DEVELOPER GUIDE | PUBLIC
Document Version: 1.75 – 2020-02-27

SAPUI5: UI Development Toolkit for HTML5

THE BEST RUN SAP

Content

SAPUI5: UI Development Toolkit for HTML5

Create apps with rich user interfaces for modern web business applications, responsive across browsers and devices, based on HTML5. (Documentation patch 1.75.0)

SAPUI5 offers powerful development concepts:

- One consistent user experience for your apps
- Responsive across browsers and devices - smartphones, tablets, desktops
- Built-in extensibility concepts at code and application level
- Data binding types and Model-View-Controller (MVC)
- Feature-rich UI controls for handling complex UI patterns and predefined layouts for typical use cases. UI controls automatically adapt themselves to the capabilities of each device.
- Full translation support
- Keyboard interaction support and accessibility features

And many more....

Check the SAPUI5 playlist in the SAP Technology YouTube channel for the latest highlights videos!

SAP Fiori apps are built with SAPUI5 and follow the SAP Fiori design guidelines to ensure consistent design and a high level of design quality. See SAP Fiori Design Guidelines.

For more information about SAP Fiori, see http://www.sap.com/fiori

> → Tip
>
> Looking for the Demo Kit for a specific SAPUI5 version?
>
> Check at https://ui5.sap.com/versionoverview.html which versions are available. You can view the version-specific Demo Kit by adding the version number to the URL, e.g. `https://ui5.sap.com/1.38.8/`
>
> For more information, see Versioning of SAPUI5 [page 29].
>
> Need a PDF version of the SAPUI5 documentation? Go to https://help.sap.com/viewer/p/SAPUI5.

What's New in SAPUI5

Find out what's new in the latest versions of SAPUI5.

Check the latest videos in the SAPUI5 playlist in the SAP Technology YouTube channel and the OpenUI5 YouTube channel.

What's New in SAPUI5 1.75

With this release SAPUI5 is upgraded from version 1.74 to 1.75.

New Features

UI Adaptation: Embedding External Content

If you have enabled an app for UI adaptation (see SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]), users of this app can now embed external content, such as maps or videos, as iFrames. This feature is available for key users at runtime in UI adaptation mode as well as for developers within the SAPUI5 Visual Editor in SAP Web IDE.

Improved Features

Card Explorer

We have introduced a schema validation feature in our samples in the Card Explorer. With this option, developers can see a more detailed report for mistakes inside the card manifest. Things like wrong names of properties, bad property types or bad structures are easily spotted. For more information, explore the samples in Card Explorer.

Currency Codes

When displaying ISO currency codes using `sap.ui.core.format.NumberFormat`, `sap.ui.model.type.Currency` or `sap.ui.model.odata.type.Currency`, the currency code is now displayed by default after the amount, ignoring locale, in order to be consistent with SAP design guidelines. The core configuration parameter `trailingCurrencyCode` can be used to switch the behavior globally.

If currency symbols are enabled (formatting option `currencyCode: false`), they continue to follow locale-specific placement.

SAPUI5 OData V2 Messages

With the new version of the SAPUI5 OData V2 model, the target of server messages is shortened by removing associated pairs of navigation properties. For example, a `/SalesOrderSet('1')/ToLineItems(SalesOrderID='1',ItemPosition='10')/ToHeader/GrossAmount` message target gets reduced to `/SalesOrderSet('1')/GrossAmount` if the `ToLineItems` and `ToHeader` navigation properties have the same relationship in the service metadata. If the second navigation property references a collection, the message target path is reduced only if the referenced entity is the same as without the navigation.

SAPUI5 OData V4 Model

The new version of the SAPUI5 OData V4 model introduces the following features:

- When displaying aggregated data with a list binding using either the `$$aggregation` binding parameter or `sap.ui.model.odata.v4.ODataListBinding#setAggregation`, you can filter by properties that are part of the original entity set but not of the result set. Note that these filters need to be provided as `sap.ui.model.Filter` objects.
- `sap.ui.model.odata.v4.Context#requestSideEffects` now supports updating in parent bindings by specifying navigation properties to the parent entities in the path expressions. The respective navigation properties from the parent binding to the binding of the context and back to the parent need to be marked as partners in the metadata.
- When using `autoExpandSelect`, paths with navigation properties can now be added to `$select`. The binding will evaluate this and automatically derive `$select` and `$expand`.
- For `sap.ui.model.odata.v4.ODataPropertyBinding`, a `$$noPatch` binding parameter is provided, so that values can be changed in the model without updating them in the back end.
- The `resume` method of the `v4.ODataContextBinding` and `v4.ODataListBinding` classes now works synchronously.

> ! Restriction
>
> Due to the limited feature scope of this version of the SAPUI5 OData V4 model, check that all required features are in place before developing applications. Double-check the detailed documentation of the features, as certain parts of a feature may be missing. While we aim to be compatible with existing controls, some controls might not work due to small incompatibilities compared to `sap.ui.model.odata.(v2.)ODataModel`, or due to missing features in the model (such as tree binding). This also applies to smart controls (`sap.ui.comp` library) that do not support the SAPUI5 OData V4 model, as well as controls such as `TreeTable` and `AnalyticalTable`, which are not supported in combination with the SAPUI5 OData V4 model. The interface for applications has been changed for easier and more efficient use of the model. For a summary of these changes, see Changes Compared to OData V2 Model [page 971].

For more information, see OData V4 Model [page 918], the API Reference, and the Samples in the Demo Kit.

Title Support in Nested Components

The `title` property can now also be defined on routing targets of type `Component`. When set with a binding syntax, it is resolved in the context of the root view of the component loaded by this target.

The router of a nested component may also have a `title` property defined on its own target(s) and eventually fire its own `titleChanged` event once such a target is displayed. A new configuration `propagateTitle` allows the `titleChanged` event to propagate from an individual `Component` target to the router of its parent component. In the routing configuration, this can also be enabled for all `Component` targets, so that it is not necessary to define the `propagateTitle` property on each `Component` target.

For more information, see Using the title Property in Targets [page 1083] and Enabling Routing in Nested Components [page 1086]. In addition, the Sample application introduced in the previous release to feature routing of nested components has been enhanced. It now shows how the new title definition and title propagation could be used in an application built with nested (or reuse) components.

Improved Controls

`sap.m.InitialPagePattern`

We have introduced the initial page floorplan as a Demo Kit sample. The floorplan allows users to navigate to a single object to view or edit it. The interaction point on the screen is a single input field and it relies on assisted input to direct the user to the object in as few steps as possible (using features such as value help and live search). For more information, see the SAP Fiori Design Guidelines and the Sample.

`sap.m.Link`

The `text` property can now be changed using UI adaptation at runtime. This enables key users to provide meaningful link text according to the application context.

For more information, see the Samples.

`sap.m.MessageBox`

We have introduced a new `emphasizedAction` property. This allows developers to specify which button in the dialog will receive the type `Emphasized`. If `emphasizedAction` is empty with no actions provided, the default value applies. For more information, see the API Reference and the Samples.

`sap.m.ObjectStatus`

We have enhanced the `sap.ui.core.IndicationColor` palette. Three new colors were added to the palette as numbers 6, 7, and 8. These colors enable developers to represent statuses that don't require a meaning in the sense of good-bad, but should be visually distinguishable. For example, statuses such as Updated, New, or Active. For more information, see the API Reference and the Sample.

`sap.ui.comp.smartfilterbar.SmartFilterBar`

- The export and import functionality of the `SelectionVariant` is now enhanced and includes the related texts that are used in the filters when fields are set. The feature is available without further configuration. For more information, see the Sample.
- We have introduced the option to define different columns in the table of the `ValueHelpDialog` and the suggestion list of the `SmartFilterBar`. In the `ValueList` annotation, each parameter can be statically annotated as important using the `Importance` annotation with `EnumMember` set to `High`. For more information, see the API Reference.
- We have enabled the `useDateRangeType` property so that it can be modified from the UI adaptation at runtime.

`sap.ui.integration.widgets.Card`

We have enhanced the capabilities of the Adaptive Card (Experimental).

- You can now load the Adaptive Card manifest/descriptor from a URL.
- The Adaptive Card supports templating, which enables the separation of data from the layout.

For more information, see the Adaptive Card in the Card Explorer.

`sap.ui.table.AnalyticalTable, sap.ui.table.Table, sap.ui.table.TreeTable`

A more comprehensive message text is now shown if no data is available because all table columns are hidden. For more information, see the Sample.

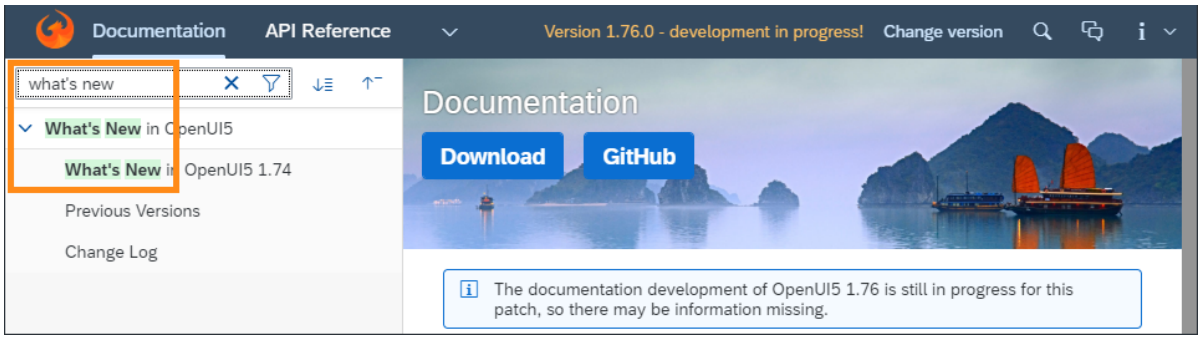
SAP Fiori Elements

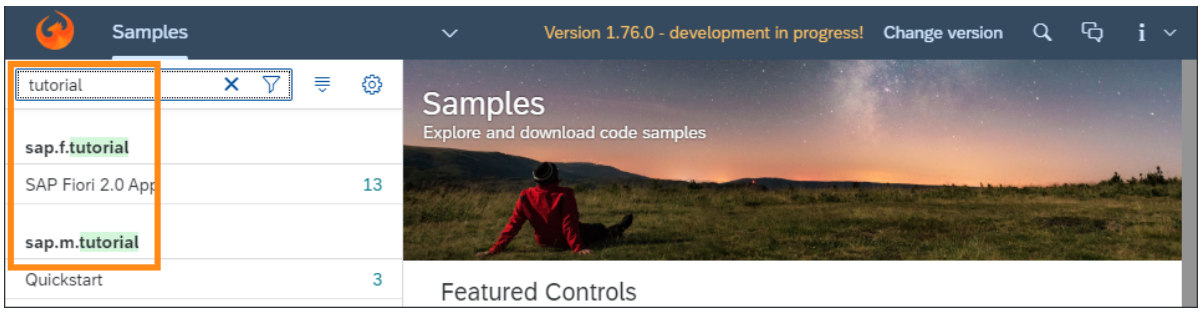
- **Option to Show an Error Message on an Empty Table in the Object Page**
  You can now show an error message strip if a table is empty or if any of the table rows contain an error in a section or subsection of an Object Page.
- **List Report Page Loaded with Data on Launch**
  By default, the list report page is loaded with data when an application is launched. For more information, see List Report Elements [page 1623].
- **Option to Set Vertical Alignment for Responsive Tables in List Report and Object Pages**
  You can now set the vertical alignment property for the whole responsive table via a manifest property `tableColumnVerticalAlignment` under the settings of `sap.ui.generic.app`. You can set the value of this string to `Top`, `Middle`, and `Bottom`.
- **Option to Define the Table Type for Each Tab of a List Report**
  You can now define the table type for tabs with different entity sets in a List Report page. For more information, see Multiple Views on List Report Tables [page 1645] and Defining Multiple Views on a List Report with Different Entity Sets and Table Settings [page 1656].

Demo Kit Improvements

Search Highlighting in the *Documentation* and *Samples* sections

The search highlighting functionality is now also available in the *Documentation* tree filter and the *Samples* list.





What's New in SAPUI5 1.74

With this release SAPUI5 is upgraded from version 1.73 to 1.74.

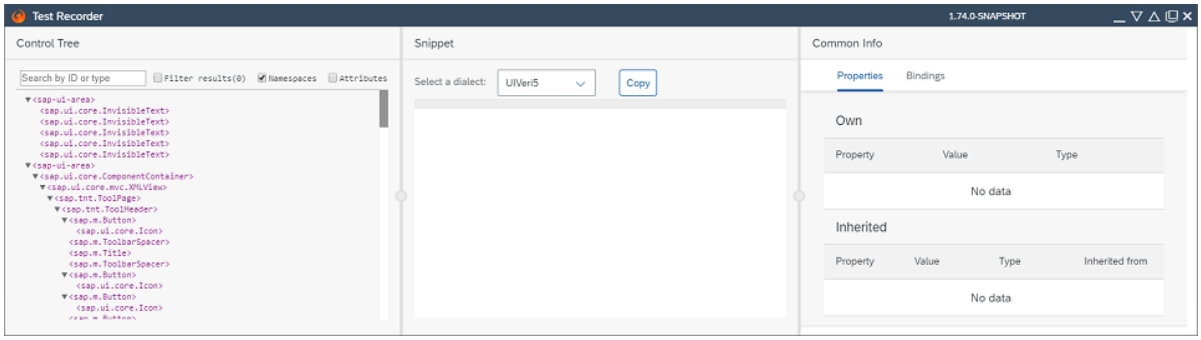
New Features

Rule Builder Control

You can model a text rule to perform an append operation on data objects or attributes in expression language 2.0. For more information, see **Operations** in Features [page 548].

Test Recorder

The Test Recorder tool is now part of the SAPUI5 framework and is available in all browsers. Use it in any SAPUI5 app to inspect the rendered user interface, view the control properties, and get hints about writing tests. The tool is aligned with the two official SAPUI5 testing tools – OPA5 and UIVeri5.



For more information, see Test Recorder [page 1251].

Improved Features

`Card Explorer`

We have enhanced the functionality to download samples from the Card Explorer, and now there are 3 optional file formats available: JSON, ZIP, and CARD. For more information, see Card Explorer.

Navigation in Nested Components

In recent releases, the capabilities to Navigate with Nested Components [page 1090] were enhanced significantly. In addition to the available documentation, a Sample has been added to showcase some fundamental possibilities of using components to structure applications and how to interconnect them via routing.

Negative Predefined CSS Margin Classes

We've introduced the following negative CSS margin classes to help you align controls that have their own default margins:

- `sapUiTinyNegativeMarginBeginEnd`
- `sapUiSmallNegativeMarginBeginEnd`
- `sapUiMediumNegativeMarginBeginEnd`
- `sapUiLargeNegativeMarginBeginEnd`

For more information, see Using Predefined CSS Margin Classes [page 1046] and the Sample.

Responsive Padding Enablement

We've introduced responsive paddings to the `sap.m.IconTabBar`, `sap.m.ObjectHeader`, and `sap.m.TabContainer` controls. For more information, see Enabling Responsive Paddings [page 1052].

SAPUI5 OData V4 Model

The new version of the SAPUI5 OData V4 model introduces the following features:

- The `autoExpandSelect:true` model setting has been enhanced for property bindings that are added later.
- We have added the `sap.ui.model.v4.ODataListBinding.getDownloadUrl` method.
- `sap.ui.model.v4.AnnotationHelper.format` can now be used for operation parameters.
- For messages returned in error responses of operation calls, targets pointing to operation parameters are now parsed correctly.
- The `sap.ui.model.v4.Context.setProperty` method can be used to set properties locally on the client by specifying `null` as the `groupId`. The set value is then not included in PATCH and POST requests to create the new entity.
- Annotation targets for (overloaded) bound operations in 4.01 Format are also supported in value list metadata.

> ! Restriction
>
> Due to the limited feature scope of this version of the SAPUI5 OData V4 model, check that all required features are in place before developing applications. Double-check the detailed documentation of the features, as certain parts of a feature may be missing. While we aim to be compatible with existing controls, some controls might not work due to small incompatibilities compared to `sap.ui.model.odata.(v2.)ODataModel`, or due to missing features in the model (such as tree binding). This also applies to smart controls (`sap.ui.comp` library) that do not support the SAPUI5 OData V4 model, as well as controls such as `TreeTable` and `AnalyticalTable`, which are not supported in combination with the SAPUI5 OData V4 model. The interface for applications has been changed for easier and more efficient use of the model. For a summary of these changes, see Changes Compared to OData V2 Model [page 971].

For more information, see OData V4 Model [page 918], the API Reference, and the Samples in the Demo Kit.

Improved Controls

`sap.m.Avatar`

We've added *remove* and *reveal* actions in the `Avatar` design-time metadata. Now, the control can be removed and revealed when using UI adaptation at runtime. For more information, see the Samples.

`sap.m.ColorPalette`

We have introducted a *Recent Colors* section, showing the last 5 recently used colors. This feature is enabled by default, making it is easier to find and select the exact colors. For more information, see the API Reference and the Sample.

`sap.m.DateRangeSelection`

We have introduced the ability to select month and year ranges. This improves the user experience when only a month or a year range has to be selected, and is defined by the `displayFormat` property. For more information, see the API Reference and the Sample.

`sap.m.List, sap.m.Table, sap.m.Tree`

The busy indicator is now displayed in the center of the visible area of the UIs of these controls and is no longer vertically centered and therefore not always visible. For more information, see the Sample.

`sap.m.NavContainer`

In version 1.69, the default value for the `defaultTransitionName` property was visually updated to behave as a slide & fade animation and the classic slide animation was no longer an option. Now, we've added the previous slide behavior as a new type of transition. To use it, set the `defaultTransitionName` property to `baseSlide`. For more information, see the API Reference and the Sample.

`sap.m.NotificationListItem`

The avatar background color can now be managed by the application developer using the new `authorAvatarColor` property. Now, if any of the `authorPicture` or `authorInitials` properties are not set, the default icon will not be displayed. For more information, see the API Reference and the Sample.

`sap.m.plugins.DataStateIndicator`

This plugin for the table controls (`sap.m.List, sap.m.Table, sap.ui.table.Table`) allows you to implement binding-related messages and show them on the UI using a message strip. For more information, see the API Reference and the Sample.

`sap.m.Select`

With the new `required` property, you can now indicate whether user input is required. This property is helpful for accessibility purposes when a single relationship between the field and a label can't be established, for example, when one label exists for multiple fields. For more information, see the API Reference.

`sap.m.StandardListItem`

The new `infoStateInverted` property changes the rendering behavior of the information state and information text. If it is set to `true`, the color defined by the `infoState` property is then shown as the background color of the information text. For more information, see the API Reference and the Sample.

`sap.m.Table`

A more comprehensive message text is now shown if no data is available because all table columns are hidden. For more information, see the Sample.

`sap.m.ViewSettingsDialog`

We have introduced a new button to display only the selected items in the filter tab, and to hide items that are not selected. This button works in combination with the Search field, so the displayed items are both filtered by title and selection. For more information, see the API Reference and the Sample.

`sap.m.Wizard`

We have enhanced the `sap.m.Wizard` control for better integration in the `sap.f.DynamicPage`. In order to make use of it, you need to make certain configurations. For more information, see the API Reference.

`sap.tnt.InfoLabel`

We have introduced the option to add an icon to the `sap.tnt.InfoLabel` content. It is defined by the new `icon` property. For more information, see the API Reference and the Sample.

`sap.ui.comp.smartfield.SmartField, sap.ui.comp.smartfilterbar.SmartFilterBar`

We've introduced support for fiscal dates. The service metadata property must be of type `Edm.String` annotated with one of the fiscal annotations (`IsFiscalYear`, `IsFiscalPeriod`, `IsFiscalYearPeriod`, `IsFiscalQuarter`, `IsFiscalYearQuarter`, `IsFiscalWeek`, `IsFiscalYearWeek`, `IsDayOfFiscalYear`). For more information, see the API Reference for `SmartField`, the API Reference for `SmartFilterBar` and the Sample for `SmartFilterBar`.

`sap.ui.comp.smartfilterbar.SmartFilterBar, sap.ui.comp.smarttable.SmartTable`

We've updated the available exclude operators to match the include operators for each field type. The change affects the *Define Conditions* tab of the `ValueHelpDialog` in `SmartFilterBar` and the *Filter* tab of `TablePersoDialog` in `SmartTable`. For more information, see the Sample for `SmartFilterBar` and the Sample for `SmartTable`.

`sap.ui.comp.valuehelpdialog.ValueHelpDialog`

We've improved the behavior of the `ValueHelpDialog` basic search for `SmartFilterBar` and `SmartField`. Now, only the entered (or modified) value is taken over into the basic search. The search is triggered automatically when the dialog opens and takes over the value into the basic search (if it's not empty). Only the entered characters are taken, regardless if the suggestion list is displaying a typeahead. For more information, see the Sample for `SmartField` and the Sample for `SmartFilterBar`.

`sap.ui.export.Spreadsheet`

This class has been further improved to facilitate the export functions. The class now supports `sap.ui.model.Binding` as a `dataSource` parameter. For more information, see the API Reference.

`sap.ui.integration.widgets.Card`

- A new experimental Calendar Card type is now available . Its purpose is to give an overview of a single entity (a person, for example). It consists of an interactive calendar, legend, and a schedule. For more information, see Calendar Card in the Card Explorer.
- We have enabled Data Sources to be used in the descriptor for `sap.ui.integration.widgets.Card`. Data Sources are named and reusable manifest entities that hold configuration settings for services. Referenced using special double-bracketed syntax, they are used to construct data request URLs. Data Sources are defined in the `sap.app` part of the manifest. For more information, see Data Sources in the Card Explorer.
- We have introduced new number formatters to represent the data on the UI in human-readable format. Now we have predefined number formatters for:
  - Currency
  - Date and Time
  - Floating-point numbers
  - Integers
  - Percent
  - Units of measurement

  For more information, see Card Formatters in the Card Explorer.
- We now also support objects as values for manifest parameters. Until now only string values were supported. For more information, see Manifest Parameters section in the Card Explorer.
- We have introduced a new experimental type of card - Adaptive Card. With this type of card, you can visualize and reuse cards created using the Microsoft Adaptive Cards specification and manifest, while achieving fully adapted SAP Fiori 3 user experience, out of the box. For more information, see Adaptive Card in the Card Explorer.

`sap.uxap.ObjectPageLayout`

You can now move the corresponding section numbers that are displayed in the `AnchorBar` when using UI adaptation at runtime. For more information, see the Samples.

SAP Fiori Elements

- **Smart Multi-Input Control in Smart Tables**
  It is now possible to include a smart multi-input field in a smart table on an object page. For more information, see Using the Smart MultiInput Control on the Object Page [page 1720].
- **Flexible Column Layout: Displayed Item on Object Page Highlighted in Master List**
  The navigation row in the table is highlighted in blue to correlate with the object being displayed in applications with flexible column layout.
- **Object Page Saves and Restores State of Control Variants**
  This means that the selected table variants are stored in the same way as the control variants for the list report page.
- **Configuration of Relevant Links**
  SAP Fiori Elements now gives you the option of configuring relevant links in Related Apps. For more information, see Enabling the Related Apps Button [page 1697].
- **Option to Set Initial Expansion Level in List Report and Analytical List Page Tables**
  You can now set `initialExpansionLevel` using the `PresentationVariant` annotation to set the number of expanded levels for tables and trees. For more information, see Initial Expansion Level for Tables in List Reports & Analytical List Pages [page 1620].
- **Themes**
  SAP Quartz Light is now available by default for applications that are generated by the SAP Web IDE wizard and plugin.

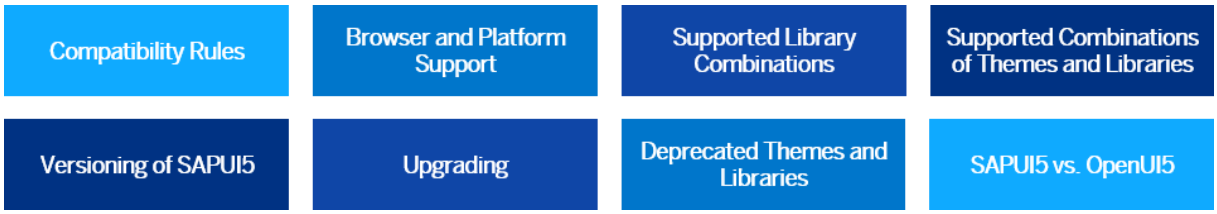
Documentation

Performance Checklist

Ensuring that your SAPUI5 apps run fast is an important topic in application development. To support you in this task, we have improved the existing performance-related documentation as well as added some new information. Please use the comprehensive Performance Checklist [page 690] as a starting point for best practices to help you to review and speed up your SAPUI5 apps.

Read Me First

Before you start using SAPUI5 productively, please read the **important information** in the section. Here you read everything you need to know about supported library combinations, the supported browsers and platforms, and so on.



- Compatibility Rules [page 17]
- Browser and Platform Support [page 20]
- Supported Library Combinations [page 26]
- Supported Combinations of Themes and Libraries [page 27]
- SAPUI5 vs. OpenUI5 [page 37]
- Deprecated Themes and Libraries [page 34]
- Upgrading [page 31]
- Versioning of SAPUI5 [page 29]

Compatibility Rules

The following sections describe what SAP can change in major, minor, and patch releases. Always consider these rules when developing apps, features, or controls with or for SAPUI5.

> ⚠ Caution
>
> As an app developer, **never** do the following:
>
> - Never manipulate HTML/CSS via JavaScript (`domRef.className = "someCSSClass";`) or directly via CSS, for example. Always follow our recommendations under CSS Styling Issues [page 1464].
> - Never use or override "private" functions that are not part of the API Reference. Private functions are typically (but not always) prefixed with a preceding "_". Always double-check the API Reference, private functions are not listed there.

API Evolution

Unless otherwise mentioned, the word "API" in this section refers to "public API", meaning functions, classes, namespaces, controls along with their declared properties, aggregations, and so on. The sole definition of the

public API is the API Reference, which is included in the SAPUI5 Demo Kit. Features that are **not** mentioned there are **not** part of the API.

The following rules apply for introducing new APIs or making incompatible changes to existing APIs:

Major release (**x**.yy.zz): A new major version can introduce new APIs or make incompatible changes to existing APIs.

Minor release (x.**yy**.zz): A new minor version can introduce new APIs but must not contain incompatible changes to **any** APIs.

Patch release (x.yy.**zz**): A new patch version only contains fixes to the existing implementation, but does not usually contain new features or incompatible API changes.

> **i** Note
>
> Exceptions to these rules are possible, but only in very urgent cases such as security issues. Such exceptions are documented in the Change Log.

Compatible Changes

The following changes to existing APIs are compatible and can be done anytime:

- Adding new libraries, controls, classes, properties, functions, or namespaces
- Generalizing properties, that is moving properties up in the inheritance hierarchy
- Adding new values to enumeration types; this means that when dealing with `enum` properties, always be prepared to accept new values, for example, by implementing a `"default"` or `"otherwise"` path when reacting on `enum` values.

Incompatible Changes

The following is not part of the public API and may **change in patch and minor releases**:

- Open source libraries (see Third-Party Open Source Libraries [page 20])
- Log messages

The following changes to existing APIs are incompatible, but **can be done in a new major release**:

- Renaming an API (library, namespace, function, property, control, events, and so on)
- Removing support for parameters
- Removing support for configuration entries
- Reducing the visibility of an API; this does not break JavaScript applications, but changes the contract
- Removing or reordering parameters in an API signature
- Reducing the accepted value range, for example, parameter of a function
- Broadening the value range of a return value (or property). Exception: enumerations
- Moving JavaScript artifacts (namespaces, functions, classes) between modules
- Replacing asserts with precondition checks
- Moving properties (and so on) down in the inheritance hierarchy

- Changing the name of `enum` values
- Changing defaults (properties, function parameters)
- Renaming or removing files

Inheritance

Inheriting from SAPUI5 objects (e.g. by calling `sap.ui.extend` on an existing control to add custom functionality) may endanger the updatability of your code.

When overriding an SAPUI5 lifecycle method (such as `init`, `exit`, `onBeforeRendering`, and `onAfterRendering`), you must make sure that the super class implementation is called, for example like this:

```
MyClass.prototype.onAfterRendering = function() {
  SuperClass.prototype.onAfterRendering.apply(this);
  // do your additional stuff AFTER calling super class
}
```

SAP might add, remove, or change the internal implementation of the parent class at any time. Especially, you should not rely on the following functionality:

- Internal structures and methods that are not part of the public API
- Any internal logic and behavior of the object that is not reflected in the public API
- The parent hierarchy of objects especially for composites where the API parent differs from the real parent (e.g. parent object > internal object > child object). For more information, see API Reference: `sap.ui.base.ManagedObject`.
- All rendering functionality of a control, including the HTML structure and CSS classes
- Naming collisions with SAPUI5 structures and methods. SAPUI5 might introduce new API or internal structures at a later point in time that collide with your implementation. To avoid collisions, a custom prefix may be applied. Don't use namespaces starting with `sap.m.*` or `sap.ui.*` in your app.

We recommend that you test inherited classes very carefully after updating SAPUI5 to make sure that the extended functionality is still working as expected.

Deprecation

If possible and appropriate, we mark old artifacts as deprecated and create new artifacts, instead of making incompatible changes. A deprecation comment in the corresponding API documentation, and perhaps also a log entry in the implementation, explain why and when an artifact has been deprecated and include tips on how to achieve the same results without using deprecated functionality.

Experimental API

Some features or controls delivered with the current SAPUI5 version are flagged as "experimental". These experimental features and controls are not part of the released scope of the delivered SAPUI5 version. Do not

use experimental features or controls in a productive environment, or with data that has not been sufficiently backed up.

Experimental features and controls can be changed or deleted at any time without notice, and without a formal deprecation process. They may also be incompatible to changes provided in an upgrade.

Third-Party Open Source Libraries

SAPUI5 contains and uses several third-party open source libraries, such as `jQuery`. These libraries can also be used by applications and/or custom control libraries, but the SAPUI5 compatibility rules described in this document do not apply to these third-party libraries.

If you want to use the third-party open source libraries included in SAPUI5, note the following restrictions:

- SAP decides which versions and modules of the used libraries are provided.
- SAP can upgrade to a higher version of the used libraries even within a patch release.
  If we change to a new default version of a library, we document our findings that might have an effect on SAPUI5 apps (see Upgrading [page 31]). Make sure that you adapt your code if necessary!
- For important reasons such as security, SAPUI5 can stop providing a library at any time.
- The third-party libraries are provided "as is". Extensions, adaptations, and support are not performed or provided by SAP.

> **i Note**
>
> Do not use different versions of these libraries as this might lead to unforeseen side effects..

For a list of the third-party open source software used in SAPUI5, see the *Included Third-Party Software* link in the *About* dialog in the Demo Kit.

Related Information

Versioning of SAPUI5 [page 29]
API Reference

Browser and Platform Support

Browser and platform support for the SAPUI5 libraries on iOS, Android, macOS, and Windows platforms.

> **i Note**
>
> The single source of truth about supported browsers and platforms is the Product Availability Matrix (PAM) that you can find at https://support.sap.com/pam. SAPUI5 is not a product of its own, so please check the PAM for the product you're using SAPUI5 with. For more information, see SAP Note 1716423.
>
> The following sections only contain additional information on restrictions and platform support information for specific SAPUI5 libraries in a summarized form.

As SAPUI5 is based on CSS3, HTML5, and the ECMAScript 5 (ES5) JavaScript API, only browsers with HTML5 capabilities are supported. In general, only major versions that are also supported by the respective platform can be supported by the SAPUI5 framework.

> **! Restriction**
>
> We currently do not guarantee that newer ECMAScript standards, such as ES6/ES2015, work with SAPUI5.

Depending on the platform your SAPUI5 apps run on, different browsers in different versions are supported. If you know which platform and which browsers are used by your users, you can decide on which libraries to use for your app.

Overview of Supported Browsers, Platforms, and Reference Devices

The following tables give a general overview of the browsers, platforms, and reference devices supported by the main SAPUI5 libraries.

Browser and Platform Support Matrix

> **i Note**
>
> SAPUI5 plans to end the support for Microsoft Internet Explorer 11 as future direction (end of 2021 or later). Legacy web applications, which use active browser plugins that require Microsoft Internet Explorer 11, cannot run embedded inside an SAP Fiori launchpad that depends on a SAPUI5 version released after support ended.
>
> Integration of SAPUI5 applications in SAP GUI for Windows through the SAP HTML Control also use the Microsoft Internet Explorer Control. Alternatives for this Microsoft Internet Explorer Control are presently under investigation.

Platform	Device Category	Platform Version	Safari	Web View	Internet Explorer	Microsoft Edge (Chromium)	Microsoft Edge (EdgeHTML)[1]	Google Chrome	Mozilla Firefox	SAP Fiori Client
Windows	Desktop	Windows 8.1	-	-	Version 11[3]	-		Latest version	Latest version and Extended Support Release (ESR)[2, 6]	-
		Windows 10	-	Latest version		Latest 2 versions				-
	Touch[4, 5]	Windows 10	-	Latest version	Version 11[3]	Latest 2 versions[6]		Latest version[6]	Latest version and Extended Support Release (ESR)[6]	Latest version

Platform	Device Category	Platform Version	Safari	Web View	Internet Explorer	Microsoft Edge (Chromium)	Microsoft Edge (EdgeHTML)[1]	Google Chrome	Mozilla Firefox	SAP Fiori Client
macOS	Desktop	Latest 2 versions	Latest 2 versions	-	-	-	Latest version[4]	-	-	
iOS	Phone and Tablet[4, 5]	Latest 2 versions	Latest 2 versions	Latest version[6]	-	-	-	-	Latest version[7]	
Android	Phone and Tablet[4, 5, 7]	As of version 5	-	-	-	-	Latest version	-	Latest version	

1) The next stable SAPUI5 release that comes after SAPUI5 1.71 will be the last SAPUI5 release to support Microsoft Edge (EdgeHTML).
2) In regards to handling touch events, there are some issues with Windows 8. For more information, see Windows 8 Support - Known Issues [page 2202].
3) Internet Explorer 11 requires add-ons *XML DOM Document* and *XML DOM Document 3.0* to be activated for XML parsing support.
4) Not supported for `sap.ui.commons` and `sap.ui.ux3`.
5) Not supported for `sap.gantt`. Note that gantt charts consuming `sap.gantt` can be displayed on tablet devices.
6) Not supported for `sap.ui.vbm`.
7) Not supported for `sap.ui.vk`.

Supported Reference Devices

For mobile operating systems, support is restricted to specific reference devices.

When creating support incidents, make sure that the device you refer to belongs to the listed ones:

> **i Note**
>
> Touch-enabled devices are not supported by the `sap.ui.commons`, `sap.ui.ux3` and `sap.gantt` libraries.

Platform	Device	End of Support Date
iOS SAP always supports the 2 latest releases of the iOS operating system, not exceeding 3 years from vendor release date. SAPUI5 supports Apple iPhone and iPad series until 3 years from the vendor device release date, except defined otherwise.	Apple iPhone 8	September 2020
	Apple iPhone X	November 2020
	Apple iPhone XS	September 2021
	Apple iPhone XR	October 2021
	Apple iPhone 11	September 2022
	Apple iPad (5th)	March 2020
	Apple iPad Pro (2nd)	June 2020
	Apple iPad (6th)	March 2021

Platform	Device	End of Support Date
	Apple iPad Pro (3rd)	October 2021
	Apple iPad Air (3rd)	March 2022
	Apple iPad Mini (5th)	March 2022
	Apple iPad (7th)	September 2022
Android Android OS based devices are very fragmented in matters of operating system variants and hardware diversity. SAPUI5 supports Samsung Galaxy S and Galaxy Tab S series until 3 years from vendor device release date, except defined otherwise.	Samsung Galaxy S8	April 2020
	Samsung Galaxy S9	March 2021
	Samsung Galaxy S10	March 2022
	Samsung Galaxy Tab S3	March 2020
	Samsung Galaxy Tab S4	August 2021
	Samsung Galaxy Tab S5e	April 2022
	Samsung Galaxy Tab S6	August 2022
Windows SAPUI5 supports Microsoft Surface Pro reference devices until 3 years from vendor device release date, except defined otherwise.	Microsoft Surface Pro (2017)	June 2020
	Microsoft Surface Pro 6	October 2021
	Microsoft Surface Pro 7	October 2022
	Microsoft Surface Pro X	November 2022

Additional Information

- **General**
  - Internet Explorer 11 (IE11) provides specific document and enterprise modes for compatibility reasons. SAPUI5 supports only the IE11 document mode. For backward compatibility, IE11 allows to enable a special enterprise mode that can simulate either an IE8 or IE7 within an IE11, which is NOT supported for SAPUI5 apps. This functionality should be used only for critical apps that require an older browser version to run. For more information, see "Fix web compatibility issues using document modes and the Enterprise Mode site list " in the Microsoft Windows IT Center.
  - The PhantomJS browser is not supported. If you are using it for testing purposes, make sure that you use version 2.x or higher. Otherwise you may get an error message, such as "`TypeError: 'undefined' is not a function (evaluating 'f.bind(null,undefined)')(line 146)`".
  - `sap.ui.core`, `sap.ui.layout`, `sap.ui.unified` are basic libraries, supporting all platforms or browsers that are supported by any of the other libraries.
- `sap.m`
  - For the `maxLines` property of the `sap.m.Text` control, multiline ellipsis handling is not supported for all browsers and devices and is not supported at all for right-to-left text direction. For more information, see Visual Degradations [page 24].
- `sap.ui.vk` - the supported browsers are all WebGL-compatible.

Visual Degradations

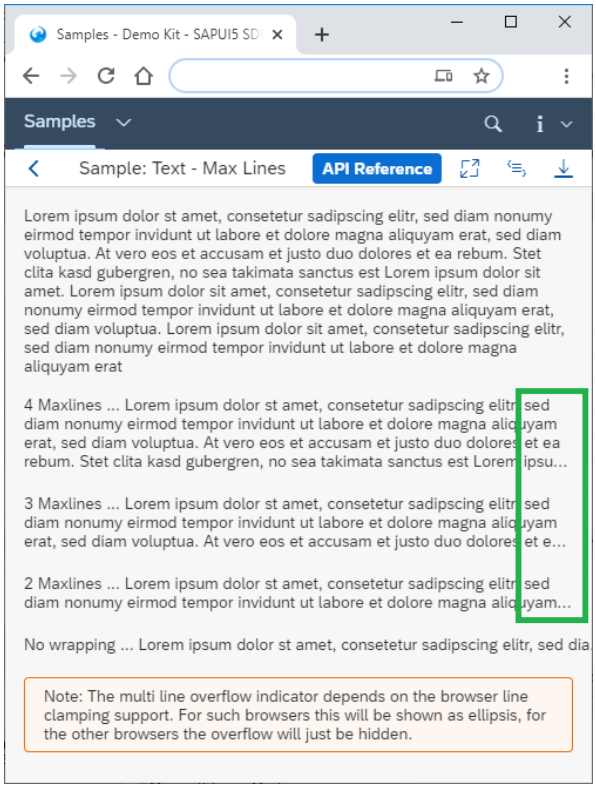
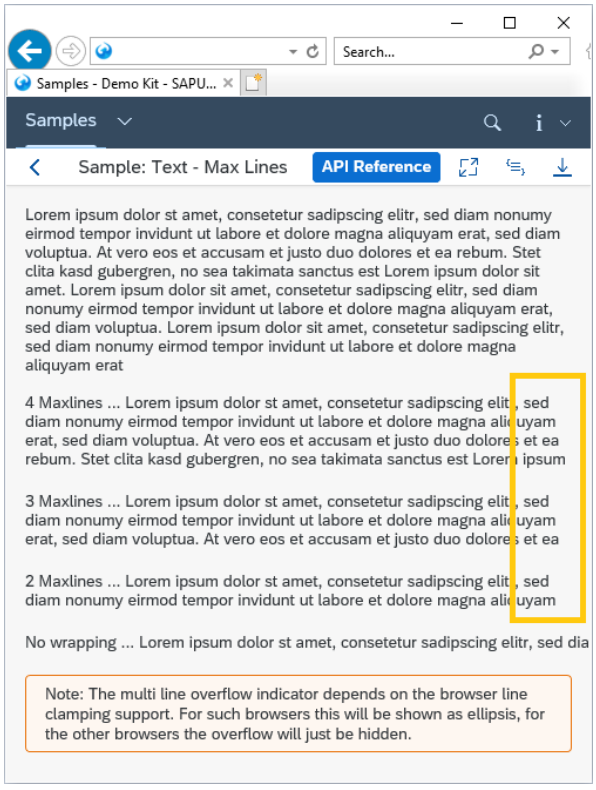
Depending on the combination of device and browser, visual degradations may occur in the `sap.m` library.

The following sections give an overview of the known degradations.

sap.m.Text - maxLines (property), sap.m.Text - text (property), sap.m.ObjectListItem - title (property), sap.m.ObjectHeader - title (property)

The visual aid for indicating multiline overflow is an ellipsis at the end of a line. This ellipsis is displayed if the text string exceeds the maximum number of lines displayed on screen. Depending on the line-clamping support offered by your browser, this visual aid may not be displayed at all, meaning the text is simply truncated without any visual indication that it is incomplete. The table below outlines which browsers fail to support the multiline ellipsis handling of the `maxLines` property, and also shows examples the visual degradation along with what the display should look like when it is supported:

Table 1: maxLines Visual Degradations

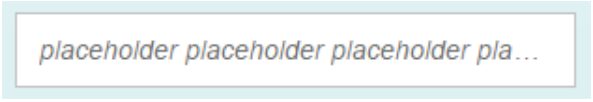
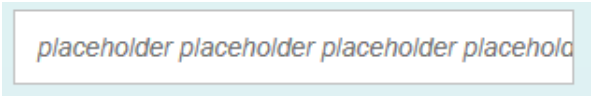
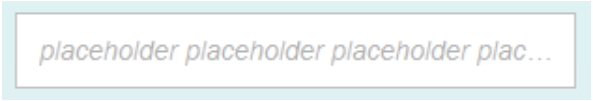
What it Should Look Like	Visual Degradation
Google Chrome	Internet Explorer 11
	

placeholder Property in sap.m.Input and sap.m.TextArea

As there is no W3C specification for how to use the placeholder property, browser handling for this property varies greatly. Some browsers use a native placeholder property, but for browsers that do not support this, SAP implements its own placeholder version.

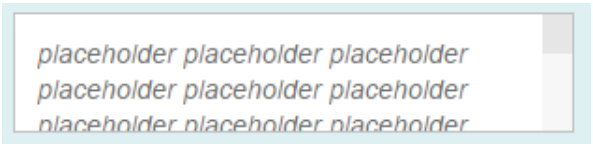
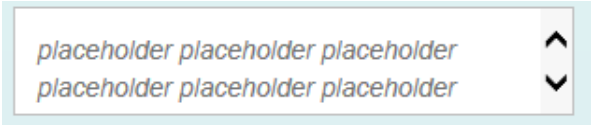
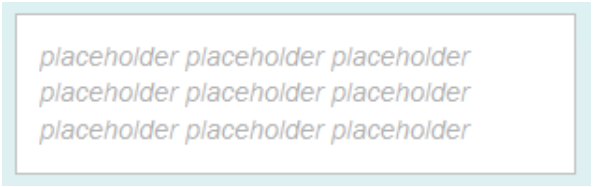
The following overview outlines which browsers use which version, and which limitations or degradations apply in each case for the `sap.m.Input` control and `sap.m.TextArea` control.

Table 2: placeholder Property in sap.m.Input

Browser	Situation
Google Chrome	Google Chrome supports the native placeholder property and displays the ellipsis correctly, indicating that the placeholder text stretches beyond the field that is currently visible 
Internet Explorer Version 11	This version supports the native placeholder property but does not display the ellipsis, instead it simply truncates the placeholder text string  **i Note** If you focus in the field, the placeholder disappears from view. If you leave the focus without typing anything, the placeholder is then displayed again.
Mozilla Firefox	Mozilla Firefox currently supports the native placeholder property and displays the ellipsis correctly, indicating that the placeholder text stretches beyond the field that is currently visible 

Whereas `sap.m.Input` contains just a single line placeholder, `sap.m.TextArea` is a multiline control, meaning it brings with it different issues to the ones listed above. These issues are different depending on the browser and are listed below.

Table 3: placeholder Property in sap.m.TextArea

Browser	Situation
Google Chrome	Google Chrome supports the native placeholder property and displays multiple lines along with a scrollbar 
Internet Explorer Version 11	Internet Explorer 11 supports the native placeholder property and displays multiple lines along with a scrollbar.  **i Note** Clicking the scrollbar sets focus from a technical perspective, meaning the placeholder text disappears and makes scrolling impossible. Scrolling with the mouse wheel does not set focus, and enables you to read the entire placeholder text.
Mozilla Firefox	Mozilla Firefox supports the native placeholder property but does not display a scrollbar or an ellipsis, instead it simply truncates the placeholder text string 

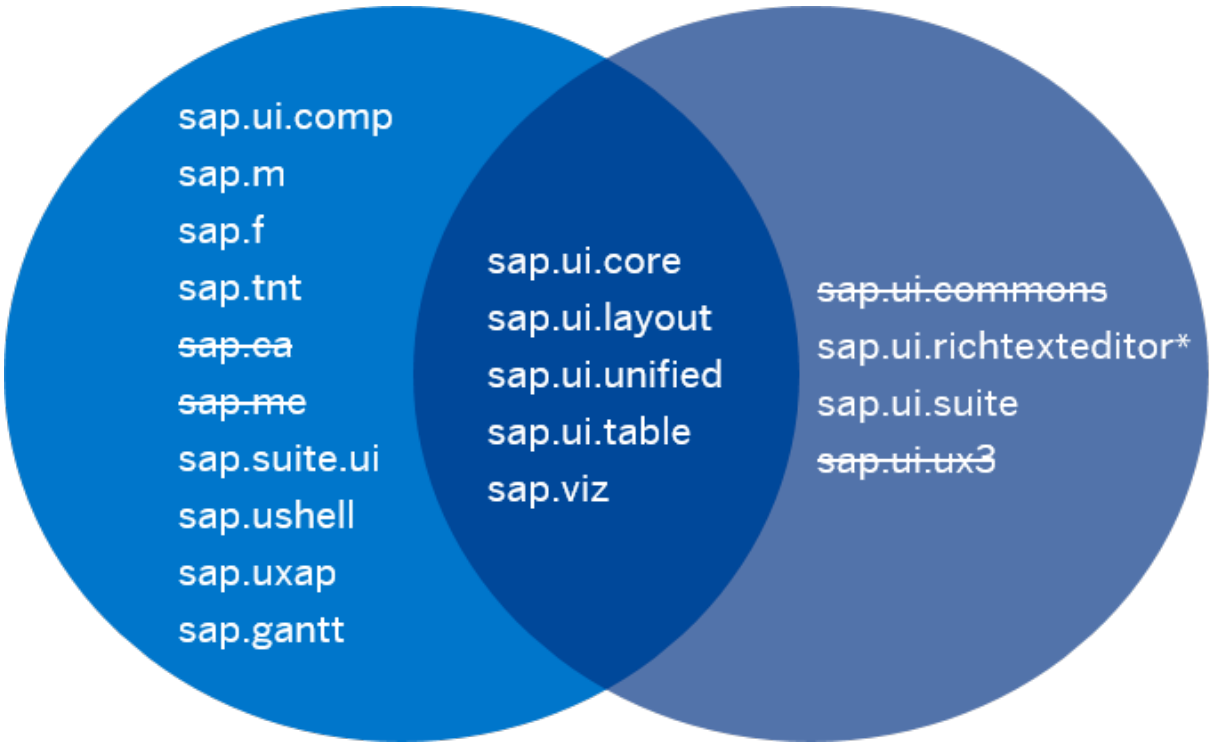
Supported Library Combinations

SAPUI5 provides a set of JavaScript and CSS libraries, which can be combined in an application using the combinations that are supported.

There are two sets of possible library combinations, which are best described using the diagram below. Any of the libraries listed on the lefthand side can be used with those listed in the middle, and any of the libraries listed on the righthand side can be used with the ones listed in the middle. The libraries listed on the lefthand side **cannot** be used in combination with the libraries listed on the right, and vice versa:

i Note

Libraries that are not mentioned explicitly, belong to the lefthand side.



*Combinations between `sap.m` and `sap.ui.richtexteditor` are allowed only for desktop apps when using version 4 of TinyMCE.

Related Information

Deprecated Themes and Libraries [page 34]

Supported Combinations of Themes and Libraries

This chapter gives an overview of the possible combinations of themes and libraries for the SAPUI5 versions that are still in maintenance.

Active Libraries

The following table shows which themes are available for the active SAPUI5 libraries. Even though the `sap_bluecrystal` and the `sap_hcb` themes are now deprecated, they currently are still available, but will not be maintained. We recommend that you migrate your existing apps to the supported themes.

Table 4: Themes for Supported Libraries

	1.38	1.44	1.46	1.65	1.69	1.74
`sap_fiori_3_dark`	n/a	n/a	n/a	n/a	n/a	■
`sap_fiori_3`	n/a	n/a	n/a	■	■	■
`sap_fiori_3_hcb` `sap_fiori_3_hcw`	n/a	n/a	n/a	n/a	■	■
`sap_belize` `sap_belize_plus`	n/a	■	■	■	■	■
`sap_belize_hcb` `sap_belize_hcw`	n/a	n/a	■	■	■	■
`sap_bluecrystal`	■	⚠	⚠	⚠	⚠	⚠
`sap_hcb`	■	■	⚠	⚠	⚠	⚠

■ = Supported, ⚠ = Not Supported

Deprecated Libraries

Should you decide to ignore the deprecation of libaries and continue, for example, using `sap.ui.commons` and `sap.ui.ux3`, we recommend you use the `sap_bluecrystal` or `sap_belize` theme.

The `sap_bluecrystal` theme is also no longer supported, but offers full coverage of the `sap.ui.commons` and the `sap.ui.ux3` library. It is currently still shipped, **but will be removed in one of the next versions.**

The `sap_belize` theme offers an initial implementation for the `sap.ui.commons` and the `sap.ui.ux3` library to allow for a smoother transition, but it is not supported for this library and will not be maintained. We recommend that you consider migrating your existing apps to actively developed libraries, such as `sap.m`, and use `sap_belize` as the default theme going forward.

Table 5: Themes for the Deprecated Libraries `sap.ui.commons` and `sap.ui.ux3`

	1.38	1.44	1.46	1.48	1.65
`sap_fiori_3`	n/a	n/a	n/a	n/a	
`sap_belize` `sap_belize_plus`	n/a				
`sap_belize_hcb` `sap_belize_hcw`	n/a	n/a			
`sap_bluecrystal`					
`sap_hcb`					
`sap_goldreflection`					
`sap_platinum`					
`sap_ux`					

= Supported, = Not Supported, = Deprecated, = Removed

Related Information

Deprecated Themes and Libraries [page 34]

Versioning of SAPUI5

Versioning and maintenance strategy for SAPUI5.

SAPUI5 uses a 3-digit version identifier, for example 1.60.2. The digits have the following meaning:

- The first digit (1) specifies the release number (major version).
- The second digit (60) specifies the version number (minor version).
- The third digit (2) specifies the patch number.

To view the documentation for a specific version, check at https://ui5.sap.com/versionoverview.html which versions are available. You can view the version-specific Demo Kit by adding the version number to the URL, for example, `https://ui5.sap.com/1.60.2/`.

To get an overview of the new features of each version, see What's New in SAPUI5 [page 6], to see the fixes contained in each patch check the Change Log.

Maintenance Strategy

Every month, SAPUI5 releases a new version for productive usage:

- On SAP Cloud Platform, all SAPUI5 versions are shipped.
- On AS ABAP, only two SAPUI5 versions per year are shipped as part of the SAP_UI component.

The release strategy follows the principle of "one innovation code line": subsequent versions ensure continuous innovation with an evolving code line.

Once a year, a version with long-term support is released and available on both SAP platforms. This SAPUI5 version is included in a release of the SAP_UI component, usually as part of SP02 of the respective SAP_UI release. All other versions do not have a maintenance period and no patches are provided. Required fixes are available with the next minor versions together with the new features. However, in exceptional cases, also the most recent version may be patched with correction code.

For the decision to consume a new version, we recommend the following guideline:

- For SAP Cloud Platform, we recommend to upgrade to the latest available SAPUI5 version.
- For AS ABAP/SAP Fiori Frontend Server, or in case regular version updates are not feasible, we recommend to update to the respective long-term maintenance versions as outlined in the *Minimal Installation Requirements* and *SAP Fiori Frontend Server* notes.

For example, the following versions have a long-term maintenance:

- 1.38 which is included in SAP_UI 7.50 and UI add-on 2.0
- 1.44 which is included in SAP_UI 7.51
- 1.52 which is included in SAP_UI 7.52
- 1.60 which is included in SAP_UI 7.53

In the version overview at https://ui5.sap.com/versionoverview.html, you can see which of the SAPUI5 versions have an extended maintenance.

For more information on the SAPUI5 maintenance strategy for SAP NetWeaver AS for ABAP, see SAP Note 2217489.

Availability of Multiple Versions on the Akamai Content Delivery Network

All SAPUI5 resources are available on the content delivery network Akamai. There, you can also find multiple SAPUI5 versions, and you can use them in your code as described in Variant for Bootstrapping from Content Delivery Network [page 696].

Check the available versions with respective maintenance status at https://ui5.sap.com/versionoverview.html.

SAPUI5 Version vs. OpenUI5 Version (Core Version)

You can find which patch versions you use in you app in the technical information dialog (Ctrl + Left Alt + Shift + P).

The patch version number of the SAPUI5 version and the OpenUI5 version that is included, may be different.

OpenUI5 includes the core runtime libraries and some additional libraries.

To access the SAPUI5 version at runtime, you can use the following code:

```
sap.ui.require([
    "sap/ui/VersionInfo",
    "sap/base/util/Version"
], function(VersionInfo, VersionUtil) {
    VersionInfo.load().then(function(oCurrentVersionInfo) {
        var oVersionUtil = new VersionUtil(oCurrentVersionInfo.version);
        // ...
    });
});
```

To access the OpenUI5 version (core version) at runtime, you use the following code:

```
var oConfig = sap.ui.getCore().getConfiguration();
var oVersion = oConfig.getVersion();
```

For more information, see SAPUI5 vs. OpenUI5 [page 37].

Upgrading

The following sections describe what you have to consider when upgrading to a new version of SAPUI5.

Upgrading from a Version Below 1.40

Older jQuery Versions Removed

As of this version, SAPUI5 only contains one version of jQuery (the current version is 2.2.3). This standard version is always used when no other jQuery version is included in the bootstrap of an app. If you need a specific jQuery version for your app, add and load it explicitly as described in noJQuery Variant for Bootstrapping [page 698]. Check the console for the related warning message if you are unsure which version you are using.

Upgrading from a Version Below 1.38

When upgrading to the current SAPUI5 version from a version below 1.38 (released in June 2016), check whether the changes listed below influence your apps.

With this SAPUI5 version, jQuery has been upgraded to version 2.2.3.

This upgrade may impact your SAPUI5 apps. The following sections give an overview of our findings and how to deal with them.

> **i** Note
>
> If you use additional open-source libraries that depend on jQuery, check whether they need to be upgraded as well.

jQuery.Event

Problem

jQuery removed some robustness checks in its event handling code. Without these checks, the `jQuery.trigger` function must only be called with events that either have no `originalEvent` property or where the `originalEvent` has all methods that `window.Event implements` (especially `preventDefault`, `stopPropagation` and `stopImmediatePropagation`).

When a `jQuery.Event` is constructed with an object literal (`properties`) or when `originalEvent` is set to some object after construction, this constraint is not fulfilled. Unfortunately, many SAPUI5 unit tests used this approach to simulate mouse or key events.

Solution

For each code that creates events, you have to apply the following fix:

The module `QUnitUtils` now rewrites the `jQuery.Event` constructor so that any given object literal is enriched with the missing methods. Most SAPUI5 unit tests include the `QUnitUtils` module early, which then fixes the issue.

Application code that needs to simulate an event, either should omit the `originalEvent` or use `Event.create` to create a native event and only then create a `jQuery.Event`.

jQuery.fn.position

Problem

`jQuery.fn.position` now takes the scroll positions of the parent element into account. This change was recoginzed as incompatible by the jQuery team and reverted with version 2.2.1.

Solution

Nothing, this is automatically fixed.

jQuery.now

Problem

`jQuery.now` is now set to `Date.now` for all browsers. But as the jQuery property represents a separate reference to that function, it is not touched by code that modifies `Date.now`, especially not by Sinon fake timers. Therefore Sinon fake timers don't work with jQuery 2.2 if Sinon is started after `jQuery`.

Solution

As a workaround, `QUnitUtils` redefines `jQuery.now` so that it delegates to the current `Date.now`. This will then use any installed fake timer.

:visible selector

Problem

Somewhere between jQuery 1.11.1 and 2.2.0, the behavior of the `:visible` selector has changed. For empty inline elements (for example, a `span` with no text), the selector now reports `:visible = true` whereas jQuery 1.1.1 reported it as `hidden`. There was only one functionality in the `sap.ui.dt` library where this change in behavior caused problems.

Solution

Instead of using `:visible`, that functionality now uses its own implementation similar to jQuery 1.11.1.

Sizzle attribute selector ([name=value])

Problem

In Microsoft Internet Explorer, the attribute selector no longer works when the attribute value is unquoted and starts with a hash (`#`). This is the case when hash-name-references are searched for, like with the `usemap` attribute of the `IMG` element.

Solution

Use quotes (`""`) for attribute values in those cases.

jQuery.isPlainObject

Problem

jQuery 2.2.0 simplified the implementation of `jQuery.isPlainObject`. As a side-effect, objects with a `constructor` property with a non-function value (like a `string` value) caused a runtime error when `jQuery.isPlainObject` was applied.

Solution

This issue is fixed with jQuery 2.2.2.

Descriptor for Applications, Components, and Libraries

If you want to add new attributes of a descriptor version higher than V2 (SAPUI5 1.30) to your existing `manifest.json` file, see Migration Information for Upgrading the Descriptor File [page 775].

Deprecated Themes and Libraries

As SAPUI5 evolves over time, some of the UI controls are replaced by others, or their concepts abandoned entirely. This chapter gives an overview on theme and library level of the most important deprecations. Individual control deprecations and more information about the controls replacing them can be found in the API reference within the Demo Kit.

Themes that are no longer supported

sap_hcb

The `sap_hcb` theme is deprecated as of version 1.48. It has been replaced by the `sap_belize_hcb` theme.

`sap_hcb` is the High Contrast Black theme used for the already deprecated `sap_goldreflection` and `sap_bluecrystal` themes. For `sap_belize` and `sap_belize_plus` there are two high contrast themes available: `sap_belize_hcb` (High Contrast Black) and `sap_belize_hcw` (High Contrast White).

sap_bluecrystal

The `sap_bluecrystal` theme is no longer supported as of version 1.40. It has been replaced by `sap_belize` as the default theme for SAPUI5 applications.

Custom themes based on `sap_bluecrystal` are no longer supported with 1.40 or higher.

sap_ux

The `sap_ux` theme is no longer supported as of version 1.40. This was one of the very first SAPUI5 themes and is only implemented by a small subset of the `sap.ui.commons` and `sap.ui.ux3` controls, which are also deprecated.This theme has been removed with SAPUI5 version 1.48.

sap_platinum

The `sap_platinum` theme is no longer supported as of version 1.40. This was one of the very first SAPUI5 themes and is only implemented by a small subset of the `sap.ui.commons` and `sap.ui.ux3` controls, which are also deprecated. This theme has been removed with SAPUI5 version 1.48.

sap_goldreflection

The `sap_goldreflection` theme is no longer supported as of version 1.40. This was one of the first SAPUI5 themes and is only implemented by the `sap.ui.commons` and `sap.ui.ux3` controls, which are also deprecated. This theme has been removed with SAPUI5 version 1.48.

Deprecated Libraries

sap.ui.commons

The `sap.ui.commons` library is deprecated as of version 1.38.

`sap.ui.commons` was available from the very beginning of SAPUI5. It contains a large number of basic UI controls like buttons, input fields and dropdowns. With version 1.16, the `sap.m` library was introduced. It contains semantically identical controls (button, input and select) that, at that time, were only supported on mobile platforms. In later versions, `sap.m` was extended to support desktop platforms as well. For more information about this, see Browser and Platform Support [page 20]. The `sap.m` controls were bigger in size to support mobile displays that require a larger touch area. The "Compact Content Density" feature explained under Content Densities [page 1142] was then added to SAPUI5, allowing you to display a control in a more compact screen size. Today, applications should be built one single time using `sap.m` (and other libraries) and their content density switched at runtime depending on the environment. As such, the redundant library `sap.ui.commons` should no longer be used.

Some of the controls in `sap.ui.commons.layout` have been replaced by the new dedicated layout library called `sap.ui.layout`, which runs on the same platforms as `sap.m`.

Some of the old controls have been made available again through the non-deprecated `sap.ui.unified` library (e.g. FileUploader, Menu), which runs on the same platforms as `sap.m`.

Some concepts such as Accordion and Row Repeater have been abandoned completely.

sap.ui.ux3

The `sap.ui.ux3` library is deprecated as of version 1.38.

This library contains more complex UI controls that were based on `sap.ui.commons` along the UX3 design approach. The `sap.m` library - successor to `sap.ui.commons` - implements SAP's new SAP Fiori design [http://experience.sap.com/fiori-design/], which supersedes UX3. As such, the `sap.ui.ux3` library is also deprecated. Some of the UX3 concepts are reflected in SAP Fiori, some are abandoned, as outlined in the following table:

Concept	What's Happened?
Feeds	Replaced by sap.m (`sap.m.Feed*`).
Notification Bar	Replaced by sap.m (`sap.m.MessagePopover` and `sap.m.semantic*`).
Thing Inspector	Indirectly replaced by a different design for displaying object data.

Concept	What's Happened?
Shell	Partially replaced by `sap.ui.unified.Shell`.
Data Set	Not part of SAP Fiori.
Exact	Not directly part of SAP Fiori. Use `sap.ui.comp.FilterBar` or `sap.m.IconTabBar` for filtering.
Quick Views	Concept abandoned as the concept of "hovering with the mouse pointer over a control" does not exist on mobile devices.

For more information about the SAP Fiori design, see the SAP Fiori design guidelines.

sap.ca

The `sap.ca` library is deprecated as of version 1.22.

This library contains a mixture of controls for various use cases. Some were replaced by `sap.m` (e.g. DatePicker, Message) and some were discontinued without being replaced (such as Hierarchy or OverflowContainer).

sap.me

The `sap.me` library is deprecated as of version 1.34.

The main feature within the `sap.me` library are the calendar controls. You find replacement controls in the `sap.ui.unified` library covering most but not all features of the `sap.me` calendars.

sap.makit

The `sap.makit` library is deprecated as of version 1.38.

The `sap.makit` library contains a few chart controls that only support mobile platforms and do not support accessibility measures. This library is replaced by `sap.viz`.

Related Information

Index of Deprecated APIs
Supported Library Combinations [page 26]
Supported Combinations of Themes and Libraries [page 27]

SAPUI5 vs. OpenUI5

With SAPUI5 and OpenUI5 we provide two deliveries of our UI development toolkit. Both are very closely related, but have their differences.

Licenses

The main difference is the license.

OpenUI5 is Open Source, free to use, released under the Apache 2.0 license. Since we also use many Open Source libraries, we try to return the favor and also benefit from the experience and knowledge of developers all over the world.

SAPUI5 is not a separate SAP product with a separate license. It's integrated, for example, in the following products:

- SAP HANA
- SAP Cloud Platform
- SAP NetWeaver 7.4 or higher (included in the UI technologies (`SAP_UI`) component)
- User interface add-on for SAP NetWeaver Application Server 7.3x

Content

The easiest way to get an overview of which libraries are delivered is to have a look at the *API Reference* of the each Demo Kit. You'll see that the list of libraries in SAPUI5 is much longer... which in no way means that OpenUI5 provides just a very limited scope!

Most importantly, the core containing all central functionality and the most commonly used control libraries is identical in both deliveries. (For example, `sap.m`, `sap.ui.layout`, `sap.ui.unified`.)

So OpenUI5 also gives you all the important features needed to build feature-rich Web applications.

The additional libraries in SAPUI5 include more controls on top, like charts, and SAPUI5 also lets you use 'smart controls', for example, which are controls that are automatically configured by OData annotations from the back end. The exact feature range of SAPUI5 also depends on the platform you're using. For example, you can only use the ABAP repository with SAP NetWeaver and not on SAP Cloud Platform.

Contributing to OpenUI5

OpenUI5 is Open Source, and is available on GitHub .

If you find a bug or have an idea for a new feature - just go ahead and propose a GitHub issue or a change. But before you do so, please just read our guidelines first: Contributing to OpenUI5 .

Resources

For the OpenUI5 version, visit http://openui5.org/ where you can download the runtime and the Demo Kit (SDK) at http://openui5.org/download.html.

For the SAPUI5 resources, check your platform installation.

Both resources are also available online via the content delivery network provider Akamai at `https://openui5.hana.ondemand.com/` and `https://sapui5.hana.ondemand.com/`.

Compatibility of OpenUI5 and SAPUI5

Technically, you can switch between OpenUI5 and SAPUI5 (providing you have the respective license), e.g. if you want to use the SAPUI5-specific features.

Just check first which SAPUI5 version you need, because the version numbers of OpenUI5 and SAPUI5 might differ on patch level (last number). You can find this information in the technical information dialog (`Ctrl` + `Alt` + `Shift` + `P`).

If you're using the content delivery network, you can simply replace the bootstrapping reference to `https://openui5.hana.ondemand.com/<1.xx.yy>/` with a reference to `https://sapui5.hana.ondemand.com/<1.xx.zz>/`. For more information, see Variant for Bootstrapping from Content Delivery Network [page 696].

For all other cases, replace the runtime. Since the technical names (of controls, libraries, etc.) and APIs are the same in both OpenUI5 and SAPUI5, the code will still work and you can start enhancing it directly.

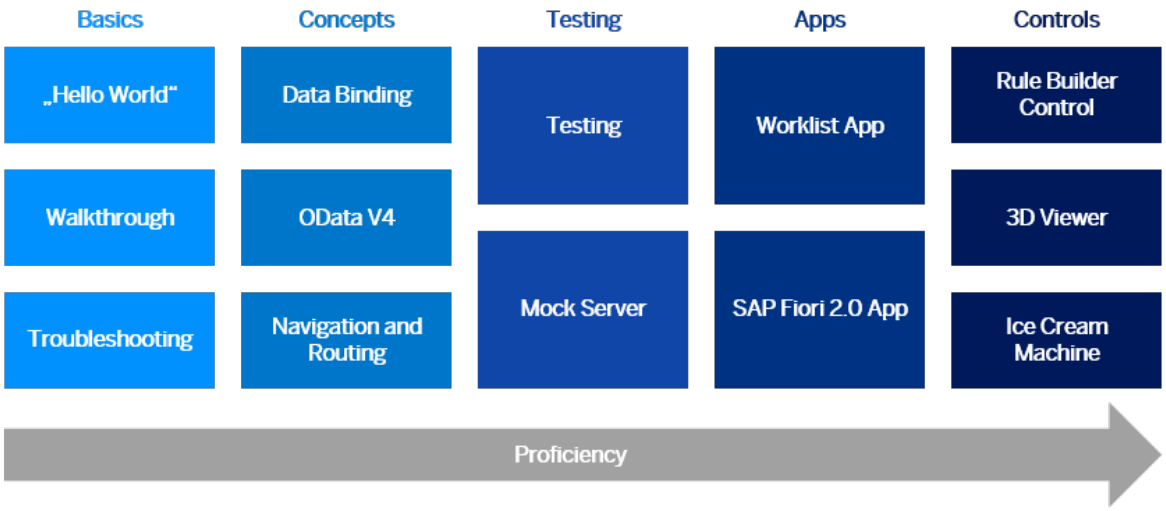
Get Started: Setup, Tutorials, and Demo Apps

Set up your development environment and go through our tutorials. They introduce you to all major development paradigms of SAPUI5 using practical examples in an interactive format. The demo apps show SAPUI5 in action.

Prerequisites and Setup

- You should be familiar with JavaScript.
- Set up your development environment: Get SAPUI5 and place it in a location, which can be accessed in the resources folder (not necessary for SAP Web IDE). For more information, see Development Environment [page 41].
- Set up a folder where you would place the application content. We will refer to this folder as the “app folder”.

Learning Path



- Quick Start [page 57]
- Walkthrough [page 69]
- Troubleshooting [page 194]
- Data Binding [page 219]
- OData V4 [page 261]
- Navigation and Routing [page 291]
- Testing [page 368]
- Mock Server [page 432]
- Worklist App [page 447]
- SAP Fiori 2.0 App [page 488]
- 3D Viewer [page 607]
- Rule Builder Control [page 536]
- Ice Cream Machine [page 640]

→ Tip

Learn with openSAP:

The openSAP course Developing Web Apps with SAPUI5 introduces you to the main concepts of SAPUI5.

The JavaScript exercises for each unit will give you the technical background needed to develop your own responsive Web apps. We'll start from scratch with the very basics and lots of hands-on coding. As we go through the weeks of this course, you'll learn more about the powerful development concepts and truly master SAPUI5.

The openSAP course Evolved Web Apps with SAPUI5 for more experienced SAPUI5 developers and ambitious beginners introduces more advanced scenarios and concepts.

Downloading Code for a Tutorial Step

To download the code from the Demo Kit, follow these steps:

1. Choose the link in the *Coding* section of the tutorial step you want to work on or find the code in the *Samples* section of the Demo Kit (filter by "*Tutorial*" to get a list of the tutorials that are available).
2. Choose the icon with the *Show source code for this sample* tooltip in the right-hand part of the header bar to display all files included in this sample.
3. Choose the *Download* button. A `zip` file is downloaded to your local machine.
4. Extract or upload the `zip` file to your development environment.
5. Adjust the project configuration files to match your development environment as described below.
6. Test the project by calling one of the HTML pages in your development environment and make sure that the app is displaying the features exactly as shown in the preview of the step.

Adapting Code to Your Development Environment

You might have to adapt parts of the coding to your local development environment to make the app work. Please check the following settings carefully:

- **Project Path and Deployment**
  All tutorials assume that the app is deployed and can be accessed under a certain path on a web server. You will not be able to run the app without a Web server as the browser does not allow you to load the required resources locally due to security restrictions.
- **SAPUI5 Resources**
  You can either download and deploy the runtime to your (local) Web server or reference the CDN version located at `https://sapui5.hana.ondemand.com/resources/sap-ui-core.js`. Some development environments such as the SAP Web IDE also provide a local runtime for testing purposes. If you download the code from the samples in the Demo Kit, you will have to adapt the **resource path in the bootstrap section** of all HTML pages included in the project. In the tutorial code, we assume that SAPUI5 can be accessed from the `/resources` path of the server.
- **Accessing Remote Services**
  Browsers typically prevent accessing remote resources due to the Cross-Origin Resource Sharing (CORS) policy. If you would like to call a real service or remote resources, you will have to either configure the development environment or the remote server to accept these requests. This strongly depends on the development environment and is described in more detail below.

Troubleshooting

If you get stuck, check the Troubleshooting [page 1314] section under *Essentials*, or refer to the Troubleshooting tutorial [page 194].

If you can't fix the problem, try downloading the solution of the previews or current step. This should get your project fixed again, just don't forget to check the resource path and the project configuration files again.

See SAPUI5 in Action - Our Demo Apps

If you want to see some practical examples for SAPUI5 apps, check out our Demo Appsour *Demo Apps* section in the Demo Kit. These are fully-functional apps that showcase certain floorplans, control libraries, or other SAPUI5 features. You can also download the source code of each demo app to find out how everything works together. Feel free to explore!

Related Information

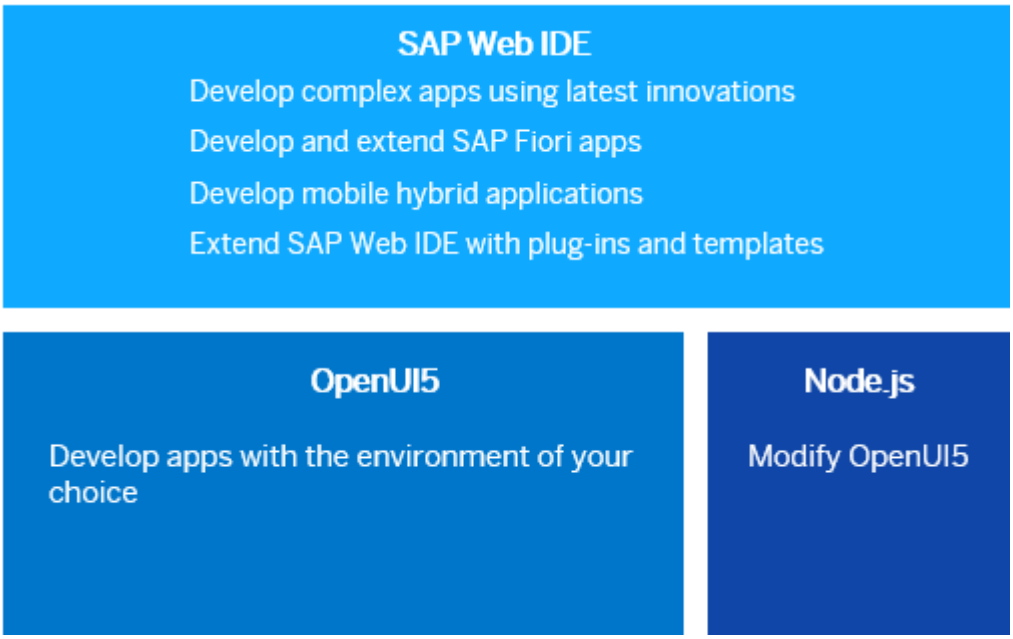
App Templates: Kick Start Your App Development [page 1399]
Demo Apps [page 671]

Development Environment

This part of the documentation gives you guidance on the most common and recommended use cases of the installation, configuration, and setup of the SAPUI5 development environment.

> **i** Note
>
> You can use SAPUI5 on different platforms. The license and maintenance conditions of the respective platforms also apply for SAPUI5. If you use SAPUI5 tools on SAP Cloud Platform, for example, the license and maintenance conditions of the SAP Cloud Platform apply.

Depending on your use case, you can choose one of the following development environments.



- App Development Using SAP Web IDE [page 44]
- App Development Using OpenUI5 [page 42]
- Developing OpenUI5 [page 51]

> **i** Note
>
> The SAPUI5 Tools for Eclipse as a development environment are no longer supported with SAPUI5 versions after 1.71. For more information, see SAPUI5 Tools for Eclipse – Now is the Time to Look for Alternatives. If you are using the SAPUI5 Tools for Eclipse with SAPUI5 version 1.71 or earlier, please consult the documentation for your specific SAPUI5 version.

App Development Using OpenUI5

Develop apps using OpenUI5 and the development environment (editor and Web server) of your choice. You can either download all of the sources or refer to the online version of OpenUI5.

Download OpenUI5

The default way of downloading and installing OpenUI5 is to get the runtime from the OpenUI5 website at http://openui5.org and deploy it on a Web server.

1. Go to https://openui5.org/releases/ and choose *Download Stable Release* to download a ZIP file containing the current stable release of the OpenUI5 sources.

2. Unzip this file and put the entire content on the Web server where your application is running (or you can even package it within your application to deploy it along with the app).
3. In your app's main HTML file, load OpenUI5 by referring to the `resources/sap-ui-core.js` file that was contained in the ZIP file. (See Standard Variant for Bootstrapping [page 694].)

Using OpenUI5 Sources from a Content Delivery Network

If you don't want to download the files and don't want to include them in your deployment, you can use the online version of OpenUI5. For more information, see Variant for Bootstrapping from Content Delivery Network [page 696].

> **i Note**
>
> You can find a list of all available OpenUI5 versions here: https://openui5.hana.ondemand.com/versionoverview.html.
>
> Only use the *Stable* version for productive apps. Nevertheless, if you also want to test the *Nightly* version, you are very welcome to send us your feedback!

Consume OpenUI5 Using the Node Package Manager

You can also use the node package manager (npm) to develop your applications. For this, you need to add the OpenUI5 dependencies to your package.json file:

```
{
  ...
  "dependencies": {
    "@openui5/sap.m": "^1.60.0",
    "@openui5/sap.ui.core": "^1.60.0",
    "@openui5/sap.ui.layout": "^1.60.0",
    "@openui5/themelib_sap_belize": "^1.60.0"
  },
  ...
}
```

To install single dependencies for your application via npm, simply run the npm install command from your terminal:

```
npm install @openui5/sap.ui.core @openui5/themelib_sap_belize [...]
```

You can find a sample application, which uses this mechanism and describes its usage, on GitHub at https://github.com/SAP/openui5-sample-app .

For more information on how to use npm and how to serve your application based on these packages, see the UI5 tooling documentation on GitHub at https://sap.github.io/ui5-tooling/ .

App Development Using SAP Web IDE

SAP Web IDE is a web-based development environment that is optimized for developing SAPUI5 complex apps using the latest innovations, developing and extending SAP Fiori apps, developing mobile hybrid apps, and extending SAP Web IDE with plug-ins and templates.

Key use cases:

- Develop new SAP Fiori apps and SAPUI5 apps
- Extend SAP Fiori apps
- Develop SAPUI5 mobile hybrid apps (HAT plug-in)
- Extend SAP Web IDE with new plug-ins and templates

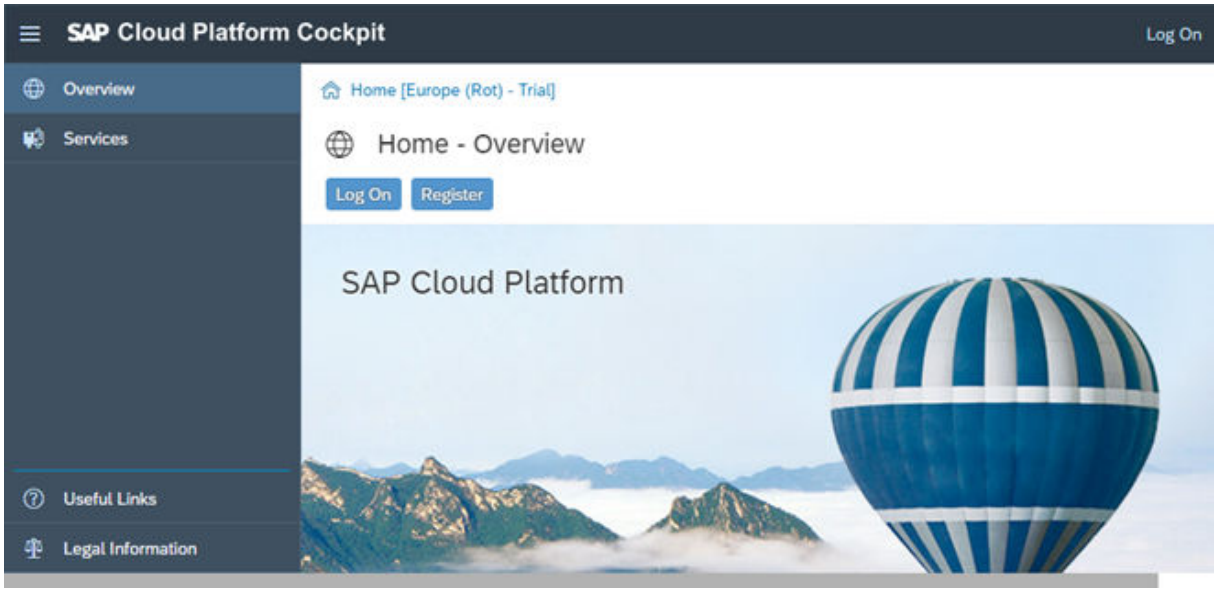
A trial version of SAP Web IDE can be accessed through the SAP Cloud Platform.

For more information about SAP Web IDE, see the documentation for SAP Web IDE on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_Web_IDE.

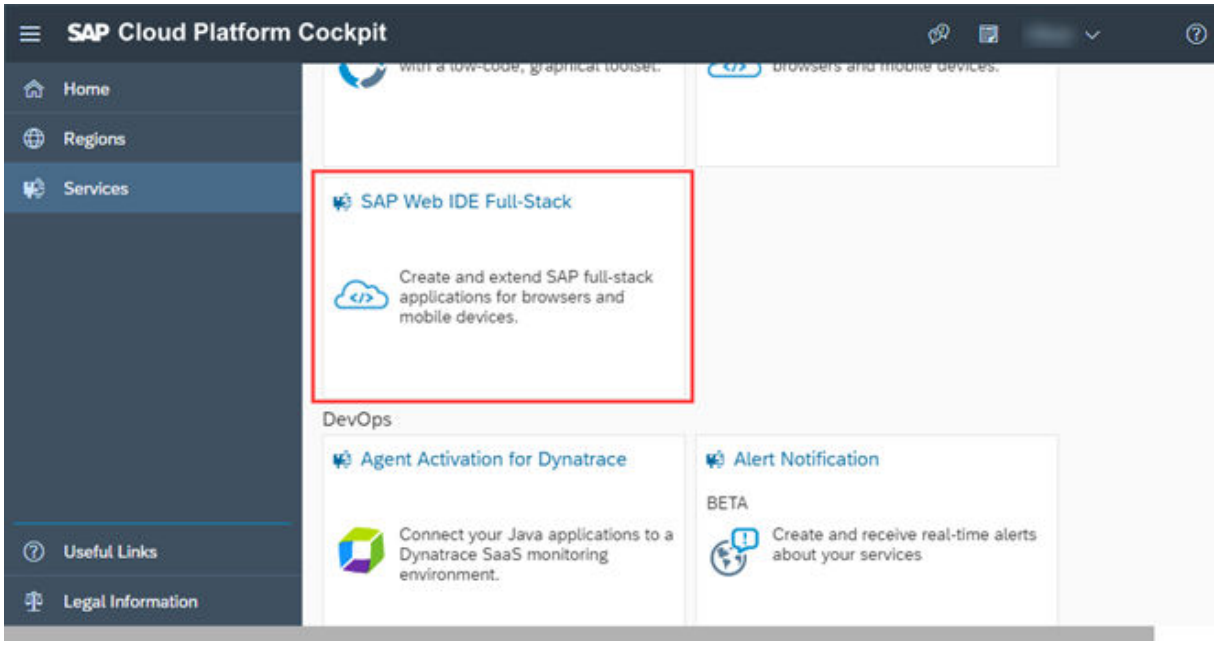
Get a Trial Account and Access SAP Web IDE

Steps for creating an SAP Cloud Platform trial account

If you do not have access to SAP Web IDE, you can create a free account. To create an account, simply register an SAP Cloud Platform trial account at https://account.hanatrial.ondemand.com/, and log on afterwards.



After you have logged on, choose *Services* in the navigation bar of the SAP Cloud Platform cockpit, and open the detailed information on your SAP Web IDE by choosing the SAP Web IDE Full-Stack tile.



> **i** Note
>
> Do **not** choose the simple SAP Web IDE tile (without "Full-Stack" in the title), because this service is not longer available for our purposes.

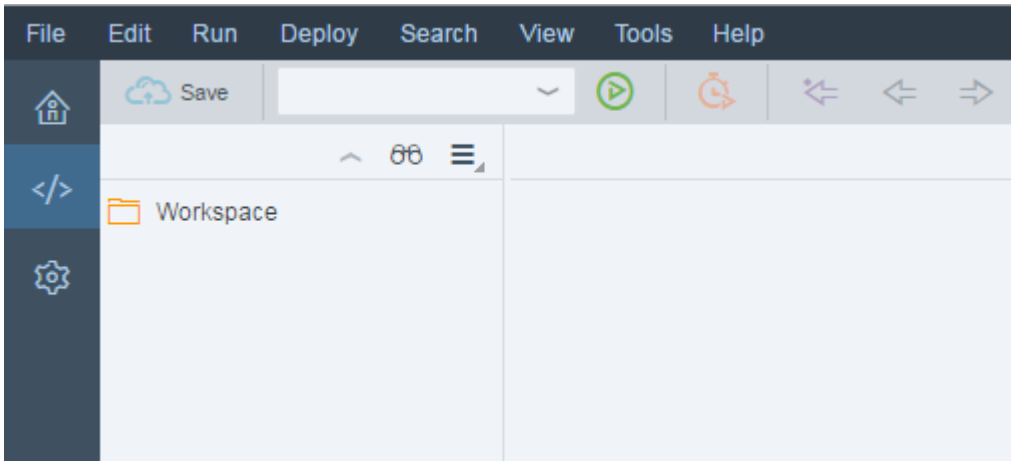
Selecting *Go to Service* leads you to your personal SAP Web IDE.

> **→** Tip
>
> You can bookmark this link to access SAP Web IDE later.

Start SAP Web IDE

Initial Steps in SAP Web IDE

1. Open SAP Web IDE and wait until the initialization has finished.
   When you start it for the first time, you will see a home screen containing more information about SAP Web IDE.
2. Change to the *Development* perspective by clicking the icon with the code symbol on the left sidebar.
   You now see a folder list with an entry *Workspace* on the left side and an empty code editor on the right side.



3. Create your project within the *Workspace* folder by choosing *File* > *New* > *Folder* from the menu or `Ctrl` + `Alt` + `Shift` + `N`. Enter, for example, **`myProject`** as the folder name.

Create a neo-app.json Project Configuration File

The `neo-app.json` file contains all project settings for SAP Web IDE and is created in the root folder of your project. It is a JSON format file consisting of multiple configuration keys. The most important setting for you to configure is the path where the SAPUI5 runtime is located when starting the app.

You do this using the “routes” key and defining an array of resource objects. For running an SAPUI5 tutorial, you only need two entries - one that configures SAPUI5 to be available with the path `/resources`, and another one that configures the test resources needed for the SAP Fiori launchpad integration with the path `/test-resources`.

Create two configuration objects that contain a `path`, a `target`, and a `description` attribute with more configuration settings. The `path` and the `entryPath` values will point to the location on the server where the SAPUI5 resources will be stored.

SAP Web IDE reads these settings automatically when running the app. You can see the whole configuration file in the code block below. Optionally, you can add the key `welcomeFile` to configure the entry point to your app. In web applications, this is typically the `index.html` file.

> **i Note**
>
> Depending on which SAP Web IDE version you are using, you might have to configure the project to run against the "snapshot" version of SAPUI5, otherwise the application will be launched with the SAPUI5 release that is delivered with SAP Web IDE. This is usually the latest version that is released publicly to customers.
>
> You can check which version of SAPUI5 is loaded by opening the SAPUI5 debugging tools with `CTRL` + `SHIFT` + `ALT` + `P`. If the version is too old for certain features of the tutorial, you have to add the `version` attribute to the target configuration entry and set the value to `snapshot`.

Procedure

1. Select the *New File* icon and enter **`neo-app.json`** as the file name.
2. Open the newly created file from the tree structure on the left side of the screen.
3. Paste the following code in the `neo-app.json` and select *Save*:

```
{
  "welcomeFile": "index.html",
  "routes": [
    {
      "path": "/resources",
      "target": {
        "type": "service",
        "name": "sapui5",
        "version": "snapshot",
        "entryPath": "/resources"
      },
      "description": "SAPUI5 Resources"
    },
    {
      "path": "/test-resources",
      "target": {
        "type": "service",
        "name": "sapui5",
        "entryPath": "/test-resources"
      },
      "description": "SAPUI5 Test Resources"
    }
  ]
}
```

Create an index.html File

A minimalistic `index.html` file is needed to test the project configuration. This file contains the SAPUI5 bootstrap and an `sap.m.Text` control that displays the text "**SAPUI5 is loaded successfully!**".

1. Choose the *New Folder* icon in the header toolbar and enter **`src`** as the folder name.
2. Select the newly created folder and create a new `index.html` file inside it by choosing the *New File* icon.
3. Paste the following code in the newly created `index.html` file and select *Save*:
   index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta charset="utf-8">
        <title>SAPUI5 Walkthrough</title>
        <script
            id="sap-ui-bootstrap"
            src="/resources/sap-ui-core.js"
            data-sap-ui-theme="sap_belize"
            data-sap-ui-libs="sap.m"
            data-sap-ui-compatVersion="edge"
               data-sap-ui-async="true"
               data-sap-ui-onInit="module:my/app/main"
               data-sap-ui-resourceRoots='{"my.app": "./"}'
             ></script>
```

```
    </head>
    <body class="sapUiBody" id="content">
    </body>
</html>
```

4. Create new file `main.js` and paste the following code into it:
   main.js

```
sap.ui.define(['sap/m/Text'], function(Text) {
    new Text({
        text: "OpenUI5 is loaded successfully!"
    }).placeAt("content");
});
```

> ⚠ Caution
>
> Adapt the path where the resources are located (`src="/resources/sap-ui-core.js"`) according to your installation. For OpenUI5 you can use `src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"`. For accessing SAPUI5 on the SAP Cloud Platform, for example, use `src="https://sapui5.hana.ondemand.com/resources/sap-ui-core.js"`.
>
> You can use this reference to the latest stable version of SAPUI5 for the tutorial or for testing purposes, but never use this for productive use. In an actual app, you always have to specify an SAPUI5 version explicitly.
>
> For more information, see Variant for Bootstrapping from Content Delivery Network [page 696].

Run the App

SAP Web IDE comes with integrated testing features that let you run the app on a central server without having to set up any additional infrastructure. You can run the app by selecting the `src/index.html` file and clicking the *run* button in the header toolbar.

This launches the app on a central server and a testing tool that allows you to configure the screen size and orientation of the device. This feature can be used to test apps that are specifically targeted for mobile, tablet, and desktop devices. You can change the resolution and the orientation in the header bar very easily.

If you don't want to run the app in the testing tool, you can adjust the *Run Configurations* for the project:

1. Right-click any file in the project and select *Run* > *Run Configurations*.
2. Choose + and select *Web Application* to add a new run configuration.
3. To save the configuration and run your project, choose *Save and Run*.

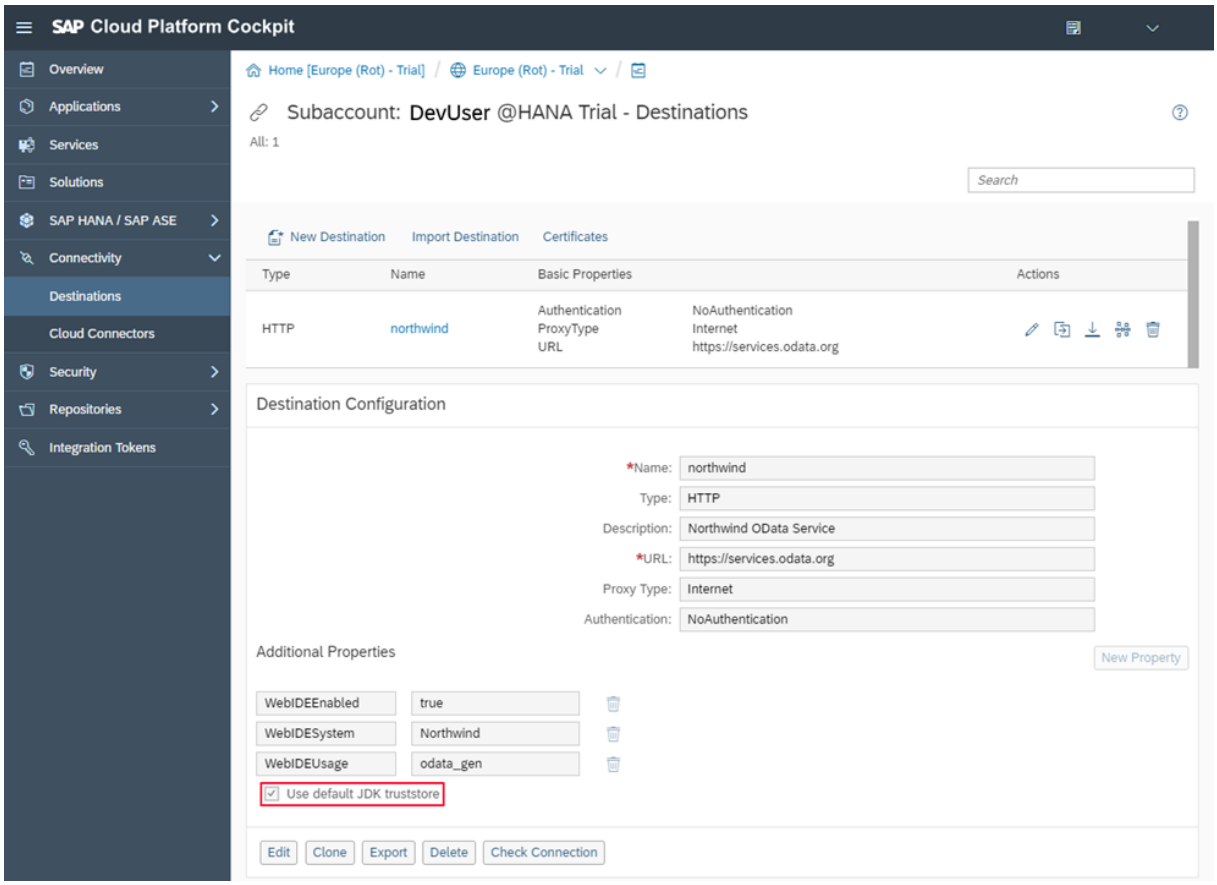
For more information on how to run projects, search for *Configuring How to Run Projects* in the SAP Web IDE Developer Guide for the SAP Cloud Platform on the SAP Help Portal at https://help.sap.com/viewer/p/CP.

Create a Northwind Destination

Configure a destination in the SAP Cloud Platform Cockpit in order to bypass the same-origin policy of the browser.

To be able to test your app, you can use a remote OData service that provides product data from the Northwind demo service of the OData group.

In the navigation bar of the SAP Cloud Platform Cockpit, choose *Destinations* and then choose *New Destination* in the toolbar.



Enter the following values into the corresponding fields:

Field	Value
Name	`northwind`
Type	`HTTP`
Description	`Northwind OData Service`
URL	`https://services.odata.org`
Proxy Type	`Internet`

Field	Value
Authentication	**NoAuthentication**

Also, enter the following properties in the section *Additional Properties*:

Property	Value
`WebIDEEnabled`	**true**
`WebIDESystem`	**Northwind**
`WebIDEUsage`	**odata_gen**
`Use default JDK truststore`	Set the checkmark.

neo-app.json

With this configuration you can use the destination for any app inside SAP Web IDE. Whenever an app calls a (local) service beginning with `/destinations/northwind/*`, the created destination becomes active as a simple proxy. This helps to prevent any possible issues related to the same-origin policy of browsers. For this, you need to add another route to the `neo-app.json`:

```
{
  "welcomeFile": "index.html",
  "routes": [
    {

      "path": "/resources",
      "target": {
        "type": "service",
        "name": "sapui5",
        "version": "snapshot",
        "entryPath": "/resources"
      },
      "description": "SAPUI5 Resources"
    },
    {
      "path": "/test-resources",
      "target": {
        "type": "service",
        "name": "sapui5",
        "entryPath": "/test-resources"
      },
      "description": "SAPUI5 Test Resources"
    },
    {
      "path": "/destinations/northwind",
      "target": {
        "type": "destination",
        "name": "northwind"
      },
      "description": "Northwind OData Service"
    }
  ]
```

```
}
```

webapp/manifest.json

In the app descriptor, the service URL is then defined relative to the destination path specified above:

```
...
"sap.app": {
 "dataSources": {
   "": {
     "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/",
     "type": "OData",
     "settings": {
       "odataVersion": "2.0"
     }
   }
 }
}
...
```

Developing OpenUI5

This section provides information for the initial setup, development workflow, and how tests are executed.

Setting Up the OpenUI5 Development Environment

OpenUI5 content is developed in an environment based on Node.js. UI5 Tooling is used as development server and build tool.

Basic Setup

The basic setup allows you to start a server for the OpenUI5 project in an easy way:

1. Install Node.js. This also includes npm, the node package manager.
2. Clone the OpenUI5 Git repository. You can download and install Git from http://git-scm.com/download :

   ```
   git clone https://github.com/SAP/openui5.git
   ```
3. Install all npm dependencies. Optionally, you can also use Yarn for this. This is described in the Advanced Setup section.

   ```
   cd openui5
   npm install
   ```
4. Start the server and open the TestSuite:

   ```
   npm run testsuite
   ```

To just start the server, enter:

```
npm start
```

5. Your default browser should open automatically and show the TestSuite: You're done!

Configuring the TestSuite Server

The OpenUI5 TestSuite server can be configured using environment variables. For example, to allow remote access to the server, that is, from an interface other than your computer's loopback/localhost, you can configure the server as follows:

```
OPENUI5_SRV_ACC_RMT_CON=true npm start
```

The following server configuration is available:

- `OPENUI5_SRV_OPEN=index.html`
- `OPENUI5_SRV_ACC_RMT_CON=true`
- `OPENUI5_SRV_PORT=9090`

Advanced Setup

The basic setup described above uses a custom setup focused on starting the UI5 server for the OpenUI5 TestSuite project in an easy way.

The advanced setup allows you to use the UI5 CLI and all of its features. The use of Yarn is required in this setup, as npm cannot handle workspaces yet.

Use the advanced setup if you plan to do any of the following:

- Build an OpenUI5 project.
- Serve a project with HTTPS or HTTP/2.
- Use any of the other UI5 CLI features and parameters.

For the advanced setup, proceed as follows:

1. Install the UI5 Tooling CLI globally, seeUI5 Tooling: Installing the UI5 CLI .
2. Install Yarn. See FAQ: What's the thing with Yarn? .
3. In the OpenUI5 repository root directory, install all dependencies using Yarn. This also links all OpenUI5 libraries between each other.

   ```
   yarn
   ```

4. Navigate into the TestSuite project and start the UI5 server:

   ```
   cd src/testsuite
   ui5 serve --open index.html
   ```

You can now use the UI5 CLI in any of your local OpenUI5 libraries. Whenever you make changes to your OpenUI5 repository's `node_modules` directory (e.g. by executing `npm install`), you may need to recreate the links between the OpenUI5 libraries. You can always do this by executing `yarn` in the OpenUI5 root directory.

Legacy Setup

You can continue to use the legacy Grunt-based setup. However, the setups described above are recommended for working with the OpenUI5 repository.

To use the legacy setup, execute `npm run start-grunt`. Note that in the past this was the default `npm start` behavior.

The legacy setup has the following differences to the standard setups as described above:

- `testsuite` path prefix:
  Standard setup: `http://localhost:8080/test-resources/testsuite/testframe.html`
  Legacy setup: `http://localhost:8080/testsuite/test-resources/testsuite/testframe.html`
- SDK documentation generated by `grunt docs` can only be served using the legacy setup.

The Development Process

For a regular development, no build is required. Just modify any source file and reload your browser. Now that's simple, isn't it?

This build-free development process does not feature optimized runtime performance. For example, there are many small requests, which would not be acceptable for remote connections. But it is the most convenient way to modify the OpenUI5 sources. Under the hood there are mainly two mechanisms applied that adapt the sources:

- The Git repository path contains a folder named like the respective control library (e.g. "sap.m"), which is omitted at runtime. The UI5 CLI server is mapping the locations.
- The CSS files are transformed (server-side) by the LESS pre-processor during the first request after a CSS file has been modified. This includes mirroring for right-to-left support.

When you work on OpenUI5 applications or libraries that already make use of the OpenUI5 npm packages, such as the OpenUI5 sample app, you can link your local OpenUI5 repository into that project. This allows you to make changes to the project itself as well as to the OpenUI5 libraries simultaneously and test them immediately.

For a detailed step-by-step guide on how to achieve such a setup with the OpenUI5 sample app, see Working With Local Dependencies .

Building OpenUI5

The UI5 Tooling is used to build a production-ready version of OpenUI5. Every library needs to be built individually.

Usage:

```
ui5 build
```

The build is responsible for the following tasks:

- Creation of the bundled `library.css` and `library-RTL.css` file for all available themes
- Minification of CSS
- Minification of JavaScript
- Bundling the JavaScript modules of the libraries into a single `library-preload.js` file
- Bundling of the most important OpenUI5 Core modules into `sap-ui-core.js`

If you encounter errors like the one below, execute `yarn` in the OpenUI5 root directory. There may be new build tools required which need to be downloaded first.

```
Error: Cannot find module 'xyz'
```

Testing OpenUI5

All OpenUI5 code must conform to a certain ruleset which is checked with ESLint.

To run an ESLint check, navigate to the root directory of the repository and execute:

```
npm run lint
```

Running Tests

Tests can be executed automatically with the Karma Test-Runner.

To run tests of a library, the `--lib` needs to be passed. The `<library-name>` corresponds to the folder within `./src/`, e.g. `sap.m`.

```
npm run karma -- --lib=<library-name>
```

This executes all tests of that library in watch mode, which will automatically re-run tests in case of file changes.

Example

```
npm run karma -- --lib=sap.m
```

Running a Specific Test

Instead of executing all tests of a library, you can also only run one test or a testsuite.

To find the URL, open `http://localhost:8080/test.html` and search for the test. Copy the URL and remove the origin part (`http://localhost:8080/`), so that it starts with `resources` or `test-resources`.

```
npm run karma -- --lib=<library-name> --ui5.testpage="<testpage-url>"
```

> **i** Note
>
> The corresponding `--lib` option still needs to be provided accordingly.

Example

```
npm run karma -- --lib=sap.m --ui5.testpage="resources/sap/ui/test/starter/
Test.qunit.html?testsuite=test-resources/sap/m/qunit/
testsuite.mobile.qunit&test=Button"
```

Coverage

You enable coverage reporting by additionally passing the `--coverage` option:

```
npm run karma -- --lib=<library-name> --coverage
```

Continuous integration (CI)

You enable the continuous integration mode by additionally passing the `--ci` option. This enables Chrome headless and disables the watch mode, so the execution stops after all tests have been executed:

```
npm run karma -- --lib=<library-name> --ci
```

You can combine the options `--ci` and `--coverage`.

Development for Hybrid Web Containers

You can develop mobile apps as hybrid app consisting of a native app wrapper, for example PhoneGap, and an HTML viewer to display the content on the user interface.

Hybrid apps have the advantage that you can publish them in app stores. Also, by embedding the application code and the SAPUI5 library files into the hybrid container, the user needs to install the files only once and does **not** need to download them every time he starts the application. But then the library size becomes important, because every user has to install the files, whereas in web applications, the library is deployed on a server and the user only needs to download the required parts of the library at runtime.

To include the resources you need in your hybrid app, you can use the static mobile runtime package `openui5-runtime-mobile*.zip`. The package is **not** contained in SAPUI5, but in the Open Source version OpenUI5.

The library size of these packages is rather small because the content that is most likely not needed has been removed, for example test pages. A package contains the debug version of all JavaScript files and the optimized and minimized version. Thus, you can use the package for productive use as well as for debugging purposes. To use this package in an app wrapper, such as PhoneGap, unzip the package in the respective resource location of the app development project. The app wrapper build then includes the files and makes them available at runtime.

To ensure that the file is small, it only contains the control libraries that are most likely used and not all control libraries. Depending on the hybrid app it may be necessary to add libraries by copying them from the respective folder of the runtime, or to delete libraries to reduce the package size and, thus, also reduce the installation size for the user.

The file contains the following control libraries:

- `sap.f`
- `sap.m`
- `sap.tnt`
- `sap.ui.core`
- `sap.ui.layout`
- `sap.ui.suite`
- `sap.ui.unified`
- `sap.uxap`

The decision, which libraries to include or not may be disputed. It is only based on a rule of thumb, and adaptations are required anyway for many apps.

Also, the mobile/hybrid package excludes certain types of files which are typically not needed. Your mileage may vary, so you might need to add the respective files for the requirements of your specific app. The

`library-preload.js` files which contain all controls from a library to reduce the number of HTTP requests are not required in hybrid apps because there is no HTTP latency. SAPUI5 will by default try to access them, so you might see a failed attempt to load these files in the log file or developer tools. These error messages do not hurt, though, and you can get rid of them by declaring that no such files exist and by setting the following configuration in the SAPUI5 bootstrap script tag:

```
data-sap-ui-preload=""
```

Optimization of the Package Size

Although the static package is small enough to be included in hybrid apps, you can reduce the size further and optimize the content for a specific application by deleting additional files. The following list gives some examples:

- You can delete all library folders if the respective control library is not needed. For example, in the OpenUI5 version you can delete the `suite` and the `unified` folder.
- In each of the `/resources/sap/* ... */themes` folders, you can delete all theme folders except the one for the theme you are using.

> **i Note**
>
> For all JavaScript files, an optimized version and a debug (`dbg`) version exists. If you delete the files, make sure that you always delete both versions. If you can do without easy debugging and want to achieve a minimum installation size, we recommend to delete all `*-dbg.js` files.

You can delete further files, but the size reduction is limited and to find out the files that are not required gets increasingly difficult.

Device Ready Event

The hybrid web container needs some time for initialization. During this time, the sending of AJAX requests is blocked, meaning that JavaScript code stops once an AJAX request is sent and the code execution stops as well. This leads to a UI freeze effect.

The OData model in SAPUI5 uses AJAX requests internally and the OData model initialization must therefore be done after the hybrid container is ready to avoid a user interface freeze. After initialization, the hybrid web containers fires an event, which is called `deviceready` in PhoneGap. To fix this issue, move the code where the OData model is created and set to the core object or any other controls' model property to the `deviceready` event listener.

Example:

```
<script>
<!-- put the following code in the beginning of the application code -->
function appReady(){
    sap.ui.getCore().setModel(new sap.ui.model.odata.v2.ODataModel(<ODATA_URL>));
}
<!-- bind to the deviceready event -->
document.addEventListener("deviceready", appReady, false);
```

```
</script>
```

Cross Domain Restrictions

If you load data from an external server or service using AJAX, the external domain has to be configured inside the hybrid web container to make the AJAX request go through the cross domain restriction. The following findings result from an integration of the demo applications into PhoneGap:

- Android
  If the AJAX code runs inside the webview in Android, no cross domain restriction exists. This means that you can load data using AJAX from everywhere. The PhoneGap documentation, however, still says that the domain needs to be configured in one XML file.
- iOS
  The restriction in webview in iOS still exists and you need to add the domain that is visited using AJAX to a whitelist file to bypass the restriction. For detailed information about the whitelist file, see the PhoneGap documentation on the PhoneGap website.

Quick Start

Unleash your SAPUI5 skills with this simple three-step tutorial. We start with a simple "Hello World" example, and convert it to a minimalist two-page app.

We create an app with two pages and a navigation button to navigate between the pages.

Preview

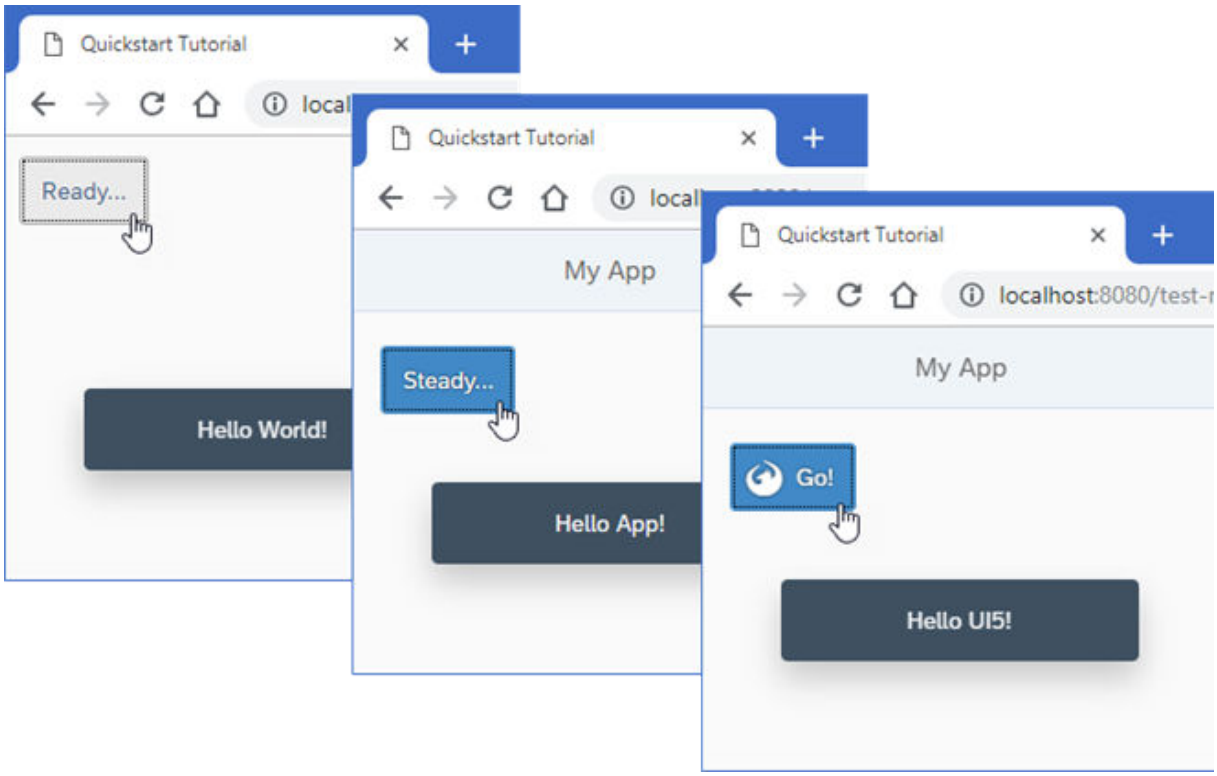


Figure 1: Ready...Steady...Go!

> → Tip
>
> If you want to skip one or more steps, you can jump directly to the step you're interested in. Then simply download the code from the previous step, and start learning from there. You can download the code for each step in the the Quick Start Sample.

> i Note
>
> All you need to build your app, is a Web browser and a development environment of your choice. For more information, see the links below.

Related Information

Get Started: Setup, Tutorials, and Demo Apps [page 38]
Development Environment [page 41]

Step 1: Ready...

Let's get you ready for your journey! We bootstrap SAPUI5 in an HTML page and implement a simple "Hello World" example.

Preview

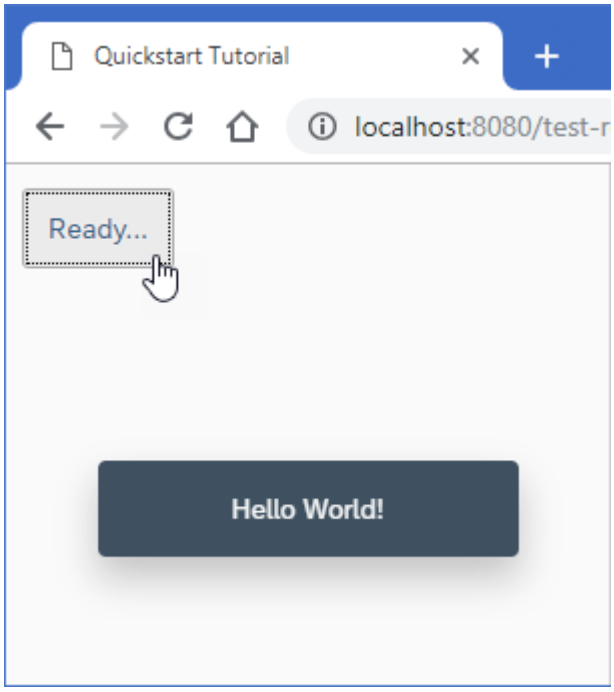


Figure 2: The browser shows a "Ready" button that triggers a "Hello World" message

Coding

You can view and download all files at Quick Start - Step 1.

webapp/index.html (new)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Quickstart Tutorial</title>
    <script id="sap-ui-bootstrap"
        src="../../../../../../../../resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-resourceroots='{"Quickstart": "./"}'
        data-sap-ui-onInit="module:Quickstart/index"
```

```
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
</head>
<body class="sapUiBody" id="content"></body>
</html>
```

In your development environment (SAP Web IDE is recommended), create the folder `webapp`. In this folder, create an `index.html` file, and paste the above code to this file.

With the `script` tag, we load and initialize SAPUI5 with typical bootstrap parameters. We define, for example, a theme, control libraries, as well as performance and compatibility flags.

First, we need a source to load SAPUI5 from. To keep things convenient, we use the path to our Content Delivery Network (CDN) for OpenUI5.

The bootstrap property `resourceroots` defines the namespace for all resources of the app. This way, we can easily reference additional files that we are about to create in this step.

The `index` module that we load with the `onInit` parameter will hold the application logic.

The `body` tag is defined with the `sapUiBody` class and the `content` ID. This is where we will add the content of the app in the next steps.

> → Tip
>
> For more information about bootstrapping from the CDN, see Variant for Bootstrapping from Content Delivery Network [page 696].

webapp/index.js (New)

```
sap.ui.define([
    "sap/m/Button",
    "sap/m/MessageToast"
], function (Button, MessageToast) {
    "use strict";

    new Button({
        text: "Ready...",
        press: function () {
            MessageToast.show("Hello World!");
        }
    }).placeAt("content");

});
```

In your `webapp` folder, create a new file `index.js` that will be called as soon as SAPUI5 is loaded and initialized.

We load two UI controls - a button and a message toast - and place the button in the element with the `content` ID. The button is defined with a `text` property and a callback attached to its `press` event.

Now open the `index.html` file in your browser. When the button is pressed, a message toast with the "Hello World" message is shown at the bottom of the screen.

neo-app.json (new, optional)

> **i** Note
>
> This file is necessary if you use SAP Web IDE as your development environment. It contains all project settings and is located in the root folder of your project. This file is **not** part of the downloadable code in the Demo Kit, so just copy the content from here.

```
{
  "welcomeFile": "/webapp/index.html",
  "routes": [
    {
      "path": "/resources",
      "target": {
        "type": "application",
        "name": "sapui5preview",
        "entryPath": "/resources"
      },
      "description": "SAPUI5 Resources"
    },
    {
      "path": "/resources",
      "target": {
        "type": "service",
        "name": "sapui5",
        "entryPath": "/resources"
      },
      "description": "SAPUI5 Resources"
    }
  ],
  "sendWelcomeFileRedirect": true
}
```

Related Information

Development Environment [page 41]
Create a neo-app.json Project Configuration File [page 46]
Variant for Bootstrapping from Content Delivery Network [page 696]

Step 2: Steady...

Now we extend our minimalist HTML page to a basic app with a view and a controller.

Preview

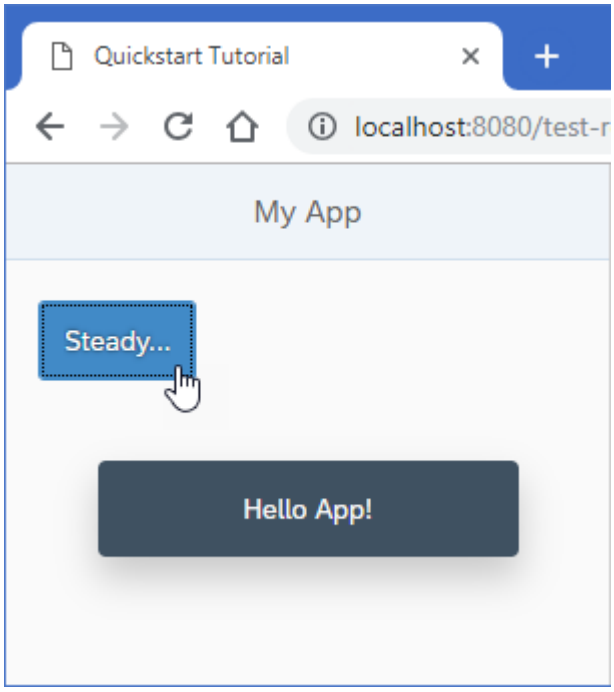


Figure 3: The browser shows a *Steady* button in an app

Coding

You can view and download all files at Quick Start - Step 2.

webapp/index.js

```
sap.ui.define([
    "sap/ui/core/mvc/XMLView"
], function (XMLView) {
    "use strict";
    XMLView.create({viewName: "Quickstart.App"}).then(function (oView) {
        oView.placeAt("content");
    });
});
```

Now we replace most of the code in this file: We remove the inline button from the previous step, and introduce a proper XML view to separate the presentation from the controller logic. We prefix the view name `Quickstart.App` with our newly defined namespace. The view is loaded asynchronously.

Similar to the step before, the view is placed in the element with the `content` ID after it has finished loading.

webapp/App.view.xml (New)

```
<mvc:View
    controllerName="Quickstart.App"
    displayBlock="true"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <App>
        <Page title="My App">
            <Button
                text="Steady..."
                press=".onPress"
                type="Emphasized"
                class="sapUiSmallMargin"/>
        </Page>
    </App>
</mvc:View>
```

The presentation logic is now defined declaratively in an XML view.

UI controls are located in libraries that we define in the `View` tag. In our case, we use the bread-and-butter controls from `sap.m`.

The new controls in the view are an `App` and a `Page`. They define a Web app with a header bar and a title.

The button from the previous examples now also defines a `type` and a `class` attribute. This improves the layout of our button and makes it stand out more.

We outsource the controller logic to an app controller. The `.onPress` event now references a function in the controller.

webapp/App.controller.js (New)

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast"
], function (Controller, MessageToast) {
    "use strict";

    return Controller.extend("Quickstart.App", {
        onPress : function () {
            MessageToast.show("Hello App!");
        }
    });

});
```

In our controller, we load the `Controller` base class and extend it to define the behavior of our app. We also add the event handler for our button.

The `MessageToast` is also loaded as a dependency. When the button is pressed, we now display a "Hello App" message.

Now reload your `index.html` file. You can see a title bar and a blue button that reacts to your input. Congratulations, you have created our very first app.

Related Information

XML View [page 787]
Controller [page 807]

Step 3: Go!

Finally, we add a second page to our app showcasing some of the key SAPUI5 concepts.

Preview

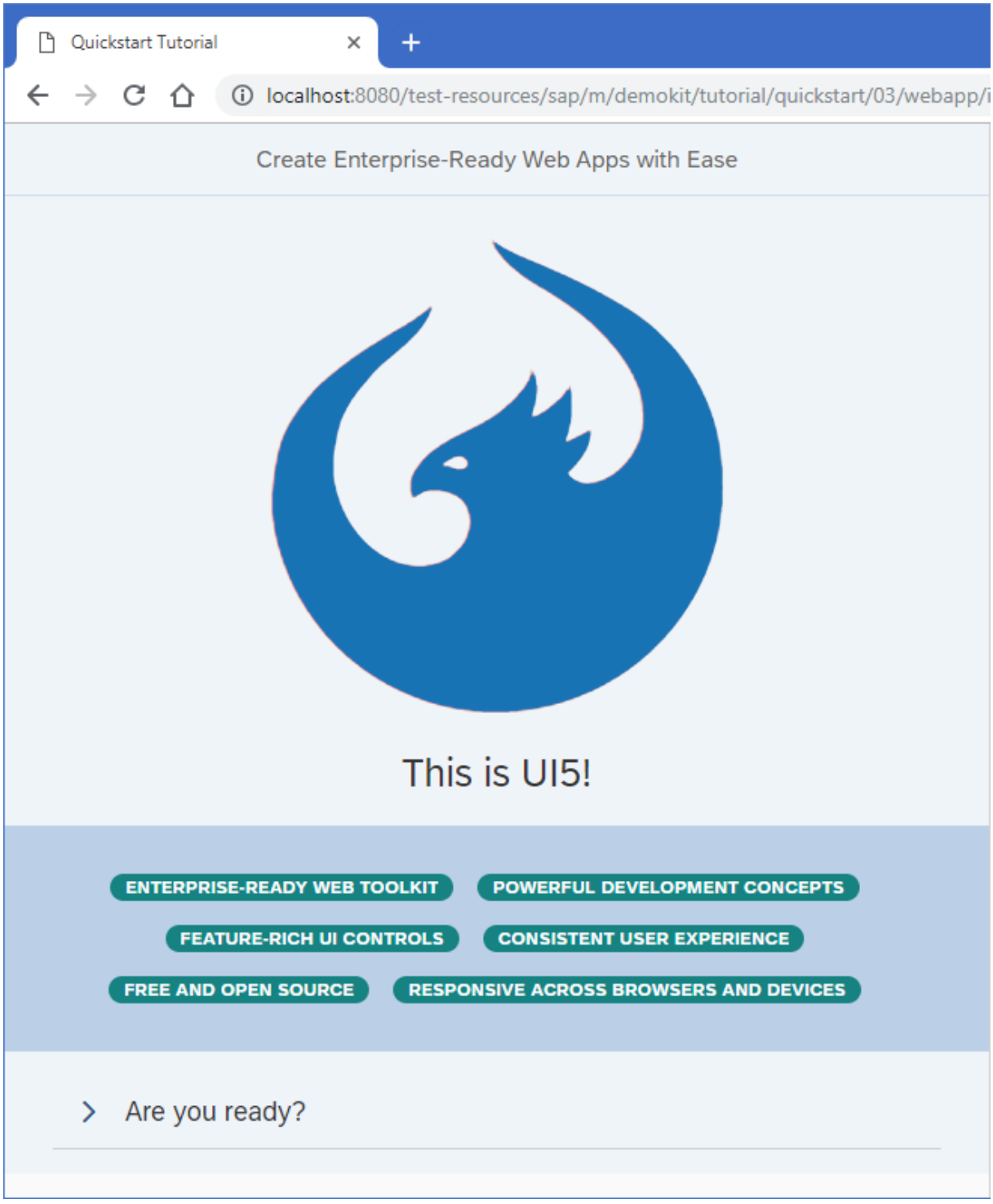


Figure 4: The second page shows plenty of UI controls and concepts to explore

Coding

You can view and download all files at Quick Start - Step 3.

webapp/index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Quickstart Tutorial</title>
    <script id="sap-ui-bootstrap"
        src="../../../../../../../../resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m, sap.ui.layout, sap.tnt"
        data-sap-ui-resourceroots='{"Quickstart": "./"}'
        data-sap-ui-onInit="module:Quickstart/index"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
</head>
<body class="sapUiBody" id="content"></body>
</html>
```

Let's spice up our app by adding some more UI controls. We add two more libraries in the bootstrap tag: `sap.ui.layout` and `sap.tnt`.

> → Tip
>
> To browse all available controls and libraries, see the Samples.

webapp/App.view.xml

```
<mvc:View
    controllerName="Quickstart.App"
    displayBlock="true"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:l="sap.ui.layout"
    xmlns:core="sap.ui.core"
    xmlns:tnt="sap.tnt">
    <App id="app">
        <Page title="My App">
            <Button
                icon="sap-icon://sap-ui5"
                text="Go!"
                press=".onPress"
                type="Emphasized"
                class="sapUiSmallMargin"/>
        </Page>
        <Page id="intro" title="Create Enterprise-Ready Web Apps with Ease">
            <l:BlockLayout background="Light">
                <l:BlockLayoutRow>
                    <l:BlockLayoutCell>
                        <core:Icon color="#1873B4" src="sap-icon://sap-ui5"
size="20rem" class="sapUiMediumMarginBottom" width="100%"/>
                        <Title level="H1" titleStyle="H1" text="This is UI5!"
width="100%" textAlign="Center"/>
                    </l:BlockLayoutCell>
                </l:BlockLayoutRow>
                <l:BlockLayoutRow>
                    <l:BlockLayoutCell>
                        <FlexBox items="{/features}" justifyContent="Center"
```

```
wrap="Wrap" class="sapUiSmallMarginBottom">
                            <tnt:InfoLabel text="{}" class="sapUiSmallMarginTop
sapUiSmallMarginEnd"/>
                        </FlexBox>
                    </l:BlockLayoutCell>
                </l:BlockLayoutRow>

                <l:BlockLayoutRow>
                    <l:BlockLayoutCell>
                        <Panel headerText="Are you ready?" expandable="true">
                            <Switch change=".onChange" customTextOn="yes"
customTextOff="no"/>
                            <l:HorizontalLayout id="ready" visible="false"
class="sapUiSmallMargin">
                                <Text text="Ok, let's get you started!"
class="sapUiTinyMarginEnd"/>
                                <Link text="Learn more" href="https://
openui5.hana.ondemand.com/"/>
                            </l:HorizontalLayout>
                        </Panel>
                    </l:BlockLayoutCell>
                </l:BlockLayoutRow>
            </l:BlockLayout>
        </Page>
    </App>
</mvc:View>
```

We also define the two new libraries in the `View` tag and give them a meaningful prefix. To the `App` control, we will assign an ID so that the controller can easily identify it.

The button now receives an icon and triggers our navigation to page two. Therefore, we change the text to "Go!".

Copy the second `Page` control with all its content into the view. It is defined with the `intro` ID and a new title. It contains several new UI controls like a `BlockLayout`, an `Icon`, and a `Panel`.

We use essential SAPUI5 concepts like navigation, data binding, and user interaction to define a nice playground on the second page of our app.

Don't worry too much about the details, we will explain them in the next tutorials.

webapp/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/ui/model/json/JSONModel"
], function (Controller, MessageToast, JSONModel) {
    "use strict";
    return Controller.extend("Quickstart.App", {
        onPress : function () {
            MessageToast.show("Hello UI5!");
            this.byId("app").to(this.byId("intro"));
        },

        onInit : function () {
            this.getView().setModel(new JSONModel({
                    features: [
                        "Enterprise-Ready Web Toolkit",
                        "Powerful Development Concepts",
                        "Feature-Rich UI Controls",
```

```
                        "Consistent User Experience",
                        "Free and Open Source",
                        "Responsive Across Browsers and Devices"
                    ]
                })
            );
        },

        onChange: function (oEvent) {
            var bState = oEvent.getParameter("state");
            this.byId("ready").setVisible(bState);
        }
    });
});
```

The `onPress` function now also triggers the navigation to the `intro` page. We fetch the `app` control by its ID and instruct it to navigate by calling the `to` method.

The `onInit` method is a lifecycle hook that is called automatically when the controller is initialized. It defines a simple JSON model with some texts located at the `features` key.

We display these texts on the second page using data binding. The `InfoLabel` tag from our view is a template that is repeated as many times as we have entries in our model.

Finally, we make the Panel in the lower part of the view interactive by attaching an `onChange` event to the switch defined there. SAPUI5 comes with a large set of feature-rich controls that you can combine as you need.

Run the app, and navigate to the second page to see some nice UI controls and interaction. If we expand the panel and click the switch, we toggle the visibility of the text and the link next to it.

You now have a little playground in your app that you can modify and extend as you wish. We intentionally did not go into all the details. If you want to know more, just continue with the Walkthrough tutorial.

Have fun with SAPUI5!

Related Information

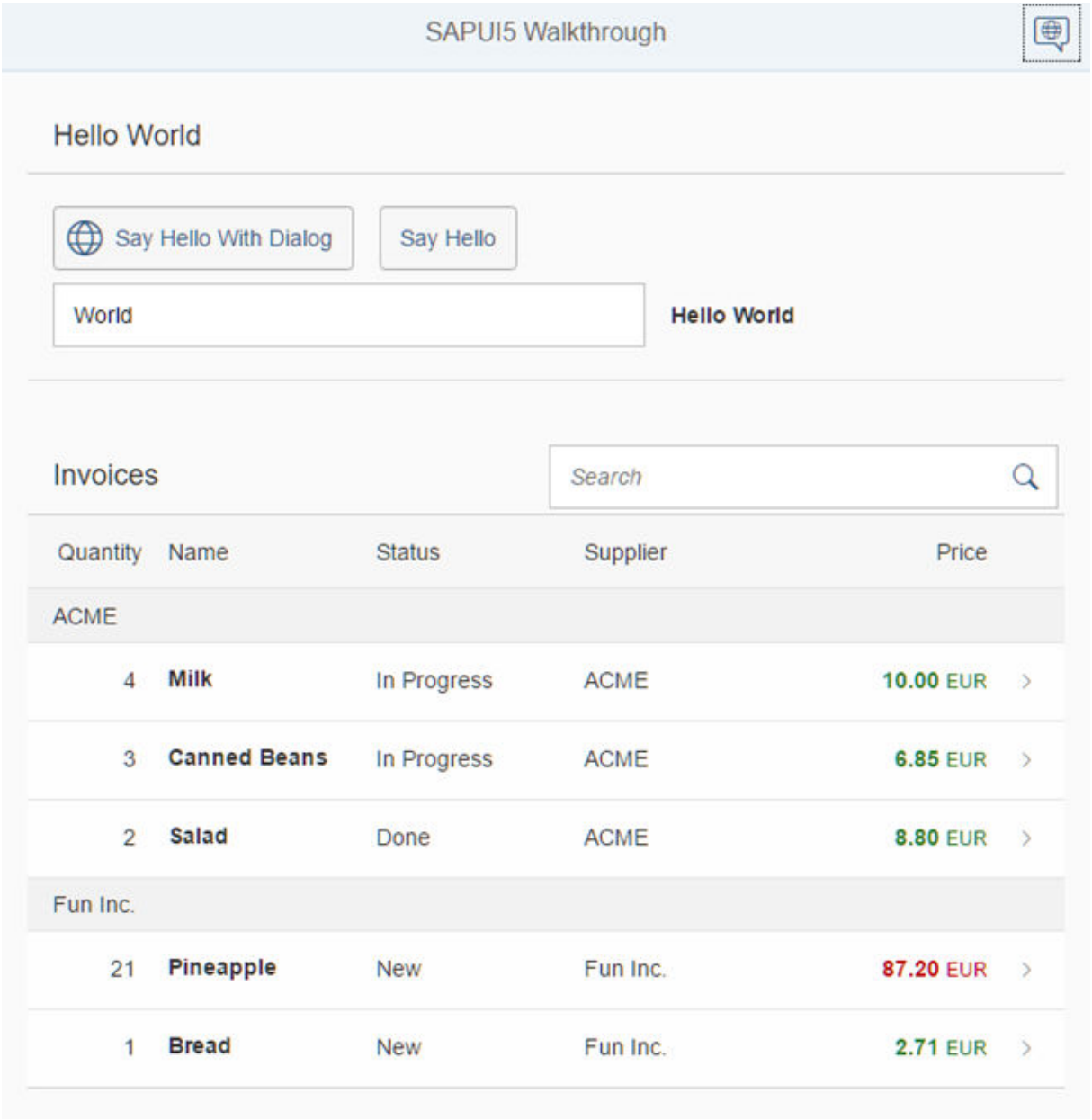
Working with Controls [page 1041]
Data Binding [page 815]
Routing and Navigation [page 1072]

Walkthrough

In this tutorial we will introduce you to all major development paradigms of SAPUI5.

We first introduce you to the basic development paradigms like *Model-View-Controller* and establish a best-practice structure of our application. We'll do this along the classic example of “Hello World” and start a new app from scratch. Next, we'll introduce the fundamental data binding concepts of SAPUI5 and extend our app to show a list of invoices. We'll continue to add more functionality by adding navigation, extending controls, and making our app responsive.Finally we'll look at the testing features and the built-in support tools of SAPUI5.

Preview



→ Tip

You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, copy it to your workspace and make sure that the application runs by calling the `webapp/index.html` file.

You can view and download the samples for all steps in the in the Demo Kit at Walkthrough. Depending on your development environment you might have to adjust resource paths and configuration entries.

For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Step 1: Hello World!

As you know SAPUI5 is all about HTML5. Let's get started with building a first "Hello World" with only HTML.

Preview

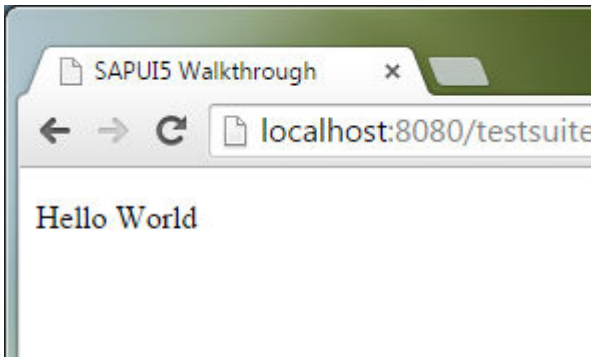


Figure 5: The browser shows the text "Hello World"

Coding

You can view and download all files at Walkthrough - Step 1.

webapp/index.html (New)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SAPUI5 Walkthrough</title>
</head>
<body>
    <div>Hello World</div>
</body>
</html>
```

Create a new folder `webapp` which will contain all sources of the app we will create throughout this tutorial. Therefore, we refer to this folder as "app folder".

Now create a new root HTML file called `index.html` in your app folder. An HTML document consists basically of two sections: head and body. The head part will be used by the browser to process the document. Using meta tags we can influence the behavior of the browser.

In this case we will tell the browser to use `UTF-8` as the document character set. We will also give our app a title that will be displayed in the browser. Be aware that our hard-coded title can be overruled by the app, for example to show a title in the language of the user.

The body part describes the layout of the page. In our case we simply display “Hello World” by using a `p` tag.

> → Tip
>
> Typically, the content of the `webapp` folder is deployed to a Web server as an application package. When deploying the `webapp` folder itself the URL for accessing the `index.html` file contains `webapp` in the path.

Conventions

- Name the root HTML file of the app `index.html` and locate it in the `webapp` folder.

Step 2: Bootstrap

Before we can do something with SAPUI5, we need to load and initialize it. This process of loading and initializing SAPUI5 is called **bootstrapping**. Once this bootstrapping is finished, we simply display an alert.

Preview

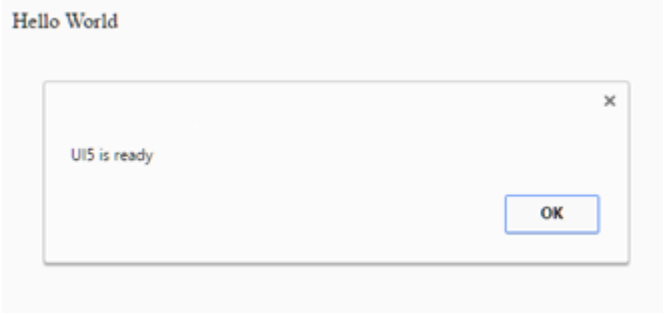


Figure 6: An alert "UI5 is ready" is displayed

Coding

You can view and download all files at Walkthrough - Step 2.

> i Note
>
> SAPUI5 is a JavaScript library that can either be loaded from the same Web server where the app resides, or from a different server. If SAPUI5 is deployed somewhere else on the server or you want to use a different server, then you need to adjust the corresponding paths in the bootstrap (here: `src="/resources/sap-ui-core.js"`) in this tutorial according to your own requirements.

You can use this reference to the latest stable version of SAPUI5 for the tutorial or for testing purposes, but never use this for productive use. In an actual app, you always have to specify an SAPUI5 version explicitly.

For more information about the CDN, see Variant for Bootstrapping from Content Delivery Network [page 696].

In case you are using SAP Web IDE, you can right-click the project and select *New* > *HTML5 Application Descriptor* to make the `/resources...` reference work. This creates the `neo-app.json` file, which configures a URL mapping for this path.

webapp/index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SAPUI5 Walkthrough</title>
    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true"
        data-sap-ui-onInit="module:sap/ui/demo/walkthrough/index"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "./"
        }'>

    </script>
</head>
<body>
<div>Hello World</div>
</body>
</html>
```

In this step, we load the SAPUI5 framework from our local webserver and initialize the core modules with the following configuration options:

- The `src` attribute of the `<script>` tag tells the browser where to find the SAPUI5 core library – it initializes the SAPUI5 runtime and loads additional resources, such as the libraries specified in the `data-sap-ui-libs` attribute.
- The SAPUI5 controls support different themes, we choose `sap_belize` as our default theme.
- We specify the required UI library `sap.m` containing the UI controls we need for this tutorial.
- To make use of the most recent functionality of SAPUI5 we define the compatibility version as `edge`.
- We configure the process of “bootstrapping” to run asynchronously.
  This means that the SAPUI5 resources can be loaded simultaneously in the background for performance reasons.
- We define the module to be loaded initially in a declarative way. With this, we avoid directly executable JavaScript code in the HTML file. This makes your app more secure. We will create the script that this references to further down in this step.
- We tell SAPUI5 core that resources in the `sap.ui.demo.walkthrough` namespace are located in the same folder as `index.html`. This is, for example, necessary for apps that run in the SAP Fiori launchpad.

webapp/index.js (New)

```
sap.ui.define([
], function () {
    "use strict";
    alert("UI5 is ready");
});
```

Now, we create a new `index.js` script that will contain the application logic for this tutorial step. We do this to avoid having executable code directly in the HTML file for security reasons. This script will be called by the `index.html`. We defined it there as a module in a declarative way.

Related Information

Bootstrapping: Loading and Initializing [page 692]
Standard Variant for Bootstrapping [page 694]
Compatibility Version Information [page 718]
Variant for Bootstrapping from Content Delivery Network [page 696]
https://jquery.org/
Content Security Policy [page 1481]

Step 3: Controls

Now it is time to build our first little UI by replacing the “Hello World” text in the HTML body by the SAPUI5 control `sap.m.Text`. In the beginning, we will use the JavaScript control interface to set up the UI, the control instance is then placed into the HTML body.

Preview

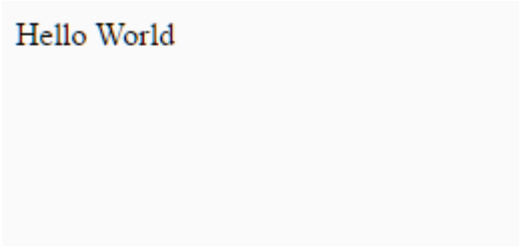


Figure 7: The "Hello World" text is now displayed by a SAPUI5 control

Coding

You can view and download all files at Walkthrough - Step 3.

webapp/index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SAPUI5 Walkthrough</title>
    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true"
        data-sap-ui-onInit="module:sap/ui/demo/walkthrough/index"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "./"
        }'>
    </script>
</head>
<body class="sapUiBody" id="content">
</body>
</html>
```

The class `sapUiBody` adds additional theme-dependent styles for displaying SAPUI5 apps.

webapp/index.js

```
sap.ui.define([
    "sap/m/Text"
], function (Text) {
    "use strict";
    new Text({
        text: "Hello World"
    }).placeAt("content");
});
```

Instead of using native JavaScript to display a dialog we want to use a simple SAPUI5 control. Controls are used to define appearance and behavior of parts of the screen.

In the example above, the callback of the `init` event is where we now instantiate a SAPUI5 text control. The name of the control is prefixed by the namespace of its control library `sap/m/` and the options are passed to the constructor with a JavaScript object. For our control we set the `text` property to the value “Hello World”.

We chain the constructor call of the control to the standard method `placeAt` that is used to place SAPUI5 controls inside a node of the document object model (DOM) or any other SAPUI5 control instance. We pass the ID of a DOM node as an argument. As the target node we use the body tag of the HTML document and give it the ID `content`.

All controls of SAPUI5 have a fixed set of properties, aggregations, and associations for configuration. You can find their descriptions in the Demo Kit. In addition, each controls comes with a set of public functions that you can look up in the API reference.

Don’t forget to remove the “Hello World” `p`.

> **i Note**
>
> Only instances of `sap.ui.core.Control` or their subclasses can be rendered stand-alone and have a `placeAt` function. Each control extends `sap.ui.core.Element` that can only be rendered inside controls. Check the API reference to learn more about the inheritance hierarchy of controls. The API documentation of each control refers to the directly known subclasses.

Related Information

Working with Controls [page 1041]
API Reference: `sap.m.Text`
Samples: `sap.m.Text`
API Reference: `sap.ui.core.Control`
API Reference: `sap.ui.core.Element`
API Reference: `sap.ui.base.ManagedObject`

Step 4: XML Views

Putting all our UI into the `index.html` file will very soon result in a messy setup and there is quite a bit of work ahead of us. So let's do a first modularization by putting the `sap.m.Text` control into a dedicated `view`.

SAPUI5 supports multiple view types (XML, HTML, JavaScript). We choose XML as this produces the most readable code and will force us to separate the view declaration from the controller logic. Yet the look of our UI will not change.

Preview

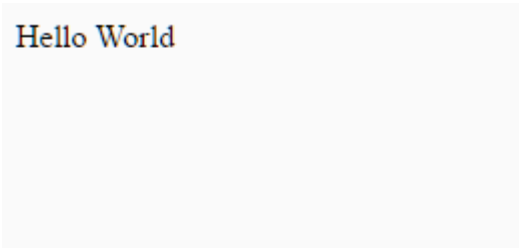


Figure 8: The "Hello World" text is now displayed by a SAPUI5 control (No visual changes to last step)

Coding

You can view and download all files at Walkthrough - Step 4.

webapp/view/App.view.xml (New)

```
<mvc:View
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
</mvc:View>
```

We create a new `view` folder in our app and a new file for our XML view inside the `app` folder. The root node of the XML structure is the `view`. Here, we reference the default namespace `sap.m` where the majority of our UI assets are located. We define an additional `sap.ui.core.mvc` namespace with alias `mvc`, where the SAPUI5 views and all other Model-View-Controller (MVC) assets are located.

> **i** Note
>
> The namespace identifies all resources of the project and has to be unique. If you develop your own application code or controls, you cannot use the namespace prefix `sap`, because this namespace is reserved for SAP resources. Instead, simply define your own unique namespace (for example, `myCompany.myApp`).

webapp/view/App.view.xml

```
<mvc:View
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Text text="Hello World"/>
</mvc:View>
```

Inside the `view` tag, we add the declarative definition of our `text` control with the same properties as in the previous step. The XML tags are mapped to controls and the attributes are mapped to the properties of the control.

webapp/index.js

```
sap.ui.define([
    "sap/ui/core/mvc/XMLView"
], function (XMLView) {
    "use strict";
    XMLView.create({
        viewName: "sap.ui.demo.walkthrough.view.App"
    }).then(function (oView) {
        oView.placeAt("content");
    });
});
```

We replace the instantiation of the `sap.m.Text` control by our new `App` XML view. The view is created by a factory function of SAPUI5 which makes sure that the view is correctly configured and can be extended by customers. The name is prefixed with the namespace `sap.ui.demo.walkthrough.view` in order to uniquely identify this resource.

> **i Note**
>
> From this step onwards, it is necessary to run the app on a Web server. We structure the app with multiple files that are loaded from the local file system. Without a Web server, this is prevented by the browser due to security reasons. If the error message "sap is not defined" appears in the developer tools of the browser, you need to check the `resource` path in the bootstrap.
>
> You can find more information about how to install a Web server for your particular environment at Development Environment [page 41].

Conventions

- View names are capitalized
- All views are stored in the `view` folder
- Names of XML views always end with `*.view.xml`
- The default XML namespace is `sap.m`
- Other XML namespaces use the last part of the SAP namespace as alias (for example, `mvc` for `sap.ui.core.mvc`)

Related Information

Model View Controller (MVC) [page 784]
Views [page 787]
XML View [page 787]

Step 5: Controllers

In this step, we replace the text with a button and show the “Hello World” message when the button is pressed. The handling of the button's `press` event is implemented in the controller of the view.

Preview

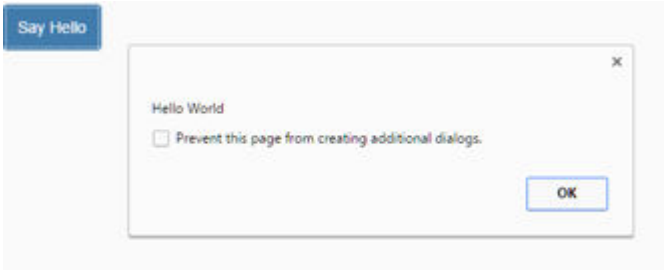


Figure 9: A *Say Hello* button is added

Coding

You can view and download all files at Walkthrough - Step 5.

webapp/view/App.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Button
      text="Say Hello"
      press=".onShowHello"/>
</mvc:View>
```

We add a reference to the controller, and replace the text control with a button with text “Say Hello”. The button triggers the `.onShowHello` event handler function when being pressed. We also have to specify the name of the controller that is connected to the view and holds the `.onShowHello` function by setting the `controllerName` attribute of the view.

A view does not necessarily need an explicitly assigned controller. You do not have to create a controller if the view is just displaying information and no additional functionality is required. If a controller is specified, it is instantiated after the view is loaded.

webapp/controller/App.controller.js (New)

```
sap.ui.define([
   "sap/ui/core/mvc/Controller"
], function (Controller) {
   "use strict";
   return Controller.extend("", {
   });
});
```

We create the folder `webapp/controller` and a new file `App.controller.js` inside. For now, we ignore the code that manages the required modules. We will explain this part in the next step.

> **i** Note
>
> The `"use strict";` literal expression was introduced by ECMAScript 5. It tells the browser to execute the code in a so called “strict mode”. The strict mode helps to detect potential coding issues at an early state at development time, that means, for example, it makes sure that variables are declared before they are used. Thus, it helps to prevent common JavaScript pitfalls and it's therefore a good practice to use strict mode.

webapp/controller/App.controller.js

```
sap.ui.define([
   "sap/ui/core/mvc/Controller"
], function (Controller) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
      onShowHello : function () {
         // show a native JavaScript alert
         alert("Hello World");
      }
   });
});
```

We define the app controller in its own file by extending the `Controller` object of the SAPUI5 core. In the beginning it holds only a single function called `onShowHello` that handles the button's `press` event by showing an alert.

Conventions

- Controller names are capitalized
- Controllers carry the same name as the related view (if there is a 1:1 relationship)
- Event handlers are prefixed with `on`
- Controller names always end with `*.controller.js`

Related Information

Model View Controller (MVC) [page 784]
Controller [page 807]
API Reference: `sap.ui.define`

Step 6: Modules

In SAPUI5, resources are often referred to as modules. In this step, we replace the alert from the last exercise with a proper Message Toast from the `sap.m` library. The required modules are enabled to be loaded asynchronously.

Preview

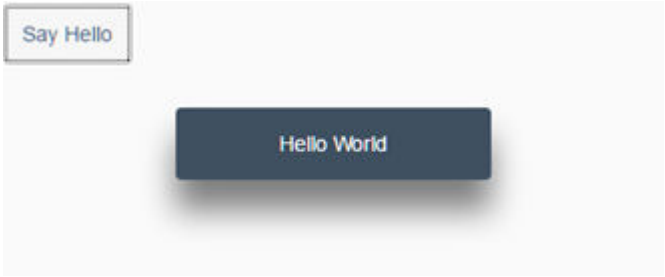


Figure 10: A message toast displays the "Hello World" message

Coding

You can view and download all files at Walkthrough - Step 6.

webapp/controller/App.controller.js

```
sap.ui.define([
   "sap/ui/core/mvc/Controller",
   "sap/m/MessageToast"
], function (Controller, MessageToast) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
      onShowHello : function () {
         MessageToast.show("Hello World");
      }
   });
});
```

We extend the array of required modules with the fully qualified path to `sap.m.MessageToast`. Once both modules, `Controller` and `MessageToast`, are loaded, the callback function is called and we can make use of both objects by accessing the parameters passed on to the function.

This Asynchronous Module Definition (AMD) syntax allows to clearly separate the module loading from the code execution and greatly improves the performance of the application. The browser can decide when and how the resources are loaded prior to code execution.

Conventions

- Use `sap.ui.define` for controllers and all other JavaScript modules to define a global namespace. With the namespace, the object can be addressed throughout the application.
- Use `sap.ui.require` for asynchronously loading dependencies but without declaring a namespace, for example code that just needs to be executed, but does not need to be called from other code.
- Use the name of the artifact to load for naming the function parameters (without namespace).

Related Information

API Reference: `sap.ui.define`
API Reference: `sap.ui.require`

Step 7: JSON Model

Now that we have set up the view and controller, it's about time to think about the M in MVC.

We will add an input field to our app, bind its value to the model, and bind the same value to the description of the input field. The description will be directly updated as the user types.

Preview

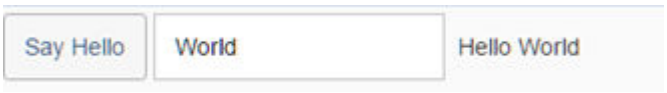


Figure 11: An input field and a description displaying the value of the input field

Coding

You can view and download all files at Walkthrough - Step 7.

webapp/controller/App.controller.js

```
sap.ui.define([
   "sap/ui/core/mvc/Controller",
   "sap/m/MessageToast",
   "sap/ui/model/json/JSONModel"
], function (Controller, MessageToast, JSONModel) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
      onInit : function () {
         // set data model on view
         var oData = {
            recipient : {
               name : "World"
            }
         };
         var oModel = new JSONModel(oData);
         this.getView().setModel(oModel);
      },
      onShowHello : function () {
         MessageToast.show("Hello World");
      }
   });
});
```

We add an init function to the controller. `onInit` is one of SAPUI5's lifecycle methods that is invoked by the framework when the controller is created, similar to a constructor function of a control.

Inside the function we instantiate a JSON model. The data for the model only contains a single property for the "recipient", and inside this it also contains one additional property for the name.

To be able to use this model from within the XML view, we call the `setModel` function on the view and pass on our newly created model. The model is now set on the view.

The message toast is just showing the static "Hello World" message. We will show how to load a translated text here in the next step.

webapp/view/App.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Button
      text="Say Hello"
      press=".onShowHello"/>
   <Input
      value="{/recipient/name}"
      description="Hello {/recipient/name}"
      valueLiveUpdate="true"
      width="60%"/>
</mvc:View>
```

We add an `sap.m.Input` control to the view. With this, the user can enter a recipient for the greetings. We bind its value to a SAPUI5 model by using the declarative binding syntax for XML views:

- The curly brackets `{...}` indicate that data is taken from the value of the `recipient`'s object name property. This is called "data binding".

- `/recipient/name` declares the path in the model.

webapp/index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SAPUI5 Walkthrough</title>
    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "./"
        }'
        data-sap-ui-oninit="module:sap/ui/demo/walkthrough/index">
    </script>
    <script src="index.js"></script>
</head>
<body class="sapUiBody" id="content">
</body>
</html>
```

The binding of the value attribute is a simple binding example that contains only a binding pattern. We can also combine texts and binding pattern to a more complex binding result as seen in the description attribute. To be able to use the so-called complex binding syntax we have to enable it globally by setting the bootstrap parameter `data-sap-ui-compatVersion` to `edge`. If this setting is omitted, then only standard binding syntax is allowed, meaning "Hello `{/recipient/name}`" would not work anymore while "`{/recipient/name}`" would work just fine.

> **i Note**
>
> You can either use `data-sap-ui-compatVersion="edge"` or `data-sap-ui-bindingSyntax="complex"` in the script. By setting the "edge" compatibility mode, the complex binding syntax is automatically enabled. The `edge` mode automatically enables compatibility features that otherwise would have to be enabled manually. For more information, see Compatibility Version Information [page 718].

Conventions

- Use Hungarian notation for variable names.

Related Information

Model View Controller (MVC) [page 784]
Data Binding [page 815]
JSON Model [page 991]
API Reference: `sap.ui.define`

Step 8: Translatable Texts

In this step we move the texts of our UI to a separate resource file.

This way, they are all in a central place and can be easily translated into other languages. This process of internationalization – in short `i18n` – is achieved in SAPUI5 by using a special resource model and the standard data binding syntax, but without preceding `/`.

Preview

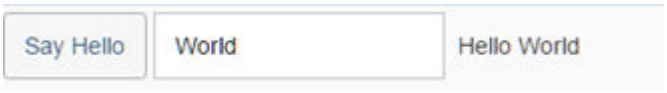


Figure 12: An input field and a description displaying the value of the input field (No visual changes to last step)

Coding

You can view and download all files at Walkthrough - Step 8.

webapp/i18n/i18n.properties (New)

```
showHelloButtonText=Say Hello
helloMsg=Hello {0}
```

We create the folder `webapp/i18n` and the file `i18n.properties` inside. The resolved bundle name is `sap.ui.demo.walkthrough.i18n`, as we will see later. The `properties` file for texts contains name-value pairs for each element. You can add any number of parameters to the texts by adding numbers in curly brackets to them. These numbers correspond to the sequence in which the parameters are accessed (starting with 0).

In this tutorial we will only have one properties file. However, in real-world projects, you would have a separate file for each supported language with a suffix for the locale, for example `i18n_de.properties` for German, `i18n_en.properties` for English, and so on. When a user runs the app, SAPUI5 will load the language file that fits best to the user's environment.

controller/App.controller.js

```
sap.ui.define([
   "sap/ui/core/mvc/Controller",
   "sap/m/MessageToast",
   "sap/ui/model/json/JSONModel",
   "sap/ui/model/resource/ResourceModel"
], function (Controller, MessageToast, JSONModel, ResourceModel) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
     onInit : function () {
         // set data model on view
         var oData = {
            recipient : {
               name : "World"
            }
         };
         var oModel = new JSONModel(oData);
         this.getView().setModel(oModel);
         // set i18n model on view
         var i18nModel = new ResourceModel({
            bundleName: "sap.ui.demo.walkthrough.i18n.i18n"
         });
         this.getView().setModel(i18nModel, "i18n");
      },
      onShowHello : function () {
         // read msg from i18n model
         var oBundle = this.getView().getModel("i18n").getResourceBundle();
         var sRecipient = this.getView().getModel().getProperty("/recipient/
name");
         var sMsg = oBundle.getText("helloMsg", [sRecipient]);
         // show message
         MessageToast.show(sMsg);
      }
   });
});
```

In the `onInit` function we instantiate the `ResourceModel` that points to the new message bundle file where our texts are now located (`i18n.properties file`). The bundle name `sap.ui.demo.walkthrough.i18n.i18n` consists of the application namespace `sap.ui.demo.walkthrough` (the application root as defined in the `index.html`), the folder name `i18n` and finally the file name `i18n` without extension. The SAPUI5 runtime calculates the correct path to the resource; in this case the path to our `i18n.properties` file. Next, the model instance is set on the view as a named model with the key `i18n`. You use named models when you need to have several models available in parallel.

In the `onShowHello` event handler function we access the `i18n` model to get the text from the message bundle file and replace the placeholder `{0}` with the recipient from our data model. The `getProperty` method can be called in any model and takes the data path as an argument. In addition, the resource bundle has a specific `getText` method that takes an array of strings as second argument.

The resource bundle can be accessed with the `getResourceBundle` method of a `ResourceModel`. Rather than concatenating translatable texts manually, we can use the second parameter of `getText` to replace parts of the text with non-static data. During runtime, SAPUI5 tries to load the correct `i18n_*.properties` file based on your browser settings and your locale. In our case we have only created one `i18n.properties` file to make it simple. However, you can see in the network traffic of your browser's developer tools that SAPUI5 tries to load one or more `i18n_*.properties` files before falling back to the default `i18n.properties` file.

webapp/view/App.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Button
      text="{i18n>showHelloButtonText}"
      press=".onShowHello"/>
   <Input
      value="{/recipient/name}"
      description="Hello {/recipient/name}"
      valueLiveUpdate="true"
      width="60%"/>
</mvc:View>
```

In the XML view, we use data binding to connect the button text to the `showHelloButtonText` property in the `i18n` model. A resource bundle is a flat structure, therefore the preceding slash (`/`) can be omitted for the path.

> **i Note**
>
> The description text is not completely localized in this example for illustration purposes. To be on the safe side, we would have to use a similar mechanism as in the controller to use a string from the resource bundle and replace parts of it. This can be done with the `jQuery.sap.formatMessage` formatter.
>
> Furthermore, `i18n` files only impact client-side application texts. Texts that are loaded from back-end systems can appear in all languages that are supported by the back-end system.

Conventions

- The resource model for internationalization is called the `i18n` model.
- The default filename is `i18n.properties`.
- Resource bundle keys are written in (lower) camelCase.
- Resource bundle values can contain parameters like `{0}`, `{1}`, `{2}`, ...
- Never concatenate strings that are translated, always use placeholders.
- Use Unicode escape sequences for special characters.

Related Information

Resource Model [page 995]

API Reference: `jQuery.sap.util.ResourceBundle`

API Reference: `sap.ui.model.resource.ResourceModel`

Samples: `sap.ui.model.resource.ResourceModel`

Step 9: Component Configuration

After we have introduced all three parts of the Model-View-Controller (MVC) concept, we now come to another important structural aspect of SAPUI5.

In this step, we will encapsulate all UI assets in a component that is independent from our `index.html` file. Components are independent and reusable parts used in SAPUI5 applications. Whenever we access resources, we will now do this relatively to the component (instead of relatively to the `index.html`). This architectural change allows our app to be used in more flexible environments than our static `index.html` page, such as in a surrounding container like the SAP Fiori launchpad.

Preview

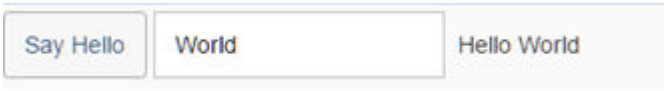


Figure 13: An input field and a description displaying the value of the input field (No visual changes to last step)

Coding

You can view and download all files at Walkthrough - Step 9.

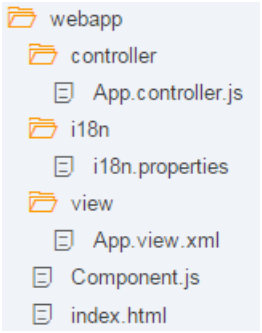


Figure 14: Folder Structure for this Step

After this step your project structure will look like the figure above. We will create the `Component.js` file now and modify the related files in the app.

webapp/Component.js (New)

```
sap.ui.define([
   "sap/ui/core/UIComponent"
], function (UIComponent) {
   "use strict";
   return UIComponent.extend("", {

      init : function () {
         // call the init function of the parent
```

```
            UIComponent.prototype.init.apply(this, arguments);
        }
    });
});
```

We create an initial `Component.js` file in the `webapp` folder that will hold our application setup. The init function of the component is automatically invoked by SAPUI5 when the component is instantiated. Our component inherits from the base class `sap.ui.core.UIComponent` and it is obligatory to make the super call to the `init` function of the base class in the overridden `init` method.

webapp/Component.js

```
sap.ui.define([
   "sap/ui/core/UIComponent",
   "sap/ui/model/json/JSONModel",
   "sap/ui/model/resource/ResourceModel"
], function (UIComponent, JSONModel, ResourceModel) {
   "use strict";
   return UIComponent.extend("sap.ui.demo.walkthrough.Component", {
      metadata : {
         rootView: {
            "viewName": "sap.ui.demo.walkthrough.view.App",
            "type": "XML",
            "async": true,
            "id": "app"
         }
      },
      init : function () {
         // call the init function of the parent
         UIComponent.prototype.init.apply(this, arguments);
         // set data model
         var oData = {
            recipient : {
               name : "World"
            }
         };
         var oModel = new JSONModel(oData);
         this.setModel(oModel);

         // set i18n model
         var i18nModel = new ResourceModel({
            bundleName : "sap.ui.demo.walkthrough.i18n.i18n"
         });
         this.setModel(i18nModel, "i18n");
      }
   });
});
```

The `Component.js` file consists of two parts now: The new `metadata` section that simply defines a reference to the root view and the previously introduced `init` function that is called when the component is initialized. Instead of displaying the root view directly in the `index.html` file as we did previously, the component will now manage the display of the app view.

In the `init` function we instantiate our data model and the `i18n` model like we did before in the app controller. Be aware that the models are directly set on the component and not on the root view of the component. However, as nested controls automatically inherit the models from their parent controls, the models will be available on the view as well.

webapp/controller/App.controller.js

```
sap.ui.define([
   "sap/ui/core/mvc/Controller",
   "sap/m/MessageToast"
], function (Controller, MessageToast) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
      onShowHello : function () {
         // read msg from i18n model
         var oBundle = this.getView().getModel("i18n").getResourceBundle();
         var sRecipient = this.getView().getModel().getProperty("/recipient/
name");
         var sMsg = oBundle.getText("helloMsg", [sRecipient]);
         // show message
         MessageToast.show(sMsg);
      }
   });
});
```

Delete the `onInit` function and the required modules; this is now done in the component. You now have the code shown above.

webapp\index.js

```
sap.ui.define([
    "sap/ui/core/ComponentContainer"
], function (ComponentContainer) {
    "use strict";
    new ComponentContainer({
        name: "sap.ui.demo.walkthrough",
        settings : {
            id : "walkthrough"
        },
        async: true
    }).placeAt("content");
});
```

We now create a component container instead of the view in our `index.js` that instantiates the view for us according to the component configuration.

Conventions

- The component is named `Component.js`.
- Together with all UI assets of the app, the component is located in the `webapp` folder.
- The `index.html` file is located in the `webapp` folder if it is used productively.

Related Information

Components [page 720]
API Reference: `sap.ui.core.mvc.ViewType`
Samples: `sap.ui.core.mvc.ViewType`
Declarative API for Initial Components [page 731]
Make Your App CSP Compliant [page 687]

Step 10: Descriptor for Applications

All application-specific configuration settings will now further be put in a separate descriptor file called `manifest.json`. This clearly separates the application coding from the configuration settings and makes our app even more flexible. For example, all SAP Fiori applications are realized as components and come with a descriptor file in order to be hosted in the SAP Fiori launchpad.

The SAP Fiori launchpad acts as an application container and instantiates the app without having a local HTML file for the bootstrap. Instead, the descriptor file will be parsed and the component is loaded into the current HTML page. This allows several apps to be displayed in the same context. Each app can define local settings, such as language properties, supported devices, and more. And we can also use the descriptor file to load additional resources and instantiate models like our `i18n` resource bundle.

Preview

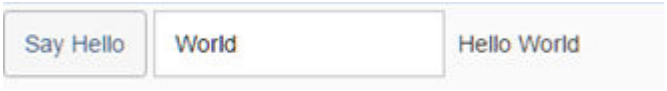


Figure 15: An input field and a description displaying the value of the input field (No visual changes to last step)

Coding

You can view and download all files at Walkthrough - Step 10.

> ⚠ Caution
>
> Automatic model instantiation is only available as of SAPUI5 version 1.30. If you are using an older version, you can manually instantiate the resource bundle and other models of the app in the `init` method of the `Component.js` file as we did in Step 9: Component Configuration [page 88].

webapp/manifest.json (New)

```
{
  "_version": "1.12.0",
```

```
  "sap.app": {
    "id": "sap.ui.demo.walkthrough",
    "type": "application",
    "i18n": "i18n/i18n.properties",
    "title": "{{appTitle}}",
    "description": "{{appDescription}}",
    "applicationVersion": {
      "version": "1.0.0"
    }
  },
  "sap.ui": {
    "technology": "UI5",
    "deviceTypes": {
        "desktop": true,
        "tablet": true,
        "phone": true
    }
  },
  "sap.ui5": {
    "rootView": {
        "viewName": "sap.ui.demo.walkthrough.view.App",
        "type": "XML",
        "async": true,
        "id": "app"
    },
    "dependencies": {
      "minUI5Version": "1.60",
      "libs": {
        "sap.m": {}
      }
    },
    "models": {
      "i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "settings": {
          "bundleName": "sap.ui.demo.walkthrough.i18n.i18n"
        }
      }
    }
  }
}
```

> **i Note**
>
> In this tutorial, we only introduce the most important settings and parameters of the descriptor file. In SAP Web IDE, you may get validation errors because some settings are missing - you can ignore those in this context.

The content of the `manifest.json` file is a configuration object in JSON format that contains all global application settings and parameters. The manifest file is called the descriptor for applications, components, and libraries and is also referred to as “descriptor” or “app descriptor” when used for applications. It is stored in the `webapp` folder and read by SAPUI5 to instantiate the component. There are three important sections defined by namespaces in the `manifest.json` file:

- `sap.app`
  The `sap.app` namespace contains the following application-specific attributes:
  - `id` (mandatory): The namespace of our application component
    The ID must not exceed 70 characters. It must be unique and must correspond to the component ID/ namespace.
  - `type`: Defines what we want to configure, here: an application
  - `i18n`: Defines the path to the resource bundle file

- `title`: Title of the application in handlebars syntax referenced from the app's resource bundle
- `description`: Short description text what the application does in handlebars syntax referenced from the app's resource bundle
- `applicationVersion`: The version of the application to be able to easily update the application later on

- `sap.ui`
  The `sap.ui namespace` contributes the following UI-specific attributes:
  - `technology`: This value specifies the UI technology; in our case we use SAPUI5
  - `deviceTypes`: Tells what devices are supported by the app: desktop, tablet, phone (all true by default)
- **sap.ui5**
  The `sap.ui5` namespace adds SAPUI5-specific configuration parameters that are automatically processed by SAPUI5. The most important parameters are:
  - `rootView`: If you specify this parameter, the component will automatically instantiate the view and use it as the root for this component
  - `dependencies`: Here we declare the UI libraries used in the application
  - `models`: In this section of the descriptor we can define models that will be automatically instantiated by SAPUI5 when the app starts. Here we can now define the local resource bundle. We define the name of the model "i18n" as key and specify the bundle file by namespace. As in the previous steps, the file with our translated texts is stored in the `i18n` folder and named `i18n.properties`. We simply prefix the path to the file with the namespace of our app. The manual instantiation in the app component's init method will be removed later in this step.

  For compatibility reasons the root object and each of the sections state the descriptor version number `1.1.0` under the internal property `_version`. Features might be added or changed in future versions of the descriptor and the version number helps to identify the application settings by tools that read the descriptor.

> **i Note**
>
> Properties of the resource bundle are enclosed in two curly brackets in the descriptor. This is not a SAPUI5 data binding syntax, but a variable reference to the resource bundle in the descriptor in handlebars syntax. The referred texts are not visible in the app built in this tutorial but can be read by an application container like the SAP Fiori launchpad.

webapp\index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SAPUI5 Walkthrough</title>
    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "./"
        }'
        data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
        data-sap-ui-compatVersion="edge"
```

```
        data-sap-ui-async="true">
    </script>
</head>
<body class="sapUiBody" id="content">
    <div data-sap-ui-component data-name="sap.ui.demo.walkthrough" data-
id="container" data-settings='{"id" : "walkthrough"}'></div>
</body>
</html>
```

Now we declare our component in the body of our `index.html`. In the bootstrapping script of our `index.html`, we enable the `ComponentSupport` module and remove the `sap.m` library. Then, we declare our component in the body via a `div` tag. This will instantiate the component when the `onInit` event is executed.

We will no longer need our `index.js` from now on, because the descriptor takes care of everything.

webapp/i18n/i18n.properties

```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important
concepts of SAPUI5

# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
```

In the resource bundle we simply add the texts for the app and add comments to separate the bundle texts semantically.

webapp/Component.js

```
sap.ui.define([
   "sap/ui/core/UIComponent",
   "sap/ui/model/json/JSONModel"
], function (UIComponent, JSONModel) {
   "use strict";
   return UIComponent.extend("sap.ui.demo.walkthrough.Component", {
      metadata : {
            manifest: "json"
      },
      init : function () {
         // call the init function of the parent
         UIComponent.prototype.init.apply(this, arguments);
         // set data model
         var oData = {
            recipient : {
               name : "World"
            }
         };
         var oModel = new JSONModel(oData);
         this.setModel(oModel);
      }
   });
});
```

In the component's `metadata` section, we now replace the `rootView` property with the property key `manifest` and the value `json`. This defines a reference to the descriptor that will be loaded and parsed automatically when the component is instantiated. We can now completely remove the lines of code containing the model instantiation for our resource bundle. It is done automatically by SAPUI5 with the help of the configuration entries in the descriptor. We can also remove the dependency to `sap/ui/model/resource/ResourceModel` and the corresponding formal parameter `ResourceModel` because we will not use this inside our anonymous callback function.

> → Tip
>
> In previous versions of SAPUI5, additional configuration settings for the app, like the service configuration, the root view, and the routing configuration, had to be added to the metadata section of the `Component.js` file. As of SAPUI5 version 1.30, we recommend that you define these settings in the `manifest.json` descriptor file. Apps and examples that were created based on an older SAPUI5 version still use the `Component.js` file for this purpose - so it is still supported, but not recommended.

Conventions

- The descriptor file is named `manifest.json` and located in the `webapp` folder.
- Use translatable strings for the title and the description of the app.

Related Information

Descriptor for Applications, Components, and Libraries [page 734]

Step 11: Pages and Panels

After all the work on the app structure it's time to improve the look of our app. We will use two controls from the `sap.m` library to add a bit more "bling" to our UI. You will also learn about control aggregations in this step.

Preview

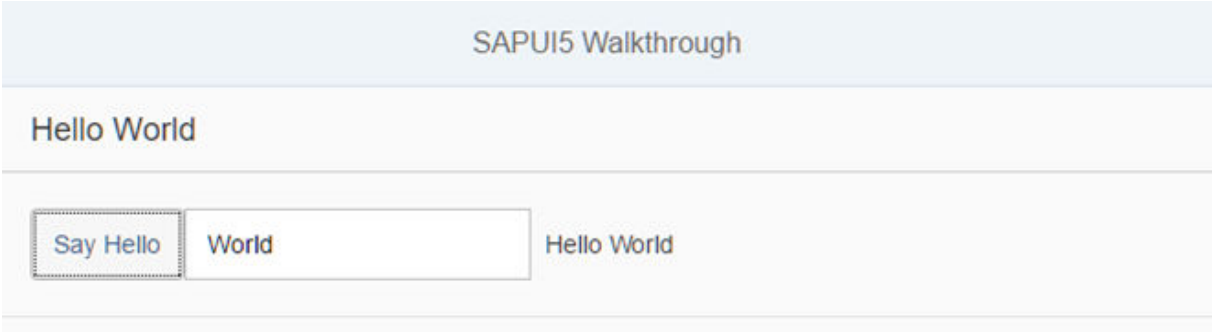


Figure 16: A panel is now displaying the controls from the previous steps

Coding

You can view and download all files at Walkthrough - Step 11.

webapp/view/App.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc"
  displayBlock="true">
   <App>
      <pages>
         <Page title="{i18n>homePageTitle}">
            <content>
               <Panel
                  headerText="{i18n>helloPanelTitle}">
                  <content>
                     <Button
                        text="{i18n>showHelloButtonText}"
                        press=".onShowHello"/>
                     <Input
                        value="{/recipient/name}"
                        description="Hello {/recipient/name}"
                        valueLiveUpdate="true"
                        width="60%"/>
                  </content>
               </Panel>
            </content>
         </Page>
      </pages>
   </App>
```

```
</mvc:View>
```

We put both the input field and the button inside a containing control called `sap.m.Page`. The page provides an aggregation to `0..N` other controls called `content`. It also displays the title attribute in a header section on top of the content. The page itself is placed into the `pages` aggregation of another control called `sap.m.App` which does the following important things for us:

- It writes a bunch of properties into the header of the `index.html` that are necessary for proper display on mobile devices.
- It offers functionality to navigate between pages with animations. We will use this soon.

In order to make the fullscreen height of the view work properly, we add the `displayBlock` attribute with the value `true` to the view. The actual content is wrapped inside a `Panel` control, in order to group related content.

webapp/i18n/i18n.properties

```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important
concepts of SAPUI5
# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
homePageTitle=Walkthrough
helloPanelTitle=Hello World
```

We add new key/value pairs to our text bundle for the start page title and the panel title.

Related Information

API Reference: `sap.m.NavContainer`
Samples: `sap.m.NavContainer`
API Reference: `sap.m.Page`
Samples: `sap.m.Page`

Step 12: Shell Control as Container

Now we use a shell control as container for our app and use it as our new root element. The shell takes care of visual adaptation of the application to the device's screen size by introducing a so-called letterbox on desktop screens.

Preview

Figure 17: The app is now run in a shell that limits the app width

Coding

You can view and download all files at Walkthrough - Step 12.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App>
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <Panel
                            headerText="{i18n>helloPanelTitle}">
                            <content>
                                <Button
                                    text="{i18n>showHelloButtonText}"
                                    press=".onShowHello"/>
                                <Input
                                    value="{/recipient/name}"
                                    description="Hello {/recipient/name}"
                                    valueLiveUpdate="true"
                                    width="60%"/>
                            </content>
                        </Panel>
                    </content>
                </Page>
            </pages>
```

```
        </App>
    </Shell>
</mvc:View>
```

The shell control is now the outermost control of our app and automatically displays a so-called letterbox, if the screen size is larger than a certain width.

> **i Note**
>
> We do not add the `Shell` control to the declarative UI definition in the XML view, because apps that run in an external shell, like the SAP Fiori launchpad, there will already be a shell around the component UI.

There are further options to customize the shell, like setting a custom background image or color and setting a custom logo. Check the related API reference for more details.

Related Information

API Reference: `sap.m.Shell`

Step 13: Margins and Paddings

Our app content is still glued to the corners of the letterbox. To fine-tune our layout, we can add margins and paddings to the controls that we added in the previous step.

Instead of manually adding CSS to the controls, we will use the standard classes provided by SAPUI5. These classes take care of consistent sizing steps, left-to-right support, and responsiveness.

Preview

Figure 18: The layout of the panel and its content now has margins and padding

Coding

You can view and download all files at Walkthrough - Step 13.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App>
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <Panel
                            headerText="{i18n>helloPanelTitle}"
                            class="sapUiResponsiveMargin"
                            width="auto">
                            <content>
                                <Button
                                    text="{i18n>showHelloButtonText}"
                                    press=".onShowHello"
                                    class="sapUiSmallMarginEnd"/>
                                <Input
                                    value="{/recipient/name}"
                                    valueLiveUpdate="true"
                                    width="60%"/>
                                <Text
                                    text="Hello {/recipient/name}"
                                    class="sapUiSmallMargin"/>
                            </content>
                        </Panel>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

To layout the panel, we add the CSS class `sapUiResponsiveMargin` that will add some space around the panel. We have to set the width of the panel to `auto` since the margin would otherwise be added to the default width of 100% and exceed the page size.

If you decrease the screen size, then you can actually see that the margin also decreases. As the name suggests, the margin is responsive and adapts to the screen size of the device. Tablets will get a smaller margin and phones in portrait mode will not get a margin to save space on these small screens.

Margins can be added to all kinds of controls and are available in many different options. We can even add space between the button and the input field by adding class `sapUiSmallMarginEnd` to the button.

To format the output text individually, we remove the description from the input field and add a new `Text` control with the same value. Here we also use a small margin to align it with the other content. Similarly, we could add the standard padding classes to layout the inner parts of container controls such as our panel, but as it already brings a padding by default, this is not needed here.

Conventions

- Use the standard SAPUI5 CSS classes for the layout if possible.

Related Information

Using Predefined CSS Margin Classes [page 1046]
Using Container Content Padding CSS Classes [page 1051]

Step 14: Custom CSS and Theme Colors

Sometimes we need to define some more fine-granular layouts and this is when we can use the flexibility of CSS by adding custom style classes to controls and style them as we like.

Preview

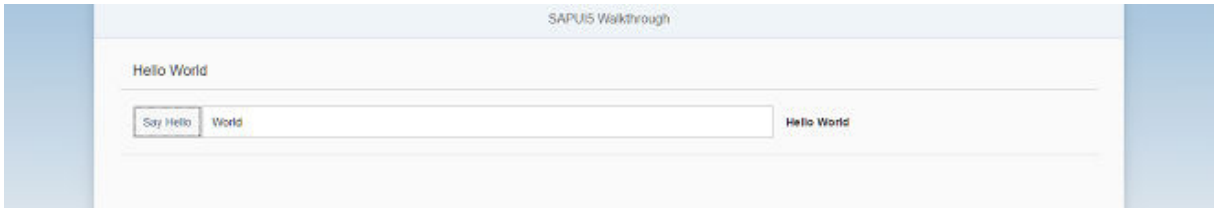


Figure 19: The space between the button and the input field is now smaller and the output text is bold

> ⚠ Caution
>
> As stated in the *Compatibility Rules*, the HTML and CSS generated by SAPUI5 is not part of the public API and may change in patch and minor releases. If you decide to override styles, you have the obligation to test and update your modifications each time SAPUI5 is updated. A prerequisite for this is that you have control over the version of SAPUI5 being used, for example in a standalone scenario. This is not possible when running your app in the SAP Fiori launchpad where SAPUI5 is centrally loaded for all apps. As such, SAP Fiori launchpad apps should not override styles.

Coding

You can view and download all files at Walkthrough - Step 14.

webapp/css/style.css (New)

```
html[dir="ltr"] .myAppDemoWT .myCustomButton.sapMBtn {
   margin-right: 0.125rem
}
```

```
html[dir="rtl"] .myAppDemoWT .myCustomButton.sapMBtn {
   margin-left: 0.125rem
}

.myAppDemoWT .myCustomText {
   display: inline-block;
   font-weight: bold;
}
```

We create a folder `css` which will contain our CSS files. In a new style definition file inside the `css` folder we create our custom classes combined with a custom namespace class. This makes sure that the styles will only be applied on controls that are used within our app.

A button has a default margin of `0` that we want to override: We add a custom margin of `2px` (or `0.125rem` calculated relatively to the default font size of `16px`) to the button with the style class `myCustomButton`. We add the CSS class `sapMBtn` to make our selector more specific: in CSS, the rule with the most specific selector "wins".

For right-to-left (rtl) languages, like Arabic, you set the left margin and reset the right margin as the app display is inverted. If you only use standard SAPUI5 controls, you don't need to care about this, in this case where we use custom CSS, you have to add this information.

In an additional class `myCustomText` we define a bold text and set the display to `inline-block`. This time we just define our custom class without any additional selectors. We do not set a color value here yet, we will do this in the view.

webapp/manifest.json

```
...
  "sap.ui5": {
    ...
    "models": {
      ...
    },
    "resources": {
      "css": [
        {
          "uri": "css/style.css"
        }
      ]
    }
  }
```

In the `resources` section of the `sap.ui5` namespace, additional resources for the app can be loaded. We load the CSS styles by defining a URI relative to the component. SAPUI5 then adds this file to the header of the HTML page as a `<link>` tag, just like in plain Web pages, and the browser loads it automatically.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
```

```
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App class="myAppDemoWT">
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <Panel
                            headerText="{i18n>helloPanelTitle}"
                            class="sapUiResponsiveMargin"
                            width="auto">
                            <content>
                                <Button
                                    text="{i18n>showHelloButtonText}"
                                    press=".onShowHello"
                                    class="myCustomButton"/>
                                <Input
                                    value="{/recipient/name}"
                                    valueLiveUpdate="true"
                                    width="60%"/>
                                <FormattedText
                                    htmlText="Hello {/recipient/name}"
                                    class="sapUiSmallMargin sapThemeHighlight-
asColor myCustomText"/>
                            </content>
                        </Panel>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

The app control is configured with our custom namespace class `myAppDemoWT`. This class has no styling rules set and is used in the definition of the CSS rules to define CSS selectors that are only valid for this app.

We add our custom CSS class to the button to precisely define the space between the button and the input field. Now we have a pixel-perfect design for the panel content.

To highlight the output text, we use a `FormattedText` control which can be styled individually, either by using custom CSS or with HTML code. We add our custom CSS class (`myCustomText`) and add a theme-dependent CSS class to set the highlight color that is defined in the theme.

The actual color now depends on the selected theme which ensures that the color always fits to the theme and is semantically clear. For a complete list of the available CSS class names, see CSS Classes for Theme Parameters [page 1262].

Conventions

- Do not specify colors in custom CSS but use the standard theme-dependent classes instead.

Related Information

Descriptor for Applications, Components, and Libraries [page 734]
CSS Classes for Theme Parameters [page 1262]

Creating Themable User Interfaces [page 1261]
Compatibility Rules [page 17]
API Reference: `sap.ui.core.theming`
Samples: `sap.ui.core.theming`

Step 15: Nested Views

Our panel content is getting more and more complex and now it is time to move the panel content to a separate view. With that approach, the application structure is much easier to understand, and the individual parts of the app can be reused.

Preview

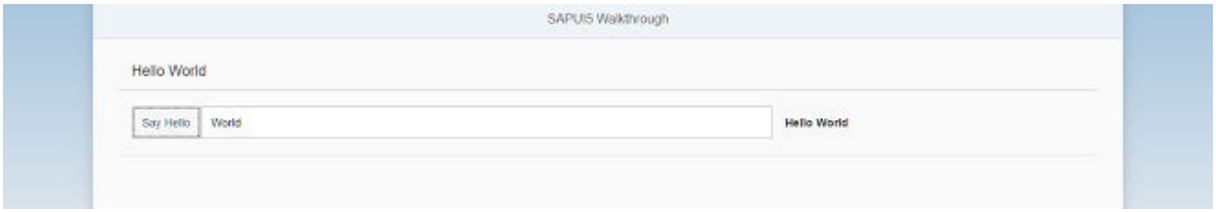


Figure 20: The panel content is now refactored to a separate view (No visual changes to last step)

Coding

You can view and download all files at Walkthrough - Step 15.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App class="myAppDemoWT">
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <mvc:XMLView
viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

Instead of putting the panel and its content directly into our `App` view, we will move it to a new separate `HelloPanel` view. We refer to this using an `XMLView` tag in the content aggregation of the panel.

webapp/view/HelloPanel.view.xml (New)

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Panel
      headerText="{i18n>helloPanelTitle}"
      class="sapUiResponsiveMargin"
      width="auto" >
      <content>
         <Button
            text="{i18n>showHelloButtonText}"
            press=".onShowHello"
            class="myAppDemoWT myCustomButton"/>
         <Input
            value="{/recipient/name}"
            valueLiveUpdate="true"
            width="60%"/>
         <FormattedText
            htmlText="Hello {/recipient/name}"
            class="sapUiSmallMargin sapThemeHighlight-asColor myCustomText"/>
      </content>
   </Panel>
</mvc:View>
```

The whole content for the panel is now added to the new file `HelloPanel.view.xml`. We also specify the controller for the view by setting the `controllerName` attribute of the XML view.

webapp/controller/HelloPanel.controller.js (New)

```
sap.ui.define([
   "sap/ui/core/mvc/Controller",
   "sap/m/MessageToast"
], function (Controller, MessageToast) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.HelloPanel", {
      onShowHello : function () {
         // read msg from i18n model
         var oBundle = this.getView().getModel("i18n").getResourceBundle();
         var sRecipient = this.getView().getModel().getProperty("/recipient/
name");
         var sMsg = oBundle.getText("helloMsg", [sRecipient]);
         // show message
         MessageToast.show(sMsg);
      }
   });
});
```

To have a reusable asset, the method `onShowHello` is also moved from the app controller to the `HelloPanel` controller.

webapp/controller/App.controller.js

```
sap.ui.define([
   "sap/ui/core/mvc/Controller"
], function (Controller) {
   "use strict";
   return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
   });
});
```

We have now moved everything out of the app view and controller. The app controller remains an empty stub for now, we will use it later to add more functionality.

Step 16: Dialogs and Fragments

In this step, we will take a closer look at another element which can be used to assemble views: the fragment.

Fragments are light-weight UI parts (UI subtrees) which can be reused but do not have any controller. This means, whenever you want to define a certain part of your UI to be reusable across multiple views, or when you want to exchange some parts of a view against one another under certain circumstances (different user roles, edit mode vs read-only mode), a fragment is a good candidate, especially where no additional controller logic is required.

A fragment can consist of 1 to n controls. At runtime, fragments placed in a view behave similar to "normal" view content, which means controls inside the fragment will just be included into the view's DOM when rendered. There are of course controls that are not designed to become part of a view, for example, dialogs.

But even for these controls, fragments can be particularly useful, as you will see in a minute.

We will now add a dialog to our app. Dialogs are special, because they open on top of the regular app content and thus do not belong to a specific view. That means the dialog must be instantiated somewhere in the controller code, but since we want to stick with the declarative approach and create reusable artifacts to be as flexible as possible, and because dialogs cannot be specified as views, we will create an XML fragment containing the dialog. A dialog, after all, can be used in more than only one view of your app.

Preview

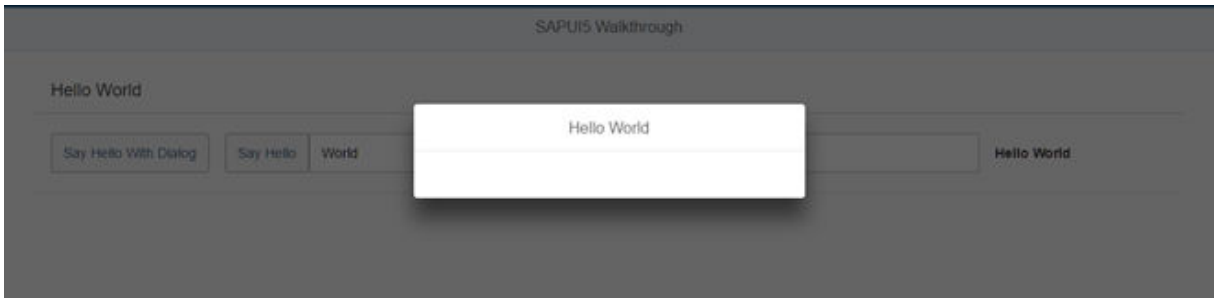


Figure 21: A dialog opens when the new "Say Hello With Dialog" button is clicked

Coding

You can view and download all files at Walkthrough - Step 16.

webapp/view/HelloPanel.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Panel
      headerText="{i18n>helloPanelTitle}"
      class="sapUiResponsiveMargin"
      width="auto" >
      <content>
      <Button
         id="helloDialogButton"
         text="{i18n>openDialogButtonText}"
         press=".onOpenDialog"
         class="sapUiSmallMarginEnd"/>
      <Button
         text="{i18n>showHelloButtonText}"
         press=".onShowHello"
         class="myCustomButton"/>
      <Input
         value="{/recipient/name}"
         valueLiveUpdate="true"
         width="60%"/>
      <FormattedText
         htmlText="Hello {/recipient/name}"
         class="sapUiSmallMargin sapThemeHighlight-asColor myCustomText"/>
      </content>
   </Panel>
</mvc:View>
```

We add a new button to the view to open the dialog. It simply calls an event handler function in the controller of the panel's content view. We will need the new `id="helloDialogButton"` in Step 29: Integration Test with OPA [page 151].

It is a good practice to set a unique ID like `helloWorldButton` to key controls of your app so that can be identified easily. If the attribute `id` is not specified, the OpenUI5 runtime generates unique but changing ID like `__button23` for the control. Inspect the DOM elements of your app in the browser to see the difference.

webapp/view/HelloDialog.fragment.xml (New)

```
<core:FragmentDefinition
   xmlns="sap.m"
   xmlns:core="sap.ui.core" >
   <Dialog
      id="helloDialog"
      title="Hello {/recipient/name}">
   </Dialog>
</core:FragmentDefinition>
```

We add a new XML file to declaratively define our dialog in a fragment. The fragment assets are located in the `core` namespace, so we add an `xml` namespace for it inside the `FragmentDefinition` tag.

The syntax is similar to a view, but since fragments do not have a controller this attribute is missing. Also, the fragment does not have any footprint in the DOM tree of the app, and there is no control instance of the fragment itself (only the contained controls). It is simply a container for a set of reuse controls.

We also add an `id` for our `Dialog` to be able to access the dialog from our `HelloPanel` controller.

webapp/controller/HelloPanel.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/ui/core/Fragment"
], function (Controller, MessageToast, Fragment) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.HelloPanel", {
        onShowHello : function () {
            …
        },
        onOpenDialog : function () {
            var oView = this.getView();

            // create dialog lazily
            if (!this.byId("helloDialog")) {
                // load asynchronous XML fragment
                Fragment.load({
                    id: oView.getId(),
                    name: "sap.ui.demo.walkthrough.view.HelloDialog"
                }).then(function (oDialog) {
                    // connect dialog to the root view of this component
(models, lifecycle)
                    oView.addDependent(oDialog);
                    oDialog.open();
                });
            } else {
                this.byId("helloDialog").open();
            }
        }
    });
});
```

If the dialog in the fragment does not exist yet, the fragment is instantiated by calling the `sap.ui.xmlfragment` method with the following arguments:

- The ID of the `HelloPanel` view
  This parameter is used to prefix the IDs inside our fragment. There, we have defined the ID `helloDialog` for the `Dialog` control, and we can access the dialog via the view by calling `oView.byId("helloDialog")`. This makes sure that even if you instantiate the same fragment in other views in the same way, each dialog will have its unique ID that is concatenated from the view ID and the dialog ID.
  Using unique IDs is important, because duplicate IDs lead to errors in the framework.
- The path of the fragment definition

We add the dialog as "dependent" on the view to be connected to the lifecycle of the view's model. A convenient side-effect is that the dialog will automatically be destroyed when the view is destroyed. Otherwise, we would have to destroy the dialog manually to free its resources.

Conventions

- Always use the `addDependent` method to connect the dialog to the lifecycle management and data binding of the view, even though it is not added to its UI tree.
- Private functions and variables should always start with an underscore.

webapp/i18n/i18n.properties

```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important
concepts of SAPUI5
# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
homePageTitle=Walkthrough
helloPanelTitle=Hello World
openDialogButtonText=Say Hello With Dialog
```

We add a new text for the open button to the text bundle.

Related Information

Reusing UI Parts: Fragments [page 1004]
Dialogs and other Popups as Fragments [page 1016]
Stable IDs: All You Need to Know [page 1442]
API Reference: `sap.ui.core.Fragment`

Step 17: Fragment Callbacks

Now that we have integrated the dialog, it's time to add some user interaction. The user will definitely want to close the dialog again at some point, so we add a button to close the dialog and assign an event handler.

Preview

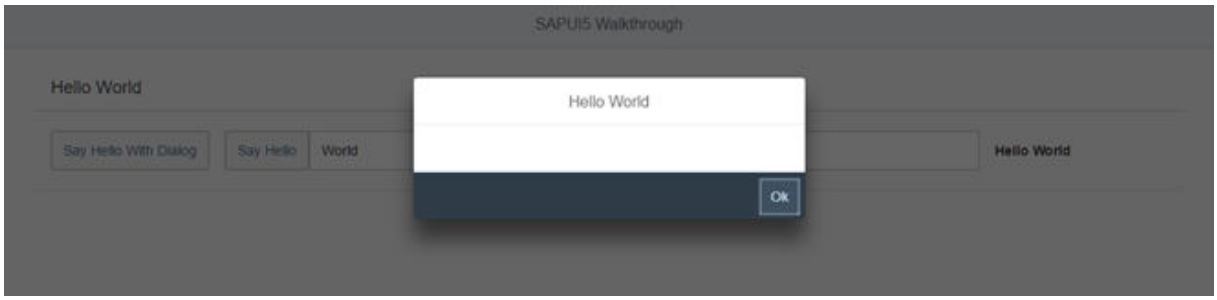


Figure 22: The dialog now has an "OK" button

Coding

You can view and download all files at Walkthrough - Step 17.

webapp/controller/HelloPanel.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/ui/core/Fragment"
], function (Controller, MessageToast, Fragment) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.HelloPanel", {
        onShowHello : function () {
            // read msg from i18n model
            var oBundle = this.getView().getModel("i18n").getResourceBundle();
            var sRecipient = this.getView().getModel().getProperty("/recipient/
name");
            var sMsg = oBundle.getText("helloMsg", [sRecipient]);
            // show message
            MessageToast.show(sMsg);
        },
        onOpenDialog : function () {
            var oView = this.getView();
            // create dialog lazily
            if (!this.byId("helloDialog")) {
                // load asynchronous XML fragment
                Fragment.load({
                    id: oView.getId(),
                    name: "sap.ui.demo.walkthrough.view.HelloDialog",
                    controller: this
                }).then(function (oDialog) {
```

```
                    // connect dialog to the root view of this component
(models, lifecycle)
                    oView.addDependent(oDialog);
                    oDialog.open();
                });
            } else {
                this.byId("helloDialog").open();
            }
        },

        onCloseDialog : function () {
            this.byId("helloDialog").close();
        }
    });
});
```

As previously described, fragments are pure UI reuse artifacts and do not have a controller. The third parameter of the `sap.ui.xmlfragment` function is optional and allows passing in a reference to a (controller) object. For our dialog we reference the `HelloPanel` controller. However, the third parameter does not necessarily have to be a controller but can be any object. Just don't forget the `this` keyword.

The event handler function is put into the same controller file and it closes the dialog by accessing the internal helper function that returns the dialog.

webapp/view/HelloDialog.fragment.xml

```
<core:FragmentDefinition
   xmlns="sap.m"
   xmlns:core="sap.ui.core" >
   <Dialog
      id="helloDialog"
      title ="Hello {/recipient/name}">
      <beginButton>
         <Button
            text="{i18n>dialogCloseButtonText}"
            press=".onCloseDialog"/>
      </beginButton>
   </Dialog>
</core:FragmentDefinition>
```

In the fragment definition, we add a button to the `beginButton` aggregation of the dialog. The press handler is referring to an event handler called `.onCloseDialog`, and since we passed in the reference to the `HelloPanel` controller, the method will be invoked there when the button is pressed. The dialog has an aggregation named `beginButton` as well as `endButton`. Placing buttons in both of these aggregations makes sure that the `beginButton` is placed before the `endButton` on the UI. What `before` means, however, depends on the text direction of the current language. We therefore use the terms `begin` and `end` as a synonym to “left” and “right". In languages with left-to-right direction, the `beginButton` will be rendered left, the `endButton` on the right side of the dialog footer; in right-to-left mode for specific languages the order is switched.

webapp/i18n/i18n.properties

```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important
concepts of SAPUI5
# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
homePageTitle=Walkthrough
helloPanelTitle=Hello World
openDialogButtonText=Say Hello With Dialog
dialogCloseButtonText=Ok
```

The text bundle is extended by the new text for the dialog's close button.

Related Information

Reusing UI Parts: Fragments [page 1004]
Instantiation of Fragments [page 1006]

Step 18: Icons

Our dialog is still pretty much empty. Since SAPUI5 is shipped with a large icon font that contains more than 500 icons, we will add an icon to greet our users when the dialog is opened.

Preview

Figure 23: An icon is now displayed in the dialog box

Coding

You can view and download all files at Walkthrough - Step 18.

webapp/view/HelloPanel.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Panel
      headerText="{i18n>helloPanelTitle}"
      class="sapUiResponsiveMargin"
      width="auto" >
      <content>
         <Button
            id="helloDialogButton"
            icon="sap-icon://world"
            text="{i18n>openDialogButtonText}"
            press=".onOpenDialog"
            class="sapUiSmallMarginEnd"/>
         <Button
            text="{i18n>showHelloButtonText}"
            press=".onShowHello"
            class="myCustomButton"/>
         <Input
            value="{/recipient/name}"
            valueLiveUpdate="true"
            width="60%"/>
           <FormattedText
              htmlText="Hello {/recipient/name}"
              class="sapUiSmallMargin sapThemeHighlight-asColor myCustomText"/>
      </content>
   </Panel>
</mvc:View>
```

We add an icon to the button that opens the dialog. The `sap-icon://` protocol is indicating that an icon from the icon font should be loaded. The identifier `world` is the readable name of the icon in the icon font.

> → Tip
>
> You can look up other icons using the *Icon Explorer* tool in the Demo Kit.
>
> To call any icon, use its name as listed in the *Icon Explorer* in `sap-icon://`*<iconname>*.

webapp/view/HelloDialog.fragment.xml

```
<core:FragmentDefinition
   xmlns="sap.m"
   xmlns:core="sap.ui.core" >
   <Dialog
      id="helloDialog"
      title ="Hello {/recipient/name}">
```

```
      <content>
         <core:Icon
            src="sap-icon://hello-world"
            size="8rem"
            class="sapUiMediumMargin"/>
      </content>
      <beginButton>
         <Button
            text="{i18n>dialogCloseButtonText}"
            press=".onCloseDialog"/>
      </beginButton>
   </Dialog>
</core:FragmentDefinition>
```

In the dialog fragment, we add an icon control to the content aggregation of the dialog. Luckily, the icon font also comes with a “Hello World” icon that is perfect for us here. We also define the size of the icon and set a medium margin on it.

Conventions

- Always use icon fonts rather than images wherever possible, as they are scalable without quality loss (vector graphics) and do not need to be loaded separately.

Related Information

Icon Explorer
API Reference: `sap.ui.core.Icon`
Samples: `sap.ui.core.Icon`

Step 19: Reuse Dialogs

In this step, we expand our reuse concept and invoke the dialog at component level.

In step 16, we created a dialog as fragment, to make it reusable across views or across our whole app. But we placed the logic for retrieving the dialog instance and for opening and closing it respectively in the controller of the `HelloPanel` view. Sticking to this approach would require copying and pasting the code to the controller of each view that needs our dialog. This would cause an undesired code redundancy which we want to avoid.

In this step, we implement the solution to this problem: We expand our reuse concept and invoke the dialog at component level.

Preview

Figure 24: The dialog is now opened by the component (no visual changes to last step)

Coding

You can view and download all files at Walkthrough - Step 19.

webapp/Component.js

```
sap.ui.define([
    "sap/ui/core/UIComponent",
    "sap/ui/model/json/JSONModel",
    "./controller/HelloDialog"
], function (UIComponent, JSONModel, HelloDialog) {
    "use strict";
    return UIComponent.extend("sap.ui.demo.walkthrough.Component", {
        metadata : {
            manifest : "json"
        },
        init : function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
            // set data model
            var oData = {
                recipient : {
                    name : "World"
                }
            };
            var oModel = new JSONModel(oData);
            this.setModel(oModel);
            // set dialog
            this._helloDialog = new HelloDialog(this.getRootControl());
        },


        exit : function() {
            this._helloDialog.destroy();
            delete this._helloDialog;
        },

        openHelloDialog : function () {
            this._helloDialog.open();
```

```
        }
    });
});
```

The dialog instantiation is refactored to a new helper object which is stored in a private property of the component. For instantiation of the helper object, we have to pass the view instance to which the dialog is added (see method call `addDependent` in the implementation of the helper object `HelloDialog.js` below).

We want to connect the reuse dialog to the lifecycle of the root view of the app, so we pass an instance of the root view on to the constructor. It can be retrieved by calling the `getRootControl` method of the component.

> **i** Note
>
> As defined in parameter `rootView` in the `manifest.json` file, our root view is `sap.ui.demo.walkthrough.view.App`. From the component, the root view can be retrieved at runtime by accessing the `rootControl` aggregation.

To be able to open the dialog from other controllers as well, we implement a reuse function `openHelloDialog` which calls the `open` method of our helper object. By doing so, we also decouple the implementation details of the reuse dialog from the application coding.

Up to this point we added the new property `_helloDialog` to the component and assigned an instance of the `HelloDialog` object to it. We want to make sure that the memory allocated for this helper object is freed up when the component is destroyed. Otherwise our application may cause memory leaks.

To do so, we use the `exit` hook. The SAPUI5 framework calls the function assigned to `exit` when destroying the component. We call the destroy function of `HelloDialog` to clean up the helper class and end its lifecycle. Nevertheless, the instance itself would still exist in the browser memory. Therefore we delete our reference to the `HelloDialog` instance by calling `delete this._helloDialog` and the garbage collection of the browser can clean up its memory.

> **i** Note
>
> We don't have to destroy the instance of `JSONModel` that we created, because we assigned it to the component with the `setModel` function. The SAPUI5 framework will destroy it together with the component.

webapp/controller/HelloDialog.js (New)

```
sap.ui.define([
    "sap/ui/base/ManagedObject",
    "sap/ui/core/Fragment"
], function (ManagedObject, Fragment) {
    "use strict";

    return
ManagedObject.extend("sap.ui.demo.walkthrough.controller.HelloDialog", {

        constructor : function (oView) {
            this._oView = oView;
        },

        exit : function () {
```

```
            delete this._oView;
        },

        open : function () {
            var oView = this._oView;

            // create dialog lazily
            if (!oView.byId("helloDialog")) {
                var oFragmentController = {
                    onCloseDialog : function () {
                        oView.byId("helloDialog").close();
                    }
                };
                // load asynchronous XML fragment
                Fragment.load({
                    id: oView.getId(),
                    name: "sap.ui.demo.walkthrough.view.HelloDialog",
                    controller: oFragmentController
                }).then(function (oDialog) {
                    // connect dialog to the root view of this component
(models, lifecycle)
                    oView.addDependent(oDialog);
                    oDialog.open();
                });
            } else {
                oView.byId("helloDialog").open();
            }
        }

    });

});
```

The implementation of the `HelloDialog` reuse object extends an `sap.ui.base.ManagedObject` object to inherit some of the core functionality of SAPUI5.

Our `open` method is refactored from the `HelloPanel` controller and instantiates our dialog fragment as in the previous steps.

> **i Note**
>
> We do not pass a controller as third parameter to function `sap.ui.xmlfragment` but a local helper object `oFragmentContoller` which included the needed event handler function `onCloseDialog` for the fragment.

The `open` method now contains our dialog instantiation. The first time the `open` method is called, the dialog is instantiated. The `oView` argument of this method is used to connect the current view to the dialog. We will call the `open` method of this object later in the controller.

The `onCloseDialog` event handler is simply moved from the `HelloPanel` controller to the reuse object.

We also add an `exit` function, just like we did in the component, that will be called automatically when the object is being destroyed. To free up all allocated memory in the helper object, we delete the property that holds the reference to the view. The view itself will be destroyed by the component, so we don't need to take care for that.

webapp/controller/HelloPanel.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast"
], function (Controller, MessageToast) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.HelloPanel", {
        onShowHello : function () {
            // read msg from i18n model
            var oBundle = this.getView().getModel("i18n").getResourceBundle();
            var sRecipient = this.getView().getModel().getProperty("/recipient/
name");
            var sMsg = oBundle.getText("helloMsg", [sRecipient]);
            // show message
            MessageToast.show(sMsg);
        },
        onOpenDialog : function () {
            this.getOwnerComponent().openHelloDialog();
        }
    });
});
```

The `onOpenDialog` method now accesses its component by calling the helper method `getOwnerComponent`. When calling the open method of the reuse object we pass in the current view to connect it to the dialog.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App class="myAppDemoWT">
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <headerContent>
                        <Button
                            icon="sap-icon://hello-world"
                            press=".onOpenDialog"/>
                    </headerContent>
                    <content>
                        <mvc:XMLView
viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

We add a button to the header area of the app view to show the reuse of the hello world dialog. When pressing the button the dialog will be opened as with the button that we previously created in the panel.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function (Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
        onOpenDialog : function () {
            this.getOwnerComponent().openHelloDialog();
        }
    });
});
```

We add the method `onOpenDialog` also to the app controller so that the dialog will open with a reference to the current view.

Conventions

- Put all assets that are used across multiple controllers in separate modules.

Related Information

Memory Management on https://developer.mozilla.org

API Reference: `sap.ui.base.ManagedObject`

Step 20: Aggregation Binding

Now that we have established a good structure for our app, it's time to add some more functionality. We start exploring more features of data binding by adding some invoice data in JSON format that we display in a list below the panel.

Preview

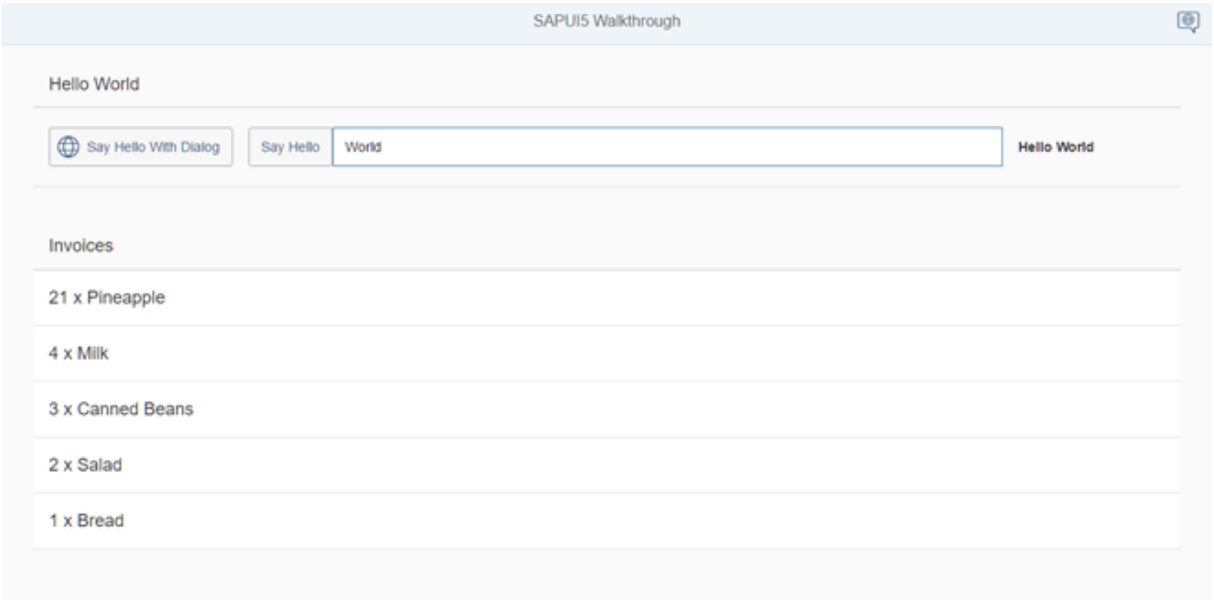


Figure 25: A list of invoices is displayed below the panel

Coding

You can view and download all files at Walkthrough - Step 20.

webapp/Invoices.json (New)

```
{
  "Invoices": [
    {
      "ProductName": "Pineapple",
      "Quantity": 21,
      "ExtendedPrice": 87.2000,
      "ShipperName": "Fun Inc.",
      "ShippedDate": "2015-04-01T00:00:00",
      "Status": "A"
    },
    {
      "ProductName": "Milk",
      "Quantity": 4,
```

```
      "ExtendedPrice": 9.99999,
      "ShipperName": "ACME",
      "ShippedDate": "2015-02-18T00:00:00",
      "Status": "B"
    },
    {
      "ProductName": "Canned Beans",
      "Quantity": 3,
      "ExtendedPrice": 6.85000,
      "ShipperName": "ACME",
      "ShippedDate": "2015-03-02T00:00:00",
      "Status": "B"
    },
    {
      "ProductName": "Salad",
      "Quantity": 2,
      "ExtendedPrice": 8.8000,
      "ShipperName": "ACME",
      "ShippedDate": "2015-04-12T00:00:00",
      "Status": "C"
    },
    {
      "ProductName": "Bread",
      "Quantity": 1,
      "ExtendedPrice": 2.71212,
      "ShipperName": "Fun Inc.",
      "ShippedDate": "2015-01-27T00:00:00",
      "Status": "A"
    }
  ]
}
```

The `invoices` file simply contains five invoices in a JSON format that we can use to bind controls against them in the app. JSON is a very lightweight format for storing data and can be directly used as a data source for SAPUI5 applications.

webapp/manifest.json

```
{
…
  "sap.ui5": {
    "rootView": "sap.ui.demo.walkthrough.view.App",
[…]
    "models": {
      "i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "settings": {
          "bundleName": "sap.ui.demo.walkthrough.i18n.i18n"
        }
      },
      "invoice": {
        "type": "sap.ui.model.json.JSONModel",
        "uri": "Invoices.json"
      }
    }
  }
}
```

We add a new model `invoice` to the `sap.ui5` section of the descriptor. This time we want a JSONModel, so we set the type to `sap.ui.model.json.JSONModel`. The `uri` key is the path to our test data relative to the component. With this little configuration our component will automatically instantiate a new `JSONModel` which

loads the invoice data from the `Invoices.json` file. Finally, the instantiated `JSONModel` is put onto the component as a named model invoice. The named model is then visible throughout our app.

> **i Note**
>
> Automatic model instantiation is only available as of SAPUI5 version 1.30. If you are using an older version, you can manually instantiate the resource bundle and other models of the app in the `onInit` method of the `Component.js` file as we did for the resource bundle in Step 9: Component Configuration [page 88].

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App class="myAppDemoWT">
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <headerContent>
                        <Button
                            icon="sap-icon://hello-world"
                            press=".onOpenDialog"/>
                    </headerContent>
                    <content>
                        <mvc:XMLView
viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
                        <mvc:XMLView
viewName="sap.ui.demo.walkthrough.view.InvoiceList"/>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

In the app view we add a second view to display our invoices below the panel.

webapp/view/InvoiceList.view.xml (New)

```
<mvc:View
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <List
      headerText="{i18n>invoiceListTitle}"
      class="sapUiResponsiveMargin"
      width="auto"
      items="{invoice>/Invoices}" >
      <items>
         <ObjectListItem
            title="{invoice>Quantity} x {invoice>ProductName}"/>
      </items>
   </List>
</mvc:View>
```

The new view is displaying a list control with a custom header text. The item aggregation of the list is bound to the root path `Invoices` of the JSON data. And since we defined a named model, we have to prefix each binding definition with the identifier `invoice>`.

In the `items` aggregation, we define the template for the list that will be automatically repeated for each invoice of our test data. More precisely, we use an `ObjectListItem` to create a control for each aggregated child of the `items` aggregation. The `title` property of the list item is bound to properties of a single invoice. This is achieved by defining a relative path (without `/` in the beginning). This works because we have bound the `items` aggregation via `items={invoice>/Invoices}` to the invoices.

webapp/i18n/i18n.properties

```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important
concepts of SAPUI5
# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
homePageTitle=Walkthrough
helloPanelTitle=Hello World
openDialogButtonText=Say Hello With Dialog
dialogCloseButtonText=Ok
# Invoice List
invoiceListTitle=Invoices
```

In the text bundle the title of the list is added.

Related Information

Lists [page 2321]
API Reference: `sap.m.List`
Samples: `sap.m.List`
List Binding (Aggregation Binding) [page 828]

Step 21: Data Types

The list of invoices is already looking nice, but what is an invoice without a price assigned? Typically prices are stored in a technical format and with a '`.`' delimiter in the data model. For example, our invoice for pineapples has the calculated price `87.2` without a currency. We are going to use the SAPUI5 data types to format the price properly, with a locale-dependent decimal separator and two digits after the separator.

Preview

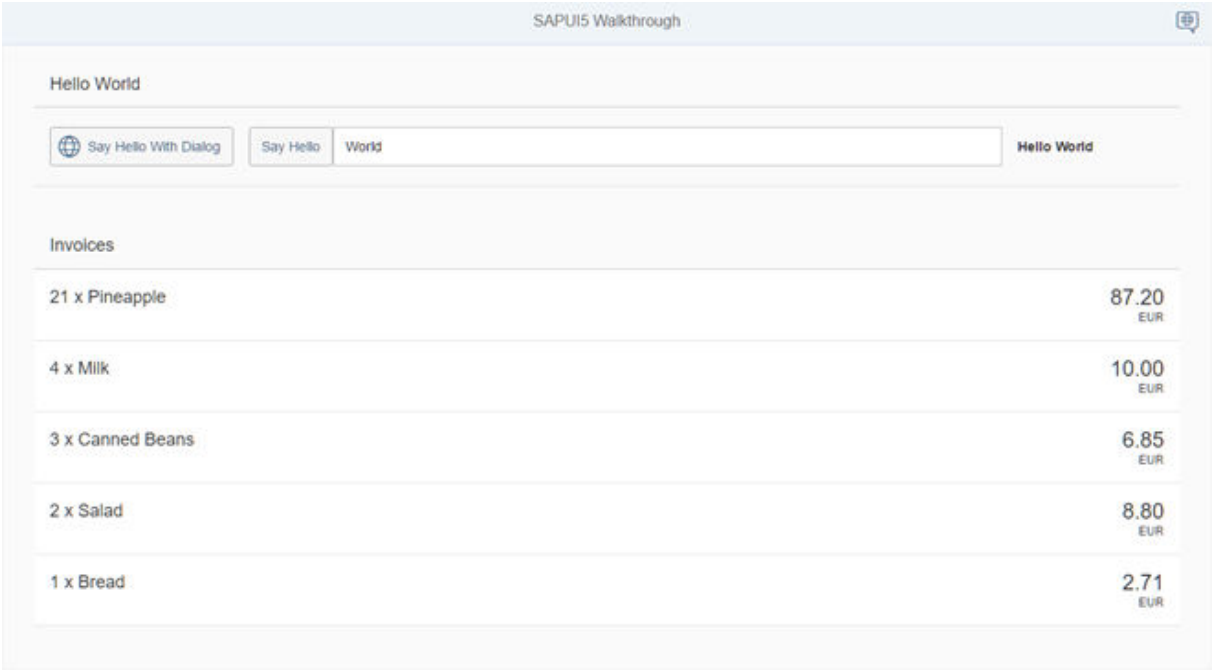


Figure 26: The list of invoices with prices and number units

Coding

You can view and download all files at Walkthrough - Step 21.

webapp/view/InvoiceList.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <List
      headerText="{i18n>invoiceListTitle}"
      class="sapUiResponsiveMargin"
      width="auto"
      items="{invoice>/Invoices}">
      <items>
         <ObjectListItem
        title="{invoice>Quantity} x {invoice>ProductName}"
        number="{
            parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/currency'}],
            type: 'sap.ui.model.type.Currency',
            formatOptions: {
                showMeasure: false
            }
        }"
        numberUnit="{view>/currency}"/>
    </items>
   </List>
```

```
</mvc:View>
```

We add a price to our invoices list in the view by adding the `number` and `numberUnit` attributes to the `ObjectListItem` control, then we apply the currency data type on the number by setting the `type` attribute of the binding syntax to `sap.ui.model.type.Currency`.

As you can see above, we are using a special binding syntax for the `number` property of the `ObjectListItem`. This binding syntax makes use of so-called "Calculated Fields", which allows the binding of multiple properties from different models to a single property of a control. The properties bound from different models are called “parts”. In the example above, the property of the control is `number` and the bound properties (“parts”) retrieved from two different models are `invoice>ExtendedPrice` and `view>/currency`.

We want to display the price in Euro, and typically the currency is part of our data model on the back end. In our case this is not the case, so we need to define it directly in the app. We therefore add a controller for the invoice list, and use the `currency` property as the second part of our binding syntax. The `Currency` type will handle the formatting of the price for us, based on the currency code. In our case, the price is displayed with 2 decimals.

Additionally, we set the formatting option `showMeasure` to `false`. This hides the currency code in the property `number`, because it is passed on to the `ObjectListItem` control as a separate property `numberUnit`.

webapp/controller/InvoiceList.controller.js (New)

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel"
], function (Controller, JSONModel) {
    "use strict";

    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {

        onInit : function () {
            var oViewModel = new JSONModel({
                currency: "EUR"
            });
            this.getView().setModel(oViewModel, "view");
        }

    });
});
```

To be able to access the currency code that is not part of our data model, we define a view model in the controller of the invoice list. It is a simple JSON model with just one key `currency` and the value `EUR`. This can be bound to the formatter of the number field. View models can hold any configuration options assigned to a control to bind properties such as the visibility.

Conventions

- Use data types instead of custom formatters whenever possible.

Related Information

Composite Binding [page 843]
Formatting, Parsing, and Validating Data [page 854]
API Reference: `sap.ui.model.type`
API Reference: `sap.ui.model.type.Currency`
Samples: `sap.ui.model.type.Currency`

Step 22: Expression Binding

Sometimes the predefined types of SAPUI5 are not flexible enough and you want to do a simple calculation or formatting in the view - that is where expressions are really helpful. We use them to format our price according to the current number in the data model.

Preview

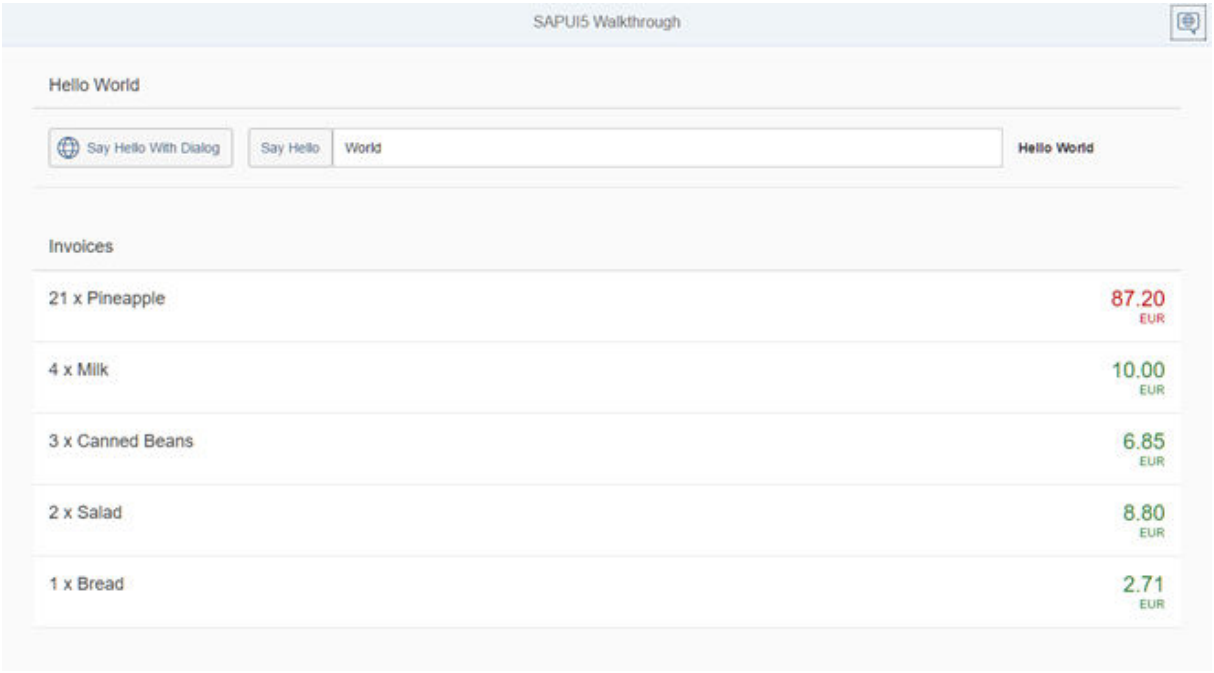


Figure 27: The price is now formatted according to its number

Coding

You can view and download all files at Walkthrough - Step 22.

webapp/view/InvoiceList.view.xml

```
<mvc:View
controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <List
      headerText="{i18n>invoiceListTitle}"
      class="sapUiResponsiveMargin"
      width="auto"
      items="{invoice>/Invoices}" >
      <items>
         <ObjectListItem
            title="{invoice>Quantity} x {invoice>ProductName}"
            number="{
        parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/currency'}],
        type: 'sap.ui.model.type.Currency',
        formatOptions: {
            showMeasure: false
        }
        }"
        numberUnit="{view>/currency}"
            numberState="{= ${invoice>ExtendedPrice} > 50 ? 'Error' :
'Success' }"/>
      </items>
   </List>
</mvc:View>
```

We add the property `numberState` in our declarative view and introduce a new binding syntax that starts with = inside the brackets. This symbol is used to initiate a new binding syntax, it's called an expression and can do simple calculation logic like the ternary operator shown here.

The condition of the operator is a value from our data model. A model binding inside an expression binding has to be escaped with the `$` sign as you can see in the code. We set the state to `'Error'` (the number will appear in red) if the price is higher than 50 and to `'Success'` (the number will appear in green) otherwise.

Expressions are limited to a particular set of operations that help formatting the data such as Math expression, comparisons, and such. You can lookup the possible operations in the documentation.

Conventions

- Only use expression binding for trivial calculations.

Related Information

Expression Binding [page 845]

Step 23: Custom Formatters

If we want to do a more complex logic for formatting properties of our data model, we can also write a custom formatting function. We will now add a localized status with a custom formatter, because the status in our data model is in a rather technical format.

Preview

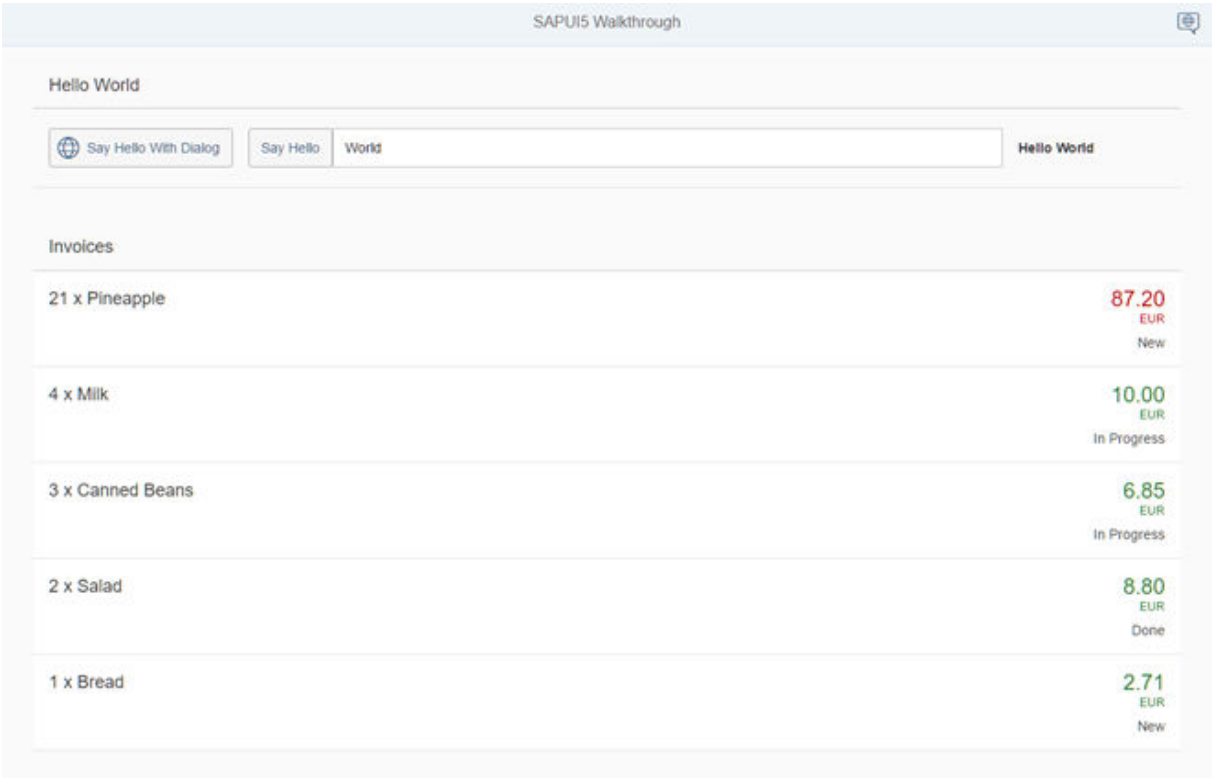


Figure 28: A status is now displayed with a custom formatter

Coding

You can view and download all files at Walkthrough - Step 23.

webapp/model/formatter.js (New)

```
sap.ui.define([], function () {
    "use strict";
    return {
        statusText: function (sStatus) {
            var resourceBundle =
this.getView().getModel("i18n").getResourceBundle();
```

```
            switch (sStatus) {
                case "A":
                    return resourceBundle.getText("invoiceStatusA");
                case "B":
                    return resourceBundle.getText("invoiceStatusB");
                case "C":
                    return resourceBundle.getText("invoiceStatusC");
                default:
                    return sStatus;
            }
        }
    };
});
```

We create a new folder `model` in our app project. The new `formatter` file is placed in the model folder of the app, because formatters are working on data properties and format them for display on the UI. So far we did not have any model-related artifacts, except for the `Invoices.json` file, we will now add the folder `webapp/model` to our app. This time we do not extend from any base object but just return a JavaScript object with our `formatter` functions inside the `sap.ui.define` call.

Function `statusText` gets the technical status from the data model as input parameter and returns a human-readable text that is read from the `resourceBundle` file.

webapp/controller/InvoiceList.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "../model/formatter"
], function (Controller, JSONModel, formatter) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {
        formatter: formatter,
        onInit : function () {
            var oViewModel = new JSONModel({
                currency: "EUR"
            });
            this.getView().setModel(oViewModel, "view");
        }
    });
});
```

To load our formatter functions, we have to add it to the `InvoiceList.controller.js`. In this controller, we first add a dependency to our custom `formatter` module. The controller simply stores the loaded formatter functions in the local property `formatter` to be able to access them in the view.

webapp/view/InvoiceList.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <List
            headerText="{i18n>invoiceListTitle}"
```

```
        class="sapUiResponsiveMargin"
        width="auto"
        items="{invoice>/Invoices}">
        <items>
            <ObjectListItem
                title="{invoice>Quantity} x {invoice>ProductName}"
                number="{
                    parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/
currency'}],
                    type: 'sap.ui.model.type.Currency',
                    formatOptions: {
                        showMeasure: false
                    }
                }"
                numberUnit="{view>/currency}"
                numberState="{=    ${invoice>ExtendedPrice} > 50 ? 'Error' :
'Success' }">
                <firstStatus>
                    <ObjectStatus text="{
                        path: 'invoice>Status',
                        formatter: '.formatter.statusText'
                    }"/>
                </firstStatus>
            </ObjectListItem>
        </items>
    </List>
</mvc:View>
```

We add a status using the `firstStatus` aggregation to our `ObjectListItem` that will display the status of our invoice. The custom formatter function is specified with the reserved property `formatter` of the binding syntax. A `"."` in front of the formatter name means that the function is looked up in the controller of the current view. There we defined a property `formatter` that holds our formatter functions, so we can access it by `.formatter.statusText`.

webapp/i18n/i18n.properties

```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important
concepts of SAPUI5
# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
homePageTitle=Walkthrough
helloPanelTitle=Hello World
openDialogButtonText=Say Hello With Dialog
dialogCloseButtonText=Ok
# Invoice List
invoiceListTitle=Invoices
invoiceStatusA=New
invoiceStatusB=In Progress
invoiceStatusC=Done
```

We add three new entries to the resource bundle that reflect our translated status texts. These texts are now displayed below the `number` attribute of the `ObjectListItem` dependent on the status of the invoice.

Related Information

Formatting, Parsing, and Validating Data [page 854]

Step 24: Filtering

In this step, we add a search field for our product list and define a filter that represents the search term. When searching, the list is automatically updated to show only the items that match the search term.

Preview

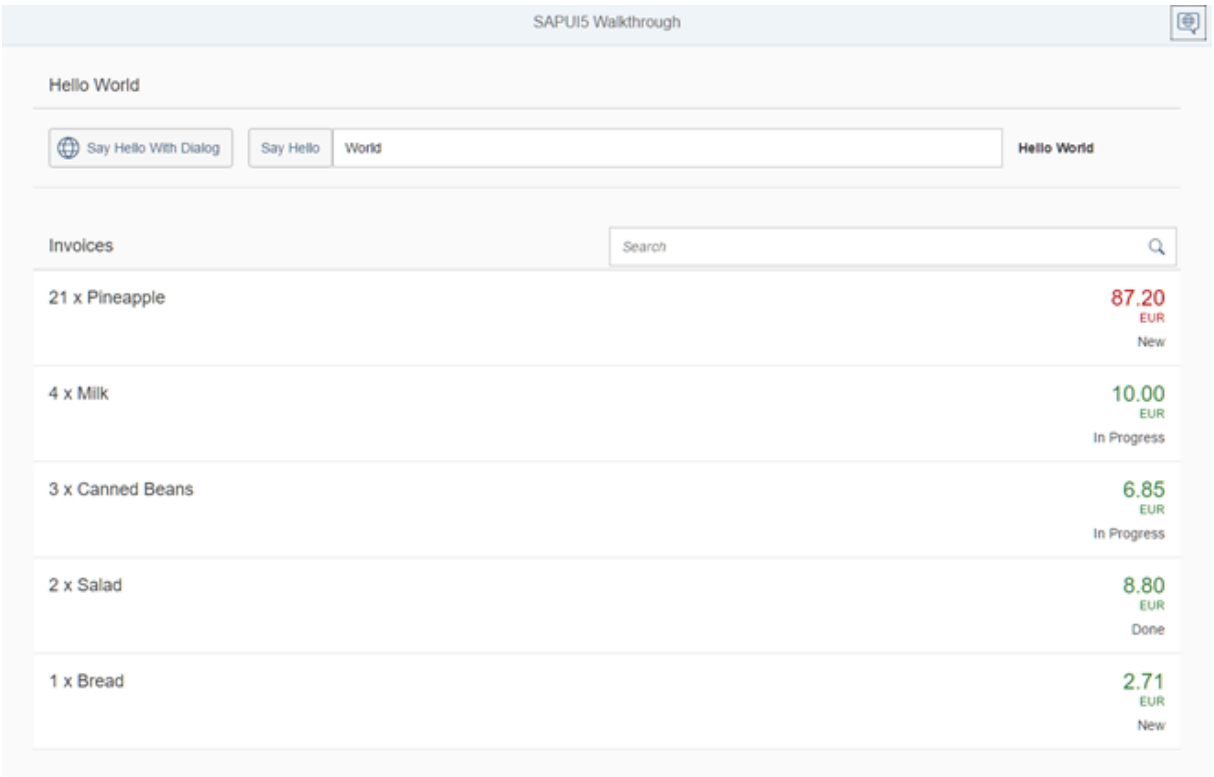


Figure 29: A search field is displayed above the list

Coding

You can view and download all files at Walkthrough - Step 24.

webapp/view/InvoiceList.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <List
      id="invoiceList"
      class="sapUiResponsiveMargin"
      width="auto"
      items="{invoice>/Invoices}" >
      <headerToolbar>
         <Toolbar>
            <Title text="{i18n>invoiceListTitle}"/>
            <ToolbarSpacer/>
            <SearchField width="50%" search=".onFilterInvoices"/>
         </Toolbar>
      </headerToolbar>
      <items>
         <ObjectListItem>
        …
         </ObjectListItem/>
      </items>
   </List>
</mvc:View>
```

The view is extended by a search control that we add to the list of invoices. We also need to specify an ID `invoiceList` for the list control to be able to identify the list from the event handler function `onFilterInvoices` that we add to the search field. In addition, the search field is part of the list header and therefore, each change on the list binding will trigger a rerendering of the whole list, including the search field.

The `headerToolbar` aggregation replaces the simple `title` property that we used before for our list header. A toolbar control is way more flexible and can be adjusted as you like. We are now displaying the title on the left side with a `sap.m.Title` control, a spacer, and the `sap.m.SearchField` on the right.

webapp/controller/InvoiceList.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "../model/formatter",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator"
], function (Controller, JSONModel, formatter, Filter, FilterOperator) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {
        formatter: formatter,
        onInit : function () {
            var oViewModel = new JSONModel({
                currency: "EUR"
            });
            this.getView().setModel(oViewModel, "view");
        },
        onFilterInvoices : function (oEvent) {

            // build filter array
            var aFilter = [];
            var sQuery = oEvent.getParameter("query");
```

```
            if (sQuery) {
                aFilter.push(new Filter("ProductName", FilterOperator.Contains,
sQuery));
            }

            // filter binding
            var oList = this.byId("invoiceList");
            var oBinding = oList.getBinding("items");
            oBinding.filter(aFilter);
        }
    });
});
```

We load two new dependencies for the filtering. The filter object will hold our configuration for the filter action and the `FilterOperator` is a helper type that we need in order to specify the filter.

In the `onFilterInvoices` function we construct a filter object from the search string that the user has typed in the search field. Event handlers always receive an event argument that can be used to access the parameters that the event provides. In our case the search field defines a parameter `query` that we access by calling `getParameter("query")` on the `oEvent` parameter.

If the query is not empty, we add a new filter object to the still empty array of filters. However, if the query is empty, we filter the binding with an empty array. This makes sure that we see all list elements again. We could also add more filters to the array, if we wanted to search more than one data field. In our example, we just search in the `ProductName` path and specify a filter operator that will search for the given query string.

The list is accessed with the ID that we have specified in the view, because the control is automatically prefixed by the view ID, we need to ask the view for the control with the helper function `byId`. On the list control we access the binding of the aggregation `items` to filter it with our newly constructed filter object. This will automatically filter the list by our search string so that only the matching items are shown when the search is triggered. The filter operator `FilterOperator.Contains` is **not** case-sensitive.

Related Information

API Reference: `sap.ui.model.Filter`

API Reference: `sap.ui.model.FilterOperator`

API Reference: `sap.m.SearchField`

Step 25: Sorting and Grouping

To make our list of invoices even more user-friendly, we sort it alphabetically instead of just showing the order from the data model. Additionally, we introduce groups and add the company that ships the products so that the data is easier to consume.

Preview

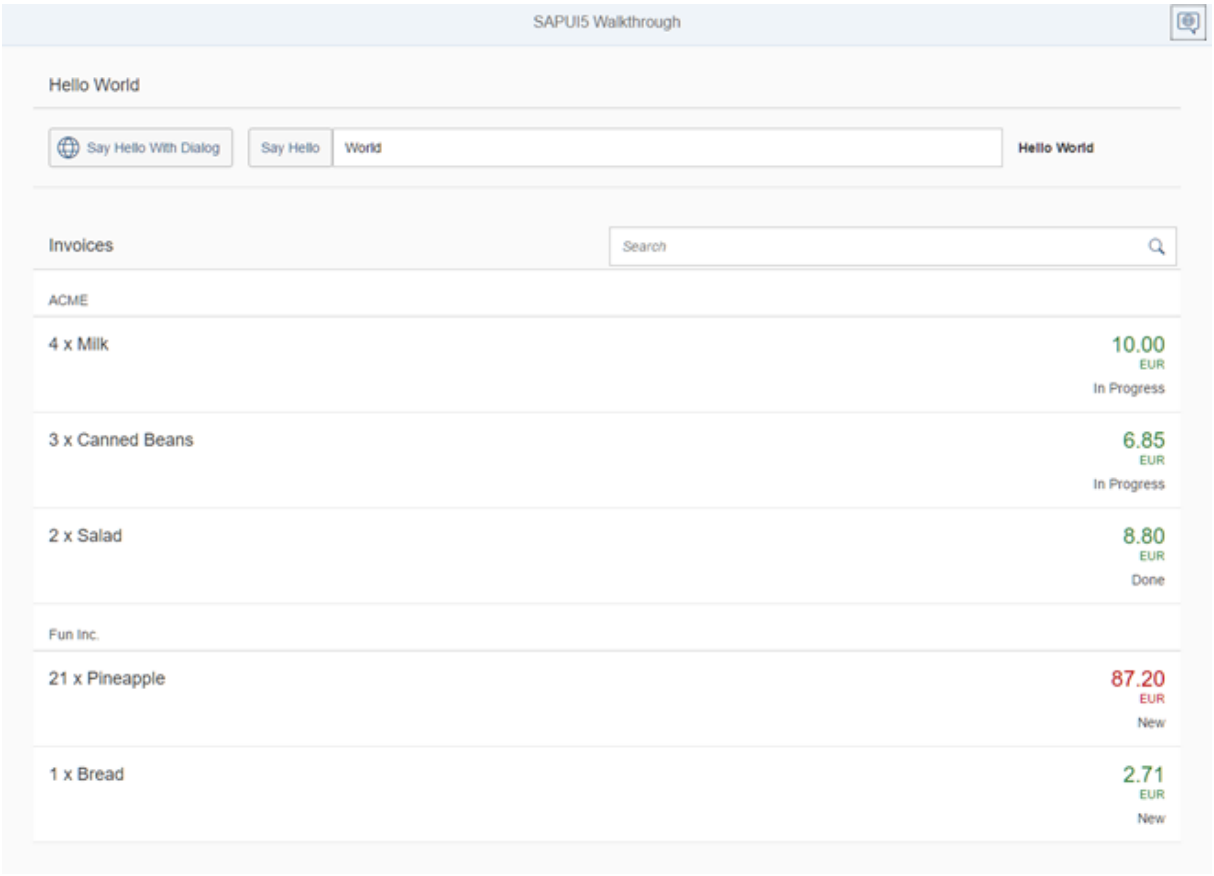


Figure 30: The list is now sorted and grouped by the shipping company

Coding

You can view and download all files at Walkthrough - Step 25.

webapp/view/InvoiceList.view.xml

```
<mvc:View
   controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
   xmlns="sap.m"
```

```
   xmlns:mvc="sap.ui.core.mvc">
   <List
      id="invoiceList"
      class="sapUiResponsiveMargin"
      width="auto"
      items="{
         path : 'invoice>/Invoices',
         sorter : {
            path : 'ProductName'
         }
      }" >
      <headerToolbar>
         ...
      </headerToolbar>
      <items>
         ...
      </items>
   </List>
</mvc:View>
```

We add a declarative sorter to our binding syntax. As usual, we transform the simple binding syntax to the object notation, specify the path to the data, and now add an additional `sorter` property. We specify the data path by which the invoice items should be sorted, the rest is done automatically. By default, the sorting is ascending, but you could also add a property `descending` with the value `true` inside the sorter property to change the sorting order.

If we run the app now we can see a list of invoices sorted by the name of the products.

webapp/view/InvoiceList.view.xml

```
<mvc:View
controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
xmlns="sap.m"
xmlns:mvc="sap.ui.core.mvc">
<List
        id="invoiceList"
        class="sapUiResponsiveMargin"
        width="auto"
        items="{
            path : 'invoice>/Invoices',
            sorter : {
                path : 'ShipperName',
                group : true
            }
        }">
    <headerToolbar>
        <Toolbar>
            <Title text="{i18n>invoiceListTitle}"/>
            <ToolbarSpacer/>
            <SearchField width="50%" search=".onFilterInvoices"/>
        </Toolbar>
    </headerToolbar>
    <items>
        …
    </items>
</List>
</mvc:View>
```

We modify the view and add a different sorter, or better; we change the sorter and set the attribute `group` to true. We also specify the path to the `ShipperName` data field. This groups the invoice items by the shipping company.

As with the sorter, no further action is required. The list and the data binding features of SAPUI5 will do the trick to display group headers automatically and categorize the items in the groups. We could define a custom grouping function if we wanted by setting the `groupHeaderFactory` property, but the result looks already fine.

Related Information

API Reference: `sap.ui.model.Sorter`
Sample: List - Grouping

Step 26: Remote OData Service

So far we have worked with local JSON data, but now we will access a real OData service to visualize remote data.

In the real world, data often resides on remote servers and is accessed via an OData service. We will add a data source configuration to the manifest and replace the JSONModel type for our `invoice` model with the publicly available Northwind OData service to visualize remote data. You will be surprised how little needs to be changed in order to make this work!

> **i** Note
>
> If you cannot get it to run, don't worry too much, the remaining steps will also work with the local JSON data you have used so far. In Step 27: Mock Server Configuration [page 139], you will learn how to simulate a back-end system to achieve a similar working scenario. However, you should at least read this chapter about remote OData services to learn about non-local data sources.

Preview

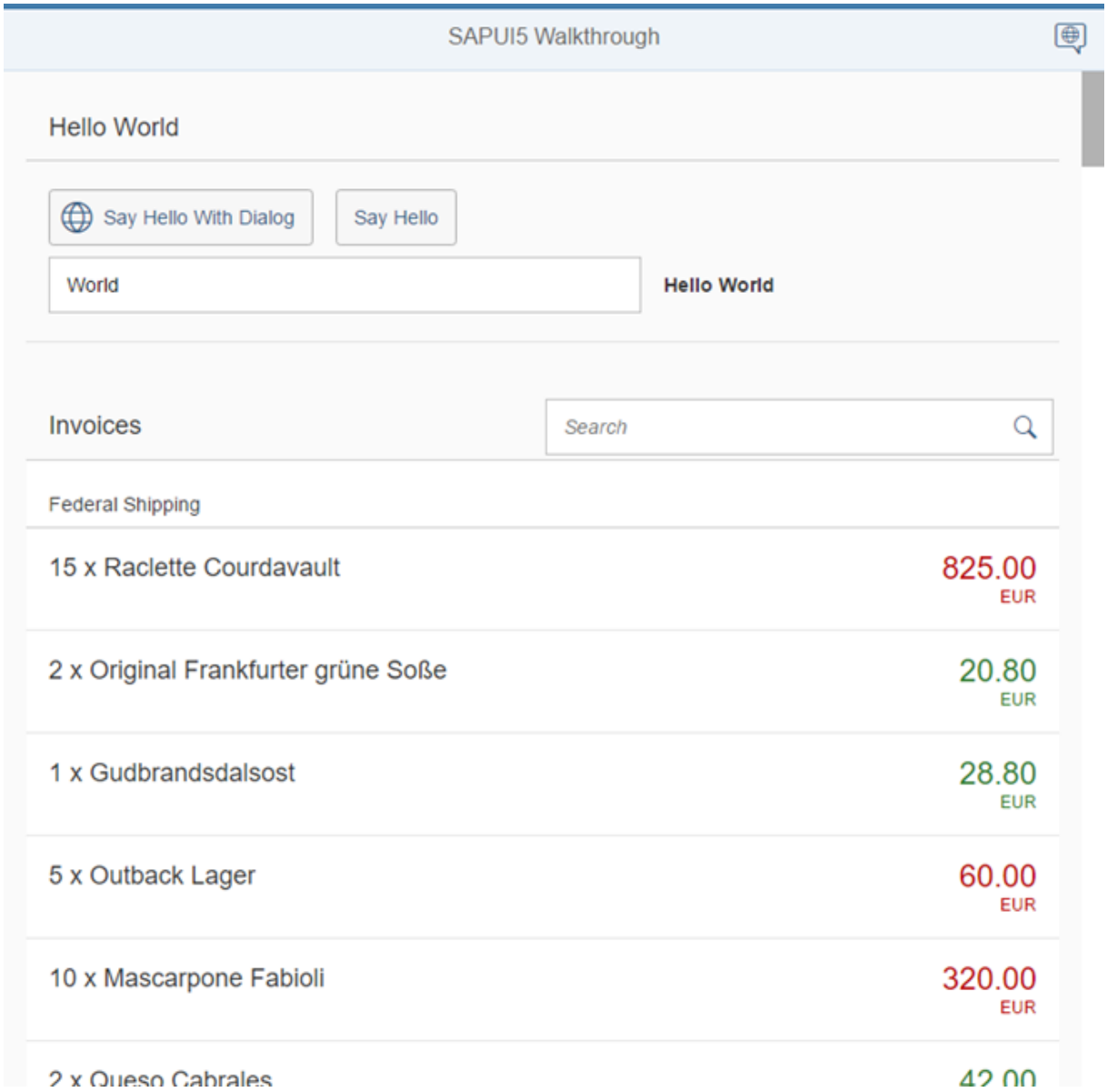


Figure 31: Products from the OData invoices test service are now shown within our app

Coding

You can view and download all files at Walkthrough - Step 26.

webapp/manifest.json

```
{
  "_version": "1.12.0",
```

```
  "sap.app": {
    ...
    "ach": "CA-UI5-DOC",
    "dataSources": {
      "invoiceRemote": {
        "uri": "https://services.odata.org/V2/Northwind/Northwind.svc/",
        "type": "OData",
        "settings": {
          "odataVersion": "2.0"
        }
      }
    }
  },
  "sap.ui": {
    ...
  },
  "sap.ui5": {
    ...
    "models": {
      "i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "settings": {
          "bundleName": "sap.ui.demo.walkthrough.i18n.i18n"
        }
      },
      "invoice": {
        "dataSource": "invoiceRemote"
      }
    }
  }
}
```

In the `sap.app` section of the descriptor file, we add a data source configuration. With the `invoiceRemote`, key we specify a configuration object that allows automatic model instantiation. We specify the type of the service (`OData`) and the model version (`2.0`). In this step, we want to use the publicly available Northwind OData service located at `https://services.odata.org/V2/Northwind/Northwind.svc/`. Therefore, the URI points to the official Northwind OData service.

In the `models` section, we replace the content of the `invoice` model. This key is still used as model name when the model is automatically instantiated during the component initialization. However, the `invoiceRemote` value of the `dataSource` key is a reference to the data source section that we specified above. This configuration allows the component to retrieve the technical information for this model during the start-up of the app.

Our component now automatically creates an instance of `sap.ui.model.odata.v2.ODataModel` according to the settings we specified above, and makes it available as a model named `invoice`. When you use the `invoiceRemote` data source, the `ODataModel` fetches the data from the real Northwind OData service. The invoices we receive from the Northwind OData service have identical properties as the JSON data we used previously (except for the `status` property, which is not available in the Northwind OData service).

> **i Note**
>
> If you want to have a default model on the component, you can change the name of the model to an empty string in the descriptor file. Automatically instantiated models can be retrieved by calling `this.getModel` in the component. In the controllers of component-based apps you can call `this.getView().getModel()` to get the automatically instantiated model. For retrieving a named model you have to pass on the model name defined in the descriptor file to `getModel`, that is, in the component you would call `this.getModel("invoice")` to get our automatically generated `invoice` model that we defined in the descriptor.

You can now try to run the app and see what happens - we will see an error related to our new configuration in the console:

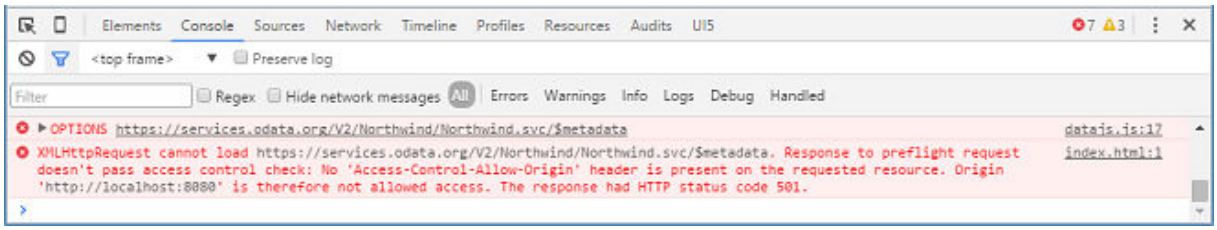


Figure 32: Violations of the same-origin policy in Google Chrome

Due to the so called same-origin policy, browsers deny AJAX requests to service endpoints in case the service endpoint has a different domain/subdomain, protocol, or port than the app. The browser refuses to connect to a remote URL directly for security reasons. Depending on your development environment you have different options to overcome this limitation:

- SAP Web IDE: Configure a destination [page 1392]
- Local Development: Configure a local proxy (CORS anywhere) [page 1394]
- Workaround: Disabling the same-origin policy in the browser [page 1395] (not recommended, only for testing)

Related Information

OData Home Page
API Reference: `sap.ui.model.odata.v2.ODataModel`
First-Aid Kit [page 1386]
Request Fails Due to Same-Origin Policy (Cross-Origin Resource Sharing - CORS) [page 1391]

Step 27: Mock Server Configuration

We just ran our app against a real service, but for developing and testing our app we do not want to rely on the availability of the “real” service or put additional load on the system where the data service is located.

This system is the so-called back-end system that we will now simulate with an SAPUI5 feature called mock server. It serves local files, but it simulates a back-end system more realistically than just loading the local data. We will also change the model instantiation part so that the model is configured in the descriptor and instantiated automatically by SAPUI5. This way, we do not need to take care of the model instantiation in the code.

Preview

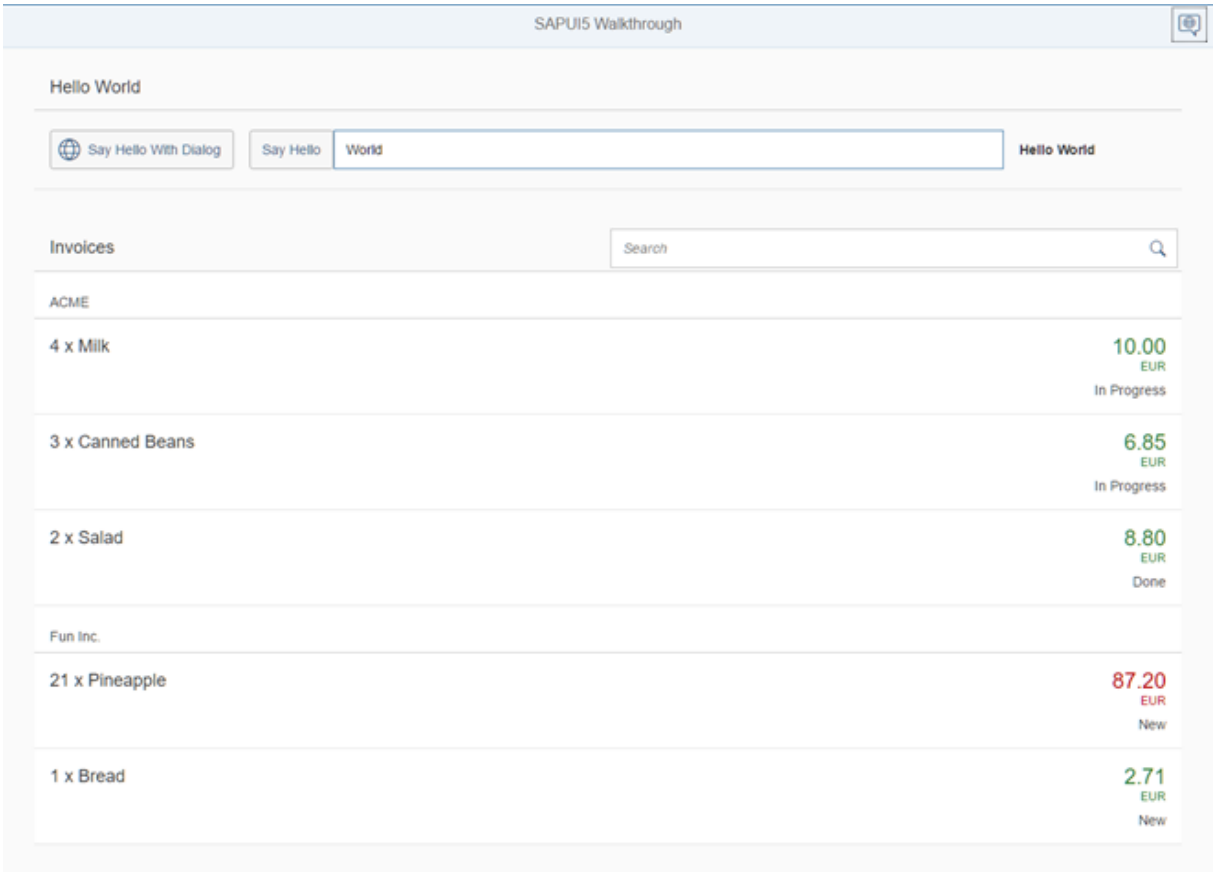


Figure 33: The list of invoices is now served by the Mock Server

Coding

You can view and download all files at Walkthrough - Step 27.

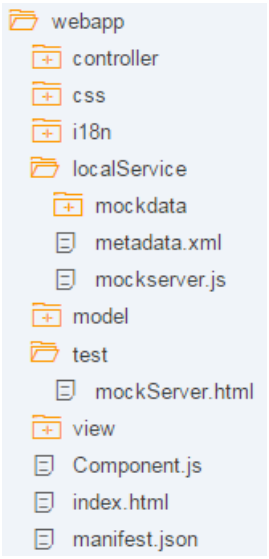


Figure 34: Folder Structure for this Step

The folder structure of our app project is clearly separating test and productive files after this step. The new `test` folder now contains a new HTML page `mockServer.html` which will launch our application in test mode without calling the real service.

The new `localService` folder contains a `metadata.xml` service description file for OData, the `mockserver.js` file that simulates a real service with local data, and the `mockdata` subfolder that contains the local test data (`Invoices.json`).

webapp/test/mockServer.html (New)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SAPUI5 Walkthrough</title>
    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "./"
        }'
        data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
</head>
<body class="sapUiBody" id="content">
    <div data-sap-ui-component data-name="sap.ui.demo.walkthrough" data-
id="container" data-settings='{"id" : "walkthrough"}'></div>
</body>
</html>
```

We copy the `index.html` to a separate file in the `webapp/test` folder and name it `mockServer.html`. We will now use this file to run our app in test mode with mock data loaded from a JSON file. Test pages should not

be placed in the application root folder but in a subfolder called `test` to clearly separate productive and test coding.

From this point on, you have two different entry pages: One for the real "connected" app (`index.html`) and one for local testing (`mockServer.html`). You can freely decide if you want to do the next steps on the real service data or on the local data within the app.

> **i** Note
>
> If no connection to the real service is available or the proxy configuration from the previous step does not work, you can always use the `mockServer.html` file. This will display the app with simulated test data. The `index.html` file will always load the data from a remote server. If the request fails, the list of invoices will stay empty.

webapp/test/mockServer.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SAPUI5 Walkthrough - Test Page</title>
    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "../"
        }'
        data-sap-ui-oninit="module:sap/ui/demo/walkthrough/test/initMockServer"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
</head>
<body class="sapUiBody" id="content">
    <div data-sap-ui-component data-name="sap.ui.demo.walkthrough" data-
id="container" data-settings='{"id" : "walkthrough"}'></div>
</body>
</html>
```

We modify the `mockServer.html` file and change the page title to distinguish it from the productive start page. In the bootstrap, the `data-sap-ui-resourceroots` property is also changed. The namespace now points to the folder above (`"../"`), because the `mockServer.html` file is now in a subfolder of the `webapp` folder. Instead of loading the app component directly, we now call a script `initMockServer.js`.

webapp/test/initMockServer.js (New)

```
sap.ui.define([
    "../localService/mockserver"
], function (mockserver) {
    "use strict";
```

```
    // initialize the mock server
    mockserver.init();

    // initialize the embedded component on the HTML page
    sap.ui.require(["sap/ui/core/ComponentSupport"]);
});
```

The first dependency is a file called `mockserver.js` that will be located in the `localService` folder later.

The `mockserver` depencency that we are about to implement is our local test server. Its `init` method is immediately called before we load the component. This way we can catch all requests that would go to the "real" service and process them locally by our test server when launching the app with the `mockServer.html` file. The component itself does not "know" that it will now run in test mode.

webapp/localService/mockdata/Invoices.json (New)

```
[
  {
    "ProductName": "Pineapple",
    "Quantity": 21,
    "ExtendedPrice": 87.2000,
    "ShipperName": "Fun Inc.",
    "ShippedDate": "2015-04-01T00:00:00",
    "Status": "A"
  },
  {
    "ProductName": "Milk",
    "Quantity": 4,
    "ExtendedPrice": 9.99999,
    "ShipperName": "ACME",
    "ShippedDate": "2015-02-18T00:00:00",
    "Status": "B"
  },
  {
    "ProductName": "Canned Beans",
    "Quantity": 3,
    "ExtendedPrice": 6.85000,
    "ShipperName": "ACME",
    "ShippedDate": "2015-03-02T00:00:00",
    "Status": "B"
  },
  {
    "ProductName": "Salad",
    "Quantity": 2,
    "ExtendedPrice": 8.8000,
    "ShipperName": "ACME",
    "ShippedDate": "2015-04-12T00:00:00",
    "Status": "C"
  },
  {
    "ProductName": "Bread",
    "Quantity": 1,
    "ExtendedPrice": 2.71212,
    "ShipperName": "Fun Inc.",
    "ShippedDate": "2015-01-27T00:00:00",
    "Status": "A"
  }
]
```

The `Invoices.json` file is similar to our previous file in the `webapp` folder. Just copy the content and remove the outer object structure with the key `invoices` so that the file consists of one flat array of invoice items. This file will automatically be read by our server later in this step.

Remove the old `Invoices.json` file from the `webapp` folder, it is no longer used.

webapp/localService/metadata.xml (New)

```
<edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/
edmx">
    <edmx:DataServices m:DataServiceVersion="1.0" m:MaxDataServiceVersion="3.0"
                       xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata">
        <Schema Namespace="NorthwindModel" xmlns="http://
schemas.microsoft.com/ado/2008/09/edm">
            <EntityType Name="Invoice">
                <Key>
                    <PropertyRef Name="ProductName"/>
                    <PropertyRef Name="Quantity"/>
                    <PropertyRef Name="ShipperName"/>
                </Key>
                <Property Name="ShipperName" Type="Edm.String" Nullable="false"
MaxLength="40" FixedLength="false"
                          Unicode="true"/>
                <Property Name="ProductName" Type="Edm.String" Nullable="false"
MaxLength="40" FixedLength="false"
                          Unicode="true"/>
                <Property Name="Quantity" Type="Edm.Int16" Nullable="false"/>
                <Property Name="ExtendedPrice" Type="Edm.Decimal" Precision="19"
Scale="4"/>
            </EntityType>
        </Schema>
        <Schema Namespace="ODataWebV2.Northwind.Model" xmlns="http://
schemas.microsoft.com/ado/2008/09/edm">
            <EntityContainer Name="NorthwindEntities"
m:IsDefaultEntityContainer="true" p6:LazyLoadingEnabled="true"
                             xmlns:p6="http://schemas.microsoft.com/ado/
2009/02/edm/annotation">
                <EntitySet Name="Invoices" EntityType="NorthwindModel.Invoice"/>
            </EntityContainer>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

The metadata file contains information about the service interface and does not need to be written manually. It can be accessed directly from the "real" service by calling the service URL and adding `$metadata` at the end (e.g. in our case `http://services.odata.org/V2/Northwind/Northwind.svc/$metadata`). The mock server will read this file to simulate the real OData service, and will return the results from our local source files in the proper format so that it can be consumed by the app (either in XML or in JSON format).

For simplicity, we have removed all content from the original Northwind OData metadata document that we do not need in our scenario. We have also added the `status` field to the metadata since it is not available in the real Northwind service.

webapp/localService/mockserver.js (New)

```
sap.ui.define([
    "sap/ui/core/util/MockServer",
    "sap/base/util/UriParameters"
], function (MockServer, UriParameters) {
    "use strict";

    return {
        init: function () {
            // create
            var oMockServer = new MockServer({
                rootUri: "https://services.odata.org/V2/Northwind/Northwind.svc/"
            });

            var oUriParameters = new UriParameters(window.location.href);

            // configure mock server with a delay
            MockServer.config({
                autoRespond: true,
                autoRespondAfter: oUriParameters.get("serverDelay") || 500
            });

            // simulate
            var sPath = "../localService";
            oMockServer.simulate(sPath + "/metadata.xml", sPath + "/mockdata");

            // start
            oMockServer.start();
        }
    };

});
```

Now that we have added the OData service description file `metadata.xml` file, we can write the code to initialize the mock server which will then simulate any OData request to the real Northwind server.

We load the standard SAPUI5 `MockServer` module as a dependency and create a helper object that defines an `init` method to start the server. This method is called before the component initialization in the `mockServer.html` file above. The `init` method creates a `MockServer` instance with the same URL as the real service calls.

The URL in configuration parameter `rootUri` has to be exactly the same as the `uri` that is defined for the data source in the `manifest.json` descriptor file. This can be an absolute or, for example in SAP Web IDE, a relative URL to a destination. The URL will now be served by our test server instead of the real service. Next, we set two global configuration settings that tell the server to respond automatically and introduce a delay of one second to imitate a typical server response time. Otherwise, we would have to call the respond method on the `MockServer` manually to simulate the call.

To simulate a service, we can simply call the `simulate` method on the `MockServer` instance with the path to our newly created `metadata.xml`. This will read the test data from our local file system and set up the URL patterns that will mimic the real service.

Finally, we call start on `oMockServer`. From this point, each request to the URL pattern `rootUri` will be processed by the `MockServer`. If you switch from the `index.html` file to the `mockServer.html` file in the browser, you can now see that the test data is displayed from the local sources again, but with a short delay. The delay can be specified with the URI parameter `serverDelay`, the default value is one second.

This approach is perfect for local testing, even without any network connection. This way your development does not depend on the availability of a remote server, i.e. to run your tests.

Try calling the app with the `index.html` file and the `mockServer.html` file to see the difference. If the real service connection cannot be made, for example when there is no network connection, you can always fall back to the local test page.

Conventions

- The `webapp/test` folder contains non-productive code only.
- Mock data and the script to start the `MockServer` are stored in the `webapp/localService` folder.
- The script to start the `MockServer` is called `mockserver.js`.

Related Information

Mock Server [page 1222]
API Reference: `sap.ui.core.util.MockServer`
Create a Northwind Destination [page 49]

Step 28: Unit Test with QUnit

Now that we have a test folder in the app, we can start to increase our test coverage.

Actually, every feature that we added to the app so far, would require a separate test case. We have totally neglected this so far, so let's add a simple unit test for our custom formatter function from Step 23. We will test if the long text for our status is correct by comparing it with the texts from our resource bundle.

> **i Note**
>
> In this tutorial, we focus on a simple use case for the test implementation. If you want to learn more about QUnit tests, have a look at our Testing [page 368] tutorial, especially Step 2: A First Unit Test [page 376].

Preview

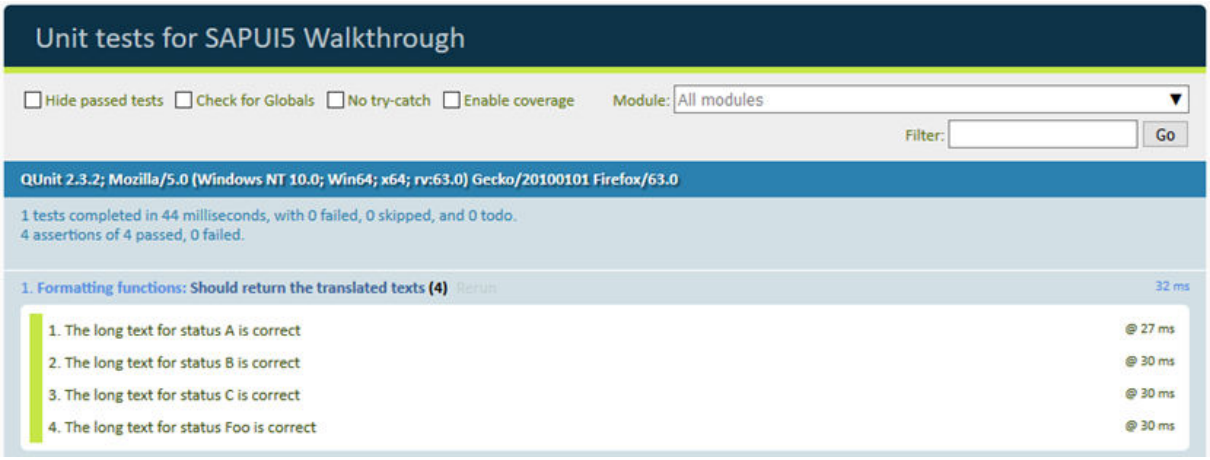


Figure 35: A unit test for our formatters is now available

Coding

You can view and download all files at Walkthrough - Step 28.

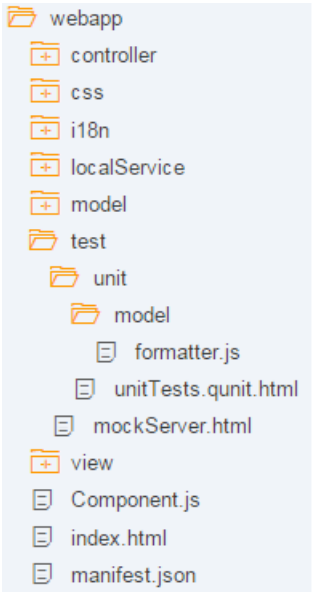


Figure 36: Folder Structure for this Step

We add a new folder `unit` under the `test` folder and a `model` subfolder where we will place our formatter unit test. The folder structure matches the app structure to easily find the corresponding unit tests.

webapp/test/unit/model/formatter.js

```
/*global QUnit*/
```

```
sap.ui.define([
    "sap/ui/demo/walkthrough/model/formatter",
    "sap/ui/model/resource/ResourceModel"
], function (formatter, ResourceModel) {
    "use strict";

    QUnit.module("Formatting functions", {
        beforeEach: function () {
            this._oResourceModel = new ResourceModel({
                bundleUrl: sap.ui.require.toUrl("sap/ui/demo/walkthrough") + "/
i18n/i18n.properties"
            });
        },
        afterEach: function () {
            this._oResourceModel.destroy();
        }
    });


    QUnit.test("Should return the translated texts", function (assert) {

        // Arrange
        // this.stub() does not support chaining and always returns the right
data
        // even if a wrong or empty parameter is passed.
        var oModel = this.stub();
        oModel.withArgs("i18n").returns(this._oResourceModel);
        var oViewStub = {
            getModel: oModel
        };
        var oControllerStub = {
            getView: this.stub().returns(oViewStub)
        };

        // System under test
        var fnIsolatedFormatter = formatter.statusText.bind(oControllerStub);

        // Assert
        assert.strictEqual(fnIsolatedFormatter("A"), "New", "The long text for
status A is correct");

        assert.strictEqual(fnIsolatedFormatter("B"), "In Progress", "The long
text for status B is correct");

        assert.strictEqual(fnIsolatedFormatter("C"), "Done", "The long text for
status C is correct");

        assert.strictEqual(fnIsolatedFormatter("Foo"), "Foo", "The long text for
status Foo is correct");
    });

});
```

We create a new `formatter.js` file under `webapp/test/unit/model` where the unit test for the custom formatter is implemented. The `formatter` file that we want to test is loaded as a dependency. We also need a dependency to the `ResourceModel`, because we want to check if the translated texts are correct.

The formatter file just contains one QUnit module for our formatter function. It instantiates our `ResourceBundle` with the localized texts in the `beforeEach` function and destroys it again in the `afterEach` function. These functions are called before and after each test is executed.

Next is our unit test for the formatter function. In the implementation of the `statusText` function that we created in step 23 we access the `ResourceBundle` with the following queued call: `var resourceBundle = this.getView().getModel("i18n").getResourceBundle();`.

Since we do not want to test the controller, the view, or the model functionality, we first remove the dependencies by replacing these calls with empty hulls with the help of `SinonJS` and its stub method. This happens in the `Arrange` section of the unit test. SinonJS injects a stub method for all objects so we can simply call `this.stub()` to create a new stub for any behavior we need to mock.

Test stubs are functions with pre-programmed behavior. They support the full SinonJS test spy API in addition to methods which can be used to alter the stub's behavior. If this part is a bit confusing have a look at the official SinonJS documentation for test spies or ignore it for now, it will become clear later on.

Then we bind our stub to the `statusText` formatter by calling the `bind` function of JavaScript. The `this` pointer is now bound to our controller stub when the function is invoked using the variable `fnIsolatedFormatter` and we can still pass in arguments as we like. This happens in the "system under test" part of the test.

Finally we perform our assertions. We check each branch of the formatter logic by invoking the isolated formatter function with the values that we expect in the data model (`A`, `B`, `C`, and everything else). We strictly compare the result of the formatter function with the hard-coded strings that we expect from the resource bundle and give a meaningful error message if the test should fail. We hard-code the strings here to identify issues with the resource bundle properties. If a property was missing, the test would still be successful if we check against the real value (that would be an empty string on both sides) from the resource bundle.

webapp/test/unit/unitTests.qunit.html (New)

```
<!DOCTYPE html>
<html>
<head>
    <title>Unit tests for SAPUI5 Walkthrough</title>
    <meta charset="utf-8">

    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "../../"
        }'
        data-sap-ui-async="true">
    </script>

    <link rel="stylesheet" type="text/css" href="https://
openui5.hana.ondemand.com/resources/sap/ui/thirdparty/qunit-2.css">

    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/thirdparty/
qunit-2.js"></script>
    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/qunit/qunit-
junit.js"></script>
    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/qunit/qunit-
coverage.js"></script>
    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/thirdparty/
sinon.js"></script>
    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/thirdparty/
sinon-qunit.js"></script>

    <script src="unitTests.qunit.js"></script>
</head>
<body>
    <div id="qunit"/>
    <div id="qunit-fixture"/>
```

```
</body>
</html>
```

The so-called QUnit test suite is an HTML page that triggers all QUnit tests for the application. Most of it is generating the layout of the result page that you can see in the preview and we won't further explain these parts but focus on the application parts instead.

Let's start with the namespaces. Since we are now in the `webapp/test/unit` folder, we actually need to go up two levels to get the `src` folder again. This namespace can be used inside the tests to load and trigger application functionality.

First, we load some basic QUnit functionality via script tags. Other QUnit tests can be added here as well. Then the HTML page loads another script called `unitTests.qunit.js`, which we will create next. This script will execute our formatter.

webapp/test/unit/unitTests.qunit.js (New)

```
/* global QUnit */
QUnit.config.autostart = false;
sap.ui.getCore().attachInit(function () {
    "use strict";
    sap.ui.require([
        "sap/ui/demo/walkthrough/test/unit/model/formatter"
    ], function () {
        QUnit.start();
    });
});
```

This script loads and executes our formatter. If we now open the `webapp/test/unit/unitTests.qunit.html` file in the browser, we should see our test running and verifying the formatter logic.

Conventions

- All unit tests are placed in the webapp/test/unit folder of the app.
- Files in the test suite end with `*.qunit.html`.
- The `unitTests.qunit.html` file triggers all unit tests of the app.
- A unit test should be written for formatters, controller logic, and other individual functionality.
- All dependencies are replaced by stubs to test only the functionality in scope.

Related Information

Unit Testing with QUnit [page 1159]
QUnit Home Page
Sinon.JS Home Page
Testing [page 368]

Step 29: Integration Test with OPA

If we want to test interaction patterns or more visual features of our app, we can also write an integration test.

We haven't thought about testing our interaction with the app yet, so in this step we will check if the dialog actually opens when we click the "Say Hello with Dialog" button. We can easily do this with OPA5, a feature of SAPUI5 that is easy to set up and is based on JavaScript and QUnit. Using integration and unit tests and running them consistently in a continuous integration (CI) environment, we can make sure that we don't accidentally break our app or introduce logical errors in existing code.

> **i Note**
>
> In this tutorial, we focus on a simple use case for the test implementation. If you want to learn more about OPA tests, have a look at our Testing [page 368] tutorial, especially Step 6: A First OPA Test [page 393].

Preview

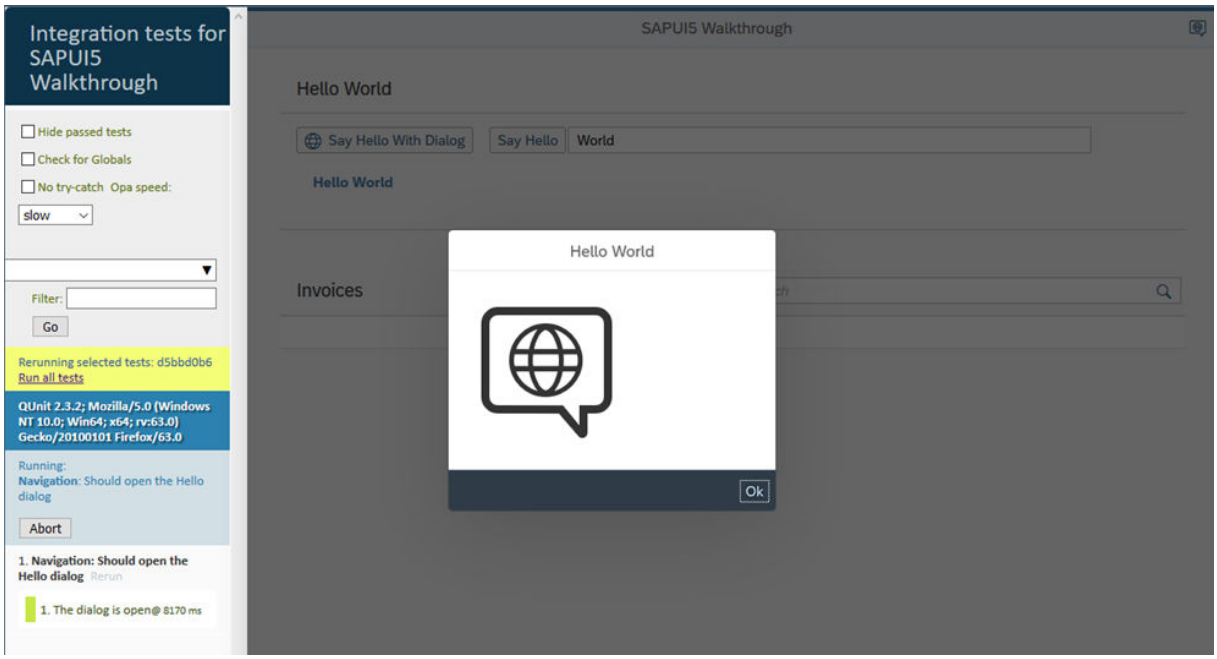


Figure 37: An OPA test opens the "Hello" dialog from step 16

Coding

You can view and download all files at Walkthrough - Step 29.

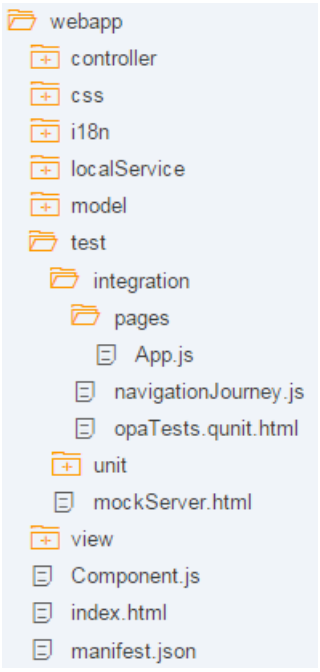


Figure 38: Folder Structure for this Step

We add a new folder `integration` below the `test` folder, where we put our new test cases. Page objects that help structuring such integration tests are put in the `pages` subfolder that we also create now.

webapp/test/integration/NavigationJourney.js (New)

```
/*global QUnit, opaTest*/

sap.ui.define([
    "sap/ui/demo/walkthrough/localService/mockserver",
    "sap/ui/test/opaQunit",
    "./pages/App"
], function (mockserver) {
    "use strict";

    QUnit.module("Navigation");

    opaTest("Should open the Hello dialog", function (Given, When, Then) {
        // initialize the mock server
        mockserver.init();

        // Arrangements
        Given.iStartMyUIComponent({
            componentConfig: {
                name: "sap.ui.demo.walkthrough"
            }
        });

        //Actions
        When.onTheAppPage.iPressTheSayHelloWithDialogButton();

        // Assertions
        Then.onTheAppPage.iShouldSeeTheHelloDialog();
```

```
        // Cleanup
        Then.iTeardownMyApp();
    });
});
```

Let's start with the `journey` first. A `journey` consists of a series of integration tests that belong to the same context such as navigating through the app. Similar to the QUnit test implementation, OPA5 uses QUnit, that's why we first set up a QUnit module `Navigation` that will be displayed on our result page.

The function `opaTest` is the main aspect for defining integration tests with OPA. Its parameters define a test name and a callback function that gets executed with the following OPA5 helper objects to write meaningful tests that read like a user story.

- `Given`
  On the given object we can call arrangement functions like `iStartMyUIComponent` to load our app component for integration testing.
- `When`
  Contains custom actions that we can execute to get the application in a state where we can test the expected behavior.
- `Then`
  Contains custom assertions that check a specific constellation in the application and the teardown function that removes our component again.

In our journey, we create a very simple test that starts the app. Inside the app, we simulate a click on a button and expect that the dialog is opened afterwards. Finally, we shut down the app again.

As you can see, the test case reads like a user story, we actually do not need the implementation of the methods yet to understand the meaning of the test case. This approach is called "Behavior Driven Development" or simply BDD and is popular in "Agile Software Development".

webapp/test/integration/pages/App.js (New)

```
sap.ui.define([
    "sap/ui/test/Opa5",
    "sap/ui/test/actions/Press"
], function (Opa5, Press) {
    "use strict";

    var sViewName = "sap.ui.demo.walkthrough.view.HelloPanel";

    Opa5.createPageObjects({
        onTheAppPage: {
            actions: {
                iPressTheSayHelloWithDialogButton: function () {
                    return this.waitFor({
                        id: "helloDialogButton",
                        viewName: sViewName,
                        actions: new Press(),
                        errorMessage: "Did not find the 'Say Hello With Dialog'
button on the HelloPanel view"
                    });
                }
            },

            assertions: {
                iShouldSeeTheHelloDialog: function () {
```

```
                    return this.waitFor({
                        controlType: "sap.m.Dialog",
                        success: function () {
                            // we set the view busy, so we need to query the
parent of the app
                            Opa5.assert.ok(true, "The dialog is open");
                        },
                        errorMessage: "Did not find the dialog control"
                    });
                }
            }
        }
    });
});
```

The implementation of the page object holds the helper functions we just called in our `journey`. We require OPA5 from the `sap.ui.test` namespace and define a page object with the helper function `createPageObjects`. We pass in an object with the key of our page `onTheAppPage` and two sections: `actions` and `assertions`.

In the actions section of the page object we define a function to click the "Hello" dialog button. This is done in OPA5 with a `waitFor` statement, it is basically a loop that checks for the conditions defined as parameters. If the conditions are met, the success callback is executed, if the test fails because the conditions have not been met, the text in the `errorMessage` property is displayed on the result page.

We define a `waitFor` statement that checks for controls of type `sap.m.Button`. As soon as a button is found on the app page the success handler is executed and we use jQuery to trigger a `tap` event on the first button that we found. This should open the `HelloDialog` similar to clicking on the button manually.

In the assertions section we define another `waitFor` statement that checks if a `sap.m.Dialog` control is existing in the DOM of the app. When the dialog has been found, the test is successful and we can immediately confirm by calling an `ok` statement with a meaningful message.

webapp/test/integration/opaTests.qunit.html (New)

```
<!DOCTYPE html>
<html>
<head>
    <title>Integration tests for SAPUI5 Walkthrough</title>
    <meta charset="utf-8">

    <script
        id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.walkthrough": "../../"
        }'
        data-sap-ui-animation="false"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>

    <link rel="stylesheet" type="text/css" href="https://
openui5.hana.ondemand.com/resources/sap/ui/thirdparty/qunit-2.css">

    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/thirdparty/
qunit-2.js"></script>
```

```
    <script src="https://openui5.hana.ondemand.com/resources/sap/ui/qunit/qunit-
junit.js"></script>

    <script src="opaTests.qunit.js"></script>
</head>
<body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
</body>
</html>
```

This file contains our test suite for all OPA tests of the app. We use the same namespace as for our application.

Then we load the basic QUnit functionality via script tags from SAPUI5 so that we can execute the test journey. The `NavigationJourney` we defined above will be loaded via a script called `opaTests.qunit.js`:

webapp/test/integration/opaTests.qunit.js (New)

```
/* global QUnit */
QUnit.config.autostart = false;
sap.ui.getCore().attachInit(function () {
    "use strict";
    sap.ui.require([
        "sap/ui/demo/walkthrough/test/integration/NavigationJourney"
    ], function () {
        QUnit.start();
    });
});
```

This script loads the `NavigationJourney`, and the test functions inside are immediately executed. When you call the `webapp/test/integration/opaTests.qunit.html` page of your project on the server, you should see the QUnit layout and a test “Should see the Hello dialog” is executed immediately. It will load the app component on the right side of the page. There you can see what operations the test is performing on the app, if everything works correctly the button click is triggered, then a dialog is shown and the test case is green.

Conventions

- OPA tests are located in the `webapp/test/integration` folder of the application.
- Use `page` objects and `journeys` for structuring OPA tests.

Related Information

Integration Testing with One Page Acceptance Tests (OPA5) [page 1182]
Samples: `sap.ui.test.Opa5`
Testing [page 368]

Step 30: Debugging Tools

Even though we have added a basic test coverage in the previous steps, it seems like we accidentally broke our app, because it does not display prices to our invoices anymore. We need to debug the issue and fix it before someone finds out.

Luckily, SAPUI5 provides a couple of debugging tools that we can use within the app to check the application logic and the developer tools of modern browsers are also quite good. We will now check for the root cause.

Preview

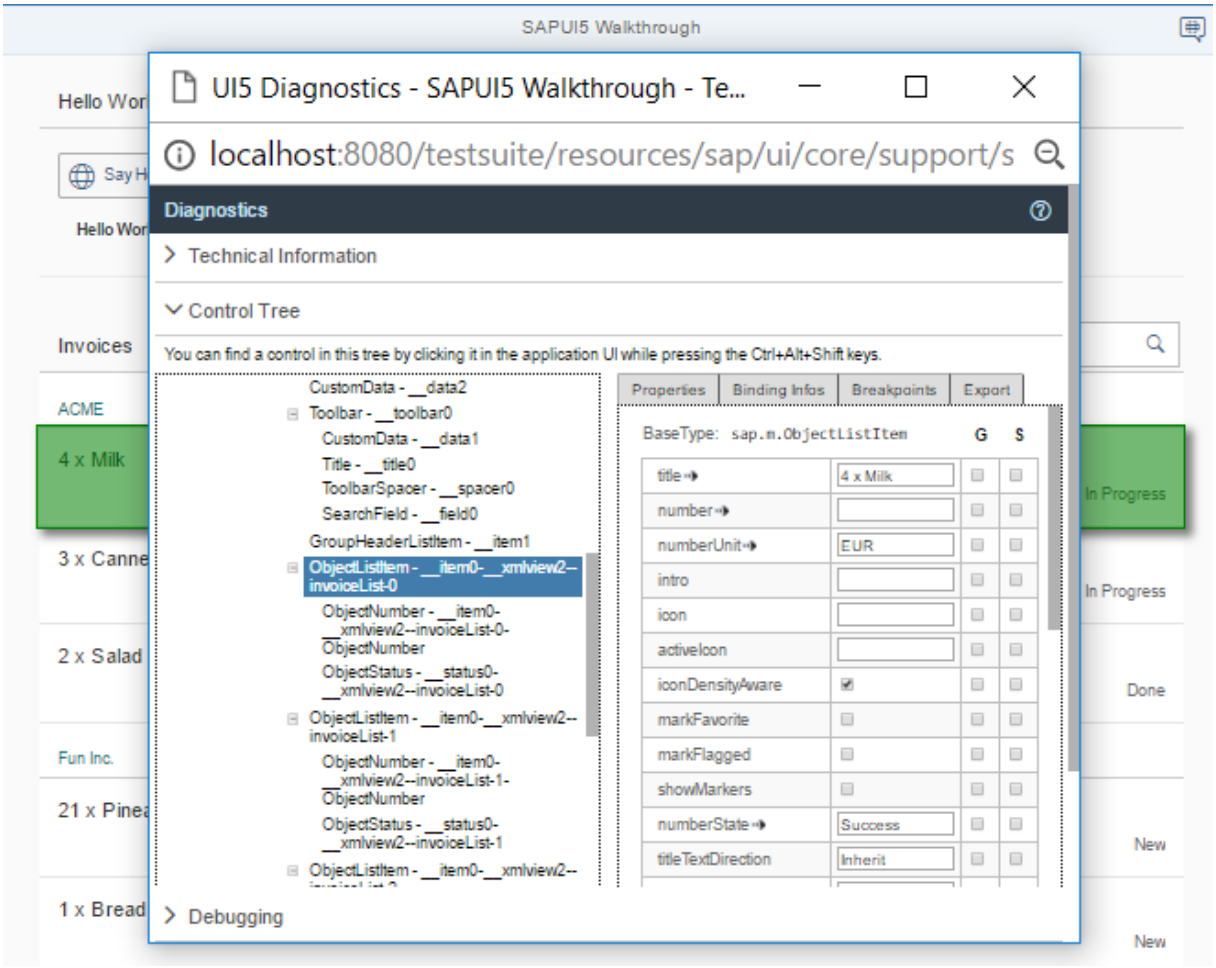


Figure 39: The diagnostics window

Coding

You can view and download all files at Walkthrough - Step 30.

webapp/view/InvoiceList.view.xml

```
<mvc:View
        controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc">
    <List
            id="invoiceList"
            class="sapUiResponsiveMargin"
            width="auto"
            items="{
            path : 'invoice>/Invoices',
            sorter : {
                path : 'ShipperName',
                group : true
            }
        }">
        <headerToolbar>
            <Toolbar>
                <Title text="{i18n>invoiceListTitle}"/>
                <ToolbarSpacer/>
                <SearchField width="50%" search=".onFilterInvoices"/>
            </Toolbar>
        </headerToolbar>
        <items>
            <ObjectListItem
                    title="{invoice>Quantity} x {invoice>ProductName}"
                    number="{
                    parts: [{path: 'invoice>ExTendedPrice'}, {path: 'view>/
currency'}],
                    type: 'sap.ui.model.type.Currency',
                    formatOptions: {
                        showMeasure: false
                    }
                }"
                numberUnit="{view>/currency}"
                    numberState="{=    ${invoice>ExtendedPrice} > 50 ? 'Error' :
'Success' }">
                <attributes>
                    <ObjectAttribute text="{
                        path: 'invoice>Status',
                        formatter: '.formatter.statusText'
                    }"/>
                </attributes>
            </ObjectListItem>
        </items>
    </List>
</mvc:View>
```

We introduced a typo in the binding of the number attribute to simulate a frequent error; instead of using `'invoice>ExtendedPrice'` we use `'invoice>ExTendedPrice'`. Now we call the app and notice that the price is actually missing. By pressing CTRL + ALT + SHIFT + S we open the SAPUI5 support diagnostics tool and check the app.

> **i Note**
>
> If you use the Google Chrome browser, you can install the *UI5 Inspector* plugin. With this plugin, you can easily debug your SAPUI5- or OpenUI5-based apps. For more information, see UI5 Inspector [page 1374].

Besides technical information about the app and a trace that is similar to the developer tools console of the browser, there is a really handy tool for checking such errors in this dialog. Open the tab *Control Tree* by clicking on the expand symbol on the right.

A hierarchical tree of SAPUI5 controls is shown on the left and the properties of the selected control are displayed on the right. If we now select the first `ObjectListItem` control of the tree and go to the *Binding Infos* tab on the right, we can actually see that the binding path of the number attribute is marked as invalid. We can now correct the error in the view and the price should appear in the list of invoices again.

Sometimes errors are not as easy to spot and you actually need to debug the JavaScript code with the tools of the browser. For performance reasons, the SAPUI5 files are shipped in a minified version, this means that all possible variable names are shortened and comments are removed.

This makes debugging harder because the code is a lot less readable. You can load the debug sources by adding the URL parameter `sap-ui-debug=true` or by pressing `CTRL` + `ALT` + `SHIFT` + `P` and select *Use Debug Sources* in the dialog box that is displayed. After reloading the page, you can see in the *Network* tab of the browser's developer tools that now a lot of files with the `-dbg` suffix are loaded. These are the source code files that include comments and the uncompressed code of the app and the SAPUI5 artifacts.

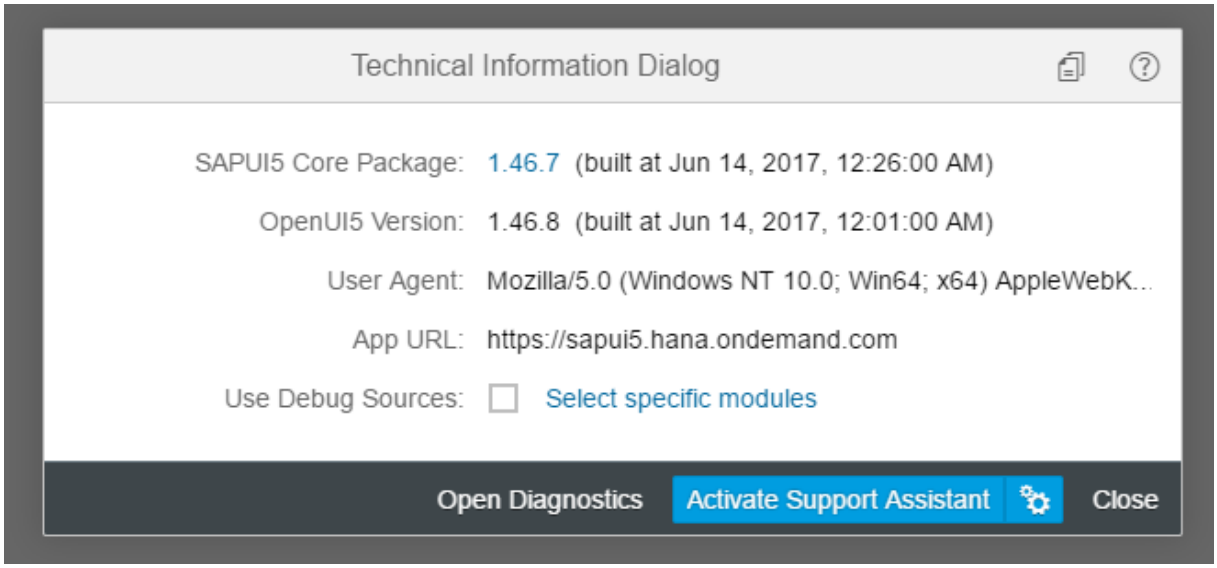


Figure 40: Technical information dialog

For a more detailed explanation of the SAPUI5 support tools, go through the Troubleshooting [page 194] tutorial.

If you're stuck and need help for some development task, you can also post a question in the SAPUI5-related forums, for example in the SAP Community or on Stack Overflow.

Conventions

- As per SAPUI5 convention uncompressed source files end with `*-dbg.js`

Related Information

Debugging [page 1315]
Diagnostics [page 1326]

Technical Information Dialog [page 1322]

Step 31: Routing and Navigation

So far, we have put all app content on one single page. As we add more and more features, we want to split the content and put it on separate pages.

In this step, we will use the SAPUI5 navigation features to load and show a separate detail page that we can later use to display details for an invoice. In the previous steps, we defined the page directly in the app view so that it is displayed when the app is loaded. We will now use the SAPUI5 router class to load the pages and update the URL for us automatically. We specify a routing configuration for our app and create a separate view for each page of the app, then we connect the views by triggering navigation events.

Preview

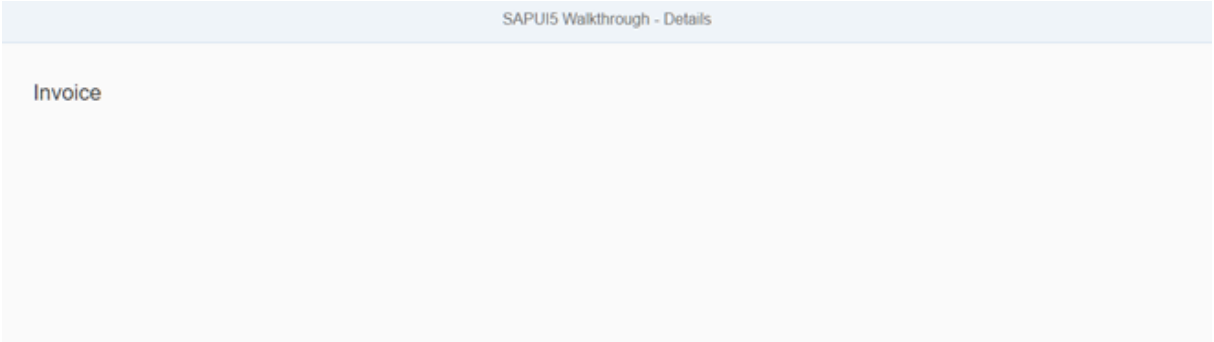


Figure 41: A second page is added to display the invoice

Coding

You can view and download all files at Walkthrough - Step 31.

webapp/manifest.json

```
{
  "_version": "1.12.0",
  …
  "sap.ui5": {
    …
    "models": {
        …
    },
    "routing": {
      "config": {
        "routerClass": "sap.m.routing.Router",
        "viewType": "XML",
```

```
        "viewPath": "sap.ui.demo.walkthrough.view",
        "controlId": "app",
        "controlAggregation": "pages",
        "async": true
      },
      "routes": [
        {
          "pattern": "",
          "name": "overview",
          "target": "overview"
        },
        {
          "pattern": "detail",
          "name": "detail",
          "target": "detail"
        }
      ],
      "targets": {
        "overview": {
          "viewId": "overview",
          "viewName": "Overview"
        },
        "detail": {
          "viewId": "detail",
          "viewName": "Detail"
        }
      }
    }
  }
```

We add a new “routing" section to the `sap.ui5` part of the descriptor. There are three subsections that define the routing and navigation structure of the app:

- `config`
  This section contains the global router configuration and default values that apply for all routes and targets. We define the router class that we want to use and where our views are located in the app. To load and display views automatically, we also specify which control is used to display the pages and what aggregation should be filled when a new page is displayed.
- `routes`
  Each route defines a name, a pattern, and one or more targets to navigate to when the route has been hit. The pattern is basically the URL part that matches to the route, we define two routes for our app. The first one is a default route that will show the overview page with the content from the previous steps, and the second is the detail route with the URL pattern `detail` that will show a new page.
- `targets`
  A target defines a view that is displayed, it is associated with one or more routes and it can also be displayed manually from within the app. Whenever a target is displayed, the corresponding view is loaded and shown in the app. In our app we simply define two targets with a view name that corresponds to the target name.

webapp/Component.js

```
sap.ui.define([
    "sap/ui/core/UIComponent",
    "sap/ui/model/json/JSONModel",
    "./controller/HelloDialog"
], function (UIComponent, JSONModel, HelloDialog) {
```

```
    "use strict";
    return UIComponent.extend("sap.ui.demo.walkthrough.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
            // set data model
            var oData = {
                recipient: {
                    name: "World"
                }
            };
            var oModel = new JSONModel(oData);
            this.setModel(oModel);
            // set dialog
            this._helloDialog = new HelloDialog(this.getRootControl());
            // create the views based on the url/hash
            this.getRouter().initialize();
        },
    });
        exit : function () {
            this._helloDialog.destroy();
            delete this._helloDialog;
        },
        openHelloDialog : function () {
            this._helloDialog.open();
        }
    });
});
```

In the component initialization method, we now add a call to initialize the router. We do not need to instantiate the router manually, it is automatically instantiated based on our `AppDescriptor` configuration and assigned to the component.

Initializing the router will evaluate the current URL and load the corresponding view automatically. This is done with the help of the routes and targets that have been configured in the `AppDescriptor`. If a route has been hit, the view of its corresponding target is loaded and displayed.

webapp/view/Overview.view.xml (New)

```
<mvc:View
        controllerName="sap.ui.demo.walkthrough.controller.App"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}">
        <headerContent>
            <Button
                    icon="sap-icon://hello-world"
                    press=".onOpenDialog"/>
        </headerContent>
        <content>
            <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
            <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.InvoiceList"/>
        </content>
    </Page>
</mvc:View>
```

We move the content of the previous steps from the `App` view to a new `Overview` view. For simplicity, we do not change the controller as it only contains our helper method to open the dialog, that means we reuse the

controller `sap.ui.demo.walkthrough.controller.App` for two different views (for the new overview and for the app view). However, two instances of that controller are instantiated at runtime. In general, one instance of a controller is instantiated for each view that references the controller.

webapp/view/App.view.xml

```
<mvc:View
        controllerName="sap.ui.demo.walkthrough.controller.App"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc"
        displayBlock="true">
<Shell>
    <App class="myAppDemoWT" id="app"/>
</Shell>
</mvc:View>
```

Our `App` view is now only containing the empty app tag. The router will automatically add the view that corresponds to the current URL into the app control. The router identifies the app control with the ID that corresponds to the property `controlId: "app"` in the `AppDescriptor`.

webapp/view/Detail.view.xml (New)

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        title="{i18n>detailPageTitle}">
        <ObjectHeader
            title="Invoice"/>
    </Page>
</mvc:View>
```

Now we add a second view for the detail view. It only contains a page and an `ObjectHeader` control that displays the static text *Invoice* for now.

webapp/i18n/i18n.properties

```
…
# Invoice List
invoiceListTitle=Invoices
invoiceStatusA=New
invoiceStatusB=In Progress
invoiceStatusC=Done
# Detail Page
detailPageTitle=Walkthrough - Details
```

We add a new string to the resource bundle for the detail page title.

webapp/view/InvoiceList.view.xml

```
<mvc:View
        controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc">
    <List    …>
        …
        <items>
            <ObjectListItem

                    title="{invoice>Quantity} x {invoice>ProductName}"
                    number="{
                    parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/
currency'}],
                    type: 'sap.ui.model.type.Currency',
                    formatOptions: {
                        showMeasure: false
                    }
                }"
                    numberUnit="{view>/currency}"
                    numberState="{=    ${invoice>ExtendedPrice} > 50 ? 'Error' :
'Success' }"
                    type="Navigation"
                    press="onPress">
                <firstStatus>
                    <ObjectStatus text="{
                        path: 'invoice>Status',
                        formatter: '.formatter.statusText'
                    }"/>
                </firstStatus>
            </ObjectListItem>
        </items>
    </List>
</mvc:View>
```

In the invoice list view we add a press event to the list item and set the item type to `Navigation` so that the item can actually be clicked.

webapp/controller/InvoiceList.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "../model/formatter",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator"
], function (Controller, JSONModel, formatter, Filter, FilterOperator) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {
        …
        onPress: function (oEvent) {
            var oRouter = sap.ui.core.UIComponent.getRouterFor(this);
            oRouter.navTo("detail");
        }
    });
});
```

We add the event handler function to the controller of our invoices list. Now it is time to navigate to the detail page by clicking an item in the invoice list. We access the router instance for our app by calling the helper

method `sap.ui.core.UIComponent.getRouterFor(this)`. On the router we call the `navTo` method to navigate to the `detail` route that we specified in the routing configuration.

You should now see the detail page when you click an item in the list of invoices.

Conventions

- Define the routing configuration in the descriptor

Related Information

Routing and Navigation [page 1072]
Tutorial: Navigation and Routing [page 291]
API Reference: `sap.m.routing.Router`
Samples: `sap.m.routing.Router`

Step 32: Routing with Parameters

We can now navigate between the overview and the detail page, but the actual item that we selected in the overview is not displayed on the detail page yet. A typical use case for our app is to show additional information for the selected item on the detail page.

To make this work, we have to pass over the information which item has been selected to the detail page and show the details for the item there.

Preview

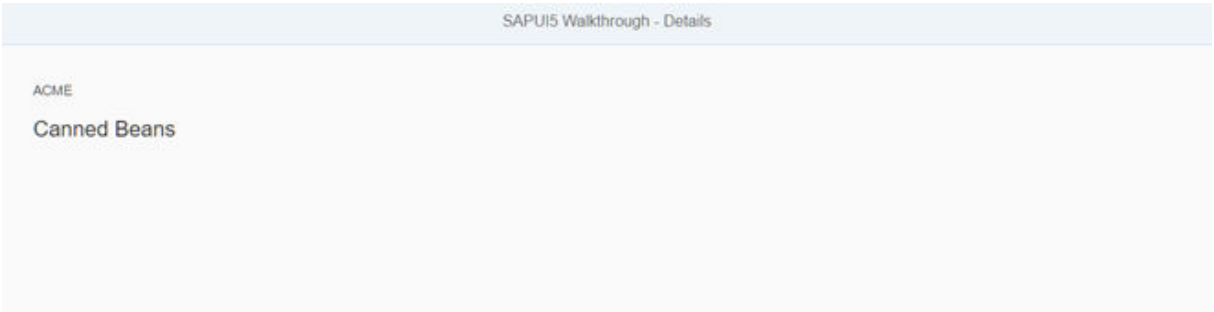


Figure 42: The selected invoice details are now shown in the details page

Coding

You can view and download all files at Walkthrough - Step 32.

webapp/manifest.json

```
{
  "_version": "1.12.0",
  …
  "sap.ui5": {
    …
    "routing": {
      "config": {
        "routerClass": "sap.m.routing.Router",
        "viewType": "XML",
        "viewPath": "sap.ui.demo.walkthrough.view",
        "controlId": "app",
        "controlAggregation": "pages",
        "async": true
      },
      "routes": [
        {
          "pattern": "",
          "name": "overview",
          "target": "overview"
        },
        {
          "pattern": "detail/{invoicePath}",
          "name": "detail",
          "target": "detail"
        }
      ],
          "targets": {
        "overview": {
          "viewID": "overview"
          "viewName": "Overview"
        },
        "detail": {
          "viewId": "detail"
          "viewName": "Detail"
        }
      }
    }
  }
}
```

We now add a navigation parameter `invoicePath` to the detail route so that we can hand over the information for the selected item to the detail page. Mandatory navigation parameters are defined with curly brackets.

webapp/view/Detail.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
```

```
    <Page
        title="{i18n>detailPageTitle}">
        <ObjectHeader
            intro="{invoice>ShipperName}"
            title="{invoice>ProductName}"/>
    </Page>
</mvc:View>
```

We add a controller that will take care of setting the item's context on the view and bind some properties of the `ObjectHeader` to the fields of our `invoice` model. We could add more detailed information from the `invoice` object here, but for simplicity reasons we just display two fields for now.

webapp/controller/InvoiceList.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "../model/formatter",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator"
], function (Controller, JSONModel, formatter, Filter, FilterOperator) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {
        …
        onPress: function (oEvent) {
            var oItem = oEvent.getSource();
            var oRouter = sap.ui.core.UIComponent.getRouterFor(this);
            oRouter.navTo("detail", {
                invoicePath:
window.encodeURIComponent(oItem.getBindingContext("invoice").getPath().substr(1))
            });
        }
    });
});
```

The control instance that has been interacted with can be accessed by the `getSource` method that is available for all SAPUI5 events. It will return the `ObjectListItem` that has been clicked in our case. We will use it to pass the information of the clicked item to the detail page so that the same item can be displayed there.

In the `navTo` method we now add a configuration object to fill the navigation parameter `invoicePath` with the current information of the item. This will update the URL and navigate to the detail view at the same time. On the detail page, we can access this `context` information again and display the corresponding item.

To identify the object that we selected, we would typically use the key of the item in the back-end system because it is short and precise. For our invoice items however, we do not have a simple key and directly use the binding path to keep the example short and simple. The path to the item is part of the binding context which is a helper object of SAPUI5 to manage the binding information for controls. The binding context can be accessed by calling the `getBindingContext` method with the model name on any bound SAPUI5 control. We need to remove the first `/` from the binding path by calling `.substr(1)` on the string because this is a special character in URLs and is not allowed, we will add it again on the detail page.

webapp/controller/Detail.controller.js (New)

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/UIComponent"
], function (Controller, UIComponent) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.Detail", {
        onInit: function () {
            var oRouter = sap.ui.core.UIComponent.getRouterFor(this);

oRouter.getRoute("detail").attachPatternMatched(this._onObjectMatched, this);
        },
        _onObjectMatched: function (oEvent) {
            this.getView().bindElement({
                path: "/" +
window.decodeURIComponent(oEvent.getParameter("arguments").invoicePath),
                model: "invoice"
            });
        }
    });
});
```

Our last piece to fit the puzzle together is the detail controller. It needs to set the context that we passed in with the URL parameter `invoicePath` on the view, so that the item that has been selected in the list of invoices is actually displayed, otherwise, the view would simply stay empty.

In the `onInit` method of the controller we fetch the instance of our app router and attach to the detail route by calling the method `attachPatternMatched` on the route that we accessed by its name. We register an internal callback function `_onObjectMatched` that will be executed when the route is hit, either by clicking on the item or by calling the app with a URL for the detail page.

In the `_onObjectMatched` method that is triggered by the router we receive an event that we can use to access the URL and navigation parameters. The `arguments` parameter will return an object that corresponds to our navigation parameters from the route pattern. We access the `invoicePath` that we set in the invoice list controller and call the `bindElement` function on the view to set the context. We have to add the root `/` in front of the path again that was removed for passing on the path as a URL parameter.

The `bindElement` function is creating a binding context for a SAPUI5 control and receives the model name as well as the path to an item in a configuration object. This will trigger an update of the UI controls that we connected with fields of the invoice model. You should now see the invoice details on a separate page when you click on an item in the list of invoices.

Conventions

- Define the routing configuration in the `AppDescriptor`

Related Information

Routing and Navigation [page 1072]

Tutorial: Navigation and Routing [page 291]
API Reference: `sap.m.routing.Router`
Samples: `sap.m.routing.Router`

Step 33: Routing Back and History

Now we can navigate to our detail page and display an invoice, but we cannot go back to the overview page yet. We'll add a back button to the detail page and implement a function that shows our overview page again.

Preview

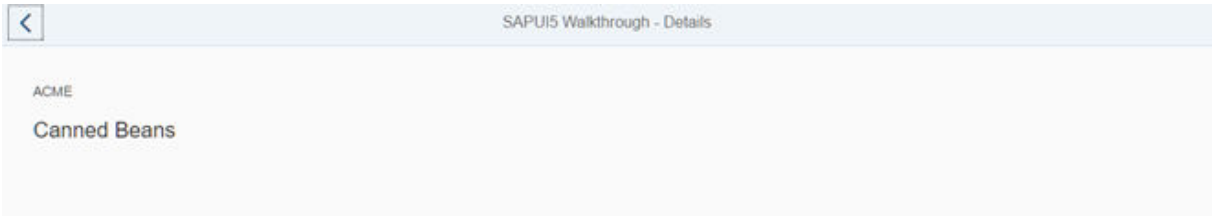


Figure 43: A back button is now displayed on the detail page

Coding

You can view and download all files at Walkthrough - Step 33.

webapp/view/Detail.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        title="{i18n>detailPageTitle}"
        showNavButton="true"
        navButtonPress=".onNavBack">
        <ObjectHeader
            intro="{invoice>ShipperName}"
            title="{invoice>ProductName}"/>
    </Page>
</mvc:View>
```

On the detail page, we tell the control to display a back button by setting the parameter `showNavButton` to true and register an event handler that is called when the back button is pressed.

webapp/controller/Detail.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/routing/History",
    "sap/ui/core/UIComponent"
], function (Controller, History, UIComponent) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.Detail", {
        onInit: function () {
            var oRouter = UIComponent.getRouterFor(this);

oRouter.getRoute("detail").attachPatternMatched(this._onObjectMatched, this);
        },
        _onObjectMatched: function (oEvent) {
            this.getView().bindElement({
                path: "/" +
window.decodeURIComponent(oEvent.getParameter("arguments").invoicePath),
                model: "invoice"
            });
        },

        onNavBack: function () {
            var oHistory = History.getInstance();
            var sPreviousHash = oHistory.getPreviousHash();

            if (sPreviousHash !== undefined) {
                window.history.go(-1);
            } else {
                var oRouter = UIComponent.getRouterFor(this);
                oRouter.navTo("overview", {}, true);
            }
        }
    });
});
```

We load a new dependency that helps us to manage the navigation history from the `sap.ui.core.routing` namespace and add the implementation for the event handler to our detail page controller.

In the event handler we access the navigation history and try to determine the previous hash. In contrast to the browser history, we will get a valid result only if a navigation step inside our app has already happened. Then we will simply use the browser history to go back to the previous page. If no navigation has happened before, we can tell the router to go to our overview page directly. The third parameter `true` tells the router to replace the current history state with the new one since we actually do a back navigation by ourselves. The second parameter is an empty array (`{}`) as we do not pass any additional parameters to this route.

This implementation is a bit better than the browser's back button for our use case. The browser would simply go back one step in the history even though we were on another page outside of the app. In the app, we always want to go back to the overview page even if we came from another link or opened the detail page directly with a bookmark. You can try it by loading the detail page in a new tab directly and clicking on the back button in the app, it will still go back to the overview page.

Conventions

- Add a path to go back to the parent page when the history state is unclear.

Step 34: Custom Controls

In this step, we are going to extend the functionality of SAPUI5 with a custom control. We want to rate the product shown on the detail page, so we create a composition of multiple standard controls using the SAPUI5 extension mechanism and add some glue code to make them work nicely together. This way, we can reuse the control across the app and keep all related functionality in one module.

Preview

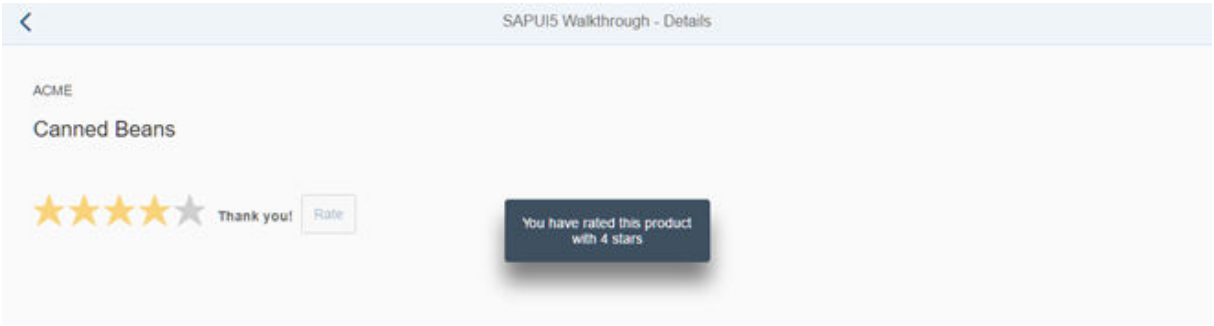


Figure 44: A custom product rating control is added to the detail page

Coding

You can view and download all files at Walkthrough - Step 34.

webapp/control/ProductRating.js (New)

```
sap.ui.define([
    "sap/ui/core/Control"
], function (Control) {
    "use strict";
    return Control.extend("sap.ui.demo.walkthrough.control.ProductRating", {
        metadata : {
        },
        init : function () {
        },
        renderer : function (oRM, oControl) {
        }
    });
});
```

We create a new folder `control` and a file `ProductRating.js` that will hold our new control. As with our controllers and views, the custom control inherits the common control functionality from a SAPUI5 base object, for controls this is done by extending the base class `sap.ui.core.Control`.

Custom controls are small reuse components that can be created within the app very easily. Due to their nature, they are sometimes also referred to as "notepad" or "on the fly" controls. A custom control is a

JavaScript object that has two special sections (`metadata` and `renderer`) and a number of methods that implement the functionality of the control.

The `metadata` section defines the data structure and thus the API of the control. With this meta information on the properties, events, and aggregations of the control SAPUI5 automatically creates setter and getter methods and other convenience functions that can be called within the app.

The renderer defines the HTML structure that will be added to the DOM tree of your app whenever the control is instantiated in a view. It is usually called initially by the core of SAPUI5 and whenever a property of the control is changed. The parameter `oRM` of the render function is the SAPUI5 render manager that can be used to write strings and control properties to the HTML page.

The `init` method is a special function that is called by the SAPUI5 core whenever the control is instantiated. It can be used to set up the control and prepare its content for display.

> **i** Note
>
> Controls always extend `sap.ui.core.Control` and render themselves. You could also extend `sap.ui.core.Element` or `sap.ui.base.ManagedObject` directly if you want to reuse life cycle features of SAPUI5 including data binding for objects that are not rendered. Please refer to the API reference to learn more about the inheritance hierarchy of controls.

webapp/control/ProductRating.js

```
sap.ui.define([
    "sap/ui/core/Control",
    "sap/m/RatingIndicator",
    "sap/m/Label",
    "sap/m/Button"
], function (Control, RatingIndicator, Label, Button) {
    "use strict";
    return Control.extend("sap.ui.demo.walkthrough.control.ProductRating", {
        metadata : {
            properties : {
                value:     {type : "float", defaultValue : 0}
            },
            aggregations : {
                _rating : {type : "sap.m.RatingIndicator", multiple: false,
visibility : "hidden"},
                _label : {type : "sap.m.Label", multiple: false, visibility :
"hidden"},
                _button : {type : "sap.m.Button", multiple: false, visibility :
"hidden"}
            },
            events : {
                change : {
                    parameters : {
                        value : {type : "int"}
                    }
                }
            }
        },
        init : function () {
            this.setAggregation("_rating", new RatingIndicator({
                value: this.getValue(),
                iconSize: "2rem",
                visualMode: "Half",
                liveChange: this._onRate.bind(this)
```

```
            }));
            this.setAggregation("_label", new Label({
                text: "{i18n>productRatingLabelInitial}"
            }).addStyleClass("sapUiSmallMargin"));
            this.setAggregation("_button", new Button({
                text: "{i18n>productRatingButton}",
                press: this._onSubmit.bind(this)
            }).addStyleClass("sapUiTinyMarginTopBottom"));
        },

        setValue: function (fValue) {
            this.setProperty("value", fValue, true);
            this.getAggregation("_rating").setValue(fValue);
        },

        reset: function () {
            var oResourceBundle = this.getModel("i18n").getResourceBundle();

            this.setValue(0);
            this.getAggregation("_label").setDesign("Standard");
            this.getAggregation("_rating").setEnabled(true);

this.getAggregation("_label").setText(oResourceBundle.getText("productRatingLabel
Initial"));
            this.getAggregation("_button").setEnabled(true);
        },

        _onRate : function (oEvent) {
            var oRessourceBundle = this.getModel("i18n").getResourceBundle();
            var fValue = oEvent.getParameter("value");

            this.setProperty("value", fValue, true);


this.getAggregation("_label").setText(oRessourceBundle.getText("productRatingLabe
lIndicator", [fValue, oEvent.getSource().getMaxValue()]));
            this.getAggregation("_label").setDesign("Bold");
        },

        _onSubmit : function (oEvent) {
            var oResourceBundle = this.getModel("i18n").getResourceBundle();

            this.getAggregation("_rating").setEnabled(false);

this.getAggregation("_label").setText(oResourceBundle.getText("productRatingLabel
Final"));
            this.getAggregation("_button").setEnabled(false);
            this.fireEvent("change", {
                value: this.getValue()
            });
        },
        renderer : function (oRm, oControl) {
            oRm.openStart("div", oControl);
            oRm.class("myAppDemoWTProductRating");
            oRm.openEnd();
            oRm.renderControl(oControl.getAggregation("_rating"));
            oRm.renderControl(oControl.getAggregation("_label"));
            oRm.renderControl(oControl.getAggregation("_button"));
            oRm.close("div");
        }
    });
});
```

We now enhance our new custom control with the custom functionality that we need. In our case we want to create an interactive product rating, so we define a value and use three internal controls that are displayed

updated by our control automatically. A `RatingIndicator` control is used to collect user input on the product, a label is displaying further information, and a button submits the rating to the app to store it.

In the `metadata` section we therefore define several properties that we make use in the implementation:

- Properties
  - Value
    We define a control property `value` that will hold the value that the user selected in the rating. Getter and setter function for this property will automatically be created and we can also bind it to a field of the data model in the XML view if we like.
- Aggregations
  As described in the first paragraph, we need three internal controls to realize our rating functionality. We therefore create three "hidden aggregations" by setting the `visibility` attribute to `hidden`. This way, we can use the models that are set on the view also in the inner controls and SAPUI5 will take care of the lifecycle management and destroy the controls when they are not needed anymore. Aggregations can also be used to hold arrays of controls but we just want a single control in each of the aggregations so we need to adjust the cardinality by setting the attribute `multiple` to `false`.
  - `_rating`: A `sap.m.RatingIndicator` control for user input
  - `_label`: A `sap.m.Label` to display additional information
  - `_button`: A `sap.m.Button` to submit the rating

  > **i** Note
  >
  > You can define `aggregations` and `associations` for controls. The difference is in the relation between the parent and the related control:
  > - An **aggregation** is a strong relation that also manages the lifecycle of the related control, for example, when the parent is destroyed, the related control is also destroyed. Also, a control can only be assigned to one single aggregation, if it is assigned to a second aggregation, it is removed from the previous aggregation automatically.
  > - An **association** is a weak relation that does not manage the lifecycle and can be defined multiple times. To have a clear distinction, an association only stores the ID, whereas an aggregation stores the direct reference to the control. We do not specify associations in this example, as we want to have our internal controls managed by the parent.
- Events
  - Change
    We specify a `change` event that the control will fire when the rating is submitted. It contains the current value as an event parameter. Applications can register to this event and process the result similar to "regular" SAPUI5 controls, which are in fact built similar to custom controls.

In the `init` function that is called by SAPUI5 automatically whenever a new instance of the control is instantiated, we set up our internal controls. We instantiate the three controls and store them in the internal aggregation by calling the framework method `setAggregation` that has been inherited from `sap.ui.core.Control`. We pass on the name of the internal aggregations that we specified above and the new control instances. We specify some control properties to make our custom control look nicer and register a `liveChange` event to the rating and a press event to the button. The initial texts for the label and the button are referenced from our `i18n` model.

Let's ignore the other internal helper functions and event handlers for now and define our renderer. With the help of the SAPUI5 render manager and the control instance that are passed on as a reference, we can now render the HTML structure of our control. We render the start of the outer `<div>` tag as `<div` and call the

helper method `writeControlData` to render the ID and other basic attributes of the control inside the `div` tag. Next, we add a custom CSS class so that we can define styling rules for the custom control in our CSS file later. This CSS class and others that have been added in the view are then rendered by calling `writeClasses` on the renderer instance. Then we close the surrounding `div` tag and render three internal controls by passing the content of the internal aggregation to the render managers `renderControl` function. This will call the renderer of the controls and add their HTML to the page. Finally, we close our surrounding `<div>` tag.

The `setValue` is an overridden setter. SAPUI5 will generate a setter that updates the property value when called in a controller or defined in the XML view, but we also need to update the internal rating control in the hidden aggregation to reflect the state properly. Also, we can skip the rerendering of SAPUI5 that is usually triggered when a property is changed on a control by calling the `setProperty` method to update the control property with true as the third parameter.

Now we define the event handler for the internal rating control. It is called every time the user changes the rating. The current value of the rating control can be read from the event parameter value of the `sap.m.RatingIndicator` control. With the value we call our overridden setter to update the control state, then we update the `label` next to the rating to show the user which value he has selected currently and also displays the maximum value. The string with the placeholder values is read from the `i18n` model that is assigned to the control automatically.

Next, we have the `press` handler for the rating button that submits our rating. We assume that rating a product is a one-time action and first disable the rating and the button so that the user is not allowed to submit another rating. We also update the label to show a "Thank you for your rating!" message, then we fire the change event of the control and pass in the current value as a parameter so that applications that are listening to this event can react on the rating interaction.

We define the `reset` method to be able to revert the state of the control on the UI to its initial state so that the user can again submit a rating.

webapp/view/Detail.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:wt="sap.ui.demo.walkthrough.control">
    <Page
        title="{i18n>detailPageTitle}"
        showNavButton="true"
        navButtonPress=".onNavBack">
        <ObjectHeader
            intro="{invoice>ShipperName}"
            title="{invoice>ProductName}"/>
        <wt:ProductRating id="rating" class="sapUiSmallMarginBeginEnd"
change=".onRatingChange"/>
    </Page>
</mvc:View>
```

A new namespace `wt` is defined on the detail view so that we can reference our custom controls easily in the view. We then add an instance of the `ProductRating` control to our detail page and register an event handler for the change event. To have a proper layout, we also add a margin style class.

webapp/controller/Detail.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/routing/History",
    "sap/m/MessageToast",
    "sap/ui/core/UIComponent"
], function (Controller, History, MessageToast, UIComponent) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.Detail", {
        …
        _onObjectMatched: function (oEvent) {
            this.byId("rating").reset();
            this.getView().bindElement({
                path: "/" +
window.decodeURIComponent(oEvent.getParameter("arguments").invoicePath),
                model: "invoice"
            });
        },
        onNavBack: function () {
            var oHistory = History.getInstance();
            var sPreviousHash = oHistory.getPreviousHash();
            if (sPreviousHash !== undefined) {
                window.history.go(-1);
            } else {
                var oRouter = UIComponent.getRouterFor(this);
                oRouter.navTo("overview", {}, true);
            }
        },

        onRatingChange: function (oEvent) {
            var fValue = oEvent.getParameter("value");
            var oResourceBundle =
this.getView().getModel("i18n").getResourceBundle();

            MessageToast.show(oResourceBundle.getText("ratingConfirmation",
[fValue]));
        }
    });
});
```

In the `Detail` controller we load the dependency to the `sap.m.MessageToast` because we will simply display a message instead of sending the rating to the backend to keep the example simple. The event handler `onRatingChange` reads the value of our custom change event that is fired when the rating has been submitted. We then display a confirmation message with the value in a `MessageToast` control.

In the `onObjectMatched` private method, we call the `reset` method to make it possible to submit another rating as soon as the detail view is displayed for a different item.

webapp/css/style.css

```
.myAppDemoWTmyCustomButton.sapMBtn {
    margin-right: 0.125rem;
}
.myAppDemoWTmyCustomText {
    font-weight: bold;
}
/*  ProductRating */
.myAppDemoWTProductRating {
```

```
    padding: 0.75rem;
}
.myAppDemoWTProductRating .sapMRI {
    vertical-align: initial;
}
```

To layout our control, we add a little padding to the root class to have some space around the three inner controls, and we override the alignment of the `RatingIndicator` control so that it is aligned in one line with the label and the button.

We could also do this with more HTML in the renderer but this is the simplest way and it will only be applied inside our custom control. However, please be aware that the custom control is in your app and might have to be adjusted when the inner controls change in future versions of SAPUI5.

webapp/i18n/i18n.properties

```
…
# Detail Page
detailPageTitle=Walkthrough - Details
ratingConfirmation=You have rated this product with {0} stars

# Product Rating
productRatingLabelInitial=Please rate this product
productRatingLabelIndicator=Your rating: {0} out of {1}
productRatingLabelFinal=Thank you for your rating!
productRatingButton=Rate
```

The resource bundle is extended with the confirmation message and the strings that we reference inside the custom control. We can now rate a product on the detail page with our brand new control.

Conventions

- Put custom controls in the `control` folder of your app.

Related Information

Developing Controls [page 2158]
Defining the Control Metadata [page 2188]
API Reference: `sap.m.RatingIndicator`
Samples: `sap.m.RatingIndicator`
API Reference: `sap.m.Label`
Samples: `sap.m.Label`
API Reference: `sap.m.Button`
Samples: `sap.m.Button`
API Reference: `sap.ui.core.Control`

API Reference: `sap.ui.core.Element`

API Reference: `sap.ui.base.ManagedObject`

Step 35: Responsiveness

In this step, we improve the responsiveness of our app. SAPUI5 applications can be run on phone, tablet, and desktop devices and we can configure the application to make best use of the screen estate for each scenario. Fortunately, SAPUI5 controls like the `sap.m.Table` already deliver a lot of features that we can use.

Preview

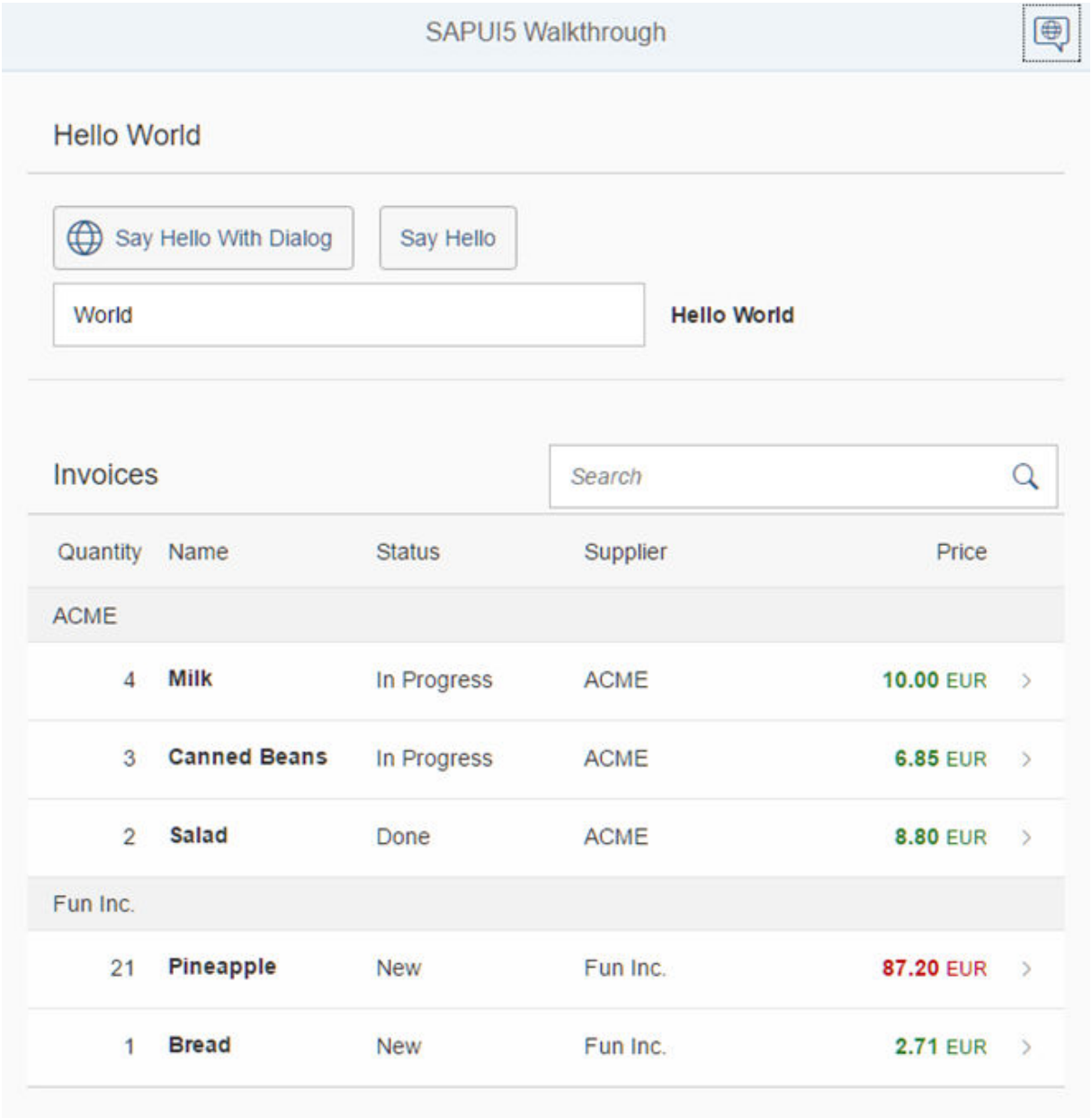


Figure 45: A responsive table is hiding some of the columns on small devices

Coding

You can view and download all files at Walkthrough - Step 35.

webapp/view/InvoiceList.view.xml

```
<mvc:View
        controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc">
    <Table
        id="invoiceList"
        class="sapUiResponsiveMargin"
        width="auto"
        items="{
            path : 'invoice>/Invoices',
            sorter : {
                path : 'ShipperName',
                group : true
            }
        }">
        <headerToolbar>
            <Toolbar>
                <Title text="{i18n>invoiceListTitle}"/>
                <ToolbarSpacer/>
                <SearchField width="50%" search=".onFilterInvoices"/>
            </Toolbar>
        </headerToolbar>
        <columns>
            <Column
                hAlign="End"
                minScreenWidth="Small"
                demandPopin="true"
                width="4em">
                <Text text="{i18n>columnQuantity}"/>
            </Column>
            <Column>
                <Text text="{i18n>columnName}"/>
            </Column>
            <Column
                minScreenWidth="Small"
                demandPopin="true">
                <Text text="{i18n>columnStatus}"/>
            </Column>
            <Column
                minScreenWidth="Tablet"
                demandPopin="false">
                <Text text="{i18n>columnSupplier}"/>
            </Column>
            <Column
                hAlign="End">
                <Text text="{i18n>columnPrice}"/>
            </Column>
        </columns>
        <items>
            <ColumnListItem
                type="Navigation"
                press=".onPress">
                <cells>
                    <ObjectNumber number="{invoice>Quantity}"
emphasized="false"/>
```

```
                    <ObjectIdentifier title="{invoice>ProductName}"/>
                    <Text text="{
                        path: 'invoice>Status',
                        formatter: '.formatter.statusText'
                    }"/>
                    <Text text="{invoice>ShipperName}"/>
                    <ObjectNumber
                        number="{
                            parts: [{path: 'invoice>ExtendedPrice'}, {path:
'view>/currency'}],
                            type: 'sap.ui.model.type.Currency',
                            formatOptions: {
                                showMeasure: false
                            }
                        }"
                        unit="{view>/currency}"
                        state="{= ${invoice>ExtendedPrice} > 50 ? 'Error' :
'Success' }"/>
                </cells>
            </ColumnListItem>
        </items>
    </Table>
</mvc:View>
```

We exchange the list with a table simply by replacing the tag `<List>` with `<Table>`. The table has a built-in responsiveness feature that allows us to make the app more flexible. The table and the list share the same set of properties so we can simply reuse these and also the sorter.

Since a table has multiple cells in each row, we have to define columns for our table and name these according to the data. We add five `sap.m.Column` controls to the column aggregation and configure each one a bit differently:

- **Quantity**
  This column will contain a short number, so we set the alignment to `End` (which means "right" in LTR languages) and the width to `4em` which is long enough for the column description. As a description text we use a `sap.m.Text` control that references a property of the resource bundle. We set the property `minScreenWidth` to `Small` to indicate that this column is not so important on phones. We will tell the table to display this column below the main column by setting the property `demandPopin` to `true`.
- **Name**
  Our main column that has a pretty large **width** to show all the details. It will always be displayed.
- **Status**
  The status is not so important, so we also display it below the `name` field on small screens by setting `minScreenWidth` to `small` and `demandPopin` to `true`
- **Supplier**
  We completely hide the `Supplier` column on phone devices by setting `minScreenWidth` to `Tablet` and `demandPopin` to `false`.
- **Price**
  This column is always visible as it contains our invoice price.

Instead of the `ObjectListItem` that we had before, we will now split the information onto the cells that match the columns defined above. Therefore we change it to a `ColumnListItem` control with the same attributes, but now with cells aggregation. Here we create five controls to display our data:

- **Quantity**
  A simple `sap.m.ObjectNumber` control that is bound to our data field.
- **Name**
  A `sap.m.ObjectIdentifier` controls that specifies the name.

- **Status**
  A `sap.m.Text control` with the same formatter as before.
- **Supplier**
  A simple `sap.m.Text` control.
- **Price**
  An `ObjectNumber` control with the same formatter as the attributes number and `numberUnit` from the previous steps.

Now we have defined our table responsively and can see the results when we decrease the browsers screen size. The *Supplier* column is not shown on phone sizes and the two columns *Quantity* and *Status* will be shown below the name.

webapp/i18n/i18n.properties

```
...
# Invoice List
invoiceListTitle=Invoices
invoiceStatusA=New
invoiceStatusB=In Progress
invoiceStatusC=Done
columnQuantity=Quantity
columnName=Name
columnSupplier=Supplier
columnStatus=Status
columnPrice=Price
# Detail Page
...
```

We add the column names and the attribute titles to our `i18n` file.

We can see the results when we decrease the browser's screen size or open the app on a small device.

Conventions

- Optimize your application for the different screen sizes of phone, tablet, and desktop devices.

Related Information

Configuring Responsive Behavior of a Table [page 2332]

Step 36: Device Adaptation

We now configure the visibility and properties of controls based on the device that we run the application on. By making use of the `sap.ui.Device` API and defining a device model we will make the app look great on many devices.

Preview

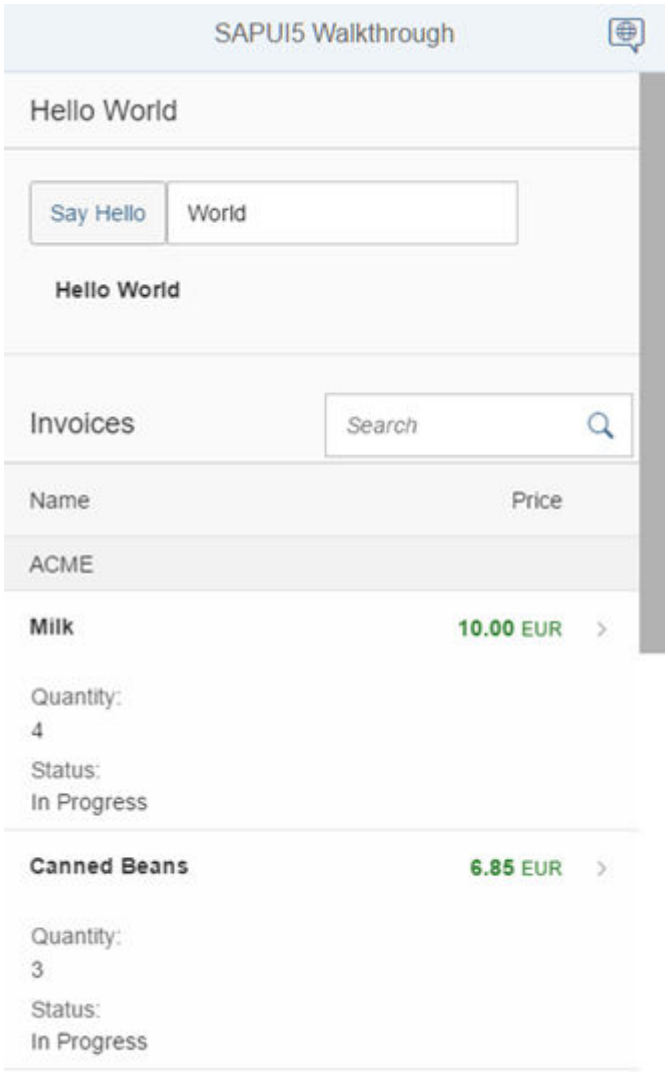


Figure 46: On phone devices, the panel is collapsed to save screen space and a button is hidden

Coding

You can view and download all files at Walkthrough - Step 36.

webapp/view/HelloPanel.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel
        headerText="{i18n>helloPanelTitle}"
        class="sapUiResponsiveMargin"
        width="auto"
        expandable="{device>/system/phone}"
        expanded="{= !${device>/system/phone} }">
        <content>
            <Button
                id="helloDialogButton"
                icon="sap-icon://world"
                text="{i18n>openDialogButtonText}"
                press=".onOpenDialog"
                class="sapUiSmallMarginEnd sapUiVisibleOnlyOnDesktop"/>
            <Button
                text="{i18n>showHelloButtonText}"
                press=".onShowHello"
                class="myCustomButton"/>
            <Input
                value="{/recipient/name}"
                valueLiveUpdate="true"
                width="60%"/>
            <FormattedText
                htmlText="Hello {/recipient/name}"
                class="sapUiSmallMargin sapThemeHighlight-asColor myCustomText"/>
        </content>
    </Panel>
</mvc:View>
```

We add two new properties `expandable` and `expanded` to the `HelloPanel`. The user can now close and open the panel to have more space for the table below on devices with small screens. The property `expandable` is bound to a model named `device` and the path `/system/phone`. So the panel can be expanded on phone devices only. The device model is filled with the `sap.ui.Device` API of SAPUI5 as we see further down. The `expanded` property controls the state of the panel and we use expression binding syntax to close it on phone devices and have the panel expanded on all other devices. The device API of SAPUI5 offers more functionality to detect various device-specific settings, please have a look at the documentation for more details.

> **i Note**
>
> The `sap.ui.Device` API detects the device type (Phone, Tablet, Desktop) based on the user agent and many other properties of the device. Therefore simply reducing the screen size will not change the device type. To test this feature, you will have to enable device emulation in your browser or open it on a real device.

We can also hide single controls by device type when we set a CSS class like `sapUiVisibleOnlyOnDesktop` or `sapUiHideOnDesktop` . We only show the button that opens the dialog on desktop devices and hide it for other devices. For more options, see the documentation linked below.

webapp/Component.js

```
sap.ui.define([
    "sap/ui/core/UIComponent",
    "sap/ui/model/json/JSONModel",
    "./controller/HelloDialog",
    "sap/ui/Device"
], function (UIComponent, JSONModel, HelloDialog, Device) {
    "use strict";
    return UIComponent.extend("sap.ui.demo.walkthrough.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
            // set data model
            var oData = {
                recipient: {
                    name: "World"
                }
            };
            var oModel = new JSONModel(oData);
            this.setModel(oModel);
            // disable batch grouping for v2 API of the northwind service
            this.getModel("invoice").setUseBatch(false);
            // set device model
            var oDeviceModel = new JSONModel(Device);
            oDeviceModel.setDefaultBindingMode("OneWay");
            this.setModel(oDeviceModel, "device");
            // set dialog
            this._helloDialog = new HelloDialog(this.getRootControl());
            // create the views based on the url/hash
            this.getRouter().initialize();
        },
        exit : function () {
            this._helloDialog.destroy();
            delete this._helloDialog;
        },
        openHelloDialog : function () {
            this._helloDialog.open();
        }
    });
});
```

In the `app` component we add a dependency to `sap.ui.Device` and initialize the device model in the `init` method. We can simply pass the loaded dependency `Device` to the constructor function of the JSONModel. This will make most properties of the SAPUI5 device API available as a JSON model. The model is then set on the component as a named model so that we can reference it in data binding as we have seen in the view above.

> **i Note**
>
> We have to set the binding mode to `OneWay` as the device model is read-only and we want to avoid changing the model accidentally when we bind properties of a control to it. By default, models in SAPUI5 are bidirectional (`TwoWay`). When the property changes, the bound model value is updated as well.

webapp/view/Detail.view.xml

> → Tip
>
> You can test the device specific features of your app with the developer tools of your browser. For example in Google Chrome, you can emulate a tablet or a phone easily and see the effects. Some responsive options of SAPUI5 are only set initially when loading the app, so you might have to reload your page to see the results.

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:wt="sap.ui.demo.walkthrough.control">
    <Page
        title="{i18n>detailPageTitle}"
        showNavButton="true"
        navButtonPress=".onNavBack">
        <ObjectHeader
            responsive="true"
            fullScreenOptimized="true"
            number="{
                parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/
currency'}],
                type: 'sap.ui.model.type.Currency',
                formatOptions: {
                    showMeasure: false
                }
            }"
            numberUnit="{view>/currency}"
            intro="{invoice>ShipperName}"
            title="{invoice>ProductName}">
            <attributes>
                <ObjectAttribute title="{i18n>quantityTitle}"
text="{invoice>Quantity}"></ObjectAttribute>
                <ObjectAttribute title="{i18n>dateTitle}" text="{
                    path: 'invoice>ShippedDate',
                    type: 'sap.ui.model.type.Date',
                    formatOptions: {
                      style: 'long',
                      source: {
                        pattern: 'yyyy-MM-ddTHH:mm:ss'
                      }
                    }
                  }"/>
            </attributes>
        </ObjectHeader>
        <wt:ProductRating id="rating" class="sapUiSmallMarginBeginEnd"
change=".onRatingChange"/>
    </Page>
</mvc:View>
```

Some controls already have built-in responsive features that can be configured. The `ObjectHeader` control can be put in a more flexible mode by setting the attribute `responsive` to `true` and `fullScreenOptimized` to true as well. This will show the data that we add to the view now at different positions on the screen based on the device size.

We add the `number` and `numberUnit` field from the list of the previous steps also to the `ObjectHeader` and use the same formatter with the `currency` type as in the previous steps. We then define two attributes: The quantity of the invoice and the shipped date which is part of the data model. We have not used this `shippedDate` field from the invoices JSON file so far, it contains a date in typical string format.

We now use the `Date` type and provide the pattern of our date format in the source section of the format options. It will display a more human-readable formatted date text that also fits to small screen devices.

webapp/controller/Detail.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/routing/History",
    "sap/m/MessageToast",
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/UIComponent"
], function (Controller, History, MessageToast, JSONModel, UIComponent) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.Detail", {
        onInit : function () {
            var oViewModel = new JSONModel({
                currency: "EUR"
            });
            this.getView().setModel(oViewModel, "view");
            var oRouter = sap.ui.core.UIComponent.getRouterFor(this);

oRouter.getRoute("detail").attachPatternMatched(this._onObjectMatched, this);
        },
        _onObjectMatched : …
});
```

In the `Detail` controller we simply add the view model with our currency definition to display the number properly. It is the same code as in the `InvoiceList` controller file.

webapp/i18n/i18n.properties

```
# Detail Page
detailPageTitle=Walkthrough - Details
ratingConfirmation=You have rated this product with {0} stars
dateTitle=Order date
quantityTitle=Quantity
```

We add the column names and the attribute titles to our `i18n` file.

We can see the results when we decrease the browser's screen size or open the app on a small device.

Conventions

Optimize your application for the different screen sizes of phone, tablet, and desktop devices.

Related Information

API Reference: `sap.ui.Device.media.RANGESETS`
API Reference: `sap.ui.Device`

Step 37: Content Density

In this step of our Walkthrough tutorial, we adjust the content density based on the user's device. SAPUI5 contains different content densities allowing you to display larger controls for touch-enabled devices and a smaller, more compact design for devices that are operated by mouse. In our app, we will detect the device and adjust the density accordingly.

Preview

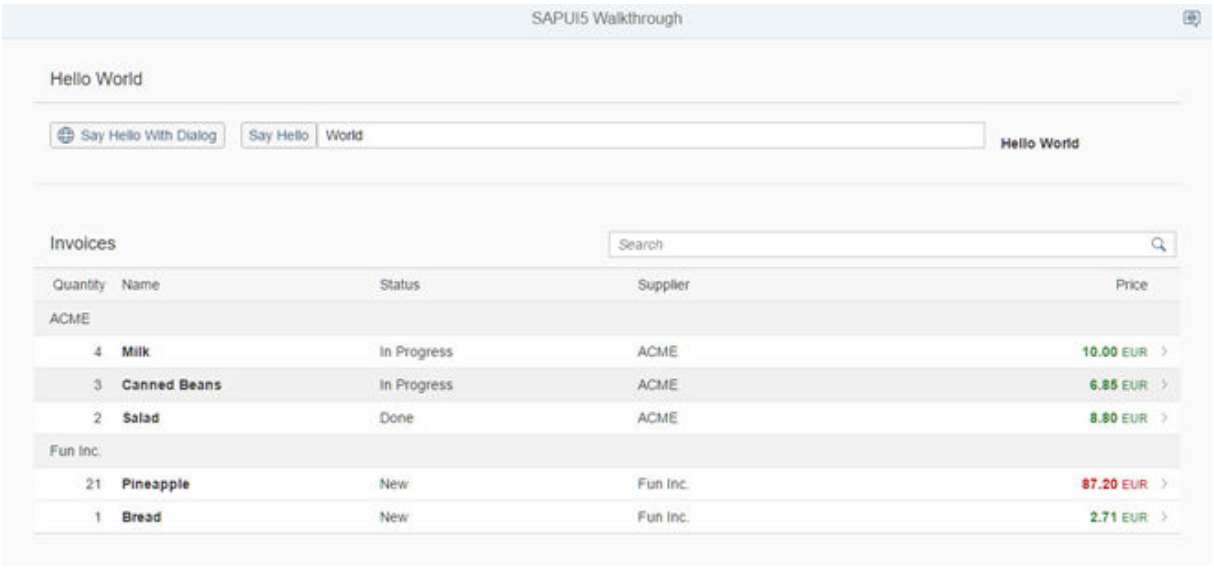


Figure 47: The content density is compact on desktop devices and cozy on touch-enabled devices

Coding

You can view and download all files at Walkthrough - Step 37.

webapp/Component.js

```
...
        init: function () {
...         },
```

```
...
        getContentDensityClass : function () {
            if (!this._sContentDensityClass) {
                if (!Device.support.touch) {
                    this._sContentDensityClass = "sapUiSizeCompact";
                } else {
                    this._sContentDensityClass = "sapUiSizeCozy";
                }
            }
            return this._sContentDensityClass;
        }
    });
});
```

To prepare the content density feature we will also add a helper method `getContentDensityClass`. SAPUI5 controls can be displayed in multiple sizes, for example in a `compact` size that is optimized for desktop and non-touch devices, and in a `cozy` mode that is optimized for touch interaction. The controls look for a specific CSS class in the HTML structure of the application to adjust their size.

This helper method queries the `Device` API directly for touch support of the client and returns the CSS class `sapUiSizeCompact` if touch interaction is not supported and `sapUiSizeCozy` for all other cases. We will use it throughout the application coding to set the proper content density CSS class.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function (Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
        onInit: function () {

this.getView().addStyleClass(this.getOwnerComponent().getContentDensityClass());
        },
        onOpenDialog: function () {
            this.getOwnerComponent().openHelloDialog();
        }
    });
});
```

We add a method `onInit` on the app controller that is called when the app view is instantiated. There we query the helper function that we defined on the app component to set the corresponding style class on the app view, All controls inside the app view will now automatically adjust either to the compact or cozy size as defined by the style.

webapp/controller/HelloDialog.js

```
sap.ui.define([
    "sap/ui/base/ManagedObject",
    "sap/ui/core/Fragment",
    "sap/ui/core/syncStyleClass"
], function (ManagedObject, Fragment, syncStyleClass) {
    "use strict";
```

```
    return
ManagedObject.extend("sap.ui.demo.walkthrough.controller.HelloDialog", {
        constructor : function (oView) {
            this._oView = oView;
        },
        exit : function () {
            delete this._oView;
        },
        open : function () {
            var oView = this._oView;
            // create dialog lazily
            if (!oView.byId("helloDialog")) {
                var oFragmentController = {
                    onCloseDialog : function () {
                        oView.byId("helloDialog").close();
                    }
                };
                // load asynchronous XML fragment
                Fragment.load({
                    id: oView.getId(),
                    name: "sap.ui.demo.walkthrough.view.HelloDialog",
                    controller: oFragmentController
                }).then(function (oDialog){
                    // connect dialog to the root view of this component
(models, lifecycle)
                    oView.addDependent(oDialog);
                    // forward compact/cozy style into dialog

syncStyleClass(oView.getController().getOwnerComponent().getContentDensityClass()
, oView, oDialog);
                    oDialog.open();
                });
            } else {
                oView.byId("helloDialog").open();
            }
        }
    });
});
```

The "Hello World" dialog is not part of the `app` view but opened in a special part of the DOM called "static area". The content density class defined on the `app` view is not known to the dialog so we sync the style class of the app with the dialog manually.

webapp/manifest.json

```
...
  "sap.ui5": {
    ...
    "dependencies": {
      ...
    },
    "contentDensities": {
      "compact": true,
      "cozy": true
    }
  }
```

In the `contentDensities` section of the `sap.ui5` namespace, we specify the modes that the application supports. Containers like the SAP Fiori launchpad allow switching the content density based on these settings.

As we have just enabled the app to run in both modes depending on the devices capabilities, we can set both to `true` in the application descriptor.

Summary

You should now be familiar with the major development paradigms and concepts of SAPUI5 and have created a very simple first app. You are now ready to build a proper app based on what you've learned.

If you want to dive deeper into specific topics, you can use the other tutorials that show some of the aspects of this Walkthrough and advanced topics in more detail.

Related Information

Content Densities [page 1142]
API Reference: `sap.ui.Device.media.RANGESETS`
API Reference: `sap.ui.Device`
API Reference: `jQuery.sap.syncStyleClass`

Step 38: Accessibility

As the last step in this tutorial, we are going to improve the accessibility of our app.

To achieve this, we will add ARIA attributes. ARIA attributes are used by screen readers to recognize the application structure and to interpret UI elements properly. That way, we can make our app more accessible for users who are limited in their use of computers, for example visually impaired persons. The main goal here is to make our app usable for as many people as we can.

> → Tip
>
> ARIA is short for **Accessible Rich Internet Applications**. It is a set of attributes that enable us to make apps more accessible by assigning semantic characteristics to certain elements. For more information, see Accessible Rich Internet Applications (ARIA) – Part 1: Introduction.

Preview

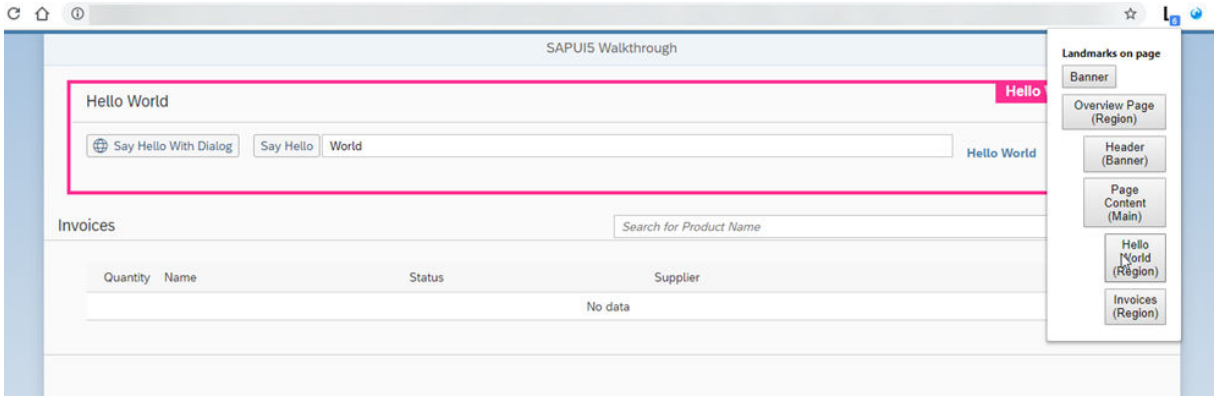


Figure 48: Landmarks in our app

Coding

You can view and download all files at Walkthrough - Step 38.

One part of the ARIA attribute set is the so-called landmarks. You can compare landmarks to maps in that they help the user navigate through an app. For this step, we will use Google Chrome with a free landmark navigation extension We will now add meaningful landmarks to our code.

webapp/view/Overview.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}">
        <landmarkInfo>
            <PageAccessibleLandmarkInfo
                rootRole="Region"
                rootLabel="{i18n>Overview_rootLabel}"
                contentRole="Main"
                contentLabel="{i18n>Overview_contentLabel}"
                headerRole="Banner"
                headerLabel="{i18n>Overview_headerLabel}"/>
        </landmarkInfo>
        <headerContent>
            ...
        </headerContent>
        <content>
            ...
        </content>
    </Page>
</mvc:View>
```

We use `sap.m.PageAccessibleLandmarkInfo` to define ARIA roles and labels for the overview page areas. For more information, see the API Reference: `sap.m.PageAccessibleLandmarkInfo`.

webapp/view/InvoiceList.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel accessibleRole="Region">
        <headerToolbar>
            <Toolbar>
                <Title text="{i18n>invoiceListTitle}"/>
                <ToolbarSpacer/>
                <SearchField
                    width="50%"
                    search=".onFilterInvoices"
                    ariaLabelledBy="searchFieldLabel"
                    ariaDescribedBy="searchFieldDescription"
                    placeholder="{i18n>searchFieldPlaceholder}"/>
            </Toolbar>
        </headerToolbar>
        <Table
            id="invoiceList"
            class="sapUiResponsiveMargin"
            width="auto"
            items="{
                path : 'invoice>/Invoices',
                sorter : {
                    path : 'ShipperName',
                    group : true
                }
            }">
            <columns>
                <Column
                    hAlign="End"
    …
            </columns>
        </Table>
    </Panel>
</mvc:View>
```

We add a `sap.m.Panel` around the invoice list, and we move the toolbar from the table into the panel so the region can take the title of the toolbar as its own. This has the effect that it will now be a region in our landmarks.

webapp/view/HelloPanel.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel
        headerText="{i18n>helloPanelTitle}"
        class="sapUiResponsiveMargin"
        width="auto"
        expandable="{device>/system/phone}"
        expanded="{= !${device>/system/phone} }"
        accessibleRole="Region">
        …
    </Panel>
</mvc:View>
```

In this view, we already have a panel, so we just add the `accessibleRole` attribute.

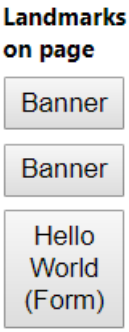
> **i** Note
>
> To add ARIA roles, labels and panels to other views, for example your `Detail.view.xml`, you can follow the same pattern. We won't go into detail in this tutorial step, but if you're interested, simply download the tutorial code and take a look at the `Detail.view.xml`.

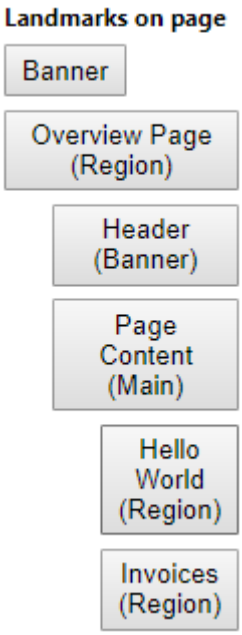
webapp/i18n/i18n.properties

```
...
#Overview Page
Overview_rootLabel=Overview Page
Overview_headerLabel=Header
Overview_contentLabel=Page Content
ratingTitle=Rate the Product
...
```

Here, we add the text for the rating panel title and the labels for the ARIA regions to the text bundle.

Result





Landmarks on the overview page - before	Landmarks on the overview page - after

As you can see, we now have four landmarks on our page. The top three landmarks structure our page:

- *Overview Page* marks the complete page.
- *Header* marks the page title.
- *Page Content* marks the content of our page. This landmark already has two children.

Congratulations!

You've completed the walkthrough, good job! You should be familiar with all major development paradigms of SAPUI5 now. Our other tutorials focus on certain aspects of SAPUI5, so feel free to explore!

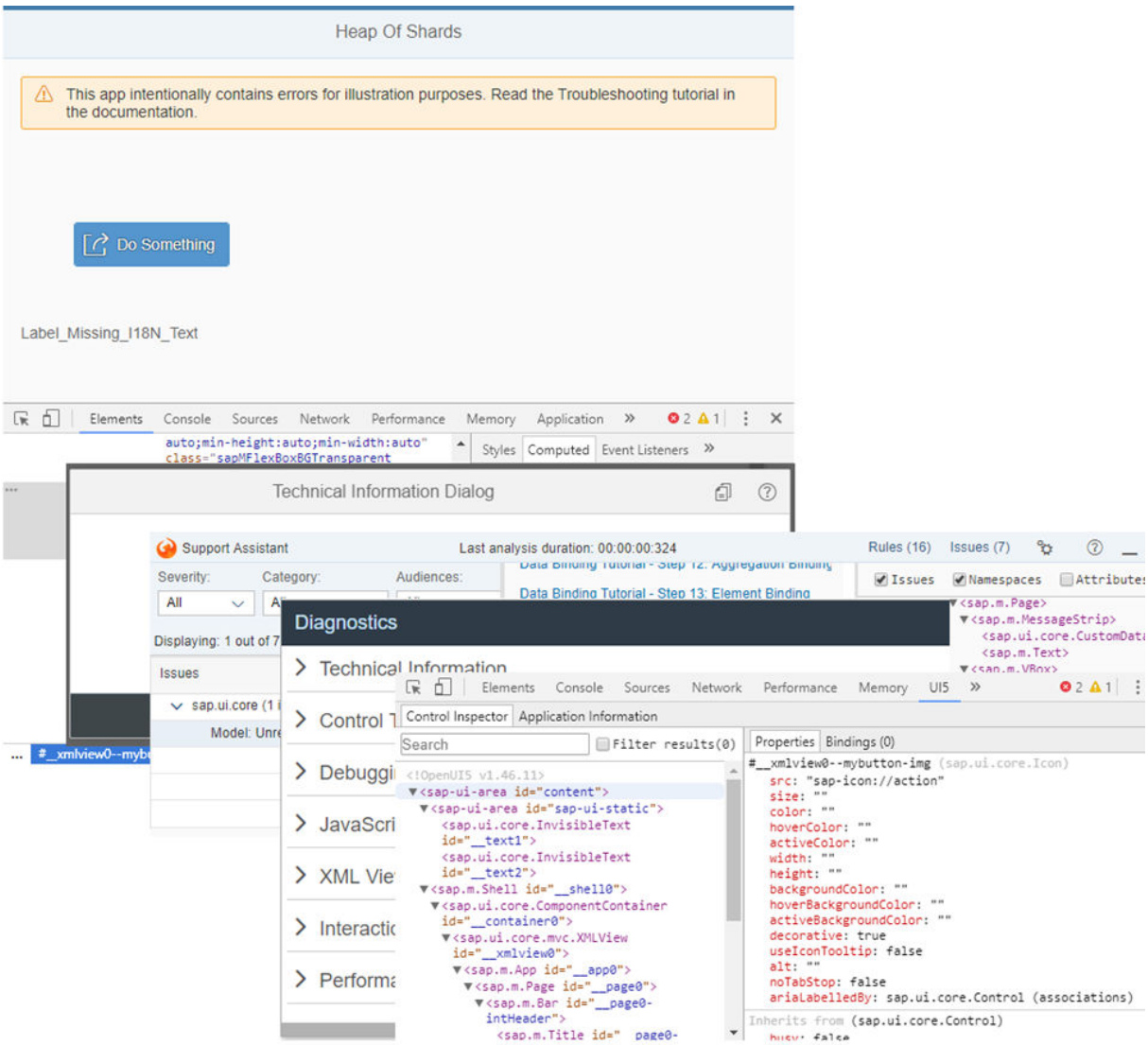
Related Information

Accessibility [page 1485]
Screen Reader Support for SAPUI5 Controls [page 2244]
SAP Software Accessibility

Troubleshooting

In this tutorial, we will show you some tools that will help you if you run into problems with your SAPUI5 app.

We will introduce you to the browser developer tools and show you the various tools that SAPUI5 offers.



For example, the SAPUI5 tools can help you with the following tasks:

- Inspect and debug apps
- Examine bugs and analyze errors
- Simulate UI changes
- Find out how to improve performance

To help you practice using the tools, we created an app with errors that we will use throughout the tutorial. You can view and download the app in the Demo Kit at Troubleshooting.

Get Help

If you're stuck and need help with a development task, you can also post a question in the SAPUI5-related forums, for example in the SAP Community or on Stack Overflow.

Step 1: Browser Developer Tools

In this step, you will learn how to use your browser's developers tools to troubleshoot your SAPUI5 app.

Most modern web browsers contain some form of Developer Tools. They allow you to examine the details of the current web page. You can also use them to debug JavaScript code, analyze network performance, live-edit DOM elements, and much more. As an example, we will show you how to use the *Developer Tools* in Google Chrome. Other browsers have similar capabilities, and you can easily adapt the examples shown here to these browsers.

Opening the Example App and the Developer Tools

1. Download the example app with errors from the Demo Kit at Troubleshooting and run the app.

   > **i Note**
   >
   > If you run the app within the Demo Kit frame, this step will not work as described. Open the app in a new tab first with  .

2. Open the *Developer Tools* by pressing `F12`.

Inspecting DOM Elements and CSS Styles in the *Elements* Tab

1. Activate the *Inspect Element* mode by pressing `Ctrl` + `Shift` + `C`.
2. Click the *Do Something* button in the app.
   The DOM tree in the *Elements* tab highlights the button's DOM element. Depending on which part of the button (icon or text) you clicked, different HTML tags are highlighted.
3. Search for the following line:

   ```
   <button id="container-HeapOfShards---app--myButton" data-sap-ui="container-
   HeapOfShards---app--myButton" aria-describedby="__text1"
       class="sapMBtn sapMBtnBase sapMBtnInverted">
   </button>
   ```

   The *Styles* section in the panel on the right shows the active and overruled (striked-through) CSS styles for the DOM element that is currently selected.
4. In the *Styles* section, switch to the *Computed* tab.
   You can see that the `margin` of the button is set to `0px`.
5. In the context menu of the element, choose *Edit as HTML* and add `sapUiLargeMargin` to the `class` section of the `button` tag.
   You can immediately see the effect on the web page.
   The edited element should now look like this:

   ```
   <button id="container-HeapOfShards---app--myButton" data-sap-ui="container-
   HeapOfShards---app--myButton" aria-describedby="__text1"
       class="sapMBtn sapMBtnBase sapMBtnInverted sapUiLargeMargin">
   ```

```
</button>
```

6. In the *Styles* section, switch again to the *Computed* tab.
   You can see that the `margin` of the button is now set to `48px`.

Analyzing Messages in the *Console* Tab

Interacting with the document

1. Switch to the *Console* tab and enter `$("#container-HeapOfShards---app--myButton")`
   The console displays the DOM element structure of the button:

```
Q.fn.init [button#container-HeapOfShards---app--
myButton.sapMBtn.sapMBtnBase.sapMBtnInverted,
context: document, selector: "#container-HeapOfShards---app--myButton"]
```

2. Examine the `button` element by expanding the structure.
3. On the *Console* tab, enter `myView=sap.ui.getCore().byId("container-HeapOfShards---app").`
4. On the *Console* tab, enter `myView.byId("myButton").`
   The console displays the SAPUI5 structure of the button control:

```
f {bAllowTextSelection: true, mEventRegistry: {…},
sId: "container-HeapOfShards---app--myButton", mProperties: PropertyBag,
mAggregations: {…}, …}
```

   Examine the SAPUI5 structure by expanding it.

> **i Note**
>
> The method of retrieval is different for SAPUI5 controls and DOM elements:

SAPUI5 Control	DOM Element
`sap.ui.getCore().byId("container-HeapOfShards---app--myButton")`	`jQuery("#container-HeapOfShards---app--myButton")`

Watching messages

1. Switch to the *Console* tab.
2. Click the *Do Something* button in the app.
   A `MessageToast` appears with the text *Sorry, an error occurred!*.
3. In the console, you see the following error message:

```
TypeError: oEvent.getSourceXYZ is not a function
at f.onPress (http://.../App.controller.js?eval:20:69)
...
```

   This means that an error occurred in the `onPress` function in the `App.controller.js` file at line `20`.
4. Click the first link in the stack trace after `f.onPress` to look at the source code where you can see that it wasn't the generic `getSource` function that was called, but an undefined `getSourceXYZ`.

```
sMessage = this.getResourceBundle().getText("buttonOk",
[oEvent.getSourceXYZ().getId()])
```

Debugging in the *Sources* Tab

1. Switch to the *Source* tab.
2. To view the source of the `App.controller` file, press `Ctrl` + `P`, enter `App.controller`, and select `App.controller.js?eval`.
3. Set a breakpoint in line 20 by clicking on the line number of the following line:

```
sMessage = this.getResourceBundle().getText("buttonOk",
[oEvent.getSourceXYZ().getId()]);
```

4. Click the *Do Something* button in the app.
   The debugger stops at line 20.
5. In line 20, replace `getSourceXYZ()` with `getSource()` and press `Ctrl` + `S`:

```
sMessage = this.getResourceBundle().getText("buttonOk",
[oEvent.getSource XYZ().getId()]);
```

6. Resume the execution of the code by pressing `F8`.
   The message toast is now displayed on the web page with the following message: *"HeapOfShards---app--myButton" pressed*

> **i Note**
>
> You can also use the *Pause on exception* button and select *Pause on caught exceptions* on the top right of the *Sources* tab to pause the execution before an exception occurs without setting breakpoints.

Checking the *Network* Tab

The *Network* tab shows the sequence and duration of files being loaded. It can be used to optimize loading performance and debug request issues.

1. Switch to the *Network* tab.
2. Press `F5` to reload the page.
   You see a list of the files that are currently loaded.
   You can see that the `NavigationBar.js` file is loaded, but the view does not contain any `NavigationBar` elements so it is not used.
3. You can't edit the code directly in this tab. You have to fix the source files in your development environment and then reload the app.
   Remove the unnecessary reference to the `NavigationBar` in the `App.controller` file:

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    'sap/m/MessageToast',
    'sap/ui/ux3/NavigationBar',
    'jquery.sap.global'
], function(Controller, MessageToast, NavigationBar, jQuery) {
...
```

Testing Responsiveness with *Device Mode*

Switch to *Device* mode by clicking the respective button or by pressing `Ctrl` + `Shift` + `M`.

Emulate different mobile devices by selecting different devices, or switch orientation from landscape to portrait.

Analyzing Performance Problems and Memory Leaks

There are additional tabs that can help you to analyze performance problems or memory leaks. For more information, refer to the documentation of the developer tools of your browser.

- *Memory* or *Profiles*
- *Performance* or *Timeline*
- *Application* or *Resources*

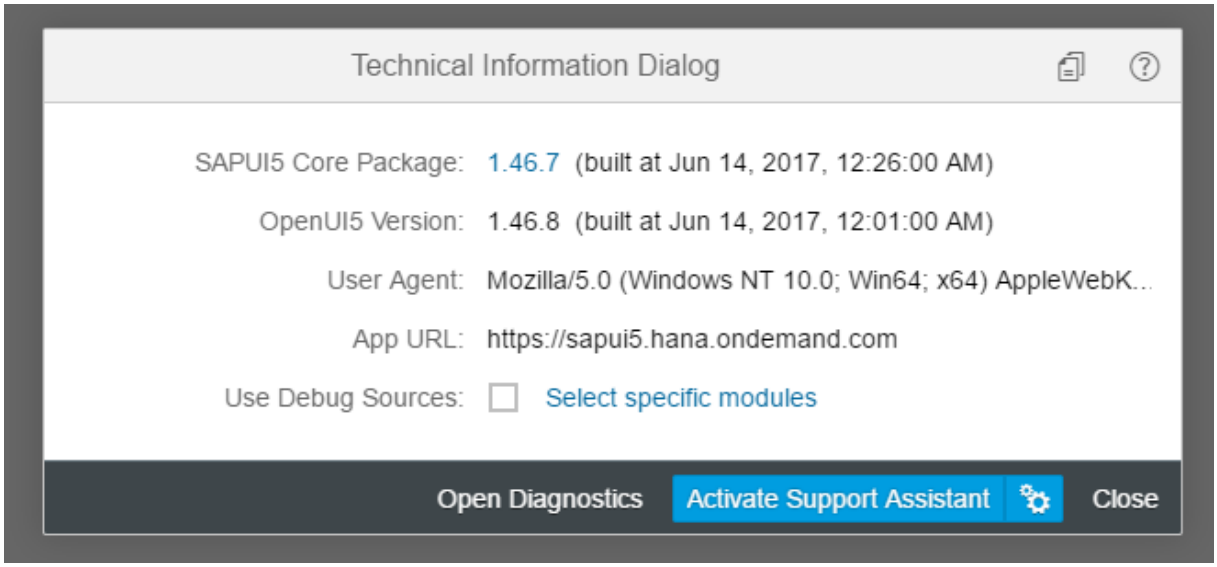
Related Information

Documentation of the Chrome DevTools on `https://developers.google.com`

Step 2: Technical Information Dialog

In this tutorial step, we will have a closer look at the "Technical Information Dialog". This tool comes in handy whenever you want to know the technical details of the running application, and also has some other useful features.

Preview



Opening the Example App and the Technical Information Dialog

1. Download the example app with errors from the Demo Kit at Troubleshooting and run the app.
2. Open the *Technical Information Dialog* by pressing `Ctrl` + `Shift` + `Alt` + `P`.

The dialog box shows information related to the app and provides access to additional support options.

Checking the SAPUI5 Version

When you run into problems with your app, you should check the SAPUI5 version that you're using. The feature that you want to use may not be available in your version or may have some bugs that are already solved in a later version.

1. Check the displayed version information for the*SAPUI5 Core Package* and the *OpenUI5 Version*.
2. Open the version overview at https://ui5.sap.com/versionoverview.html to see if there are newer patch levels or releases of SAPUI5.
3. Read the What's New [page 6] section in the documentation and check the Change Log to find information about new features and bug fixes.

> **i Note**
>
> You can view a specific version of the Demo Kit by adding the version number to the URL, for example, `https://ui5.sap.com/1.38.8/`.

For more information, see Versioning of SAPUI5 [page 29].

Checking the Device

The device on which you run the app may not be supported or might be detected incorrectly by SAPUI5. This can lead to issues with responsiveness or device adaption.

1. Verify that the *User Agent* shown in the dialog box matches your device, browser, and operating system. If the information is truncated because there is not enough space, you can see the full string as a tooltip. To copy the information, use the *Copy technical information to clipboard* button.
2. Test the functionality on another device or by using the device emulation features that are offered in the developer tools of your browser.

Turning On Debug Sources

The SAPUI5 libraries are included in your app in a compressed form. To be able to efficiently debug these libraries, they have to be reloaded in their source format and with developer comments.

1. Select the *Use Debug Sources* checkbox and confirm reloading the app.

2. Open the developer tools of your browser
3. Choose Crtl + O and type the name of an SAPUI5 framework artifact to display its source code in debug mode.

> **i Note**
>
> You may see additional errors and warnings in the developer console. These can help you investigate the problem further.
>
> For performance reasons, you should deactivate this feature again when you're done.

You can also select debug mode only for specific packages:

1. Next to the *Use Debug Sources* checkbox, choose *Select specific modules* to open the selection dialog box.
2. Select one or more modules in the module tree and notice that the value of the input field changes accordingly.
3. Apply the selection and reload the app.

Only the selected modules are now loaded in debug mode.

Copying Technical Info

If you're really stuck or have found a bug, you can open a ticket. Choose the *Copy technical information to clipboard* button to copy the technical details from this dialog box and then attach them to your message.

Accessing Other Tools

The *Technical Information Dialog* also includes links to *Diagnostics* and *Support Assistant* that we will discuss in the following steps of this tutorial.

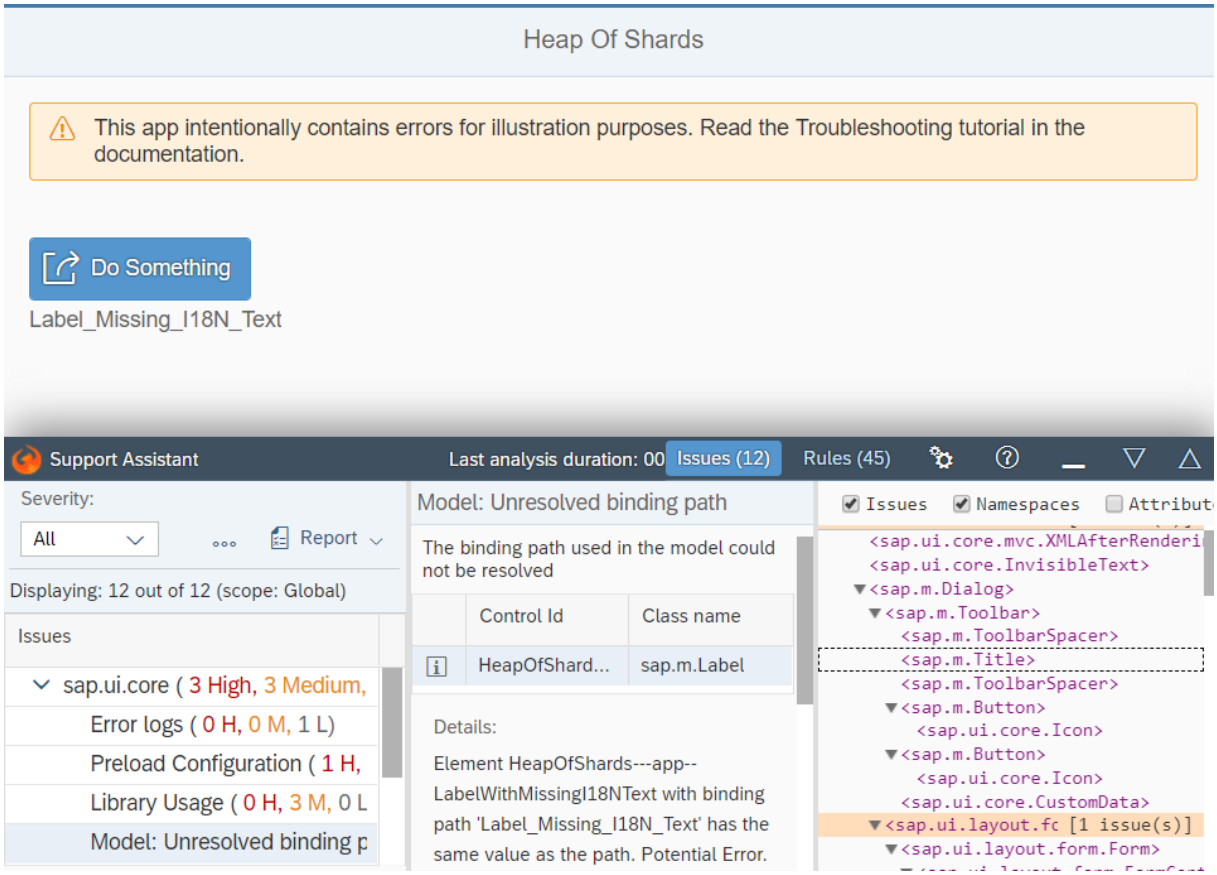
Related Information

Technical Information Dialog [page 1322]

Step 3: Support Assistant

In this tutorial step, we will have a closer look at Support Assistant. You can use this tool to check whether your app is built according to the best practices with predefined rules.

Preview



Opening the Example App and Support Assistant

1. Download the example app with errors from the Demo Kit at Troubleshooting and run the app.
2. Activate the Support Assistant using one of the following options:
   - Open the *Technical Information Dialog* by pressing `Ctrl` + `Shift` + `Alt` + `P` and choose *Activate Support Assistant*.
   - Use the URL parameter: `sap-ui-support=true`.

   The Support Assistant toolbar opens in the footer of the app.

Analyzing and Fixing Issues

1. In the *Support Assistant* toolbar, choose *Rules*.
2. In the *Available Rules* tab, select all rules, and choose *Analyze*.
   You now see a list of issues.
3. Select *Model: Unresolved binding path* in the list of issues. In the issue details, you see the following message: *Element HeapOfShards---app--LabelWithMissingI18NText with binding path 'Label_Missing_I18N_Text' has the same value as the path. Potential Error.*
4. Open the `i18n.properties` file in your development environment and add the missing text.

   ```
   […]
   item1Text=Item 1
   item2Text=Item 2
   selectEventMessage=Event "{0}" fired.
   Label_Missing_I18N_Text=Label Text
   ```

   For more information, see Walkthrough Step 8: Translatable Texts [page 85]
5. Restart the app and start the analysis again. This issue should now be gone. We have intentionally hidden some more errors in the code - check and see if you can find and correct them.

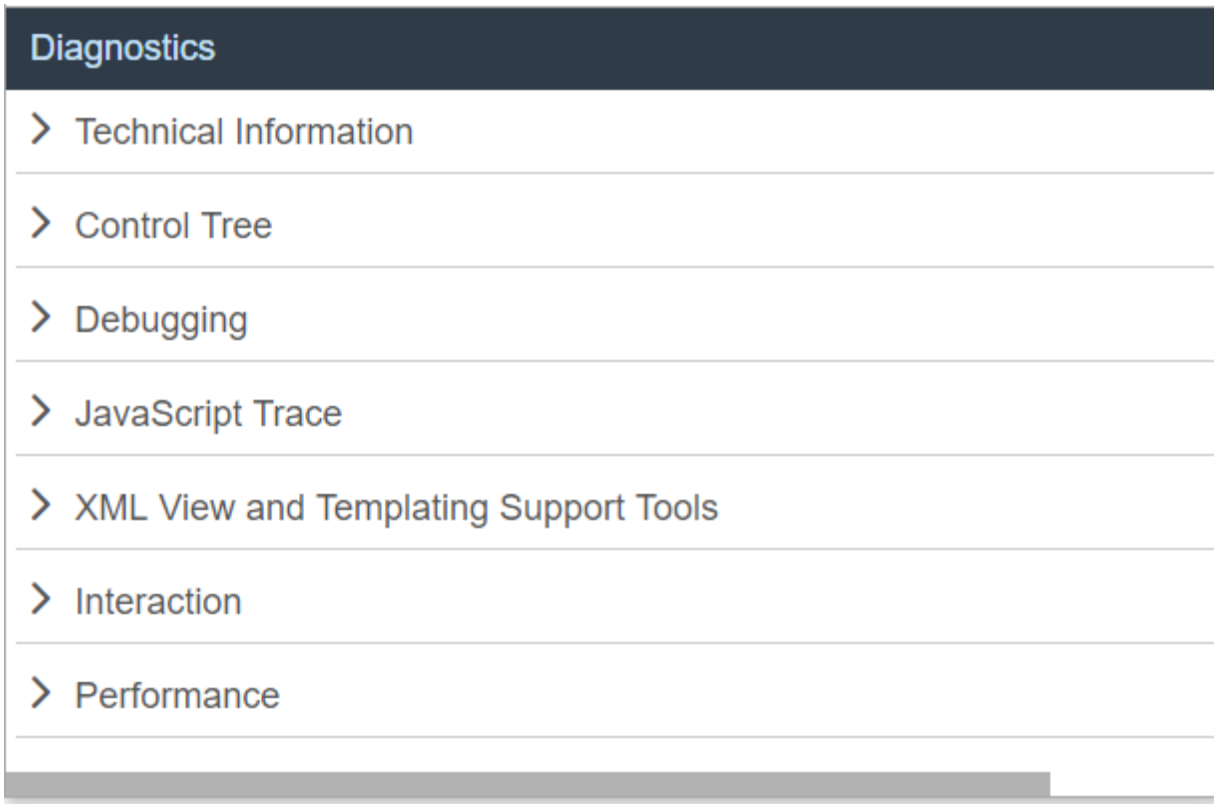
Related Information

Support Assistant [page 1339]

Step 4: Diagnostics Window

In this tutorial step, we have a closer look at the Diagnostics window. It offers a wealth of information including comprehensive technical information, a control tree, and debugging features.

Preview



Opening the Example App and the Diagnostics Window

1. Download the example app with errors from the Demo Kit at Troubleshooting and run the app.
2. Open the *Diagnostics* window by pressing `Ctrl` + `Shift` + `Alt` + `S`.

Improving App Performance

Let's say that you are facing a performance issue in your app, so let's check some performance-relevant settings in the *Diagnostics* window:

1. Expand the *Technical Information* section and scroll down to view the loaded libraries.
2. If you spot any libraries that you originally defined, but you don't actually use, remove them from the `manifest.json` file in your development environment to prevent them from loading. In this case, you can see that the example app loads the `sap.ui.layout` library, even though the `layout` control is not used.
3. Scroll to the *Configuration (bootstrap)* section. You see that the `preload` method is set to synchronous processing.
4. To improve performance, set the bootstrap parameter `data-sap-ui-async` to `true` in the `index.html` file.

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Heap Of Shards</title>
    <!-- set data-sap-ui-async="true" to enable the user to fix it in the
    troubleshooting tutorial step 4: Diagnostics Window -->
    <script id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.f"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.HeapOfShards": "./",
            "sap.ui.demo.DoesNotExist": "./DoesNotExist"
        }'
        data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
[…]
```

Simulating UI Changes

The app contains a *Do Something* button and you want to make the button bigger. The control tree allows you to test which width is the best.

1. Expand the *Control Tree* section. Make sure that you display both the app and the *Diagnostics* windows side-by-side or on different monitors. Otherwise the diagnostics window will go to the background.
2. Press and hold the `Ctrl` + `Shift` + `Alt` keys and click the *Do Something* button in the app. You see the button blinking green.
3. In the control tree of the *Diagnostics* window, the button is selected and you can see its properties on the right.
4. Change the value of the `width` property to `100%` and confirm with `Enter`.
   The button width is automatically increased.
5. The changes that you make in the *Diagnostics* window are only temporary. To make your change permanent, you have to change the app code.

Trying Different SAPUI5 Versions

If you find a bug in your application, you can easily check whether it has already been fixed in a newer SAPUI5 version. Just try the app with a different SAPUI5 version:

1. Expand the *Technical Information* section and check the loaded version.
2. Expand the *Debugging* section.
3. Choose *Other* from the *Boot application with different UI5 version on next reload* dropdown list.
4. Enter a custom URL, for example `https://openui5.hana.ondemand.com/1.46.6/resources/sap-ui-core.js`.
5. Choose *Activate Reboot URL*, confirm the dialog box, and reload the app.
6. Reopen the *Diagnostics* window and expand the *Technical Information* section. The loaded SAPUI5 version is now changed.

More Features

More features are waiting for you to discover in the *Diagnostics* window. For more information, see Diagnostics [page 1326].

Step 5: UI5 Inspector

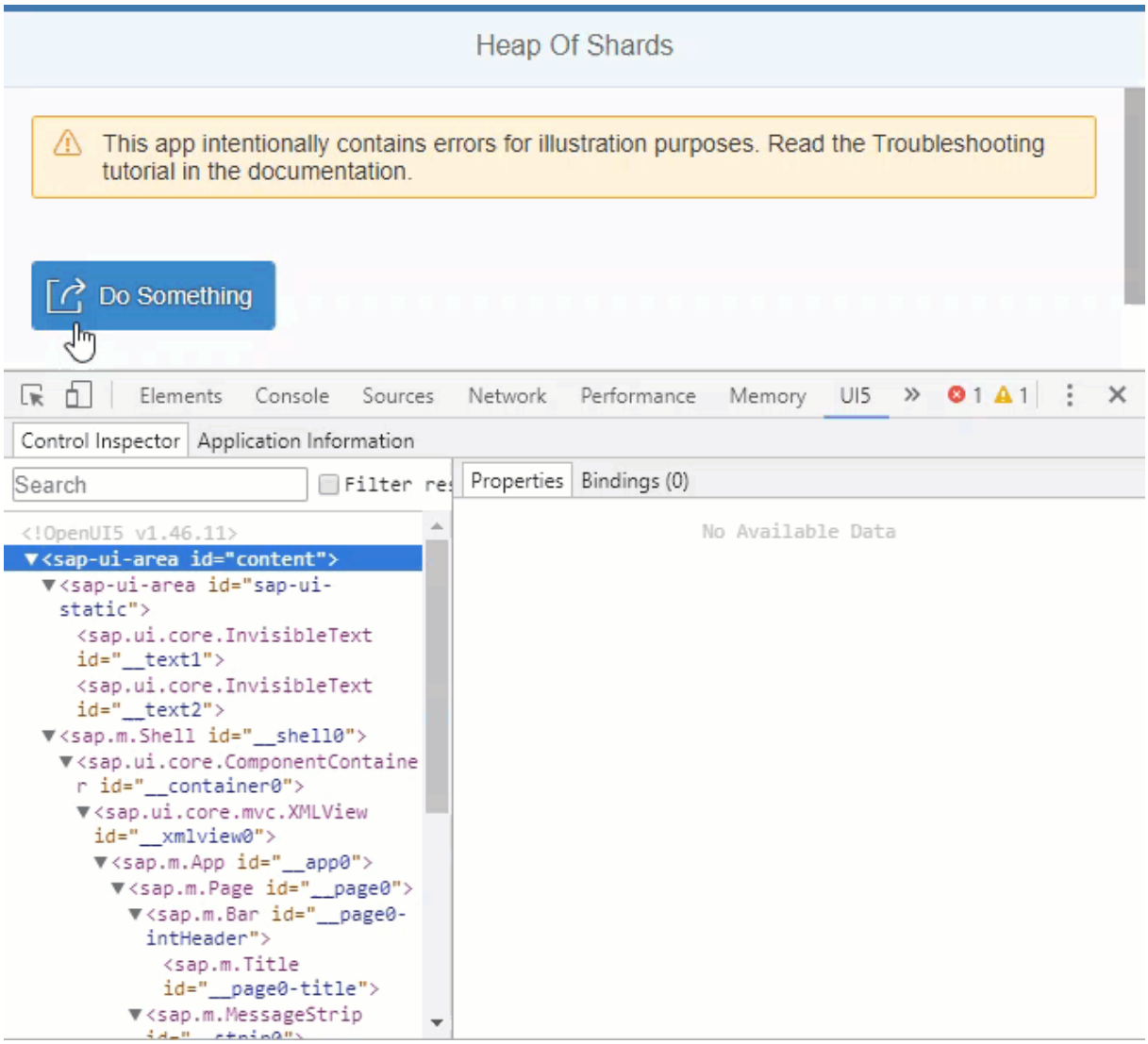
In this tutorial step, we will have a closer look at UI5 Inspector - a plug-in specifically created for analyzing and debugging SAPUI5 code.

With UI Inspector, you can find answers to the following questions, for example:

- What is the structure of your app?
- How are the elements related to each other?
- Which controls are involved when a function is performed?
- Which data is bound to a specific control and how (model and path)?

> **i Note**
>
> UI5 Inspector is only available for the Google Chrome browser.

Preview



Opening the Example App and the UI5 Inspector

1. Download UI5 Inspector from the Chrome Web Store and add it to your Chrome browser as a standard extension .
2. Download the example app with errors from the Demo Kit at Troubleshooting and run the app.
3. Open the *Developer Tools* in Google Chrome by pressing `F12`.
4. Choose the *UI5* tab on the right side of the developer tools panel.
5. Choose *Control Inspector*.
   You now see a list of all of the controls that are used in the current view of the app. When you select an entry, you see the properties and their values in the *Properties* area on the right. You can analyze line by line without being overwhelmed by too much information.

Simulating UI Changes

The app contains a *Do Something* button with meaningless icon (`sap-icon://action`) and text. We want to use the `sap-icon://activate` icon instead and change the text. With UI Inspector, we want to simulate how that will effect the UI change.

1. Right-click the *Do Something* button and from the context menu select *Inspect UI5 Control*.
   The corresponding line in the *Control Inspector* is highlighted and you can view its properties.
2. Double-click the value for the `icon` property, which is currently `sap-icon://action`.
3. Replace `action` with `activate` and confirm with `Enter`.

   The icon on the button in the app is updated to show the new icon  .
4. Double-click the value for the `text` property and change the value to `Activate`.
5. The changes that you make in the UI5 Inspector are only temporary, and the icon will be reset to the default when the page is reloaded. To make your change permanent, you have to change the app code.

Related Information

UI5 Inspector [page 1374]

First-Aid Kit

This section contains the most common issues that you might face when developing SAPUI5 apps and how to solve them.

An Empty Page Comes Up

You find yourself in one of these situations:

- The browser shows an empty page: there's no content and no error message is displayed
- An `Uncaught Error` message is shown in the developer console

Preview

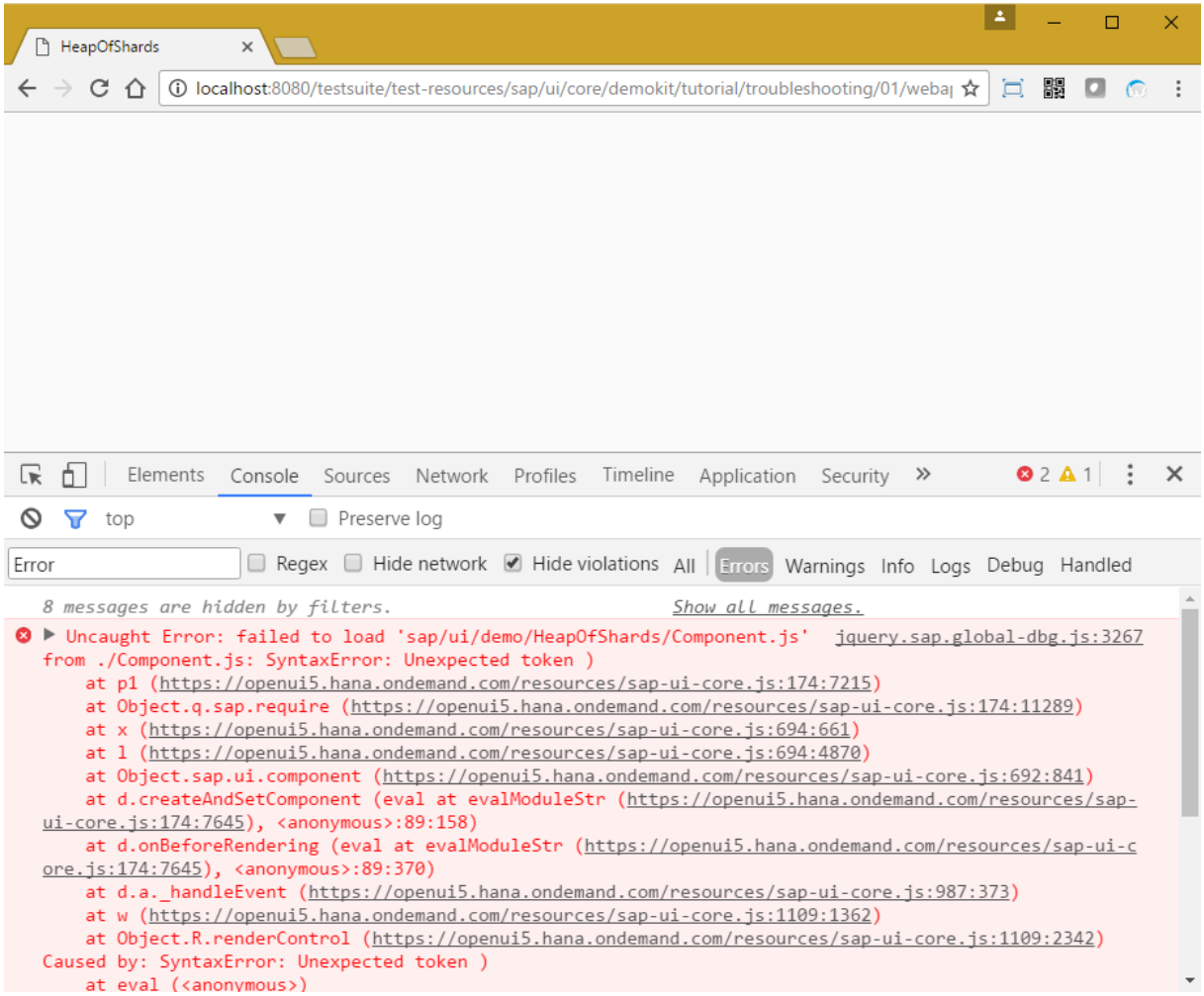


Figure 49: The browser displays an empty page and an `Uncaught Error` is issued in the console

Root Cause

This can happen for one of the following reasons:

- A critical reference error is prohibiting the app from starting.
- A syntax error is stopping the execution of your application code.
- A parsing error has occurred in an XML view.
- The tag of the control is written with lowercase letters.
- The root view is missing a root control.

Resolution

Console shows `"ReferenceError: sap is not defined"`

Have a look at the `resource` path in the bootstrap of the HTML page you are trying to open. The path to the file `sap-ui-core.js` is probably incorrect and needs to point to the path where the SAPUI5 resources are located (typically globally under `/resources` or locally under `resources`).

If you are running the code in SAP Web IDE, you have to configure the `neo-app.json` project descriptor (see Create a neo-app.json Project Configuration File [page 46]).

Other development environments might need the resources to be copied to the server and referenced relatively to the app (see Standard Variant for Bootstrapping [page 694]).

Alternatively, you can use the CDN version (see Variant for Bootstrapping from Content Delivery Network [page 696]).

Console shows `SyntaxError: <error details>`

A JavaScript error in the application code throws an exception and stops all subsequent execution. Take a look at the error details: In most cases, the root cause is mentioned in the first line of the error message.

The stack trace can provide more context on the execution scope. Analyze it from thoroughly to find a line referencing your application code and start debugging there.

Console shows `Error: Invalid XML`

If the XML view to be displayed cannot be parsed, SAPUI5 stops the execution and throws a parse error. Check the XML view for namespace issues, typos, and missing closing tags. Do a schema validation with an XML validator tool.

Console shows `Uncaught Error: failed to load 'sap/m/xxxxx.js'`

During the development on Microsoft Windows, your app works fine, but a soon as you deploy it on a Linux system, only an empty page comes up.

This could happen if you wrote the tag of the control with lowercase letters, because Linux systems use case-sensitive file names.

Correct Example	Incorrect Example
`<Button text="Click me" />`	`<button text="Click me" />` Error message: `Uncaught Error: failed to load 'sap/m/button.js'`

> → Tip
>
> Control tags always start with capital letters after the namespace like `<Button>`, `<l:FixFlex>`, `<f:SimpleForm>`.
>
> Aggregations always start with lowercase letters like `<content>`, `<l:fixContent>`, `<f:content>`

Console shows no error

Your root view is missing a root control. In the context of SAPUI5, `sap.m.App` or `sap.m.SplitApp` function as root controls. Please check your root view (for example `App.view.xml`) and add the missing root control.

Content or Control Is Not Visible

You find yourself in the situation that a control or the content of a control is not visible, but you don't see an error message in the console.

Root Cause

This can happen for one of the following reasons:

- The element is not properly bound
- The `visible` property is set to `false`
- The `height` or `width` dimension is set to `0`

Resolution

First, you should check if your control was rendered properly by using the developer tool of your browser to check the DOM element. For information about how to use your browser tools, see the documentation of you browser or check our Troubleshooting Tutorial Step 1: Browser Developer Tools [page 196].

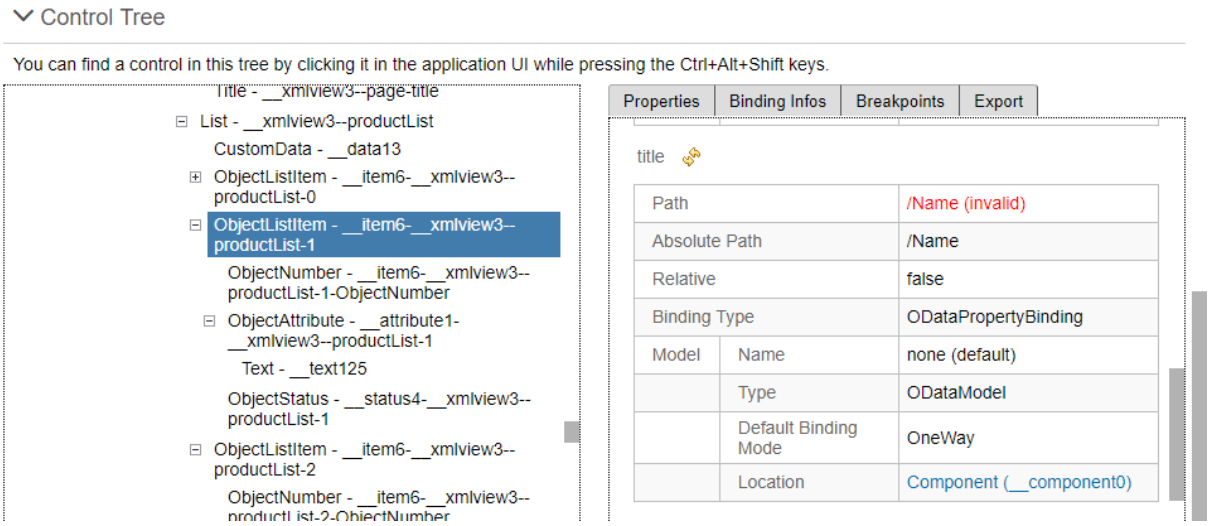
Wrong binding

If you bound your control to a source, for example, an image control, the binding may not be resolved properly. This can be caused by minor mistakes such as typos. We recommend using *Diagnostics* to debug your bindings. For more information, see Diagnostics [page 1326].

In the *Diagnostics* window, you can check whether you used a relative binding instead of an absolute one or vice versa.

If you, for example, use a `List` control, you bind the list itself to an absolute path like `items="{/Products}"` whereas the aggregations are bound to a relative path like `title="{Name}"`. The actual path of the `title` property is now `{/Products/*Product_Index*/Name}`.

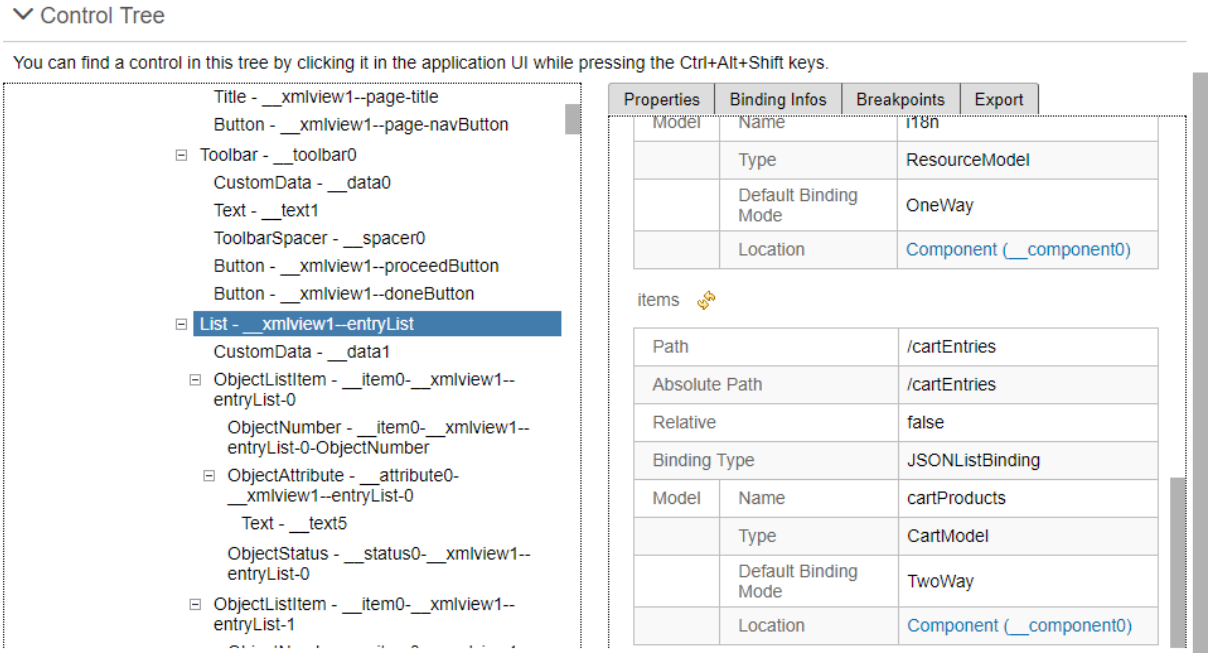
If you used an absolute binding path like `title="{/Name}` for an aggregation instead of a relative one, the result in the window would look like this:



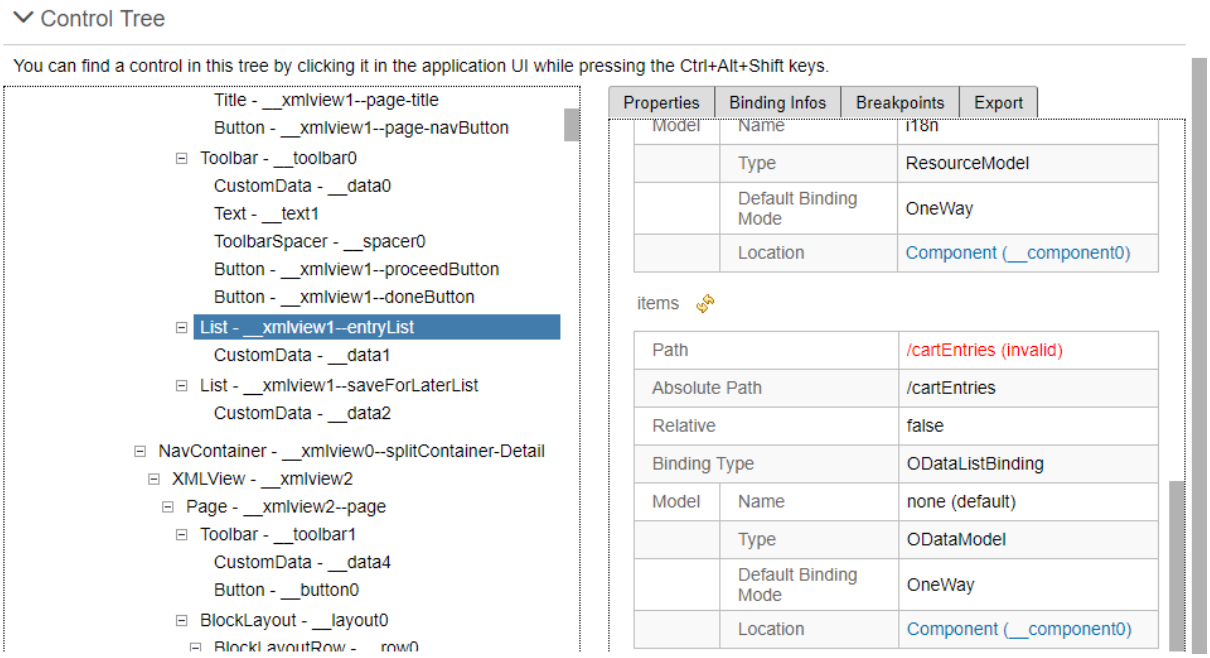
Another common error related to binding is to refer to the default model instead of referring to a specific model. This happens, for examples, if you forgot to add the model name to the binding declaration.

For example, you have two models in your application: the default model, which has no name and another model named `cartProducts`. To bind to the `cartProducts` model you have to write the model name explicitly like `items="{cartProducts>/cartEntries}"`.

If you used the binding correctly *Diagnostics* displays the following:



If the model name is missing, you see the following:



`visible` property set to `false`

If you set the `visible` property of a control to `false`, it will not be rendered at all.

Nested controls inherit the value of the `visible` property from their parents. Therefore, if the control that you are missing is nested in a parent control that is set to invisible, the nested control will also not be rendered.

You can fix this by setting the `visible` property of the parent control to `true` or by moving your missing control in the XML view so that it is not longer nested inside an invisible control.

Dimensions set to `0`

Most controls have the properties `width` and `height`. If one of them is explicitly set to `0` some controls may not be displayed at all. Similar to the `visible` property, the value of `width` and `height` are also inherited from parent controls, as long as you don't set an explicit value for these dimensions. If you, for example, set one of the dimension values for a control to `100%` it will have the same size as the parent control. And if the parent's width is `0` the nested control will also be `0`.

As with the `visible` property, you can solve this by either increasing the size of the parent or setting fixed values for the child (for example, `100px`) instead of a relative value.

Request Fails Due to Same-Origin Policy (Cross-Origin Resource Sharing - CORS)

If you use a remote URL in your code, for example a remote OData service, such as the publicly available Northwind OData service, the browser may refuse to connect to a remote URL. Due to the same-origin policy, browsers deny AJAX requests to service endpoints in case the service endpoint has a different domain/ subdomain, protocol, or port than the app.

Preview

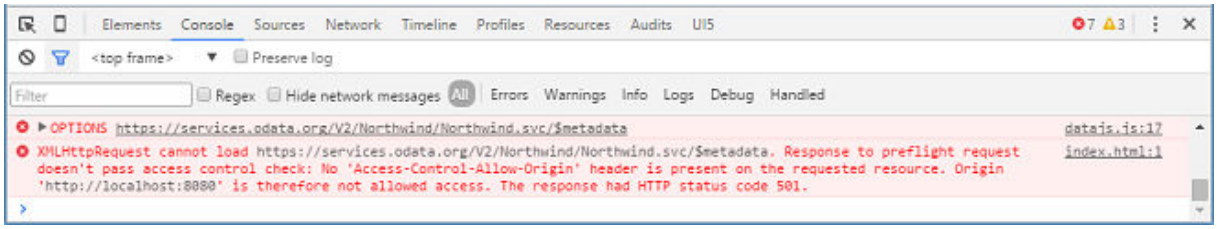


Figure 50: Violations of the same-origin policy in Google Chrome

Root Cause

Normally, the remote system would be configured to send the cross-origin resource sharing (CORS) headers to make the browser also allow direct access to remote URLs. However, if you, for example, use a Northwind OData service, you cannot modify the publicly available service. Then when you try to execute XHR requests (`XMLHttpRequest`) the browser prevents the call due to the same-origin policy.

Resolution

To solve the issue, you have the following options:

- SAP Web IDE: Configure a destination as described below (recommended)
- Local Development: Configure a local proxy (CORS anywhere)
- Workaround: Disable the same-origin policy in the browser for local testing (not recommended, only for testing)
- Set the CORS-relevant response headers on the remote system (if possible)

SAP Web IDE: Configure a destination

SAP Web IDE and the SAP Cloud Platform offer destinations that allow you to easily connect to remote systems. The destination to the Northwind OData service is an internet proxy made available inside the app at `<protocol>://<domain>/destinations/northwind/*`. Any request that is sent to this location is forwarded to `https://services.odata.org` automatically.

Create Destination in SAP Cloud Platform Cockpit

Requested URL	Forwarded To
`/destinations/northwind/V2/Northwind/Northwind.svc/$metadata`	`https://services.odata.org/V2/Northwind/Northwind.svc/$metadata`

Requested URL	Forwarded To
`/destinations/northwind/V2/Northwind/Northwind.svc/Invoices`	`https://services.odata.org/V2/Northwind/Northwind.svc/Invoices`

The destination itself is configured inside the SAP Cloud Platform Cockpit. For more information, see Create a Northwind Destination [page 49].

neo-app.json

For SAP Web IDE, a `neo-app.json` file is needed to make sure that the destination and resource mapping are available in the app. It has to be located in the root folder (`webapp`), on the same level as the `user.project.json` file that is automatically created.

If it does not exist yet, create a `neo-app.json` file and reference the Northwind destination there. Just copy the content of the code into that file and try to run the app again.

```
{
  "routes": [
    {
      "path": "/destinations/northwind",
      "target": {
        "type": "destination",
        "name": "northwind"
      },
      "description": "Northwind OData Service"
    }
  ]
}
```

> **i** Note
>
> If the file already exists, for example, because you already created it to map the SAPUI5 resources, just append the destination to the existing `route` definitions.

manifest.json

In the `manifest.json` descriptor file of your app, you can now change the data source to use the remote destination, for example:

```
{
  "_version": "1.12.0",
  "sap.app": {
    ...
    "dataSources": {
      "invoiceRemote": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/",
        "type": "OData",
        "settings": {
          "odataVersion": "2.0"
        }
      }
    }
  },
  "sap.ui": {
    ...
  },
  "sap.ui5": {
    ...
```

```
    }
}
```

After this change, you can run the app in SAP Web IDE without disabling the same-origin policy of your browser. The destination now manages the connection to the remote service.

Local Development: Configure a local proxy (CORS anywhere)

A proxy is simply a service end point on the same domain of your app to overcome the restrictions. It receives requests from the app, forwards them to another server, and finally returns the corresponding response from the remote service.

Follow the below steps to configure such a proxy in your poject.

Prerequisites: NodeJS is installed on your machine.

package.json

```
{
    "name": "Sample-Package",
    "version": "1.0.0",
    "description": "Sample package.json",
    "scripts": {
        "proxy": "node proxy.js"
    },
    "devDependencies": {
        "cors-anywhere": "^0.4.1"
    },
    "dependencies": {
    }
}
```

Add the `devDependency` called `"cors-anywhere": "^0.4.1"` to your existing `package.json`. Run `node install` to install the npm module. Add the `proxy` script to the `scripts` section in the `package.json` so that you can run a script via `npm run <script_name>`.

proxy.js (new)

```
var cors_proxy = require('cors-anywhere');

// Listen on a specific IP Address
var host = 'localhost';

// Listen on a specific port, adjust if necessary
var port = 8081;

cors_proxy.createServer({
    originWhitelist: [], // Allow all origins
    requireHeader: ['origin', 'x-requested-with'],
    removeHeaders: ['cookie', 'cookie2']
}).listen(port, host, function() {
    console.log('Running CORS Anywhere on ' + host + ':' + port);
});
```

Create a new file `proxy.js`, and copy the above script into your project directory. This is the pre-configured proxy server we are going to use to prevent the occurrence of **same-origin policy error**. We can start it by running the command `node proxy.js` or `npm run proxy. It runs a local proxy on port` in the console.

manifest.json

```
{
    "sap.app": {
        ...
        "dataSources": {
            "northwind": {
                "uri": "http://localhost:8081/https://services.odata.org/V2/
Northwind/Northwind.svc/",
                "type": "OData",
                "settings": {
                    "odataVersion": "2.0"
                }
            }
        }
    }
}
```

To use a service in the `local ui5 application` we have to change the `uri` in the `manifest` file.

> **i** Note
>
> The `uri` must start with `http://localhost:<port>`.

> **i** Note
>
> By default, you can't run the request in your browser with the `proxy.js` script. It throws the following exception: `exception Missing required request header. Must specify one of: origin,x-requested-with`. If you want to test the service in your browser, you can temporarily comment out the line `requireHeader: ['origin', 'x-requested-with']` from your `proxy.js`.

For more information on CORS Anywhere, see https://www.npmjs.com/package/cors-anywhere

Workaround: Disable the same-origin policy in the browser (not recommended, only for testing)

. It runs a local proxyIn Google Chrome, you can easily disable the same-origin policy of Chrome by running Chrome with the following command: `[your-path-to-chrome-installation-dir]\chrome.exe --disable-web-security --user-data-dir`. Make sure that all instances of Chrome are closed before you run the command. This allows all web sites to break out of the same-origin policy and connect to the remote service directly.

> ⚠ Caution
>
> **This approach is not recommended for productive apps. Running Chrome this way for surfing on the internet poses a security risk.** However, it allows you to avoid the need of setting up a proxy at development time or for testing purposes.

App or Control Looks Odd

You find yourself in a situation that your app or a control looks different than you expected.

Root Cause

This can happen for one of the following reasons:

- An HTML file is missing the `DOCTYPE` specification (this leads, for example, to exceptionally high table headers)
- Custom styles aren't working properly
- The theme you are using does not support the used libraries

Resolution

To solve the issue, you have the following options:

- Check whether the `<!DOCTYPE html>` tag is placed at the beginning of each HTML file, before the `<html>` tag.
- Check if you have used a custom CSS in your app.
  If you have used a custom CSS, it is probably interfering with the styling in the standard SAPUI5 theming libraries.
  Use the developer tools of your browser to inspect the element that has the wrong styling. In the HTML tab, you can usually see which styles are applied to a DOM element. If you have styles in the list that are added by your app, disable these styles in the debugger to see whether this solves the problem.

  > **i Note**
  >
  > SAPUI5-specific CSS classes and IDs all have an `sapUi` prefix, for example, `sapUiButton`.

  If this does not solve the issue, check for inline styles that are applied to the element in the HTML code. You can also try to isolate the control from the app to see whether there is an issue with the control instead of a collision of styles.
- Check whether the theme you that you are using is supported in combination with the libraries that you are using in your app. For more information, see Supported Combinations of Themes and Libraries [page 27] and Deprecated Themes and Libraries [page 34].

Data Binding

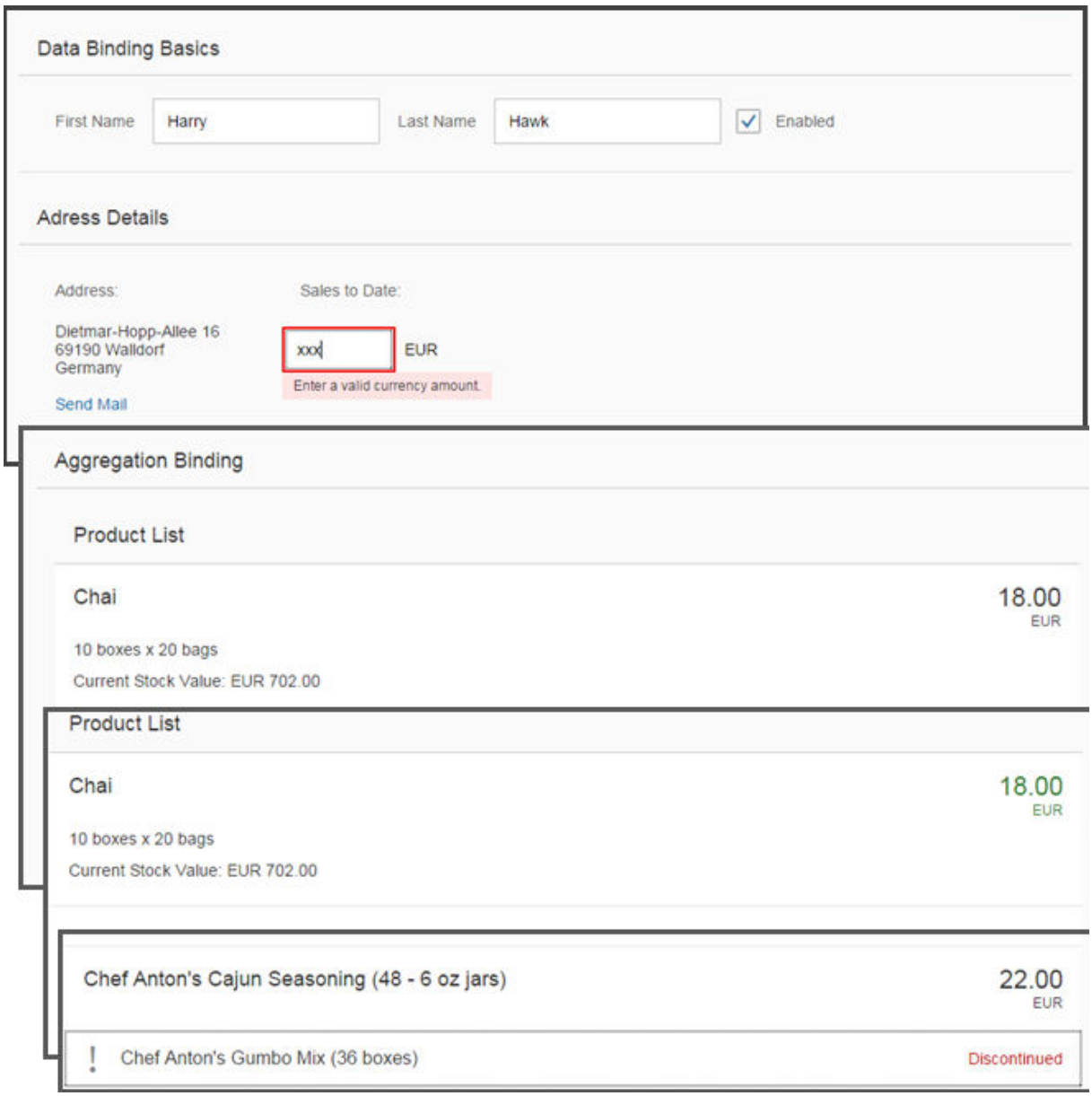
In this tutorial, we will explain the concepts of data binding in SAPUI5.

You use data binding to bind UI elements to data sources to keep the data in sync and allow data editing on the UI.

For data binding, you need a model and a binding instance: The model instance holds the data and provides methods to set the data or to retrieve the data from a server. It also provides a method for creating bindings to the data. When this method is called, a binding instance is created, which contains the binding information and provides an event, which is fired whenever the bound data changes. An element can listen to this event and update its visualization according to the new data.

The UI uses data binding to bind controls to the model which holds the application data, so that the controls are updated automatically whenever application data changes. Data binding is also used the other way round, when changes in the control cause updates in the underlying application data, for example data entered by the user. This is called two-way binding.

Preview



→ Tip

You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, copy it to your workspace and make sure that the application runs by calling the `webapp/index.html` file.

You can view and download the files for all steps in the Demo Kit at Data Binding. Depending on your development environment you might have to adjust resource paths and configuration entries.

For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Related Information

Data Binding [page 815]
Model View Controller (MVC) [page 784]

Step 1: No Data Binding

In this step, we simply place some text on the screen using a standard `sap.m.Text` control. The text in this control is a hard-coded part of the control's definition; therefore, this is not an example of data binding!

Preview

Figure 51: Screen with text

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 1.

webapp/index.html (New)

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Data Binding</title>
    <script id="sap-ui-bootstrap"
        src="resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
    <script src="index.js"></script>
</head>
<body class="sapUiBody" id="content"></body>
</html>
```

Create a new folder `webapp` which will contain all sources of the app that we will create throughout this tutorial, and create the `index.html` file within this folder.

webapp/index.js (New)

```
sap.ui.require([
    "sap/m/Text"
], function (Text) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        // Create a text UI element that displays a hardcoded text string
        new Text({text: "Hi, my name is Harry Hawk"}).placeAt("content");
    });
});
```

Create a new `index.js` file that will contain the application logic for this tutorial. We start by placing the `sap.m.Text` control into the html content. Since the value of the control's text property has been hard-coded, it is unrelated to any data that might exist within a model object. Therefore, data binding is **not** being used here.

Step 2: Creating a Model

In this step, we create a model as container for the data on which your application operates.

The business data within a model can be defined using various formats:

- JavaScript Object Notation (JSON)
- Extensible Markup Language (XML)
- OData
- Your own custom format (not covered in this tutorial)

> **i Note**
>
> There is also a special type of model called a "resource model". This model type is used as a wrapper object around a resource bundle file. The names of such files must end with `.properties` and are used typically for holding language-specific text.
>
> We will use this in Step 6: Resource Models [page 229].

When JSON, XML and resource models are created, the data they contain is loaded in a single request (either from a file stored locally on the client or by requesting it from a Web server). In other words, after the model's data has been requested, the entire model is known to the application. These models are known as client-side models and tasks such as filtering and sorting are performed locally on the client.

An OData model however, is a server-side model. This means that whenever an application needs data from the model, it must be requested from the server. Such a request will almost never return all the data in the model, typically because this would be far more data than is required by the client application. Consequently, tasks such as sorting and filtering should always be delegated to the server.

In this tutorial, we will focus on JSON models since they are the simplest ones to work with.

Preview

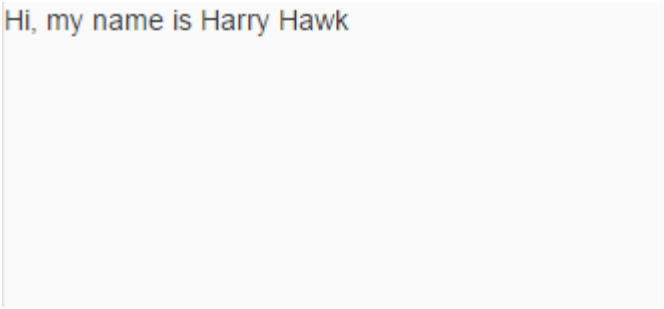


Figure 52: Screen with text derived from a model object (No visual changes to last step)

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 2.

webapp/index.js

```
sap.ui.require([
    "sap/m/Text",
    "sap/ui/model/json/JSONModel"
], function (Text, JSONModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        // Create a JSON model from an object literal
        var oModel = new JSONModel({
```

```
            greetingText: "Hi, my name is Harry Hawk"
        });

        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        // Create a text UI element that displays a hardcoded text string
        new Text({text: "Hi, my name is Harry Hawk"}).placeAt("content");
    });
});
```

Create a new JSON model passing the data as object literal and store the resulting model instance in a local variable called `oModel`.

Set `oModel` to be the default model within the entire SAPUI5 core.

This makes the model object globally available to all controls used within the application.

In this case we have bound the model object to the SAPUI5 core. This has been done for simplicity, but is not considered good practice. Generally speaking, a model object holding business data should be bound to the view that displays the data. We will correct this part of the code in the following steps.

> **i** Note
>
> Models can be set on every control by calling `setModel()`. The model is then propagated to all aggregated child controls (and their children, and so on...). All child control will then have access to that model

The text that is displayed on the UI is still hard-coded and not taken from the model - we will bind the property `greetingText` to our UI control in the next step.

Related Information

Models [page 882]
JSON Model [page 991]

Step 3: Create Property Binding

Although there is no visible difference, the text on the screen is now derived from model data.

Preview

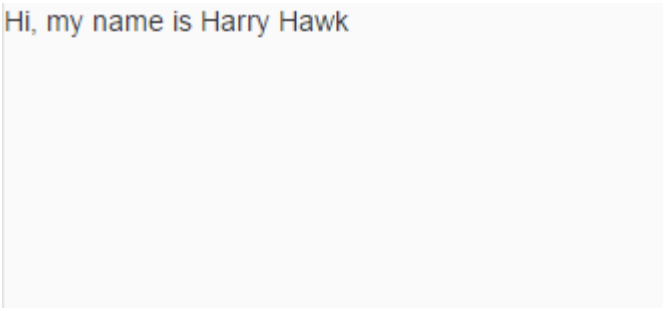


Figure 53: Screen with text derived from various sources (No visual changes to last step)

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 3.

webapp/index.js

```
sap.ui.require([
    "sap/m/Text",
    "sap/ui/model/json/JSONModel"
], function (Text, JSONModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        // Create a JSON model from an object literal
        var oModel = new JSONModel({
            greetingText: "Hi, my name is Harry Hawk"
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        // Display a text element whose text is derived
        // from the model object
        new Text({text: "{/greetingText}"}).placeAt("content");
    });
});
```

The `text` property of the `sap.m.Text` control is set to the value `{/greetingText}`. The curly brackets enclosing a binding path (binding syntax) are automatically interpreted as a binding. These binding instances are called `PropertyBindings`. In this case, the control's `text` property is bound to the `greetingText` property at the root of the default model, as the slash (`/`) at the beginning of the binding path denotes an absolute binding path.

Related Information

Binding Types [page 817]
Property Binding [page 818]

Step 4: Two-Way Data Binding

In the examples used so far, we have used a read-only field to display the value of a model property. We will now change the user interface so that the first and last name fields are displayed using `sap.m.Input` fields and an additional check box control is used to enable or disable both input fields. This arrangement illustrates a feature known as "two-way data binding". Now that the view contains more controls, we will also move the view definition into an XML file.

Preview

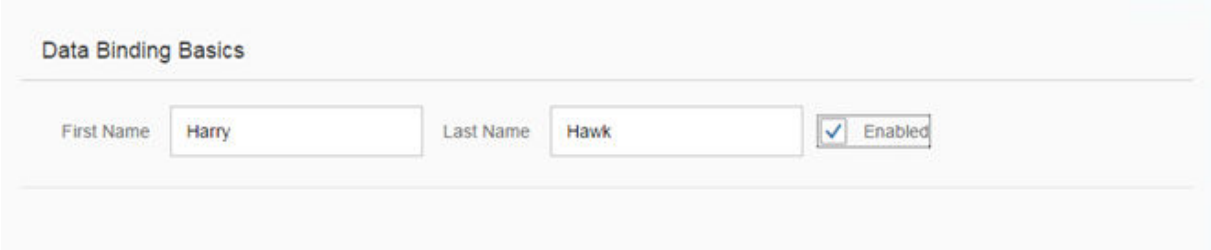


Figure 54: Input fields can be enabled or disabled

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 4.

webapp/view/App.view.xml (New)

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
  <Panel headerText="{/panelHeaderText}" class="sapUiResponsiveMargin"
width="auto">
    <content>
      <Label text="First Name" class="sapUiSmallMargin" />
      <Input value="{/firstName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}" />
      <Label text="Last Name" class="sapUiSmallMargin" />
      <Input value="{/lastName}" valueLiveUpdate="true" width="200px" enabled="{/
enabled}" />
      <CheckBox selected="{/enabled}" text="Enabled" />
    </content>
  </Panel>
</mvc:View>
```

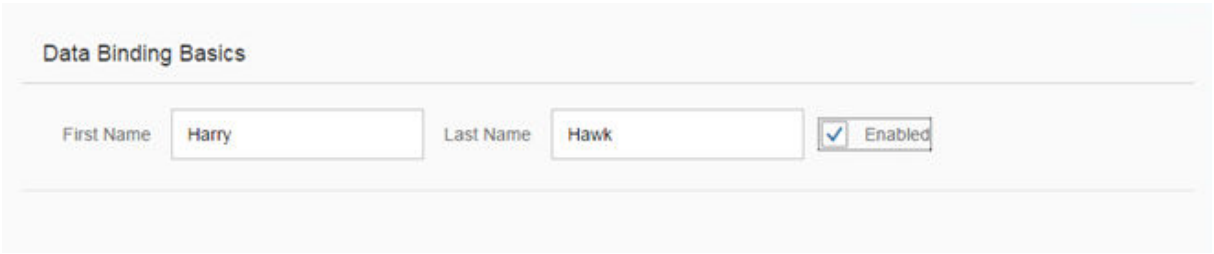
We create a new `view` folder in our app and a new file for our XML view inside the app folder.

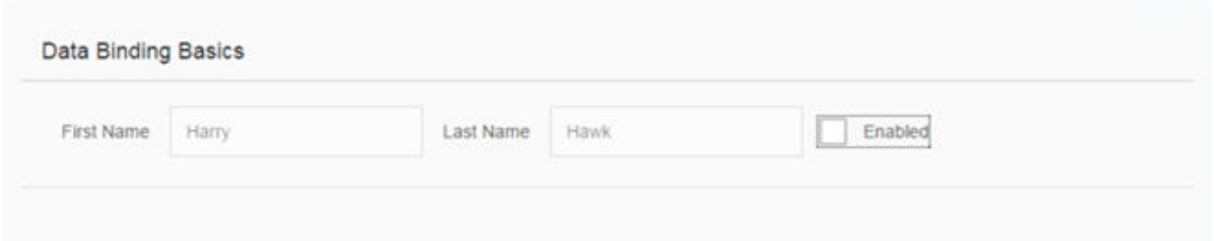
webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView"
], function (JSONModel, XMLView) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        // Create a JSON model from an object literal
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true,
            panelHeaderText: "Data Binding Basics"
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        // Display the XML view called "App"
        new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
    });
});
```

We delete the code that assigned the `sap.m.Text` field to the UI and add an XML view that is identified by its resource name.

You can now refresh the application preview and select or deselect the checkbox. You will see that the input fields are automatically enabled or disabled in response to the state of the checkbox.





It is clear that we have not written any code to transfer data between the user interface and the model, yet the `Input` controls are enabled or disabled according to the state of the checkbox. This behaviour is the result of the fact that all SAPUI5 models implement two-way data binding, and for JSON Models, two-way binding is the default behavior.

Two things are happening here:

- Data binding allows the property of a control to derive its value from any suitable property in a model.
- SAPUI5 automatically handles the transport of data both from the model to the controls, and back from the controls to the model. This is called two-way binding.

Related Information

Data Binding [page 815]

Step 5: One-Way Data Binding

In contrast to the two-way binding behavior shown above, one-way data binding is also possible. Here, data is transported in one direction only: from the model, through the binding instance to the consumer (usually the property of a control), but never in the other direction. In this example, we will change the previous example to use one-way data binding. This will illustrate how the flow of data from the user interface back to the model can be switched off if required.

Preview

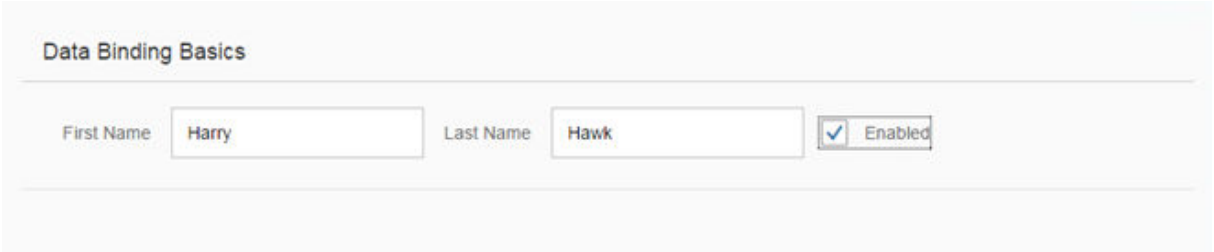


Figure 55: Two-way data binding disabled for the checkbox

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 5.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/BindingMode"
], function (JSONModel, XMLView, BindingMode) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
```

```
    sap.ui.getCore().attachInit(function () {
        // Create a JSON model from an object literal
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true,
            panelHeaderText: "Data Binding Basics"
        });
        oModel.setDefaultBindingMode(BindingMode.OneWay);

tml
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        // Display the XML view called "App"
        new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
    });
});
```

Insert the single highlighted line immediately after the creation of the model object in `index.js`.

Now, no matter what state the checkbox is in, the input fields remain open for input because one-way data binding ensures that data flows only from the model to the UI, but never in the other direction.

The binding mode (one-way or two-way) is set on the model itself. Therefore, unless you specifically alter it, a binding instance will always be created using the model's default binding mode.

Should you wish to alter the binding mode, then there are two ways of doing this:

- Alter the model's default binding mode. This is the approach used above.
- Specify the data binding mode for a specific binding instance by using the `oBindingInfo.mode` parameter. This change applies only to this data binding instance. Any other binding instances will continue to use the model's default binding mode. For more information, see API Reference: `sap.ui.base.ManagedObject.bindProperty`.

> **i Note**
>
> There are two important points to understand about alterations to a model object's data binding mode:
>
> - If you alter the default binding mode of a model (as in the example above), then unless you explicitly say otherwise, all binding instances created after that point in time will use the altered binding mode.
> - Altering a model's default binding mode has no effect on already existing binding instances.

Step 6: Resource Models

Business applications also require language-specific (translatable) texts used as labels and descriptions on the user interface.

The example we used at the start of this tutorial was overly simplistic as we stored language-specific text directly in a JSON model object. Generally speaking, unless language-specific text is derived directly from a back-end system, it is not considered good programming practice to place translatable texts directly into a model. So let's correct this situation by placing all translatable texts (such as field labels) into a resource bundle.

Preview

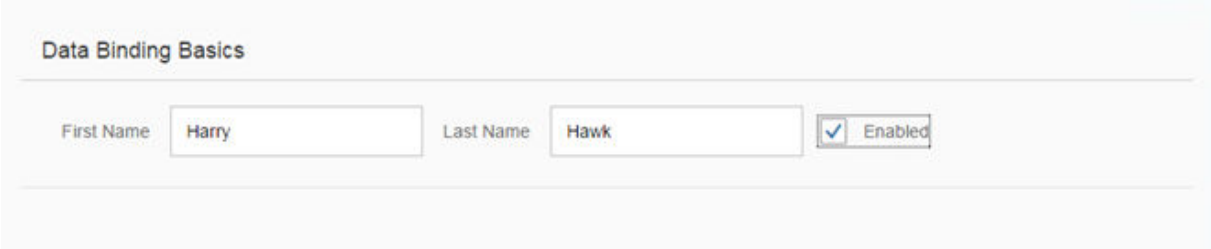


Figure 56: Texts derived from the resource model (No visual change to last step)

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 6.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/resource/ResourceModel"
], function (JSONModel, XMLView, ResourceModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        // Create a JSON model from an object literal
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        // Create a resource bundle for language specific texts
        var oResourceModel = new ResourceModel({
            bundleName: "sap.ui.demo.db.i18n.i18n"
        });

        // Assign the model object to the SAPUI5 core using the name "i18n"
        sap.ui.getCore().setModel(oResourceModel, "i18n");
        // Display the XML view called "App"
        new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
    });
});
```

Since we are creating a resource model, the file name is assumed to have the extension `.properties`; this does not need to be stated explicitly. The resource model is set to the core using the model name `i18n`.

> **i** Note
>
> Remove `, panelHeaderText : "Data Binding Basics"` from the model definition in the `index.js` file. This text is now moved to the resource model.

webapp/i18n/i18n.properties (New)

```
# Field labels
firstName=First Name
lastName=Last Name
enabled=Enabled

# Screen titles
panelHeaderText=Data Binding Basics
```

Create a new folder `i18n`, and a new file `i18n.properties` within and add the code above.

The `panelHeaderText` property has been moved from the JSON model into the `i18n` resource bundle, also the field labels are no longer hard coded in the XML view. This is because all of these text fields need to be translated.

Language-specific text stored in resource models obeys the Java convention for internationalization (i18n).

webapp/view/App.view.xml

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{i18n>panelHeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <Label text="{i18n>firstName}" class="sapUiSmallMargin"/>
            <Input value="{/firstName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <Label text="{i18n>lastName}" class="sapUiSmallMargin"/>
            <Input value="{/lastName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <CheckBox selected="{/enabled}" text="{i18n>enabled}"/>
        </content>
    </Panel>
</mvc:View>
```

Modify the data binding for the panel header and the labels in `App.view.xml` to include the model name. Notice that a "greater than" character separates the model name and the property name, and that i18n property names **must not** start with a slash character.

You could use multiple model instances by using different model names. The model name could be set as second parameter using the `setModel(oResourceModel,"i18n")` method. The model is then propagated under this name to all aggregated child controls (and their children, and so on...). All these controls have access to this model under the name `i18n` as well as to the `JSONModel` (default model, which has no name).

Related Information

Resource Model [page 995]

Step 7: (Optional) Resource Bundles and Multiple Languages

The reason we have resource bundles is to allow an app to run in multiple languages without the need to change any code. To demonstrate this feature, we will create a German version of the app – in fact all we need to do is create a German version of the resource bundle file. No code changes are needed.

Preview

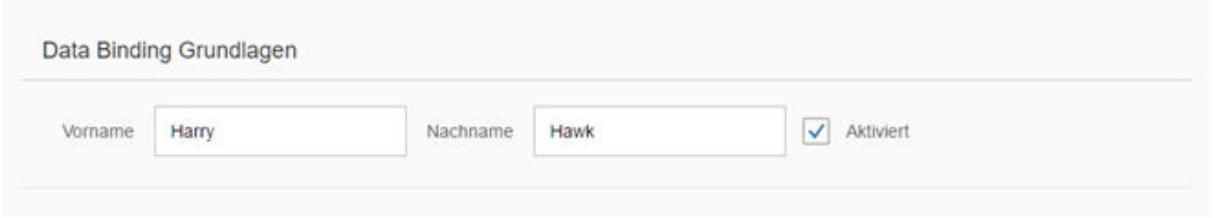


Figure 57: German version of our UI

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 7.

webapp/i18n/i18n_de.properties (New)

```
# Field labels
firstName=Vorname
lastName=Nachname
enabled=Aktiviert
# Screen titles
panelHeaderText=Data Binding Grundlagen
```

In the `i18n` folder, take a copy of the file `i18n.properties` and call it `i18n_de.properties`. Change the English text to the German text.

To test the outcome, change the default language of your browser to German and refresh your preview.

Related Information

Localization [page 1269]

Step 8: Binding Paths: Accessing Properties in Hierarchically Structured Models

In step 6 , we stated that the fields in a resource model are arranged in a flat structure; in other words, there can be no hierarchy of properties; however, this is true only for resource models. The properties within JSON and OData models almost always are arranged in a hierarchical structure. Therefore, we should take a look at how to reference fields in a hierarchically structured model object.

Preview

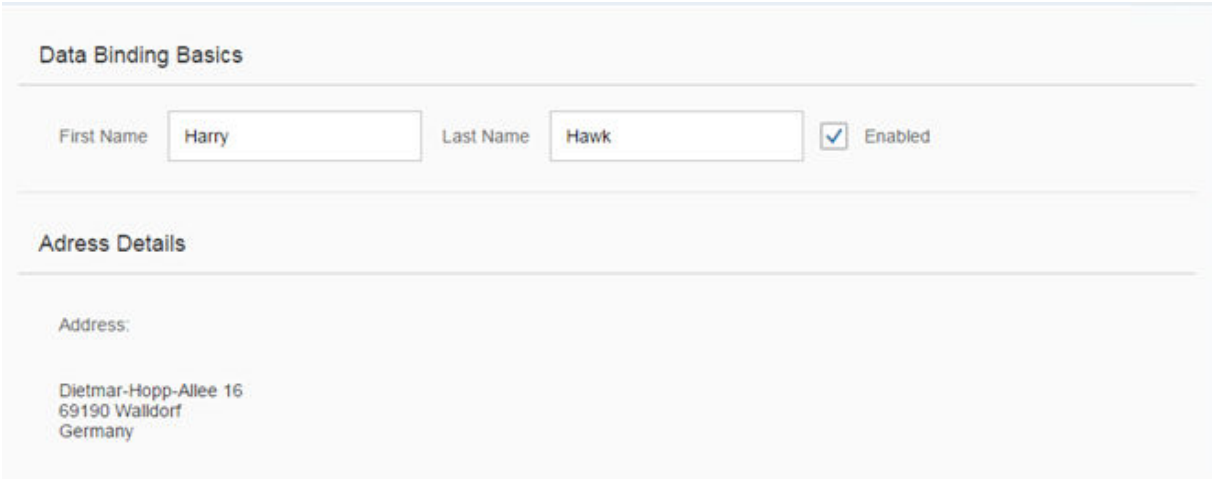


Figure 58: Second panel with additional data

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 8.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/resource/ResourceModel"
], function (JSONModel, XMLView, ResourceModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true,
            address: {
                street: "Dietmar-Hopp-Allee 16",
                city: "Walldorf",
```

```
                zip: "69190",
                country: "Germany"
            }
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        var oResourceBundle = new ResourceModel({
            bundleName: "sap.ui.demo.db.i18n.i18n"
        });
        sap.ui.getCore().setModel(oResourceBundle, "i18n");
        // Display the XML view called "App"
        new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
    });
});
```

The JSON model object now contains an additional sub-object called `address`. Within this object are four properties: `street`, `city`, `zip`, and `country`.

webapp/view/App.view.xml

```
<mvc:View
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{i18n>panel1HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <Label text="{i18n>firstName}" class="sapUiSmallMargin"/>
            <Input value="{/firstName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <Label text="{i18n>lastName}" class="sapUiSmallMargin"/>
            <Input value="{/lastName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <CheckBox selected="{/enabled}" text="Enabled"/>
        </content>
    </Panel>
    <Panel headerText="{i18n>panel2HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <l:VerticalLayout>
                <Label class="sapUiSmallMargin" text="{i18n>address}:"/>
                <FormattedText class="sapUiSmallMarginBegin
sapUiSmallMarginBottom" htmlText="{/address/street}&lt;br&gt;{/address/zip} {/
address/city}&lt;br&gt;{/address/country}" width="200px"/>
            </l:VerticalLayout>
        </content>
    </Panel>
</mvc:View>
```

We add a new panel to the XML view with a new `Label` and `Text` pair of elements.

The text property of the `Label` element is bound to the i18n resource bundle field `address`.

The text property of the `Text` element is bound to three i18n properties: `/address/street`, `/address/zip`, `/address/city`, and `/address/country`. The resulting address format is achieved by separating each one of these model property references with a hard-coded newline character while `zip` and `city` are separated by a space.

webapp/i18n/i18n.properties

```
# Field labels
firstName=First Name
lastName=Last Name
enabled=Enabled
address=Address
# Screen titles
panel1HeaderText=Data Binding Basics
panel2HeaderText=Address Details
```

webapp/i18n/i18n_de.properties

```
# Field labels
firstName=Vorname
lastName=Nachname
enabled=Aktiviert
address=Adresse
# Screen titles
panel1HeaderText=Data Binding Grundlagen
panel2HeaderText=Adressdetails
```

> **i** Note
>
> The resource bundle files now contain new properties for the **Address** and a new panel header text. Both panel properties have been numbered.
>
> In the XML view, inside the curly brackets for the binding path of the `Text` element, notice that the first character is a forward slash. This is required for binding paths that make absolute references to properties in JSON and OData models, but must not be used for resource models. After the first forward slash character, the binding path syntax uses the object names and the property name separated by forward slash characters (`{/address/street}`).
>
> As has been mentioned previously, all binding path names are case-sensitive.

Related Information

JSON Model [page 991]

Step 9: Formatting Values

We also want to provide our users a way of contacting Harry Hawk. Therefore we will add a link that sends an e-mail to Harry. To achieve that we will convert our data in the model to match the

`sap.m.URLHelper.normalizeEmail` API. As soon as the user changes the name, the e-mail will also change. We will need a custom formatter function for this.

Preview

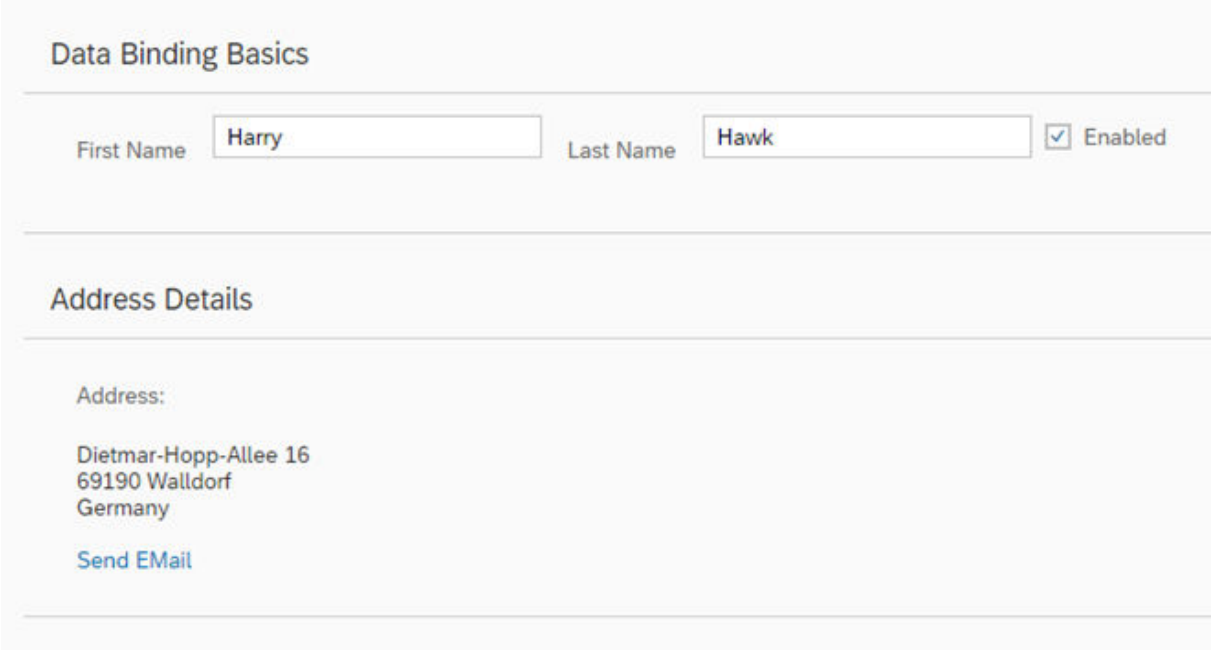


Figure 59: Address with e-mail link

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 9.

webapp/controller/App.controller.js (New)

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/library"
], function (Controller, mobileLibrary) {
    "use strict";

    return Controller.extend("sap.ui.demo.db.controller.App", {
        formatMail: function(sFirstName, sLastName) {
            var oBundle = this.getView().getModel("i18n").getResourceBundle();
            return mobileLibrary.URLHelper.normalizeEmail(
                sFirstName + "." + sLastName + "@example.com",
                oBundle.getText("mailSubject", [sFirstName]),
                oBundle.getText("mailBody"));
        }
    });
});
```

Create a new folder `controller` within your `webapp` folder as a general location for all controller files for this app and create a new file `App.controller.js`.

In our custom formatter, we define the first and last name that are currently in the model as function parameters. When a user changes the data in the model by entering a different name in the input fields, our formatter will be invoked automatically by the framework. This makes sure that the UI is in sync with the data model.

In the `formatMail` function, we use the `sap.m.URLHelper.normalizeEmail` function that expects an e-mail address, a mail subject and a text body. When a user chooses the link, the default email client will open with these parameters.For more information, see API Reference: `sap.m.URLHelper.normalizeEmail`. The `mailSubject` resource bundle text will contain a placeholder for the first name of the recipient (see below). Therefore, we provide the name with `[sFirstName]`.

> **i Note**
>
> For a detailed description of the e-mail link format, see https://developer.mozilla.org/de/docs/Web/Guide/HTML/Email_links .

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.db.controller.App"
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{i18n>panel1HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <Label text="{i18n>firstName}" class="sapUiSmallMargin"/>
            <Input value="{/firstName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <Label text="{i18n>lastName}" class="sapUiSmallMargin"/>
            <Input value="{/lastName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <CheckBox selected="{/enabled}" text="Enabled"/>
        </content>
    </Panel>
    <Panel headerText="{i18n>panel2HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <l:VerticalLayout>
                <Label class="sapUiSmallMargin" text="{i18n>address}:"/>
                <FormattedText class="sapUiSmallMarginBegin
sapUiSmallMarginBottom" htmlText="{/address/street}&lt;br&gt;{/address/zip} {/
address/city}&lt;br&gt;{/address/country}" width="200px"/>
                <Link class="sapUiSmallMarginBegin"
                    href="{
                            parts: [
                                '/firstName',
                                '/lastName'
                            ],
                            formatter: '.formatMail'
                        }"
                    text="{i18n>sendEmail}"/>
            </l:VerticalLayout>
        </content>
    </Panel>
```

```
</mvc:View>
```

For more complex bindings we cannot use the simple binding syntax with the curly braces anymore. The `href` property of the `Link` element now contains an entire object inside the string value. In this case, the object has two properties:

- `parts`
  This is a JavaScript array in which each element is an object containing a `path` property. The number and order of the elements in this array corresponds directly to the number and order of parameters expected by the `formatMail` function.
- `formatter`
  A reference to the function that receives the parameters listed in the `parts` array. Whatever value is returned by the formatter function becomes the value set for this property. The dot (`formatMail`) at the beginning of the formatter tellsSAPUI5 to look for a `formatMail` function on the controller instance of the view. If you do not use the dot, the function will be resolved by looking into the global namespace.

> **i** Note

webapp/i18n/i18n.properties

```
…
# Screen titles
panel1HeaderText=Data Binding Basics
panel2HeaderText=Address Details
When using formatter functions, the binding is automatically switched to
                    "one-way". So you can’t use a formatter function for "two-
way" scenarios, but
                    you can use data types (which will be explained in the
following steps).# E-mail
sendEmail=Send Mail
mailSubject=Hi {0}!
mailBody=How are you?
```

webapp/i18n/i18n_de.properties

```
…
# Screen titles
panel1HeaderText=Data Binding Grundlagen
panel2HeaderText=Adressdetails
# E-mail
sendEmail=E-mail versenden
mailSubject=Hallo {0}!
mailBody=Wie geht es dir?
```

And we add the missing texts to the `properties` files

Related Information

Formatting, Parsing, and Validating Data [page 854]

Step 10: Property Formatting Using Data Types

SAPUI5 provides a set of simple data types such as `Boolean`, `Currency`, `Date` and `Float`. These data types can then be applied to controls in order to ensure that the value presented on the screen is formatted correctly, and, if the field is open for input, that the value entered by the user adheres to the requirements of that data type. We will now add a new field called *Sales to Date* of type `Currency`.

Preview

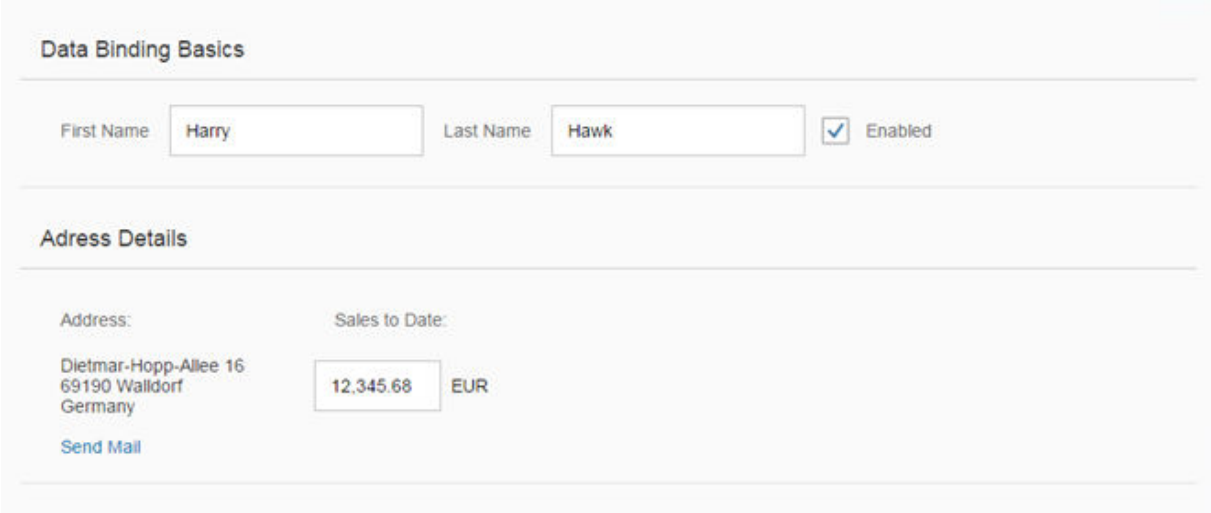


Figure 60: New *Sales to Date* input field

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 10.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/resource/ResourceModel"
], function (JSONModel, XMLView, ResourceModel) {
    "use strict";
```

```
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true,
            address: {
                street: "Dietmar-Hopp-Allee 16",
                city: "Walldorf",
                zip: "69190",
                country: "Germany"
            },
            "salesToDate" : 12345.6789,
            "currencyCode" : "EUR"
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        var oResourceBundle = new ResourceModel({
            bundleName: "sap.ui.demo.db.i18n.i18n"
        });
        sap.ui.getCore().setModel(oResourceBundle, "i18n");
        // Display the XML view called "App"
        new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
    });
});
```

We create two new model properties `salesToDate` and `currencyCode`.

webapp/view/App.view.xml

```
...
    <Panel headerText="{i18n>panel1HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <Label text="{i18n>firstName}" class="sapUiSmallMargin"/>
            <Input value="{/firstName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <Label text="{i18n>lastName}" class="sapUiSmallMargin"/>
            <Input value="{/lastName}" valueLiveUpdate="true" width="200px"
enabled="{/enabled}"/>
            <CheckBox selected="{/enabled}" text="Enabled"/>
        </content>
    </Panel>
    <Panel headerText="{i18n>panel2HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <l:HorizontalLayout>
                <l:VerticalLayout>
                    <Label class="sapUiSmallMargin" text="{i18n>address}:"/>
                    <FormattedText class="sapUiSmallMarginBegin
sapUiSmallMarginBottom" htmlText="{/address/street}&lt;br&gt;{/address/zip} {/
address/city}&lt;br&gt;{/address/country}" width="200px"/>
                    <Link class="sapUiSmallMarginBegin"
                        href="{
                            parts: [
                                '/firstName',
                                '/lastName'
                            ],
                            formatter: '.formatMail'
                        }"
                        text="{i18n>sendEmail}"/>
```

```
                </l:VerticalLayout>
                <l:VerticalLayout>
                    <Label text="{i18n>salesToDate}:" class="sapUiSmallMargin"/>
                    <Input width="200px" enabled="{/enabled}" description="{/
currencyCode}"
                        value="{
                            parts: [
                                {path: '/salesToDate'},
                                {path: '/currencyCode'}
                            ],
                            type: 'sap.ui.model.type.Currency',
                            formatOptions: {showMeasure: false}
                        }"/>
                </l:VerticalLayout>
            </l:HorizontalLayout>
        </content>
    </Panel>
</mvc:View>
```

A new pair of `Label` and `Input` elements have been created for the `salesToDate` model property. The description property of the `Input` element has been bound to the `currencyCode` model property. The value property of the `Input` element has been bound to the model properties `salesToDate` and `currencyCode`. The `{showMeasure: false}` parameter switches off the display of the currency symbol within the input field itself. This is not needed because it is being displayed using the `Input` element's description property.

webapp/i18n/i18n.properties

```
# Field labels
firstName=Vorname
lastName=Nachname
enabled=Enabled
address=Address
salesToDate=Sales to Date...
```

webapp/i18n/i18n_de.properties

```
# Field labels
firstName=Vorname
lastName=Nachname
enabled=Aktiviert
address=Adresse
salesToDate=Verk\u00e4ufe bis zum heutigen Datum
...
```

Add the missing texts to the `properties` files. Please note that special characters (non-Latin-1) have to be entered by using Unicode escape characters.

Related Information

Formatting, Parsing, and Validating Data [page 854]

Step 11: Validation Using the Message Manager

So far, we have created a currency field that can format itself correctly. The currency data type also has the ability to validate that user input adheres to to the requirements of a currency; however, data type validation functions are managed by SAPUI5, which of itself has no mechanism for reporting error messages back to the UI; therefore, we need a mechanism for reporting error messages raised by validation functions back to the user. In this step, we will connect the entire view to a feature known as the "Message Manager". Once this connection is established, any validation error messages generated based on the user input will be passed to the message manager which in turn will connect them to the appropriate view and control that caused the error.

Preview

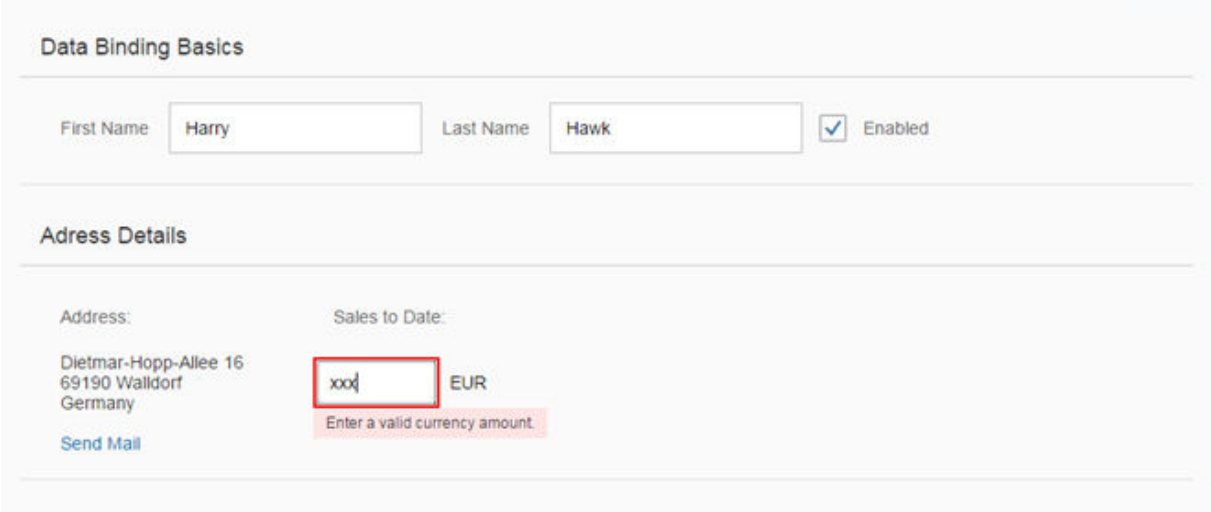


Figure 61: A message appears

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 11.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/resource/ResourceModel"
], function (JSONModel, XMLView, ResourceModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        var oModel = new JSONModel({
            firstName: "Harry",
```

```
            lastName: "Hawk",
            enabled: true,
            address: {
                street: "Dietmar-Hopp-Allee 16",
                city: "Walldorf",
                zip: "69190",
                country: "Germany"
            },
            "salesToDate": 12345.6789,
            "currencyCode": "EUR"
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        var oResourceBundle = new ResourceModel({
            bundleName: "sap.ui.demo.db.i18n.i18n"
        });
        sap.ui.getCore().setModel(oResourceBundle, "i18n");
        // Display the XML view called "App"
        var oView = new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
        // Register the view with the message manager
        sap.ui.getCore().getMessageManager().registerObject(oView, true);
        // Insert the view into the DOM
        oView.placeAt("content");
    });
});
```

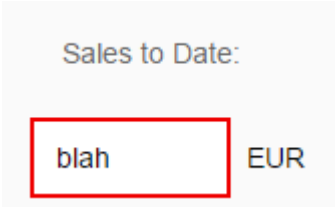
The changes to the coding are minimal:

- The XML view is now created as a named object called `oView`.
- The view object `oView` is registered with the `MessageManager`.
- Once registered, the XML view is then inserted into the DOM as before.

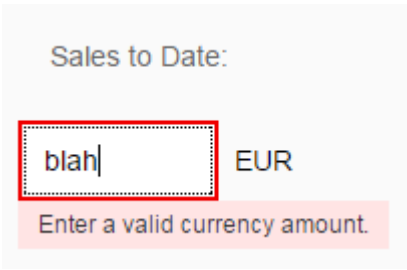
You can now enter a non-numeric value into the *Sales To Date* field and either press `Enter` or move the focus to a different UI control. This action triggers either the `onenter` or `onchange` event and then SAPUI5 executes the validation function belonging to the `sap.ui.model.type.Currency` data type.

Now that the view has been registered with the `MessageManager`, any validation error messages will be picked up by the `MessageManager`, which in turn checks its list of registered objects and then passes the error message back to the correct view for display.

Note that the field in error has a red border:



However, the error message itself will only be displayed when that particular field has focus:



Related Information

Error, Warning, and Info Messages [page 1063]

Step 12: Aggregation Binding Using Templates

Aggregation binding (or "list binding") allows a control to be bound to a list within the model data and allows relative binding to the list entries by its child controls.

It will automatically create as many child controls as are needed to display the data in the model using one of the following two approaches:

- Use template control that is cloned as many times as needed to display the data.
- Use a factory function to generate the correct control per bound list entry based on the data received at runtime.

Preview

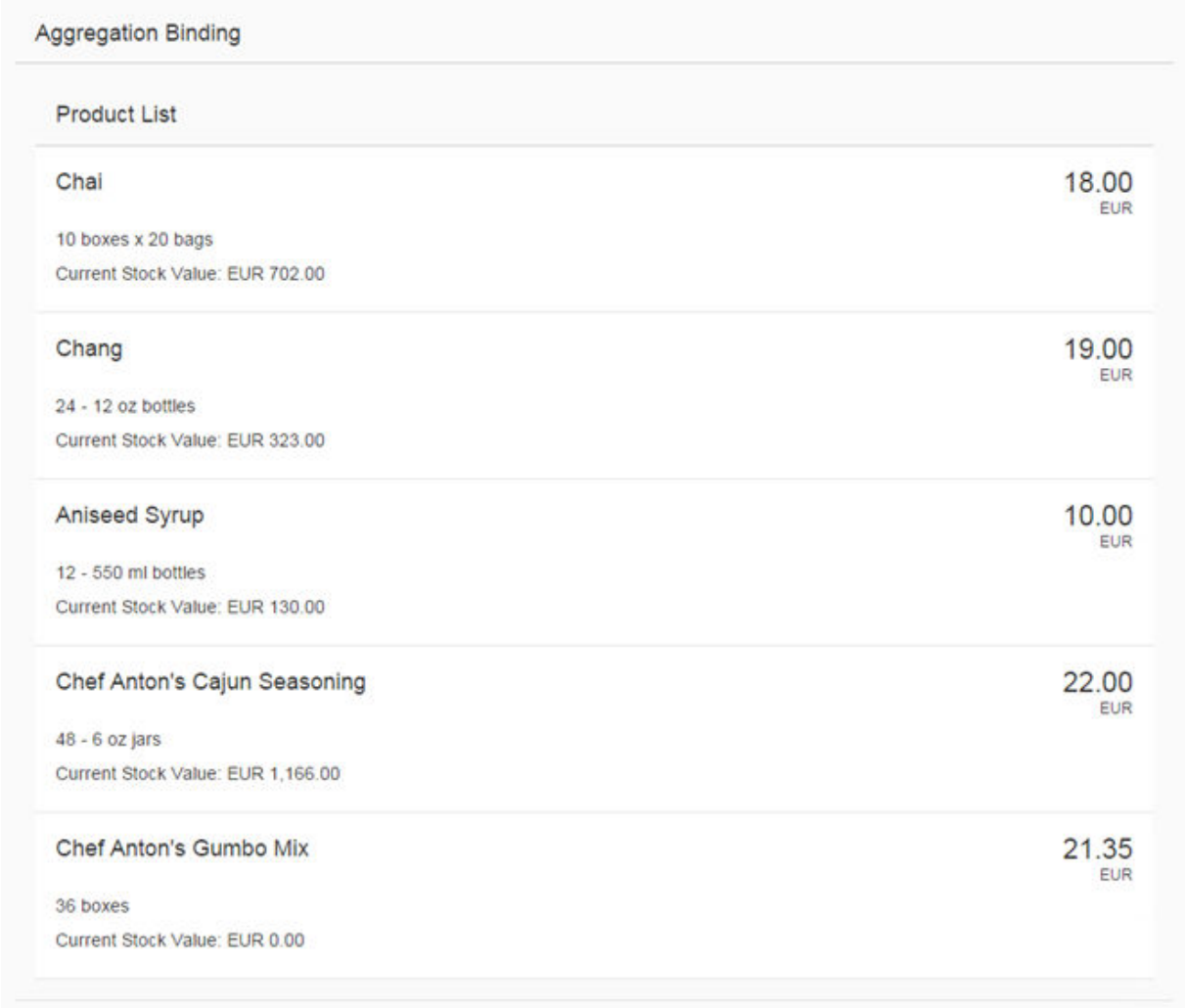


Figure 62: List with aggregation binding

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 12.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/resource/ResourceModel"
], function (JSONModel, XMLView, ResourceModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
```

```
        var oProductModel = new JSONModel();
        oProductModel.loadData("./model/Products.json");
        sap.ui.getCore().setModel(oProductModel, "products");
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true,
            address: {
                street: "Dietmar-Hopp-Allee 16",
                city: "Walldorf",
                zip: "69190",
                country: "Germany"
            },
            "salesToDate": 12345.6789,
            "currencyCode": "EUR"
        });
        // Assign the model object to the SAPUI5 core
        sap.ui.getCore().setModel(oModel);
        var oResourceBundle = new ResourceModel({
            bundleName: "sap.ui.demo.db.i18n.i18n"
        });
        sap.ui.getCore().setModel(oResourceBundle, "i18n");
        // Create the XML view called "App"
        var oView = new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        });
        // Register the view with the message manager
        sap.ui.getCore().getMessageManager().registerObject(oView, true);
        // Display the view
        oView.placeAt("content");
    });
});
```

webapp/view/App.view.xml

```
...
                    <Input width="200px" enabled="{/enabled}" description="{/
currencyCode}"
                        value="{
                            parts: [
                                {path: '/salesToDate'},
                                {path: '/currencyCode'}
                            ],
                            type: 'sap.ui.model.type.Currency',
                            formatOptions: {showMeasure: false}
                        }"/>
                </l:VerticalLayout>
            </l:HorizontalLayout>
        </content>
    </Panel>
    <Panel headerText="{i18n>panel3HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <List headerText="{i18n>productListTitle}" items="{products>/
Products}">
                <items>
                    <ObjectListItem title="{products>ProductName}"
                        number="{
                            parts: [
                                {path: 'products>UnitPrice'},
                                {path: '/currencyCode'}
                            ],
                            type: 'sap.ui.model.type.Currency',
```

```
                            formatOptions: { showMeasure: false }
                        }"
                        numberUnit="{/currencyCode}">
                        <attributes>
                            <ObjectAttribute text="{products>QuantityPerUnit}"/>
                            <ObjectAttribute title="{i18n>stockValue}"
                                text="{
                                    parts: [
                                        {path: 'products>UnitPrice'},
                                        {path: 'products>UnitsInStock'},
                                        {path: '/currencyCode'}
                                    ],
                                    formatter: '.formatStockValue'
                                }"/>
                        </attributes>
                    </ObjectListItem>
                </items>
            </List>
        </content>
    </Panel>
...
```

We add a new panel to the view.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/library",
    "sap/ui/core/Locale",
    "sap/ui/core/LocaleData",
    "sap/ui/model/type/Currency"
], function (Controller, mobileLibrary, Locale, LocaleData, Currency) {
    "use strict";
    return Controller.extend("sap.ui.demo.db.controller.App", {
        formatMail: function(sFirstName, sLastName) {
            var oBundle = this.getView().getModel("i18n").getResourceBundle();
            return mobileLibrary.URLHelper.normalizeEmail(
                sFirstName + "." + sLastName + "@example.com",
                oBundle.getText("mailSubject", [sFirstName]),
                oBundle.getText("mailBody"));
        },
        formatStockValue: function(fUnitPrice, iStockLevel, sCurrCode) {
            var sBrowserLocale =
sap.ui.getCore().getConfiguration().getLanguage();
            var oLocale = new Locale(sBrowserLocale);
            var oLocaleData = new LocaleData(oLocale);
            var oCurrency = new Currency(oLocaleData.mData.currencyFormat);
            return oCurrency.formatValue([fUnitPrice * iStockLevel, sCurrCode],
"string");
        }
    });
});
```

webapp/model/Products.json (New)

```
{ "Products": [ {
     "ProductID": 1,
```

```
      "ProductName": "Chai",
      "SupplierID": 1,
      "CategoryID": 1,
      "QuantityPerUnit": "10 boxes x 20 bags",
      "UnitPrice": "18.0000",
      "UnitsInStock": 39,
      "UnitsOnOrder": 0,
      "ReorderLevel": 10,
      "Discontinued": false
    }, {
      "ProductID": 2,
      "ProductName": "Chang",
      "SupplierID": 1,
      "CategoryID": 1,
      "QuantityPerUnit": "24 - 12 oz bottles",
      "UnitPrice": "19.0000",
      "UnitsInStock": 17,
      "UnitsOnOrder": 40,
      "ReorderLevel": 25,
      "Discontinued": true
    }, {
      "ProductID": 3,
      "ProductName": "Aniseed Syrup",
      "SupplierID": 1,
      "CategoryID": 2,
      "QuantityPerUnit": "12 - 550 ml bottles",
      "UnitPrice": "10.0000",
      "UnitsInStock": 0,
      "UnitsOnOrder": 70,
      "ReorderLevel": 25,
      "Discontinued": false
    }, {
      "ProductID": 4,
      "ProductName": "Chef Anton's Cajun Seasoning",
      "SupplierID": 2,
      "CategoryID": 2,
      "QuantityPerUnit": "48 - 6 oz jars",
      "UnitPrice": "22.0000",
      "UnitsInStock": 53,
      "UnitsOnOrder": 0,
      "ReorderLevel": 0,
      "Discontinued": false
    }, {
      "ProductID": 5,
      "ProductName": "Chef Anton's Gumbo Mix",
      "SupplierID": 2,
      "CategoryID": 2,
      "QuantityPerUnit": "36 boxes",
      "UnitPrice": "21.3500",
      "UnitsInStock": 0,
      "UnitsOnOrder": 0,
      "ReorderLevel": 0,
      "Discontinued": true
    }]
  }
```

We now use a new JSON model file for product data.

webapp/i18n/i18n.properties

```
...
# Screen titles
panel1HeaderText=Data Binding Basics
```

```
panel2HeaderText=Address Details
panel3HeaderText=Aggregation Binding
# Invoice List
invoiceListTitle=Invoices
statusA=New
statusB=In Progress
statusC=Done
# Product list
productListTitle=Product List
stockValue=Current Stock Value
```

webapp/i18n/i18n_de.properties

```
...
# Screen titles
panel1HeaderText=Data Binding Basics
panel2HeaderText=Adressdetails
panel3HeaderText=Aggregation Binding
# Invoice List
invoiceListTitle=Rechnungen
statusA=Neu
statusB=Laufend
statusC=Abgeschlossen
# Product list
productListTitle=Artikelliste
stockValue=Lagerbestand Wert
```

We add the missing texts.

Related Information

List Binding (Aggregation Binding) [page 828]

Step 13: Element Binding

Now we want to do something with that newly generated list. In most cases you will use a list to allow the selection of an item and then show the details of that item elsewhere. In order to achieve this, we use a form with relatively bound controls and bind it to the selected entity via element binding.

Preview

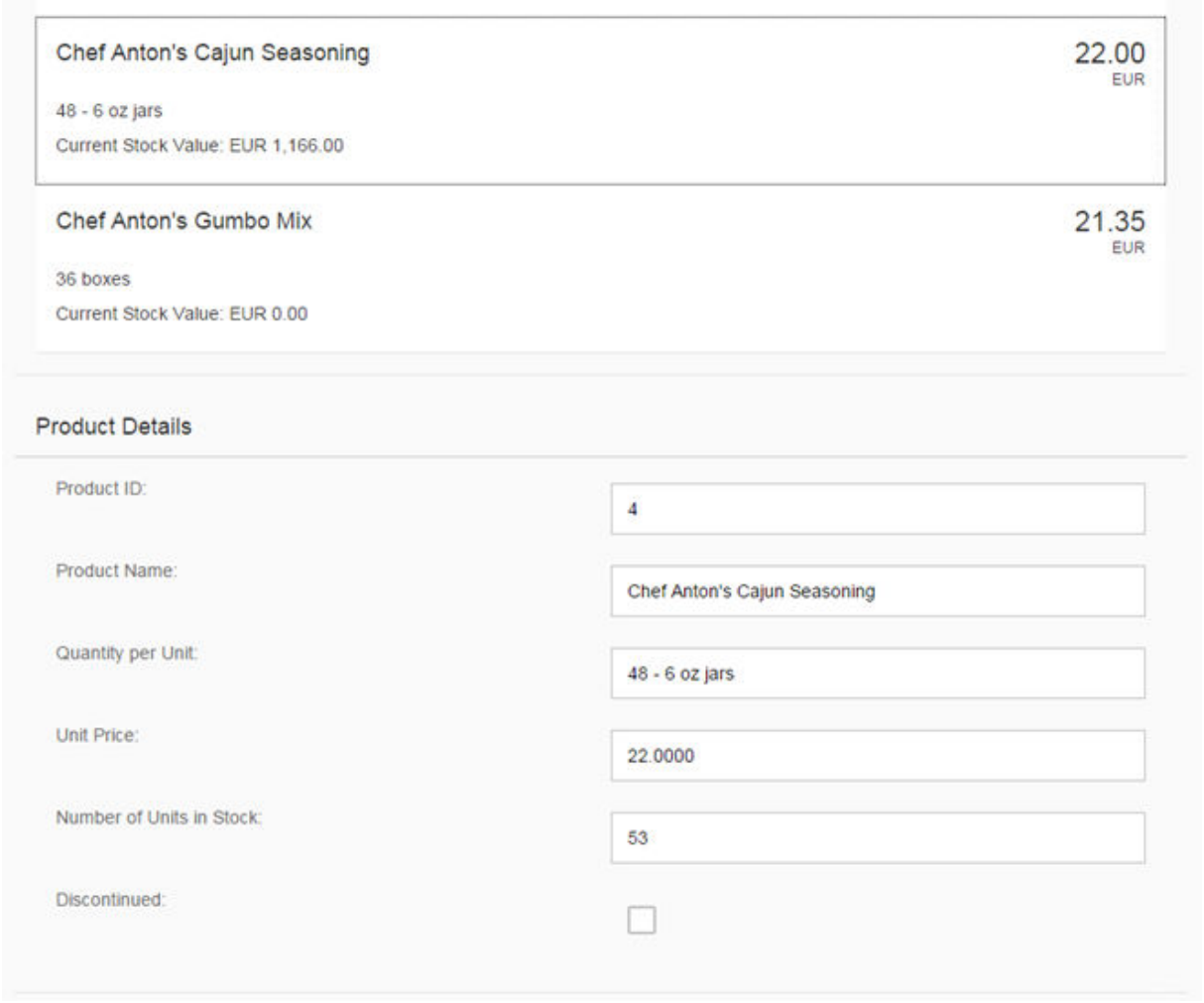


Figure 63: Element binding implemented, product details displayed per item

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 13.

webapp/view/App.view.xml

```
...
        </items>
      </List>
    </content>
  </Panel>
  <Panel id="productDetailsPanel" headerText="{i18n>panel4HeaderText}"
class="sapUiResponsiveMargin" width="auto">
    <l:Grid defaultSpan="L3 M6 S12" containerQuery="true">
        <Label text="{i18n>ProductID}:" />
        <Input value="{products>ProductID}" />

        <Label text="{i18n>ProductName}:" />
        <Input value="{products>ProductName}" />

        <Label text="{i18n>QuantityPerUnit}:" />
        <Input value="{products>QuantityPerUnit}" />

        <Label text="{i18n>UnitPrice}:" />
        <Input value="{products>UnitPrice}" />

        <Label text="{i18n>UnitsInStock}:" />
        <Input value="{products>UnitsInStock}" />

        <Label text="{i18n>Discontinued}:" />
        <CheckBox selected="{products>Discontinued}" />
    </l:Grid>
  </Panel>
</mvc:View>
```

Now we have an empty form. In order to fill this form with data, we will bind the whole panel to the path of the element which we clicked in the list. We need to add a `press`-event handler to the items in the list.

webapp/view/App.views.xml

```
...
  <Panel headerText="{i18n>panel4HeaderText}" class="sapUiResponsiveMargin"
width="auto">
    <content>
      <List headerText="{i18n>productListTitle}" items="{products>/Products}">
        <items>
          <ObjectListItem
              press=".onItemSelected"
            type="Active"
              title="{products>ProductName}"
              number="{ parts: [{path: 'products>UnitPrice'},
                                {path: '/currencyCode'}],
                        type: 'sap.ui.model.type.Currency',
                        formatOptions: { showMeasure: false }
                      }"
              numberUnit="{/currencyCode}">
              <attributes>
...
```

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/library",
    "sap/ui/core/Locale",
    "sap/ui/core/LocaleData",
    "sap/ui/model/type/Currency"
], function (Controller, mobileLibrary, Locale, LocaleData, Currency) {
    "use strict";
    return Controller.extend("sap.ui.demo.db.controller.App", {
        formatMail: function(sFirstName, sLastName) {
            var oBundle = this.getView().getModel("i18n").getResourceBundle();
            return mobileLibrary.URLHelper.normalizeEmail(
                sFirstName + "." + sLastName + "@example.com",
                oBundle.getText("mailSubject", [sFirstName]),
                oBundle.getText("mailBody"));
        },
        formatStockValue: function(fUnitPrice, iStockLevel, sCurrCode) {
            var sBrowserLocale =
sap.ui.getCore().getConfiguration().getLanguage();
            var oLocale = new Locale(sBrowserLocale);
            var oLocaleData = new LocaleData(oLocale);
            var oCurrency = new Currency(oLocaleData.mData.currencyFormat);
            return oCurrency.formatValue([fUnitPrice * iStockLevel, sCurrCode],
"string");
        },
        onItemSelected: function(oEvent) {
            var oSelectedItem = oEvent.getSource();
            var oContext = oSelectedItem.getBindingContext("products");
            var sPath = oContext.getPath();
            var oProductDetailPanel = this.byId("productDetailsPanel");
            oProductDetailPanel.bindElement({ path: sPath, model: "products" });
        }
    });
});
```

In the controller, we bind the newly created panel to the correct item whenever it is pressed.

We can now click on an element in the list and see its details in the panel below. We can even edit these details and these changes are directly shown in the list because we use two-way binding.

> **i** Note
>
> Element bindings can also be relative to its parent context.

webapp/i18n/i18n.properties

```
...
# Screen titles
panel1HeaderText=Data Binding Basics
panel2HeaderText=Address Details
panel3HeaderText=Aggregation Binding
panel4HeaderText=Product Details
# Product list
productListTitle=Product List
stockValue=Current Stock Value
# Product Details
ProductID=Product ID
```

```
ProductName=Product Name
QuantityPerUnit=Quantity per Unit
UnitPrice=Unit Price
UnitsInStock=Number of Units in Stock
Discontinued=Discontinued
```

webapp/i18n/i18n_de.properties

```
# Screen titles
panel1HeaderText=Data Binding Grundlagen
panel2HeaderText=Adressdetails
panel3HeaderText=Aggregation Binding
panel4HeaderText=Produktdetails

# Product list
productListTitle=Artikelliste
stockValue=Lagerbestand Wert
# Product Details
ProductID=Produkt-ID
ProductName=Produktname
QuantityPerUnit=Menge pro Einheit
UnitPrice=Preis der Einheit
UnitsInStock=Lagerbestand
Discontinued=Eingestellt
```

Add the missing texts to the `properties` files.

Related Information

Context Binding (Element Binding) [page 824]

Step 14: Expression Binding

Expression binding allows you to display a value on the screen that has been calculated from values found in some model object. This way simple formatting or calculations can be inserted directly into the data binding string. In this example, we will change the color of the price depending on whether it is above or below some arbitrary threshold. The threshold value is also stored in the JSON model.

Preview

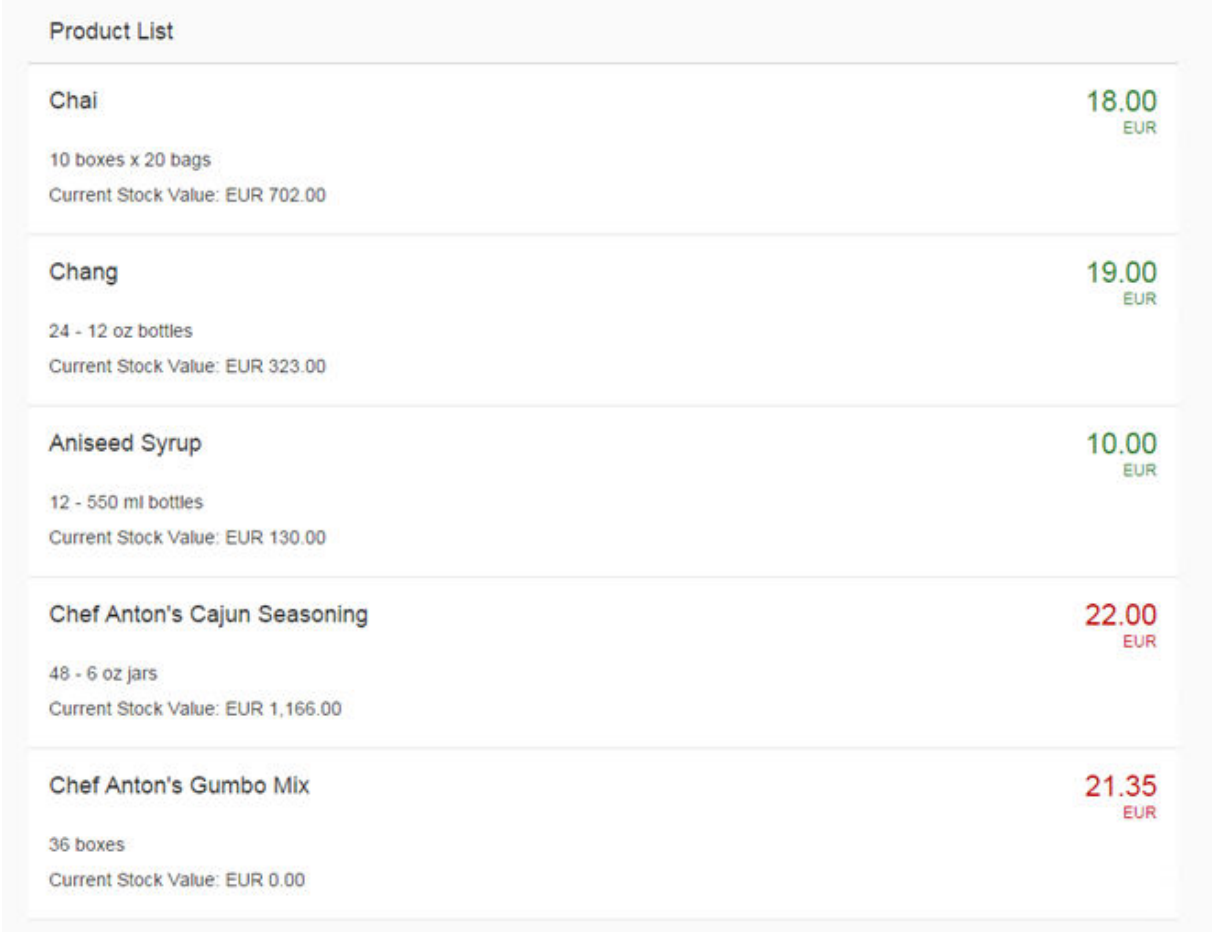


Figure 64: Values formatted

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 14.

webapp/view/App.view.xml

```
...
        </content>
    </Panel>
    <Panel headerText="{i18n>panel3HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <List headerText="{i18n>productListTitle}" items="{products>/
Products}">
                <items>
                    <ObjectListItem
                        press=".onItemSelected"
                        type="Active"
```

```
                        title="{products>ProductName}"
                        number="{
                            parts: [
                                {path: 'products>UnitPrice'},
                                {path: '/currencyCode'}
                            ],
                            type: 'sap.ui.model.type.Currency',
                            formatOptions: { showMeasure: false }
                        }"
                        numberUnit="{/currencyCode}"
                        numberState="{= ${products>UnitPrice} > ${/
priceThreshold} ? 'Error' : 'Success' }">
                        <attributes>
                            <ObjectAttribute text="{products>QuantityPerUnit}"/>
                            <ObjectAttribute title="{i18n>stockValue}"
                                text="{
                                    parts: [
                                        {path: 'products>UnitPrice'},
                                        {path: 'products>UnitsInStock'},
                                        {path: '/currencyCode'}
                                    ],
                                    formatter: '.formatStockValue'
                                }"/>
                        </attributes>
                    </ObjectListItem>
                </items>
            </List>
        </content>
    </Panel>
...
```

In the XML view, we add a new `numberState` property to the `ObjectListItem` element within the `List`. The value of this property is an expression that will be evaluated for each item.

webapp/index.js

```
sap.ui.require([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/XMLView",
    "sap/ui/model/resource/ResourceModel"
], function (JSONModel, XMLView, ResourceModel) {
    "use strict";
    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
        var oProductModel = new JSONModel();
        oProductModel.loadData("./model/Products.json");
        sap.ui.getCore().setModel(oProductModel, "products");
        var oModel = new JSONModel({
            firstName: "Harry",
            lastName: "Hawk",
            enabled: true,
            address: {
                street: "Dietmar-Hopp-Allee 16",
                city: "Walldorf",
                zip: "69190",
                country: "Germany"
            },
            "salesToDate": 12345.6789,
            "priceThreshold": 20,
            "currencyCode": "EUR"
        });
        // Assign the model object to the SAPUI5 core
```

```
        sap.ui.getCore().setModel(oModel);
        var oResourceBundle = new ResourceModel({
            bundleName: "sap.ui.demo.db.i18n.i18n"
        });
        sap.ui.getCore().setModel(oResourceBundle, "i18n");
        // Display the XML view called "App"
        var oView = new XMLView({
            viewName: "sap.ui.demo.db.view.App"
        }).placeAt("content");
        // Register the view with the message manager
        sap.ui.getCore().getMessageManager().registerObject(oView, true);
        // Insert the view into the DOM
        oView.placeAt("content");
    });
});
```

We add a new property called `priceThreshold` against which each invoice value will be checked.

As a result of binding an expression to the `numberState` property, the error status (color) of the price field will change depending on the invoice value.

Look at the following two expressions:

- `numberState="{= ${products>UnitPrice} > ${/priceThreshold} ? 'Error' : 'Success' }"`
- `numberState="{= ${products>UnitPrice} <= ${/priceThreshold} ? 'Success' : 'Error' }"`

Can you see why one of these expressions will work, and the other will not?

Logically, both expressions are identical; yet the first one works, and the second does not: it produces only an empty screen and an "Invalid XML" message in the browser's console... Hmmm, what's going on here?

In order to understand why this situation occurs, you must understand how XML files are parsed.

When an XML file is parsed, certain characters have a special (that is, high priority) meaning to the XML parser. When such characters are encountered, they are **always** interpreted to be part of the XML definition itself and not part of any other content that might exist within the XML document.

As soon as the XML parser encounters one of these high-priority characters (in this case, a less-than (<) character), it will always be interpreted as the start of a new XML tag – irrespective of any other meaning that character might have within the context of the expression. This is known as a **syntax collision**.

In this case, the collision occurs between the syntax of XML and the syntax of the JavaScript-like expression language used by SAPUI5.

Therefore, this statement fails because the less-than character is interpreted as the start of an XML tag:
`numberState="{= ${products>UnitPrice} <= ${/priceThreshold} ? 'Success' : 'Error' }"`

This particular problem can be avoided in one of two ways:

- Reverse the logic of the condition (use "greater than or equal to" instead of "less than")
- Use the escaped value for the less-than character: `numberState="{= ${products>UnitPrice} <= ${/priceThreshold} ? 'Success' : 'Error' }"`

Since the use of an escaped character is not so easy to read, the preferred approach is to reverse the logic of the condition and use a greater-than character instead.

The ampersand (`&`) character also has a high priority meaning to the XML parser. This character will always be interpreted to mean "The start of an escaped character". So if you wish to use the Boolean `AND` operator (`&&`) in a condition, you must escape both ampersand characters (`&&`).

Related Information

Expression Binding [page 845]

Step 15: Aggregation Binding Using a Factory Function

Instead of hard-coding a single template control, we use a factory function to generate different controls based on the data received at runtime. This approach is much more flexible and allows complex or heterogeneous data to be displayed.

Preview

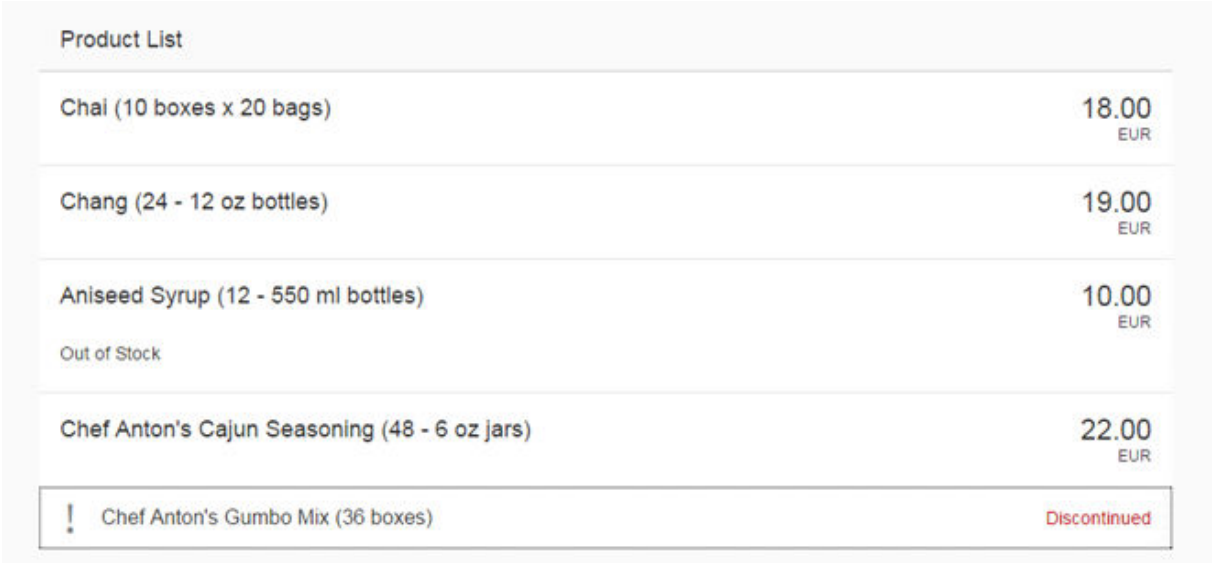


Figure 65: Controls generated based on data

Coding

You can view and download all files in the Demo Kit at Data Binding - Step 15.

webapp/view/App.view.xml

```
...
    <Panel headerText="{i18n>panel3HeaderText}" class="sapUiResponsiveMargin"
width="auto">
        <content>
            <List
                id="ProductList"
                headerText="{i18n>productListTitle}"
                items="{
                    path: 'products>/Products',
                    factory: '.productListFactory'
                }">
                <dependents>
                    <core:Fragment
fragmentName="sap.ui.demo.db.view.ProductSimple" type="XML"/>
                    <core:Fragment
fragmentName="sap.ui.demo.db.view.ProductExtended" type="XML"/>
                </dependents>
            </List>
        </content>
    </Panel>
...
```

The `List` XML element that previously held the product list is now reduced simply to a named, but otherwise empty placeholder. Without a factory function to populate it, this `List` would always remain empty.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/library",
    "sap/ui/core/Locale",
    "sap/ui/core/LocaleData",
    "sap/ui/model/type/Currency",
    "sap/m/ObjectAttribute"
], function (Controller, mobileLibrary, Locale, LocaleData, Currency,
ObjectAttribute) {
    "use strict";
    return Controller.extend("sap.ui.demo.db.controller.App", {
        formatMail: function(sFirstName, sLastName) {
            var oBundle = this.getView().getModel("i18n").getResourceBundle();
            return mobileLibrary.URLHelper.normalizeEmail(
                sFirstName + "." + sLastName + "@example.com",
                oBundle.getText("mailSubject", [sFirstName]),
                oBundle.getText("mailBody"));
        },
        formatStockValue : function(fUnitPrice, iStockLevel, sCurrCode) {
            var sBrowserLocale =
sap.ui.getCore().getConfiguration().getLanguage();
            var oLocale = new Locale(sBrowserLocale);
            var oLocaleData = new LocaleData(oLocale);
            var oCurrency = new Currency(oLocaleData.mData.currencyFormat);
            return oCurrency.formatValue([fUnitPrice * iStockLevel, sCurrCode],
"string");
        },
        onItemSelected : function(oEvent) {
            var oSelectedItem = oEvent.getSource();
            var oContext = oSelectedItem.getBindingContext("products");
            var sPath = oContext.getPath();
```

```
            var oProductDetailPanel = this.byId("productDetailsPanel");
            oProductDetailPanel.bindElement({ path: sPath, model: "products" });
        },
        productListFactory : function(sId, oContext) {
            var oUIControl;

            // Decide based on the data which dependent to clone
            if (oContext.getProperty("UnitsInStock") === 0 &&
oContext.getProperty("Discontinued")) {
                // The item is discontinued, so use a StandardListItem
                oUIControl = this.byId("productSimple").clone(sId);
            } else {
                // The item is available, so we will create an ObjectListItem
                oUIControl = this.byId("productExtended").clone(sId);

                // The item is temporarily out of stock, so we will add a status
                if (oContext.getProperty("UnitsInStock") < 1) {
                    oUIControl.addAttribute(new ObjectAttribute({
                        text : {
                            path: "i18n>outOfStock"
                        }
                    }));
                }
            }

            return oUIControl;
        }
    });
});
```

In the `App` controller, we create a new function called `productListFactory`. A factory function returns a control for the associated binding context, similar to the XML templates we have defined in the previous steps. The types of controls returned by this factory function must suit the items aggregation of the `sap.m.List` object. In this case, we return either a `StandardListItem` or an `ObjectListItem` based on the data stored in the context of the item to be created.

We decide which type of control to return by checking the current stock level and whether or not the product has been discontinued. For both options, we prepare and load an XML fragment so that we can define the view logic declaratively and assign the current controller. If the stock level is zero and the product has also been discontinued, then we use the `ProductSimple` XML fragment, otherwise the `ProductExtended` XML fragment.

The XML fragments need to be loaded only once for each case, so we create a Singleton by storing a helper variable on the controller and only loading it once. For each item of the list, we clone the corresponding control stored on the controller. This protected method creates a fresh copy of a control that we can bind to the context of the list item. Please note: In a factory function, you are responsible for the life cycle of the control you create.

If the product is not discontinued but the stock level is zero, we are temporarily out of stock. In this case, we add a single `ObjectAttribute` that adds the *Out of Stock* message to the control using JavaScript. Similar to declarative definitions in the XML view or fragments, we can bind properties using data binding syntax. In this case, we bind the text to a property in the resource bundle. Since the `Attribute` is a child of the list item, it has access to all assigned models and the current binding context.

Finally, we return the control that is displayed inside the list.

webapp/view/ProductSimple.fragment.xml (new)

```
<core:FragmentDefinition
    xmlns="sap.m"
    xmlns:core="sap.ui.core">
    <StandardListItem
        id="productSimple"

        icon="sap-icon://warning"
        title="{products>ProductName} ({products>QuantityPerUnit})"
        info="{i18n>Discontinued}"
        type="Active"
        infoState="Error"
        press=".onItemSelected">
    </StandardListItem>
</core:FragmentDefinition>
```

The XML fragment defines a `StandardListItem` that is used if the stock level is zero and the product has also been discontinued. This is our simple use case where we just define a warning icon and a *Product Discontinued* message in the `info` property.

webapp/view/ProductExtended.fragment.xml (new)

```
<core:FragmentDefinition
    xmlns="sap.m"
    xmlns:core="sap.ui.core">
    <ObjectListItem
        id="productExtended"
        title="{products>ProductName} ({products>QuantityPerUnit})"
        number="{
            parts: [
                {path: 'products>UnitPrice'},
                {path: '/currencyCode'}
            ],
            type: 'sap.ui.model.type.Currency',
            formatOptions : {
                showMeasure : false
            }
        }"
        type="Active"
        numberUnit="{/currencyCode}"
        press=".onItemSelected">
    </ObjectListItem>
</core:FragmentDefinition>
```

In our extended use case, we create an `ObjectListItem` to display more details of the product. The properties are bound to the fields of the current data binding context and therefore can use types, formatters, and all handlers that are defined in the assigned controller.

However, more complex logic can't be defined declaratively in XML. Therefore, when the stock level is zero, we add a single `ObjectAttribute` that displays the *Out of Stock* message in the controller using JavaScript.

webapp/i18n/i18n.properties

```
...
# Product Details
...
outOfStock=Out of Stock
```

webapp/i18n/i18n_de.properties

```
...
# Product Details
...
outOfStock=Nicht vorr\u00e4tig
```

We add the missing texts to the `properties` files.

That's all - you completed the Data Binding tutorial!

Related Information

List Binding (Aggregation Binding) [page 828]
XML Fragments [page 1005]
Using Factory Functions [page 831]

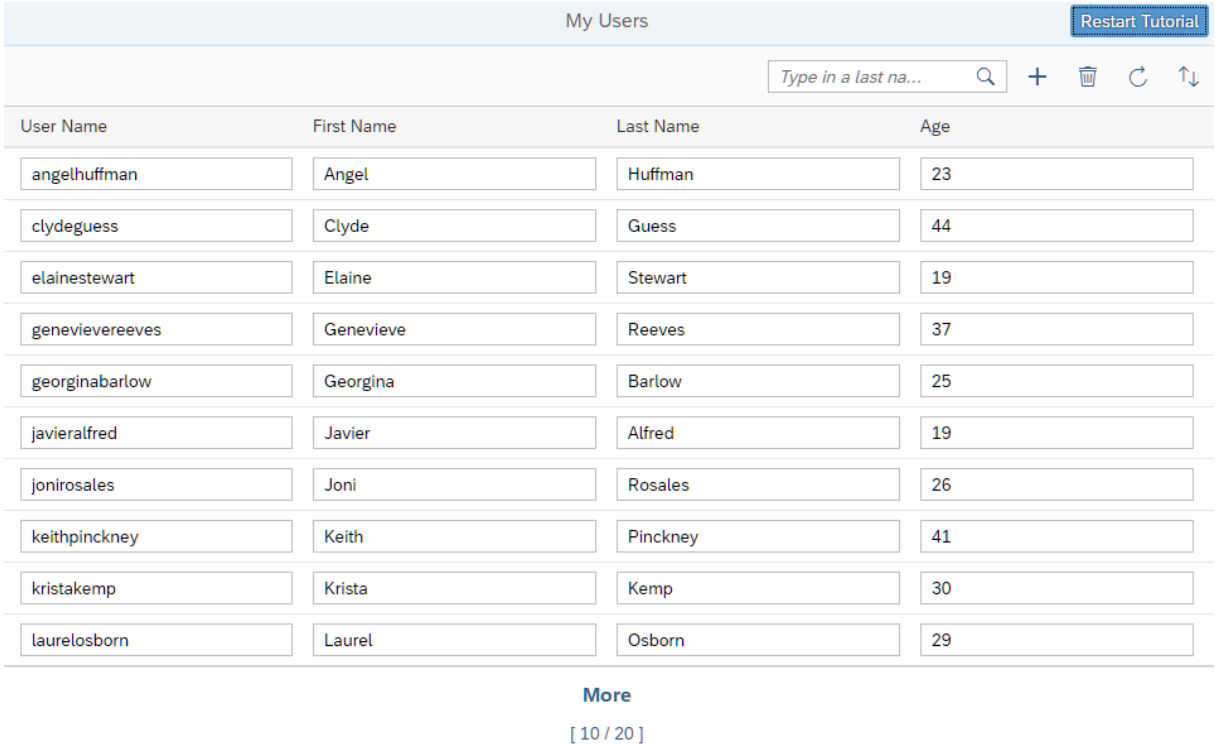
OData V4

In this tutorial, we explore how features of OData V4 can be used in SAPUI5. We write a small app that consumes data from an OData V4 service to understand how to access, modify, aggregate, and filter data in an OData V4 model.

OData is a standard protocol for creating and consuming data by using simple HTTP and REST APIs for create, read, update, delete (CRUD) operations.

We start with an initial app that simply retrieves data from an OData V4 service and displays it as a plain list.

Preview



→ Tip

You don't have to do all tutorial steps sequentially, you can jump directly to any step you want. In each step, download the code from the previous step, copy it to your workspace, and make sure that the application runs by calling the `webapp/index.html` file.

You can view and download the samples for all steps in the Demo Kit at OData V4. Depending on your development environment you might have to adjust resource paths and configuration entries.

For more information, check the following sections of the tutorial overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Related Information

OData Standard Protocol
OData V4 Model [page 918]
Basic Tutorial on the *OData* Home Page

Step 1: The Initial App

We start by setting up a simple app that loads data from an OData service and displays it in a table. We use a mock server to simulate requests to and responses from the service.

The structure and data model created in this step will be used throughout this tutorial to illustrate the OData V4 features in SAPUI5.

Preview

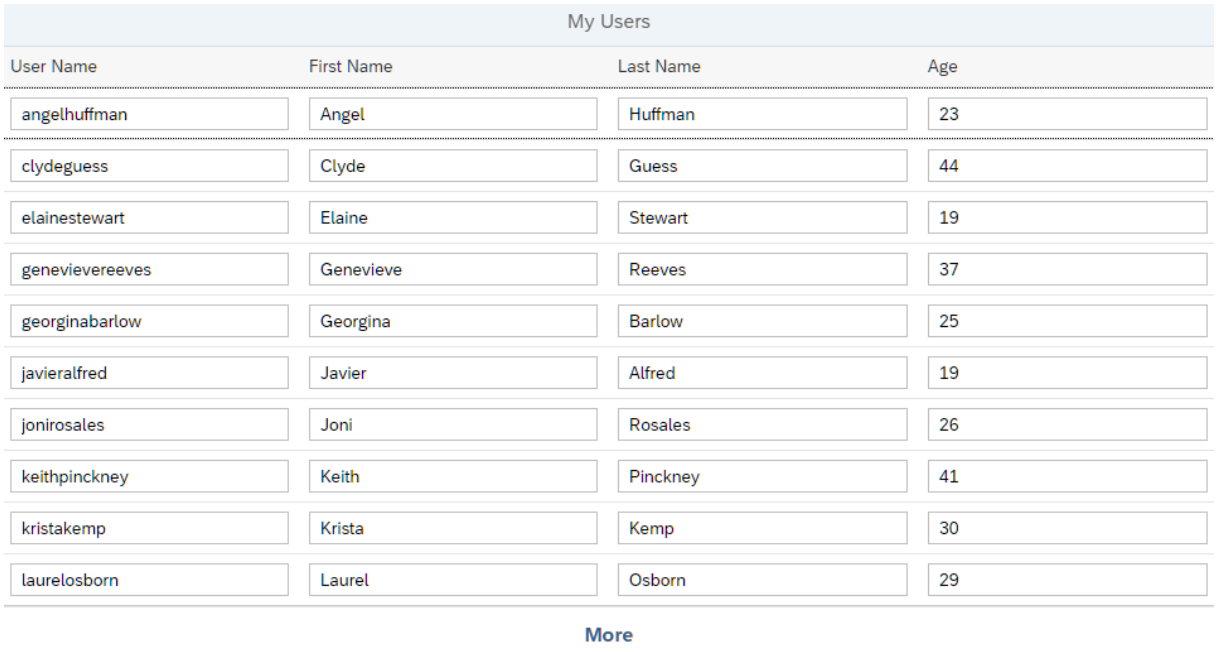


Figure 66: Initial app with a simple table

Setup

To set up your project for this tutorial, download the files at OData V4 - Step 1. Copy or import the code to your workspace and make sure that the application runs by calling the `webapp/index.html` file.

Depending on your development environment, you might have to adjust resource paths and configuration entries. The project structure and the files provided with this tutorial are explained in detail in the Walkthrough [page 69] tutorial.

You should now have the following files:

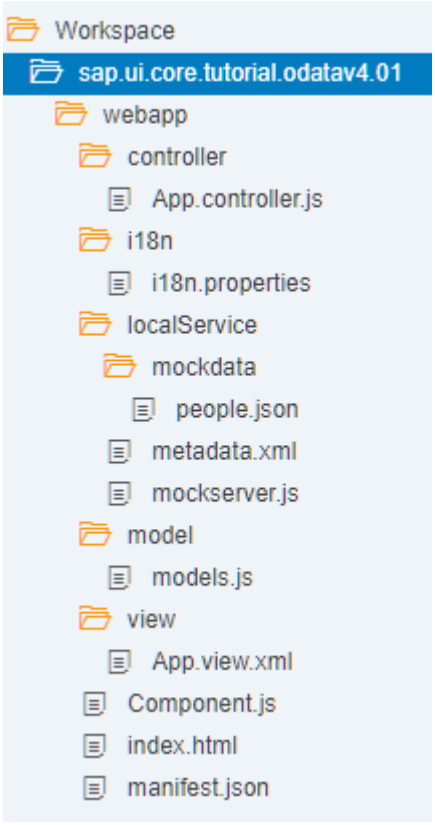


Figure 67: Folder structure with downloaded files

The Initial App

The downloaded code includes an app that displays a table containing a table of users. For performance reasons, the table only loads 10 users at a time. More data can be retrieved by using the *More* button at the bottom of the page.

During the implementation of the app, we use local mock data so that we can concentrate on the application logic without dealing with back-end readiness or connectivity issues. We use the *TripPin* sample service as a "real" OData service.

The most important files are the following:

`webapp/index.html`

This file defines the home page of the app. It contains the bootstrap script and tells the runtime where to find our custom resources. It also initializes the mock server that intercepts all requests to the real *TripPin* service and sends back mock responses.

`webapp/manifest.json`

The `manifest.json` descriptor file contains the app configuration. In the `sap.app` section, the OData V4 service is configured as the default service:

```
"dataSources": {
    "default": {
        "uri": "https://services.odata.org/TripPinRESTierService/(S(id))/",
        "type": "OData",
```

```
        "settings": {
            "odataVersion": "4.0"
        }
    }
}
```

Mock server (`webapp/localService/*`)

> **i** Note
>
> The mock server included in this tutorial is only meant to support the features needed in this tutorial. Currently, there is no "general-purpose mock server" for application development available with OData V4 (like there is for OData V2).

The `mockserver.js` file contains the implementation of the mock server. It is quite simple since the mock server is only used to simulate certain types of requests to the *TripPin* service.

The `metadata.xml` file contains the service metadata that includes, for example, entity types and entity sets. Those define the possible requests as well as the structure of responses.

To be able to add data to the emulated OData responses, we have to store the entities for each entity type we use in a JSON file: The `people.json` file contains some data that is used for the mock service responses.

In this tutorial, we only use the entity type `Person` of the *TripPin* service. The entities of type `Person` are collected in the entity set `People`. Each `Person` has a key property `UserName` and the properties `Age`, `FirstName`, and `LastName`.

Related Information

OData Reference Services including *TripPin*
Bootstrapping: Loading and Initializing [page 692]
Descriptor for Applications, Components, and Libraries [page 734]

Step 2: Data Access and Client-Server Communication

In this step, we see how the `Table` that is bound to the `People` entity set initially requests its data, and how the data can be refreshed. We use the *Console* tab in the browser developer tools to monitor the communication between the browser and the server. We see the initial request as well as the requests for refreshing the data.

Preview

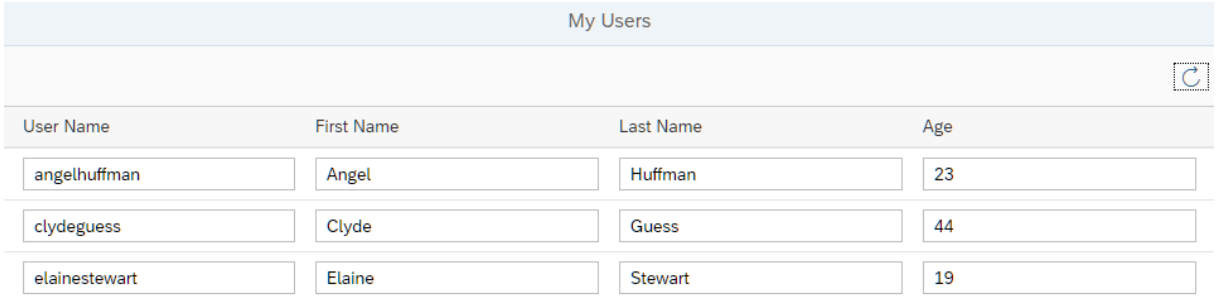


Figure 68: App with a toolbar that contains a *Refresh* button

Coding

You can view and download all files at OData V4 - Step 2.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/m/MessageBox",
    "sap/ui/model/json/JSONModel"
], function (Controller, MessageToast, MessageBox, JSONModel) {
    "use strict";
    return Controller.extend("sap.ui.core.tutorial.odatav4.controller.App", {
        onInit : function () {
            var oJSONData = {
                busy : false
            };
            var oModel = new JSONModel(oJSONData);
            this.getView().setModel(oModel, "appView");
        },

        onRefresh : function () {
            var oBinding = this.byId("peopleList").getBinding("items");

            if (oBinding.hasPendingChanges()) {
                MessageBox.error(this._getText("refreshNotPossibleMessage"));
                return;
            }
            oBinding.refresh();
            MessageToast.show(this._getText("refreshSuccessMessage"));
```

```
        },

        _getText : function (sTextId, aArgs) {
            return
this.getOwnerComponent().getModel("i18n").getResourceBundle().getText(sTextId,
aArgs);

        }
    });
});
```

We add the event handler `onRefresh` to the controller. In this method, we retrieve the current data binding of the table. If the binding has unsaved changes, we display an error message, otherwise we call `refresh()` and display a success message.

> **i** Note
>
> At this stage, our app cannot have unsaved changes. We will change this in Step 6.

We also add the private method `_getText` to retrieve translatable texts from the resource bundle (`i18n` model).

webapp/view/App.view.xml

```
...
<Page title="{i18n>peoplePageTitle}">
    <content>
        <Table
            id="peopleList"
            growing="true"
            growingThreshold="10"
            items="{
                path: '/People'
            }">
            <headerToolbar>
                <OverflowToolbar>
                    <content>
                        <ToolbarSpacer/>
                        <Button
                            id="refreshUsersButton"
                            icon="sap-icon://refresh"
                            tooltip="{i18n>refreshButtonText}"
                            press=".onRefresh"/>
                    </content>
                </OverflowToolbar>
            </headerToolbar>
            <columns>
...
```

We add the `headerToolbar` with a single `Button` to the `Table`. The button has a `press` event to which we attach an event handler called `onRefresh`.

webapp/i18n/i18n.properties

```
# App Descriptor
...
# Toolbar
#XTOL: Tooltip for refresh data
refreshButtonText=Refresh Data
# Table Area
...
# Messages
#XMSG: Message for refresh failed
refreshNotPossibleMessage=Before refreshing, please save or revert your changes

#XMSG: Message for refresh succeeded
refreshSuccessMessage=Data refreshed
```

We add the tooltip and message texts to the `properties` file.

Under the Hood

To get more insight into the client-server communication, we open the *Console* tab of the browser developer tools and then reload the app.

> **i Note**
>
> To monitor the client-server communication in a productive app, you would use the *Network* tab of the developer tools.
>
> In this tutorial, we are using a mock server instead of a real OData service so that we can execute the code in every environment. The mock server does not generate any network traffic, so we use the *Console* tab to monitor the communication.
>
> If you want to switch to the real service, do the following:
>
> 1. In the `index.html` file, remove the line `data-sap-ui-oninit="module:sap/ui/core/tutorial/odatav4/initMockServer"`.
> 2. Check the URI of the default data source in the `manifest.json` file. Depending on the environment, change it to something that avoids cross-origin resource sharing (CORS) problems. For more information, see Request Fails Due to Same-Origin Policy (Cross-Origin Resource Sharing - CORS) [page 1391]

We search for the following mock server requests:

- http://services.odata.org/TripPinRESTierService/(S(id))/$metadata
  This first request fetches the metadata that describes the entities of the service (see also OData Version 4.0. Part 3: Common Schema Definition Language (CSDL) Plus Errata 03).
  The server responds with an XML file that describes the entities, for example, entity type `"Person"` has several properties such as `UserName`, `FirstName`, `LastName`, and `Age`.

  > **i Note**
  >
  > The URL contains the session ID `(S(id))`. Since the public *TripPin* service can be used by multiple persons at the same time, the session ID separates read and write requests from different sources. You

> could use a different ID or request the service without a specified session ID. In the latter case, you will get a response with a new, random session ID.

- http://services.odata.org/TripPinRESTierService/(S(id))/People?$select=Age,FirstName,LastName,UserName&$skip=0&$top=10 .
  The second request fetches the first 10 entities from the OData service. The `growingThreshold="10"` setting in the implementation of the `Table` control in the `App.view.xml` file defines that only 10 entities are fetched at the same time from the `'/people'` path. Further data is only loaded when requested from the user interface (`growing="true"`). Therefore, there are only 10 entities requested at the same time by using `$skip=0&$top=10` (see System Query Option $top and $skip in the Basic Tutorial on the OData home page.)
  This request explicitly lists the fields that should be included in the response by using the `$select` query option. Although the *TripPin* service has more fields in its `People` entity set, only those four are included in the response. This is a feature of the OData V4 Model called "automatic determination of `$select`", or "auto-`$select`". It helps restricting the size of responses to what is really needed. The `ODataModel` computes the required fields from binding paths specified for controls. This feature is not active by default. In our case, this is activated by setting the `autoExpandSelect` property to `true` when instantiating the model in the `manifest.json` descriptor file .

Related Information

Bindings [page 922]
API Reference: `sap.ui.model.odata.v4.ODataMetaModel`
API Reference: `sap.ui.model.odata.v4.ODataListBinding.refresh`
Troubleshooting Tutorial Step 1: Browser Developer Tools [page 196]

Step 3: Automatic Data Type Detection

In this step, we use the automatic data type detection of the OData V4 model to parse, validate, and format user entries. The service metadata contains type information for the properties of each entity.

The OData V4 Model utilizes this information to compute the corresponding SAPUI5 type, including constraints, and sets this type to the SAPUI5 property binding for the entity property. For example, for `<Input value={Age}/>` the SAPUI5 type `Int64` is used, which corresponds to the OData type `Edm.Int64`.

Preview

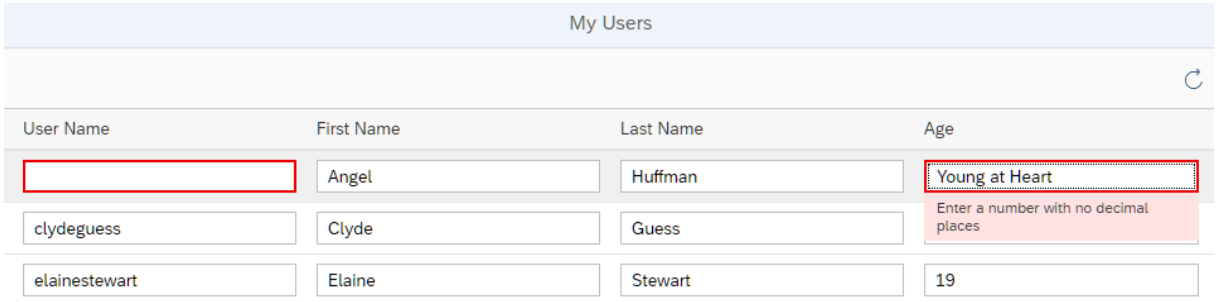


Figure 69: Input does not match the underlying data type

Coding

You can view and download all files at OData V4 - Step 3.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {...
    },
    "sap.ui": {
        "technology": "UI5",
        "deviceTypes": {

        }
    },
    "sap.ui5": {
        "rootView": {
        ...
        },
        "dependencies": {
            ...
            }
        },
        "contentDensities": {
            ...
        },
        "handleValidation": true,

        "models": {
            ...
            }
        }
    },
    "sap.platform.hcp": {
        ...
    }
```

In the `manifest.json` descriptor file, we add the `"handleValidation": true` setting. This makes sure that any validation errors that are detected by the SAPUI5 types are shown on the UI using the message manager.

We now run the app using the `index.html` file and enter values that don't match the type and constraints given in the metadata file. For example, enter the string value **`Young at Heart`** in field *Age*, which requires an integer input (SAPUI5 type `sap.ui.model.odata.type.Int64`, corresponding to OData type `Edm.Int64`), or remove an entry from the *User Name* or *First Name* fields, which are mandatory. Fields with incorrect entries are highlighted and an error message is displayed.

> **i** Note
>
> If you explicitly define a type in the binding info of a control, the automatic type detection for that binding will be turned off. For example, if you change the `Input` for `Age` in the view to `<Input value="{path:'Age', type:'sap.ui.model.type.String'}/>`, the `String` type will be used, not the `Int64` type from the service metadata.

localService/metadata.xml

```
<EntityType Name="Person">
    <Key>
        <PropertyRef Name="UserName"/>
    </Key>
    <Property Name="UserName" Type="Edm.String" Nullable="false" />
    <Property Name="FirstName" Type="Edm.String" />
    <Property Name="LastName" Type="Edm.String"/>
    <Property Name="MiddleName" Type="Edm.String"/>
    <Property Name="Gender"
Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.PersonGender"
             Nullable="false"/>
    <Property Name="Age" Type="Edm.Int64" />
```

To make the *First Name* optional, we remove the parameter `Nullable="false"` from the `FirstName` property. You can play around with the settings for the other properties, for example, change the type of property `Age` to `Type="Edm.String"` to allow free text.

> **→** Tip
>
> To see the metadata of an OData service, you append the `$metadata` variable to the URL of the service. You can try this, for example, with http://services.odata.org/TripPinRESTierService/ and http://services.odata.org/TripPinRESTierService/$metadata

Related Information

Type Determination [page 931]

API Reference: `sap.ui.model.odata.type`

Sample for `sap.ui.core.mvc.XMLView`: *XML Templating: UI5 OData types*

Step 4: Filtering, Sorting, and Counting

In this step, we add features to filter, sort, and count the user data by using the OData V4 model API to apply OData system query options `$filter`, `$orderby`, and `$count`.

Preview

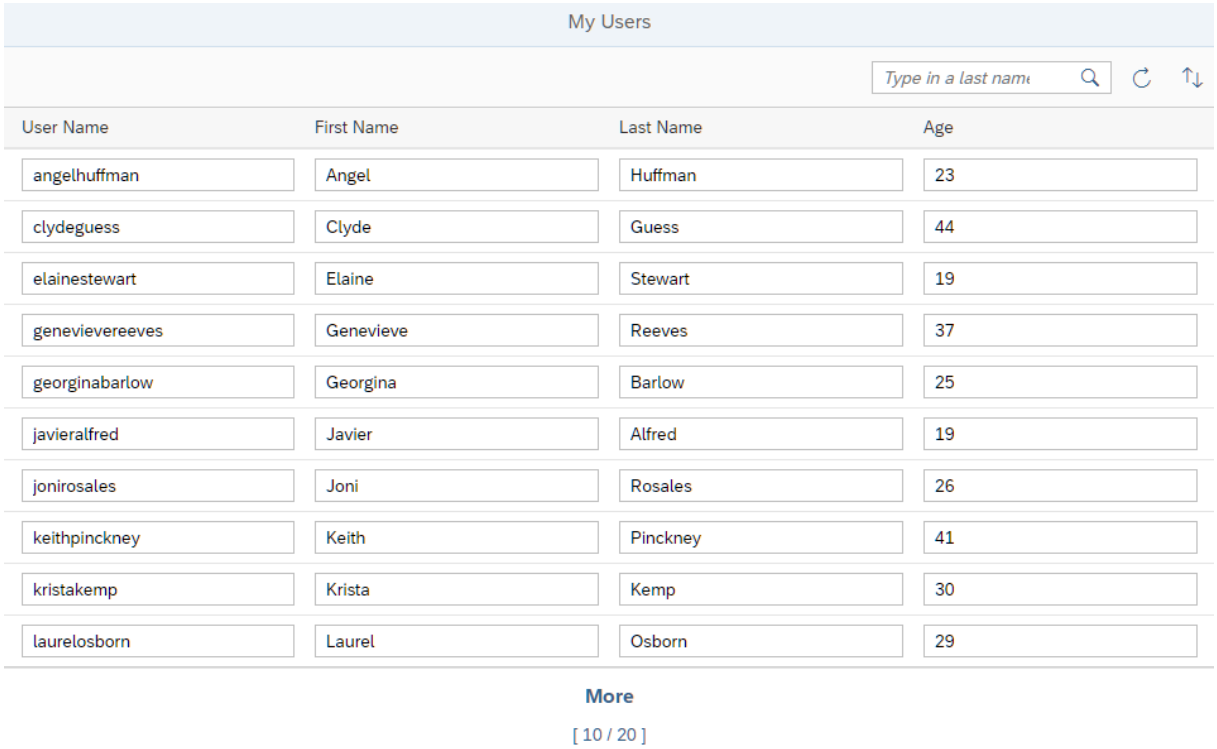


Figure 70: App now has a search field, the entries can be sorted, and you can see how many entities are loaded and how many more are available

Coding

You can view and download all files at OData V4 - Step 4.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/m/MessageBox",
    "sap/ui/model/Sorter",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/ui/model/FilterType",
```

```
    "sap/ui/model/json/JSONModel"
], function (Controller, MessageToast, MessageBox, Sorter, Filter,
FilterOperator, FilterType, JSONModel) {
    "use strict";
    return Controller.extend("sap.ui.core.tutorial.odatav4.controller.App", {
        onInit : function () {
            var oJSONData = {
                busy : false,
                order : 0
            };
            var oModel = new JSONModel(oJSONData);
            this.getView().setModel(oModel, "appView");
        },
        onRefresh : function () {
        ...
        },

        onSearch : function () {
            var oView = this.getView(),
                sValue = oView.byId("searchField").getValue(),
                oFilter = new Filter("LastName", FilterOperator.Contains,
sValue);

            oView.byId("peopleList").getBinding("items").filter(oFilter,
FilterType.Application);
        },

        onSort : function () {
            var oView = this.getView(),
                aStates = [undefined, "asc", "desc"],
                aStateTextIds = ["sortNone", "sortAscending", "sortDescending"],
                sMessage,
                iOrder = oView.getModel("appView").getProperty("/order");

            iOrder = (iOrder + 1) % aStates.length;
            var sOrder = aStates[iOrder];

            oView.getModel("appView").setProperty("/order", iOrder);
            oView.byId("peopleList").getBinding("items").sort(sOrder && new
Sorter("LastName", sOrder === "desc"));

            sMessage = this._getText("sortMessage",
[this._getText(aStateTextIds[iOrder])]);
            MessageToast.show(sMessage);
        },
        _getText : function (sTextId, aArgs) {
        ...
        }
    });
});
```

We add the `onSearch` and `onSort` event handlers for the *Search* field and the *Sort* button to the controller. We also enhance the `appView` model to store the active sorting order.

The `onSearch` event handler filters the table for people whose last name contains any string value entered in the *Search* field. We define a `sap.ui.model.Filter` and apply it to the binding of the `Table` using the `filter` method. The binding will then automatically retrieve filtered data from the OData V4 service and update the `Table`.

When the request is triggered, only entities that match the given filter criteria are requested from the OData V4 service.

> **i Note**
>
> Filters of OData services are case-sensitive. If you prefer a non case-sensitive search, implement it in the controller logic.

The `onSort` event handler requests the data unordered, or in ascending order, or descending order. Each time the *Sort* button is clicked, the next sort order is applied. The sorting is applied to the table by calling the `sort` method of the list binding with a new `sap.ui.model.Sorter`.

> **i Note**
>
> The features of filtering and sorting can also be combined.

We add the `order` property to variable `oJSONData` in `onInit` method. This property stores the current sort order.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.core.tutorial.odatav4.controller.App"
    displayBlock="true"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Shell>
        <App busy="{appView>/busy}" class="sapUiSizeCompact">
            <pages>
                <Page title="{i18n>peoplePageTitle}">
                    <content>
                        <Table
                            id="peopleList"
                            growing="true"
                            growingThreshold="10"
                            items="{
                                path: '/People',
                                parameters: {
                                    $count: true
                                }
                            }">
                            <headerToolbar>
                                <OverflowToolbar>
                                    <content>
                                        <ToolbarSpacer/>
                                        <SearchField
                                            id="searchField"
                                            width="20%"

placeholder="{i18n>searchFieldPlaceholder}"
                                            search=".onSearch"/>
                                        <Button
                                            id="refreshUsersButton"
                                            icon="sap-icon://refresh"
                                            tooltip="{i18n>refreshButtonText}"
                                            press=".onRefresh"/>
                                        <Button
                                            id="sortUsersButton"
                                            press="onSort"
                                            tooltip="{i18n>sortButtonText}"
                                            icon="sap-icon://sort"/>
```

```
                                    </content>
                                </OverflowToolbar>
                            </headerToolbar>
                            <columns>
                                <Column id="userNameColumn">
                                    <Text text="{i18n>userNameLabelText}"/>
                                </Column>
                                <Column id="firstNameColumn">
                                    <Text text="{i18n>firstNameLabelText}"/>
                                </Column>
                                <Column id="lastNameColumn">
                                    <Text text="{i18n>lastNameLabelText}"/>
                                </Column>
                                <Column id="ageColumn">
                                    <Text text="{i18n>ageLabelText}"/>
                                </Column>
                            </columns>
                            <items>
                                <ColumnListItem>
                                    <cells>
                                        <Input value="{UserName}"/>
                                    </cells>
                                    <cells>
                                        <Input value="{FirstName}"/>
                                    </cells>
                                    <cells>
                                        <Input value="{LastName}"/>
                                    </cells>
                                    <cells>
                                        <Input value="{Age}"/>
                                    </cells>
                                </ColumnListItem>
                            </items>
                        </Table>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

We add the `$count : true` parameter to tell the OData service to send the number of entities. With this setting, we automatically get the full number of entities (20) and the number of displayed entities (10) beneath the *More* button.

> **i Note**
>
> The live TripPin service does not support the `$count` parameter yet. If you use the live service instead of the mock server, as described in Step 2, leave out the `$count` parameter.

In the `OverflowToolbar`, we add a *Search* field and a *Sort* button with their events.

webapp/i18n/i18n.properties

```
...
#XTOL: Tooltip for refresh data
refreshButtonText=Refresh Data
#XTOL: Tooltip for sort
sortButtonText=Sort by Last Name
```

```
#XTXT: Placeholder text for search field
searchFieldPlaceholder=Type in a last name
...
# Messages
...
#XMSG: Message for refresh succeeded
refreshSuccessMessage=Data refreshed
#MSG: Message for sorting
sortMessage=Users sorted by {0}

#MSG: Suffix for sorting by LastName, ascending
sortAscending=last name, ascending

#MSG: Suffix for sorting by LastName, descending
sortDescending=last name, descending

#MSG: Suffix for no sorting
sortNone=the sequence on the server
```

We add the missing texts to the properties file.

Related Information

Filtering [page 939]
Sorting [page 942]
Query Options under *Querying Data* in the Basic Tutorial on the OData home page

Step 5: Batch Groups

In this step, we have a closer look at batch groups. Batch groups are used to group multiple requests into one server request to improve the overall performance.

Preview

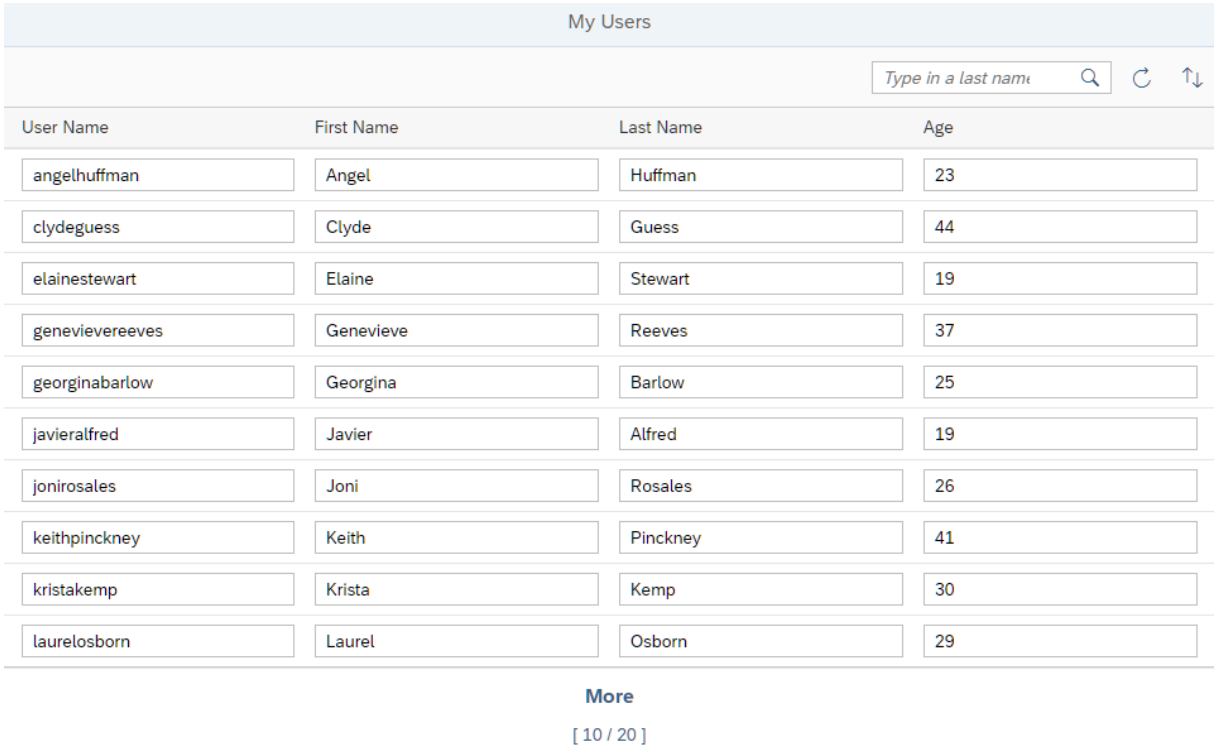


Figure 71: No visual change compared to the last step

Coding

You can view and download all files at OData V4 - Step 5.

webapp/manifest.json

```
...
                "": {
                "dataSource": "default",
                "settings": {
                    "autoExpandSelect": true,
                    "operationMode": "Server",
                    "groupId": "$auto",
                    "synchronizationMode": "None"
```

```
                    }
...
```

In the previous steps, batch processing was turned off, so that we could monitor the network traffic between our app and the service more easily. Now we turn on batch processing by changing the `groupID` to `$auto`. You can also just remove the line from the code as this is the default.

We now run the app and open the browser developer tools. On the *Console* tab, we clear all messages and choose the *Refresh* button.

> → Tip
>
> Change the settings of the *Console* so that it only displays information messages, not warnings and errors, to make it easier to find the messages we're looking for.

We see that the request is now bundled: To read the user data, the app now sends a `POST` request instead of a `GET` request to the server. The URL of the `POST` request does not include the `path` to the data we want. Instead it ends with `$batch` that indicates that this is a batch request.

A `$batch` request uses multipart MIME to put several requests into one. This makes it harder to analyze when looking at the request in the browser developer tools. To overcome this issue, you can:

- Switch the group ID to `$direct` temporarily by changing the source code or changing the default value in the debugger.
- Copy the relevant part of the request or response from the developer tools to an editor and auto-format it as JSON to analyze it.

Related Information

Batch Control [page 952]
Performance Aspects [page 967]

Step 6: Create and Edit

In this step, we will make it possible to create and edit (update) user data from the user interface and send the data to the back end.

Preview

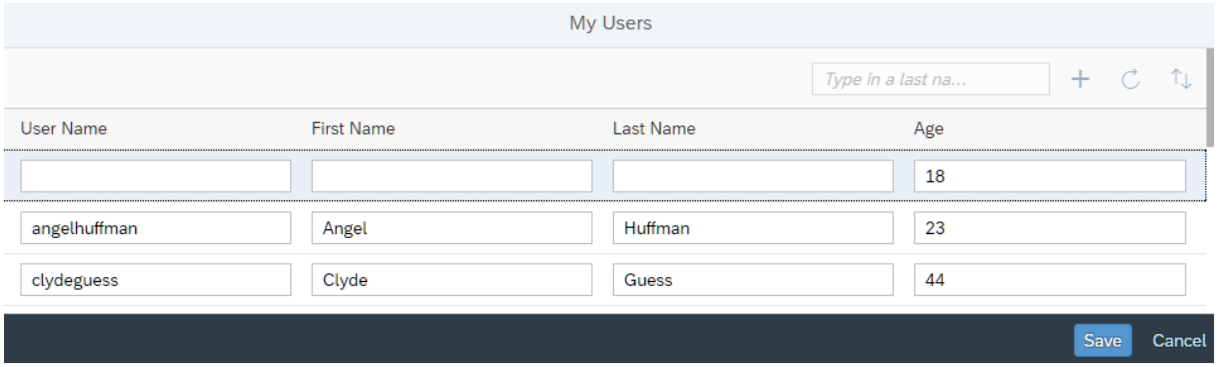


Figure 72: Data can now be edited and added.

Coding

You can view and download all files at OData V4 - Step 6.

webapp/controller/App.controller.js

```
...
        onInit : function () {
            var oMessageManager = sap.ui.getCore().getMessageManager(),
                oMessageModel = oMessageManager.getMessageModel(),
                oMessageModelBinding = oMessageModel.bindList("/", undefined, [],
                    new Filter("technical", FilterOperator.EQ, true)),
                oViewModel = new JSONModel({
                    busy : false,
                    hasUIChanges : false,
                    usernameEmpty : true,
                    order : 0
                });
            this.getView().setModel(oViewModel, "appView");
            this.getView().setModel(oMessageModel, "message");

            oMessageModelBinding.attachChange(this.onMessageBindingChange, this);
            this._bTechnicalErrors = false;
},
...
```

We change the `onInit` method: The `appView` model receives two additional properties, which we will use to control whether certain controls in the view are enabled or visible during user entries. We also make the `MessageModel` available to the view and add a `ListBinding`. When the OData service reports errors while

writing data, the OData Model adds them to the `MessageModel` as technical messages. Therefore we apply a filter to the `ListBinding`. We register our own handler to the `change` event of that `ListBinding` in order to capture any errors.

```
...
        onSort : function () {
            ...
        },
        _getText : function (sTextId, aArgs) {
            ...
        },
        _setUIChanges : function (bHasUIChanges) {
            if (this._bTechnicalErrors) {
                // If there is currently a technical error, then force 'true'.
                bHasUIChanges = true;
            } else if (bHasUIChanges === undefined) {
                bHasUIChanges = this.getView().getModel().hasPendingChanges();
            }
            var oModel = this.getView().getModel("appView");
            oModel.setProperty("/hasUIChanges", bHasUIChanges);
        }
    });
});
```

We add the `_setUIChanges` private method that lets us set the property `hasUIChanges` of the `appView` model. Unless there are currently technical messages in the `MessageModel` or it is called with a given value for its `bHasUIChanges` parameter, the method uses `ODataModel.hasPendingChanges`. That method returns `true` if there are any changes that have not yet been written to the service.

```
...
        onInit: function () {
            ...
        },
        onCreate : function () {
            var oList = this.byId("peopleList"),
                oBinding = oList.getBinding("items"),
                oContext = oBinding.create({
                    "UserName" : "",
                    "FirstName" : "",
                    "LastName" : "",
                    "Age" : "18"
                });

            this._setUIChanges();
            this.getView().getModel("appView").setProperty("/usernameEmpty",
true);

            oList.getItems().some(function (oItem) {
                if (oItem.getBindingContext() === oContext) {
                    oItem.focus();
                    oItem.setSelected(true);
                    return true;
                }
            });
        },
        onRefresh
...
```

We add the `onCreate` event handler that responds to the `press` event of the *Add User* button. We use the `create` method of the `ODataListBinding` API to create a new user with some initial data and insert it at the top of the table. The `create` method returns the binding context of the new user. That context provides a

`created` method which returns a `Promise`. The `Promise` is resolved when the new user is successfully transferred to the OData service.

We also use the binding context returned by the `create` method to focus and select the new row in which the new data can be entered.

```
...
        onRefresh: function () {
            ...
        },
        onSave : function () {
            var fnSuccess = function () {
                this._setBusy(false);
                MessageToast.show(this._getText("changesSentMessage"));
                this._setUIChanges(false);
            }.bind(this);

            var fnError = function (oError) {
                this._setBusy(false);
                this._setUIChanges(false);
                MessageBox.error(oError.message);
            }.bind(this);

            this._setBusy(true); // Lock UI until submitBatch is resolved.
            this.getView().getModel().submitBatch("peopleGroup").then(fnSuccess,
fnError);
            this._bTechnicalErrors = false; // If there were technical errors, a
new save resets them.
        },
        onSearch: function () {
            ...
        },
        ...
        _setUIChanges : function (bHasUIChanges) {
            ...
        },
        _setBusy : function (bIsBusy) {
            var oModel = this.getView().getModel("appView");
            oModel.setProperty("/busy", bIsBusy);
        }
    });
});
```

We create the `onSave` event handler, in which we call the `submitBatch` method of the `ODataModel` API to submit our changes. Because the changes that we submit refer to the table, we need to pass the update group `peopleGroup` that we declared in the table binding.

The `submitBatch` method returns a `Promise` that is rejected only if the batch request itself fails, for example, if the OData service is unavailable or if there were authorization problems. It is resolved in all other cases, also if the service returns errors for single requests that are contained in the batch request. Therefore, we have to implement the error handling for single requests differently.

We also define a `_setBusy` private function to lock the whole UI while the data is submitted to the back end.

```
...
        onSort : function () {
            ...
        },
        onMessageBindingChange : function (oEvent) {
            var aContexts = oEvent.getSource().getContexts(),
                aMessages,
                bMessageOpen = false;

            if (bMessageOpen || !aContexts.length) {
```

```
                return;
            }

            // Extract and remove the technical messages
            aMessages = aContexts.map(function (oContext) {
                return oContext.getObject();
            });
            sap.ui.getCore().getMessageManager().removeMessages(aMessages);

            this._setUIChanges(true);
            this._bTechnicalErrors = true;
            MessageBox.error(aMessages[0].message, {
                id : "serviceErrorMessageBox",
                onClose : function () {
                    bMessageOpen = false;
                }
            });

            bMessageOpen = true;
        },
...
```

We implement the event handler for the `change` event of the `ListBinding` to the `MessageModel`. We created the `ListBinding` with a filter to only include technical messages. That means that the `change` event will be fired with every change but only technical messages will have a binding context. In case of technical messages, we get the first one and display it as an error. We also make sure that the toolbar for saving or discarding changes stays visible. We delete the technical messages so that they do not accumulate.

```
...
        onRefresh: function () {
            ...
        },
        onResetChanges : function () {
            this.byId("peopleList").getBinding("items").resetChanges();
            this._bTechnicalErrors = false;
            this._setUIChanges();
        },
        onSearch: function () {
            ...
        },
...
```

The `onResetChanges` method handles discarding pending changes. It uses the `resetChanges` method of the `ODataListBinding` API to remove any such changes. Then it calls the `_setUIChanges` private method to enable the elements of the header toolbar again and hide the footer.

```
...
        onCreate: function () {
            ...
        },
        onInputChange : function (oEvt) {
            if (oEvt.getParameter("escPressed")) {
                this._setUIChanges();
            } else {
                this._setUIChanges(true);
                if
(oEvt.getSource().getParent().getBindingContext().getProperty("UserName")) {
                    this.getView().getModel("appView").setProperty("/
usernameEmpty", false);
                }
            }
        },
        onRefresh : function () {
            ...
```

```
        },
...
```

The `onInputChange` event handler manages entries in any of the `Input` fields and triggers updates to the `appView` model as needed. It does an extra check on the `UserName` field to make sure that users cannot be saved without a `UserName`. Otherwise the OData service would return errors because `UserName` is a mandatory field.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.core.tutorial.odatav4.controller.App"
    displayBlock="true"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Shell>
        <App busy="{appView>/busy}" class="sapUiSizeCompact">
            <pages>
                <Page title="{i18n>peoplePageTitle}">
                    <content>
                        <Table
                            id="peopleList"
                            growing="true"
                            growingThreshold="10"
                            items="{
                                path: '/People',
                                parameters: {
                                    $count: true,
                                    $$updateGroupId : 'peopleGroup'
                                }
                            }">
                            <headerToolbar>
                                <OverflowToolbar>
                                    <content>
                                        <ToolbarSpacer/>
                                        <SearchField
                                            id="searchField"
                                            width="20%"

placeholder="{i18n>searchFieldPlaceholder}"
                                            enabled="{= !${appView>/
hasUIChanges}}"

                                            search=".onSearch"/>
                                        <Button
                                            id="addUserButton"
                                            icon="sap-icon://add"
                                            tooltip="{i18n>createButtonText}"
                                            press=".onCreate">
                                            <layoutData>
                                                <OverflowToolbarLayoutData
priority="NeverOverflow"/>
                                            </layoutData>
                                        </Button>
                                        <Button
                                            id="refreshUsersButton"
                                            icon="sap-icon://refresh"
                                            enabled="{= !${appView>/
hasUIChanges}}"

                                            tooltip="{i18n>refreshButtonText}"
                                            press=".onRefresh"/>
                                        <Button
                                            id="sortUsersButton"
```

```
                                        icon="sap-icon://sort"
                                        enabled="{= !${appView>/
hasUIChanges}}"
                                        tooltip="{i18n>sortButtonText}"
                                        press=".onSort"/>
                                </content>
                            </OverflowToolbar>
                        </headerToolbar>
                        <columns>
                            <Column id="userNameColumn">
                                <Text text="{i18n>userNameLabelText}"/>
                            </Column>
                            <Column id="firstNameColumn">
                                <Text text="{i18n>firstNameLabelText}"/>
                            </Column>
                            <Column id="lastNameColumn">
                                <Text text="{i18n>lastNameLabelText}"/>
                            </Column>
                            <Column id="ageColumn">
                                <Text text="{i18n>ageLabelText}"/>
                            </Column>
                        </columns>
                        <items>
                            <ColumnListItem>
                                <cells>
                                    <Input
                                        value="{UserName}"
                                        valueLiveUpdate="true"
                                        liveChange=".onInputChange"/>
                                </cells>
                                <cells>
                                    <Input
                                        value="{FirstName}"
                                        liveChange=".onInputChange"/>
                                </cells>
                                <cells>
                                    <Input
                                        value="{LastName}"
                                        liveChange=".onInputChange"/>
                                </cells>
                                <cells>
                                    <Input
                                        value="{Age}"
                                        valueLiveUpdate="true"
                                        liveChange=".onInputChange"/>
                                </cells>
                            </ColumnListItem>
                        </items>
                    </Table>
                </content>
                <footer>
                    <Toolbar visible="{appView>/hasUIChanges}">
                        <ToolbarSpacer/>
                        <Button
                            id="saveButton"
                            type="Emphasized"
                            text="{i18n>saveButtonText}"
                            enabled="{= ${message>/}.length === 0 &amp;amp;&amp;
${appView>/usernameEmpty} === false }"
                            press=".onSave"/>
                        <Button
                            id="doneButton"
                            text="{i18n>cancelButtonText}"
                            press=".onResetChanges"/>
                    </Toolbar>
                </footer>
            </Page>
        </pages>
```

```
        </App>
    </Shell>
</mvc:View>
```

We add the `$$updateGroupId: 'peopleGroup'` parameter to the table. This means that changes in the table are not sent to the service immediately but instead are collected until we explicitly send them.

We add a new *Add User* button to the overflow toolbar in the table header, and define a footer toolbar that contains *Save* and *Cancel* buttons that we can display or hide through the `appView` model. We can disable the *Save* button separately, for example when a user enters invalid data.

Finally, we add the `liveChange="onInputChange"` event handler to the table cells to make it possible to react to user input. In addition, we set the `valueLiveUpdate` properties for the fields for `UserName` and `Age`. That makes sure that the SAPUI5 types validate the field content with each keystroke.

webapp/i18n/i18n.properties

```
# Toolbar
#XBUT: Button text for save
saveButtonText=Save

#XBUT: Button text for cancel
cancelButtonText=Cancel
#XTOL: Tooltip for sort
sortButtonText=Sort by Last Name
#XBUT: Button text for add user
createButtonText=Add User
...
# Messages
#XMSG: Message for user changes sent to the service
changesSentMessage=User data sent to the server
...
```

We add the new message texts.

Related Information

Model Instantiation and Data Access [page 918]
Batch Control [page 952]
OData Operations [page 945]
Creating an Entity [page 974]
Message Model [page 1070]
API Reference: `sap.ui.model.odata.v4.ODataContextBinding`

Step 7: Delete

In this step, we make it possible to delete user data.

Preview

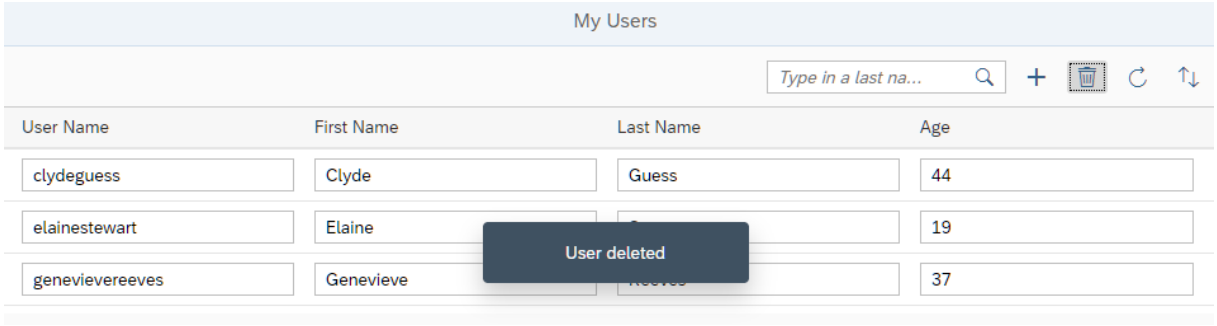


Figure 73: A new *Delete User* button is added

Coding

You can view and download all files at OData V4 - Step 7.

webapp/App.controller.js

```
        onDelete : function () {
            var oSelected = this.byId("peopleList").getSelectedItem();

            if (oSelected) {
                oSelected.getBindingContext().delete("$auto").then(function () {
                    MessageToast.show(this._getText("deletionSuccessMessage"));
                }.bind(this), function (oError) {
                    MessageBox.error(oError.message);
                });
            }
        },
```

We add the `onDelete` event handler to the controller. In the event handler, we check whether an item is selected in the table and if so, the related data is deleted from the model. To do that, we retrieve the binding context of the selection and call its `delete` method.

We explicitly set the update group ID for the deletion to `$auto` to make sure that the request to the service is sent immediately as a batch request. Otherwise the `delete` function would apply the deferred batch processing that we defined for the table's list binding event though, `delete` does not currently support deferred batch processing.

webapp/App.view.xml

```
<mvc:View
    controllerName="sap.ui.core.tutorial.odatav4.controller.App"
    displayBlock="true"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Shell>
        <App busy="{appView>/busy}" class="sapUiSizeCompact">
            <pages>
                <Page title="{i18n>peoplePageTitle}">
                    <content>
                        <Table
                            id="peopleList"
                            growing="true"
                            growingThreshold="10"
                            items="{
                                path: '/People',
                                parameters: {
                                    $count: true,
                                    $$updateGroupId : 'peopleGroup'
                                }
                            }"
                            mode="SingleSelectLeft">
                            <headerToolbar>
                                <OverflowToolbar>
                                    <content>
                                        <ToolbarSpacer/>
                                        <SearchField
                                        .../>
                                        <Button
                                        .../>
                                        <Button
                                            id="deleteUserButton"
                                            icon="sap-icon://delete"
                                            tooltip="{i18n>deleteButtonText}"
                                            press=".onDelete">
                                            <layoutData>
                                                <OverflowToolbarLayoutData
priority="NeverOverflow"/>
                                            </layoutData>
                                        </Button>
                                        <Button
                                        .../>
                                        <Button
                                        ...>
                                    </content>
                                </OverflowToolbar>
                            </headerToolbar>
                            <columns>
                                ...
                            </columns>
                            <items>
                                ...
                            </items>
                        </Table>
                    </content>
                    <footer>
                        ...
                    </footer>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

We change the `mode` of the table to `SingleSelectLeft` to make it possible to select a row.

We add the *Delete* button to the toolbar. With the `OverflowToolbarLayoutData priority="NeverOverflow"` parameter, we make sure that the button is always visible.

webapp/i18n/i18n.properties

```
...
# Toolbar
...
#XBUT: Button text for delete user
deleteButtonText=Delete User
...
# Messages
...
#XMSG: Message for user deleted
deletionSuccessMessage=User deleted
...
```

We add the missing texts to the properties file.

Related Information

Deleting an Entity [page 976]

Step 8: OData Operations

Our OData service provides one OData operation: the `ResetDataSource` action. In this step, we add a button that resets all data changes we made during the tutorial to their original state using this action.

Preview

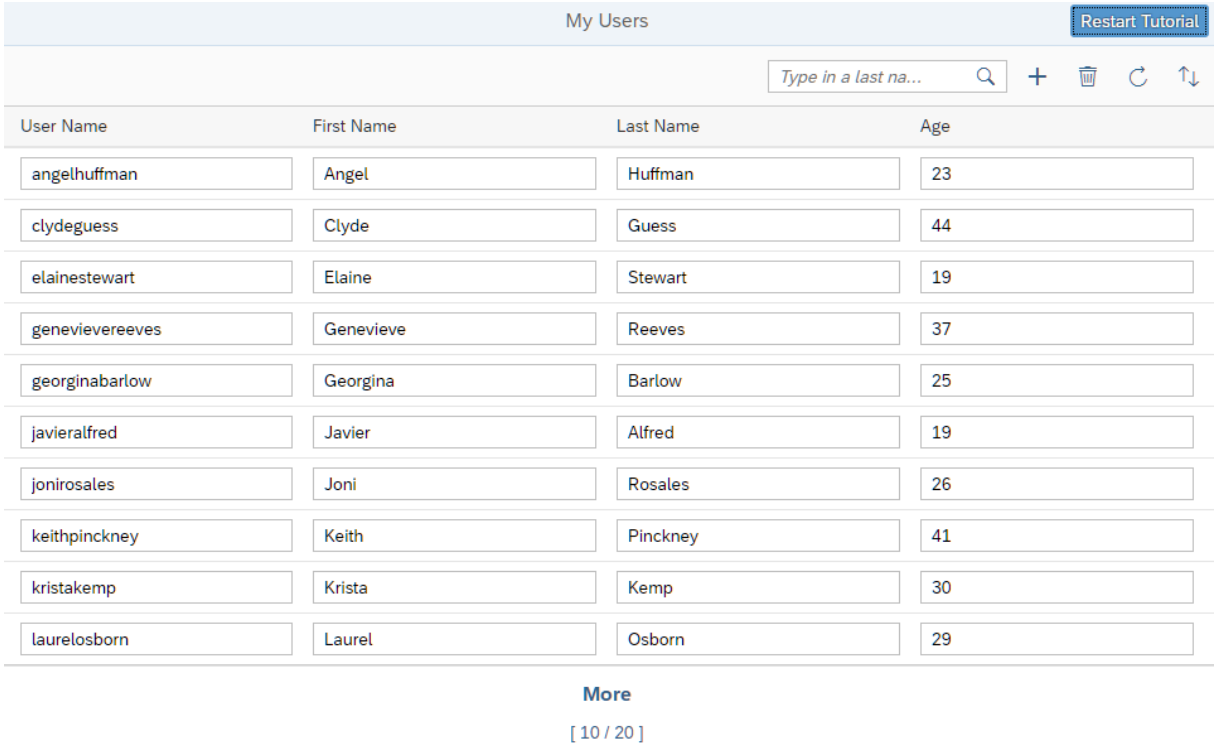


Figure 74: A *Restart Tutorial* button is added

Coding

You can view and download all files at OData V4 - Step 8.

webapp/controller/App.controller.js

```
...
        onResetChanges : function () {
            this.byId("peopleList").getBinding("items").resetChanges();
            this._setUIChanges();
        },
        onResetDataSource : function () {
            var oModel = this.getView().getModel(),
                oOperation = oModel.bindContext("/ResetDataSource(...)");
```

```
            oOperation.execute().then(function () {
                    oModel.refresh();

MessageToast.show(this._getText("sourceResetSuccessMessage"));
                }.bind(this), function (oError) {
                    MessageBox.error(oError.message);
                }
            );
        },
        onSave : function () {
...
```

The `onResetDataSource` event handler calls the `ResetDataSource` action, which is an action of the *TripPin* OData service that resets the data of the service to its original state.

We call that action by first creating a deferred operation binding on the model. The `(...)` part of the binding syntax marks the binding as deferred. We use a deferred binding because we want to control when the action is executed. Since it is deferred, we need to explicitly call its `execute` method.

The execution is asynchronous, therefore the execute method returns a `Promise`. We attach simple success and error handlers to that `Promise` by calling its `then` method.

> **i** Note
>
> Many of the methods in the OData V4 API of SAPUI5 return a `Promise` to manage asynchronous processing

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.core.tutorial.odatav4.controller.App"
    displayBlock="true"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Shell>
        <App busy="{appView>/busy}" class="sapUiSizeCompact">
            <pages>
                <Page title="{i18n>peoplePageTitle}">
                    <headerContent>
                        <Button
                            id="resetChangesButton"
                            text="{i18n>resetChangesButtonText}"
                            enabled="{= !${appView>/hasUIChanges}}"
                            press="onResetDataSource"
                            type="Emphasized">
                        </Button>
                    </headerContent>
...
```

We add the **headerContent** aggregation to the **Page** and insert the new **Button**. We add the **onResetDataSource** event handler to the **press** event.

webapp/i18n/i18n.properties

```
...
# Toolbar
...
#XBUT: Button text for reset changes
resetChangesButtonText=Restart Tutorial
...
# Messages
...
#XMSG: Message for changes reverted
sourceResetSuccessMessage=All changes reverted back to start
```

We add the missing texts to the properties file.

And now we are done! We built a simple application with user data from an OData V4 service. We can display, edit, create, and delete users. And we use OData V4 features such as batch groups and automatic type detection.

Related Information

Bindings [page 922]
OData Operations [page 945]

Navigation and Routing

SAPUI5 comes with a powerful routing API that helps you control the state of your application efficiently. This tutorial will illustrate all major features and APIs related to navigation and routing in SAPUI5 apps by creating a simple and easy to understand mobile app. It represents a set of best practices for applying the navigation and routing features of SAPUI5 to your applications.

In classical Web applications, the server determines which resource is requested based on the URL pattern of the request and serves it accordingly. The server-side logic controls how the requested resource or page is displayed in an appropriate way.

In single-page applications, only one page is initially requested from the server and additional resources are dynamically loaded using client-side logic. The user only navigates within this page. The navigation is persisted in the hash instead of the server path or URL parameters.

For example, a classical Web application might display the employee's resume page when URL `http://<your-host>/<some-path-to-the-app>/employees/resume.html?id=3` or `http://<your-host>/<some-path-to-the-app>/employees/3/resume` is called. A single-page application instead would do the same thing by using a hash-based URL like `http://<your-host>/<some-path-to-the-app>/#/employees/3/resume`.

The information in the hash, namely everything that is following the `#` character, is interpreted by the router.

i Note

This tutorial does not handle cross-app navigation with the SAP Fiori launchpad. However, the concepts described in this tutorial are also fundamental for navigation and routing between apps in the SAP Fiori launchpad.

We will create a simple app displaying the data of a company's employees to show typical navigation patterns and routing features. The complete flow of the application can be seen in the figure below. We'll start with the home page which lets users do the following:

- Display a *Not Found* page
- Navigate to a list of employees and drill further down to see a *Details* page for each employee
- Show an *Employee Overview* that they can search and sort

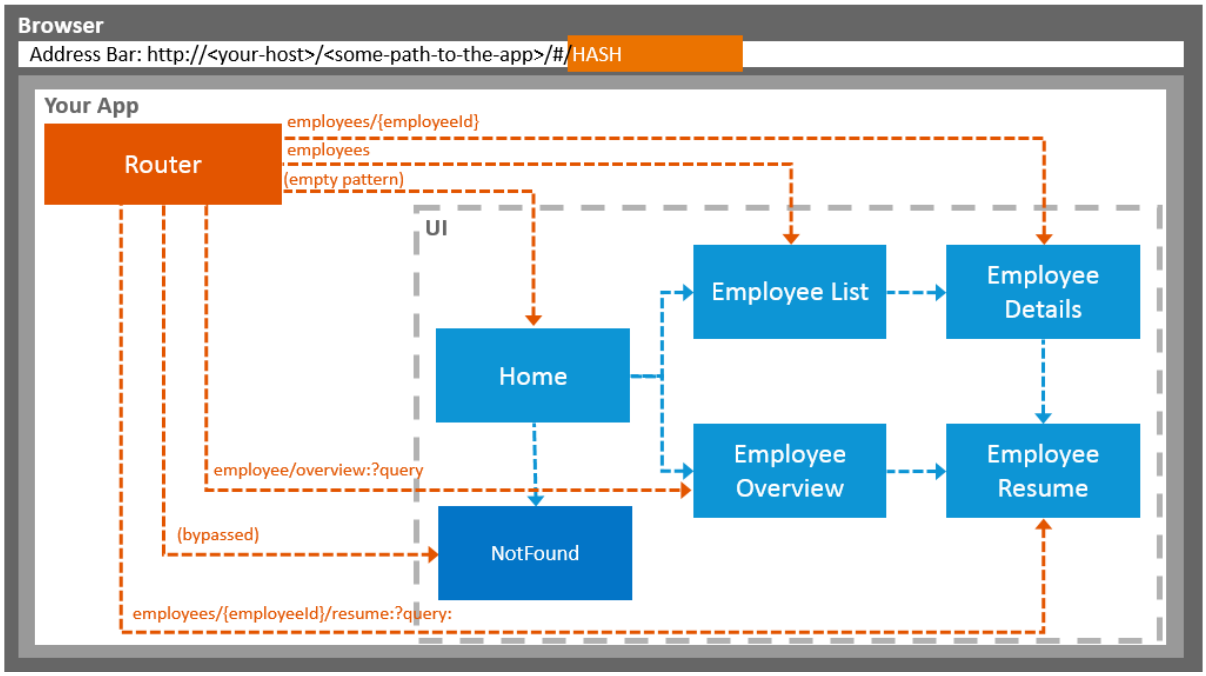


Figure 75: Page flow of the final app

Throughout this tutorial we will add features for navigating to pages and bookmarking them. We will add backward and forward navigation with common transition animations (slide, show, flip, etc.). We will add more pages to the app and navigate between them to show typical use cases. We will even learn how to implement features for bookmarking a specific search, table sorting via filters, and dialogs.

→ Tip

You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, and start there.

You can view and download the files for all steps in the Demo Kit at Navigation and Routing. Copy the code to your workspace and make sure that the application runs by calling the `webapp/index.html` file. Depending on your development environment you might have to adjust resource paths and configuration entries.

For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Step 1: Set Up the Initial App

We start by setting up a simple app for this tutorial. The app displays mock data only and mimics real OData back-end calls with the mock server as you have seen in the *Walkthrough* tutorial.

The structure and data model created in this step will be used throughout the rest of this tutorial. The initial app created in this step will be extended in the subsequent steps to illustrate the navigation and routing features of SAPUI5.

Preview

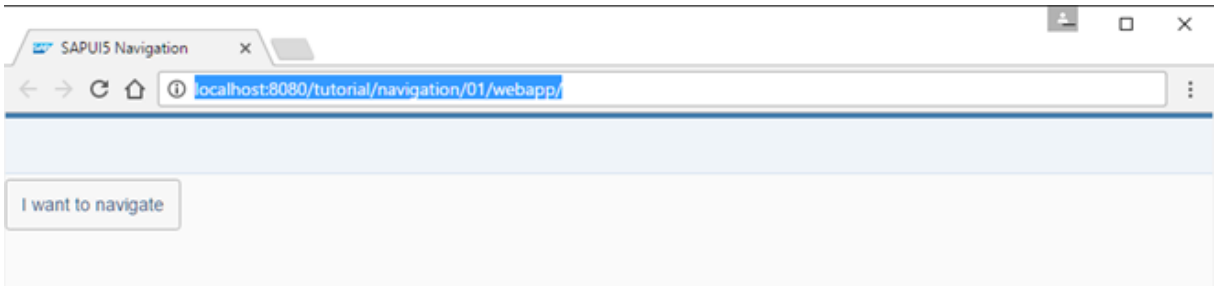


Figure 76: Initial app with a simple button

Setup

To set up your project for this tutorial, download the files for *Step 1* from the *Samples* in the Demo Kit at Navigation and Routing - Step 1. Copy the code to your workspace and make sure that the application runs by calling the `webapp/index.html` file.

Depending on your development environment you might have to adjust resource paths and configuration entries. The project structure and the files coming with this tutorial are explained in detail in the Walkthrough [page 69] tutorial.

You should have the same files as displayed in the following figure:

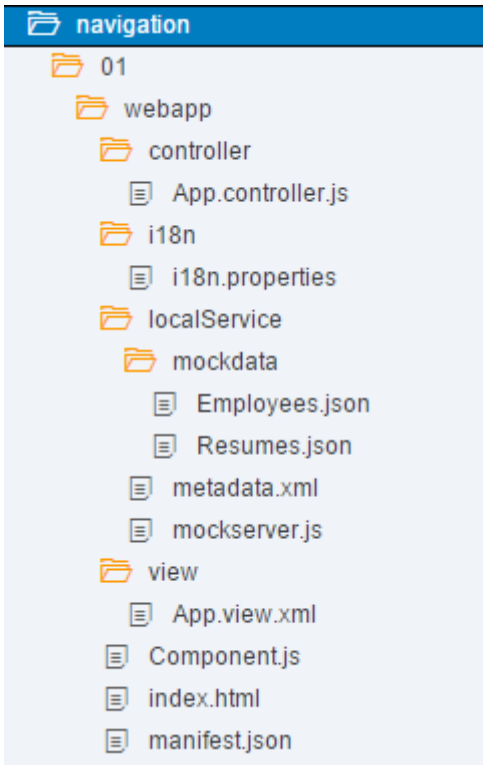


Figure 77: Folder structure with downloaded files

> **i** Note
>
> The content of the `localService` folders will not be changed in this tutorial. The `i18n` folder will always contain the `i18n.properties` file only. Therefore, we will show both subfolders collapsed in the following steps.

The Initial App

With the downloaded coding, you have an initial app with recommended settings that provides the basic features of an SAPUI5 app:

- **Home Page**
  The home page of our app is defined in the `webapp/index.html` file. In this file we bootstrap SAPUI5 and tell the runtime where to find our custom resources. Furthermore, we initialize the `MockServer` to simulate back-end requests as we do not have a real back-end service throughout this tutorial. Finally, we instantiate the application component, assign it to a `sap.m.Shell` control, and place the shell into the body. The corresponding `Component.js` file in the `webapp` folder will be extended throughout this tutorial.
- **Data**
  In the `webapp/localService/mockserver.js` file, we configure the mock server. Using the mock server in this tutorial allows us to easily run the code even without network connection and without the need of having a remote server for our application data.

The `metadata.xml` file used by the mock server describes our OData service. The service only has two OData entities:

- Employee
  An `employee` has typical properties like `FirstName` and `LastName` as well as a navigation property to a resume entity referenced by a `ResumeID`. Of course, the entity also has an ID property: `EmployeeID`. The corresponding `EntitySet` is `Employees`. The actual test data containing several employees is located in the `webapp/localService/mockdata/Employees.json` file.
- Resume
  In our case, we want to keep the resume of employees very simple. Therefore, we just have simple properties of type `Edm.String`. The properties are `Information`, `Projects`, `Hobbies` and `Notes`; all of them contain textual information. The entity has an ID property `ResumeID` and the corresponding `EntitySet` is `Resumes`. The resume data for an employee is located in file `webapp/localService/mockdata/Resumes.json`.

- **Configuration of the App**
  In the `webapp/manifest.json` descriptor file, we configure our app. The descriptor file contains the following most interesting sections:
  - `sap.app`
    In this section we reference an `i18n.properties` file and use a special syntax to bind the texts for the `title` and `description` properties.
    In the `dataSources` part, we tell our app where to find our OData service `employeeRemote`. As you might guess, the `uri` correlates to the `rootUri` of our mock server instance which can be found in `webapp/localService/mockserver.js`. It is important that these two paths match to allow our mock server to provide the test data we defined above. The `localUri` is used to determine the location of the `metadata.xml` file.
  - `sap.ui5`
    Under `sap.ui5` we declare with the `rootView` parameter that our `sap.ui.demo.nav.view.App` view shall be loaded and used as the `rootView` for our app. Furthermore, we define two `models` to be automatically instantiated and bound to the `i18n` component and a default model `""`. The latter references our `employeeRemote dataSource` which is declared in our `sap.app` section as an OData 2.0 data source. The `i18n` file can be found at `webapp/i18n/i18n.properties`. This data source will be mocked by our mock server.

So far we have a basic app that does not really have any navigation or routing implemented. This will change in the next steps when we implement our first navigation features.

Step 2: Enable Routing

In this step we will modify the app and introduce routing. Instead of having the home page of the app hard coded we will configure a router to wire multiple views together when our app is called. The routing

configuration controls the application flow when the user triggers a navigation action or opens a link to the application directly.

Preview

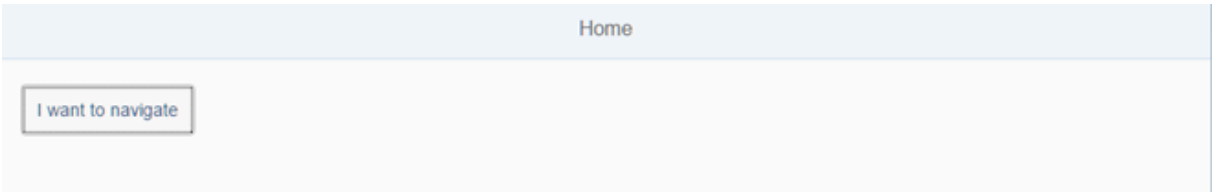


Figure 78: Views are wired together using the router

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 2.

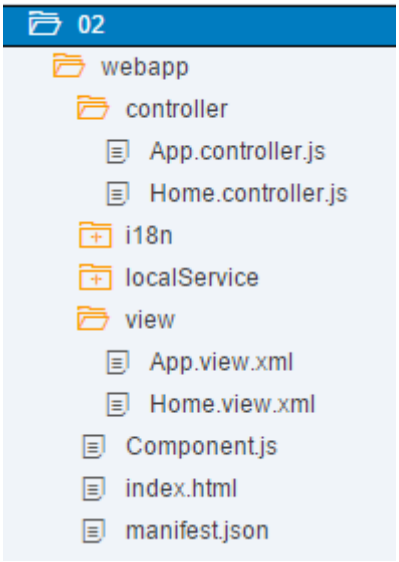


Figure 79: Folder structure for this step

webapp/manifest.json

```
{
   "_version": "1.12.0",
   "sap.app": {
      ...
   },
   "sap.ui": {
      ...
   },
   "sap.ui5": {
      "rootView": {
          "viewName": "sap.ui.demo.nav.view.App",
          "type": "XML",
          "async": true,
```

```
            "id": "app"
        },
        "dependencies": {
            ...
        },
        "models": {
            ...
        },
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "async": true
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
                "target": "home"
            }],
            "targets": {
                "home": {
                    "viewId": "home",
                    "viewName": "Home",
                    "viewLevel" : 1
                }
            }
        }
    }
}
```

Single-page applications based on SAPUI5 can use a so-called “router” to dispatch hash-based URLs to one or more views of the app. Therefore, the router needs to know how to address and show the views. In SAPUI5, we can simply add a `routing` section to our existing `sap.ui5` section in the descriptor file to configure the router. There are three properties that can be used to configure the routing of your application:

- `config`
  This section contains the global router configuration and default values that apply for all routes and targets. The property `routerClass` is special as it determines the router implementation. The default value is `sap.ui.core.routing.Router`. Here, we set the `routerClass` to `sap.m.routing.Router`, because we implement an app based on `sap.m`. All other properties in `config` are given to the router instance. For example, we define where our views are located in the app. To load and display views automatically, we also specify the `controlId` of the control that is used to display the pages and the aggregation (`controlAggregation`) that will be filled when a new page is displayed. We will create only XMLviews in this tutorial, so we can set the `viewType` property to `XML`. All our views will be available in the `view` folder of the namespace `sap.ui.demo.nav`, so we can set the `viewPath` to `sap.ui.demo.nav.view`. The `transition` allows us to set a default value for how the transition should happen; you can choose between `slide` (default), `flip`, `fade`, and `show`. All parameters of the config section can be overruled in the individual route and target definitions if needed.

  > **i Note**
  >
  > The possible values for `routerClass` are `sap.ui.core.routing.Router`, `sap.m.routing.Router`, or any other subclasses of `sap.ui.core.routing.Router`. Compared to `sap.ui.core.routing.Router` the `sap.m.routing.Router` is optimized for mobile apps and adds the properties `viewLevel`, `transition` and `transitionParameters` which can be specified

for each route or target created by the `sap.m.routing.Router`. The `transitionParameters` can also be used for custom transitions. Please check the *API Reference* for more information.

- `routes`
  Each route defines a name, a pattern, and one or more targets to navigate to when the route has been hit. The pattern is basically the hash part of the URL that matches the route. The sequence of the routes is important because only the first matched route is used by the router. In our case, we have an empty pattern to match the empty hash. The `name` property allows you to choose a unique route name that helps you to navigate a specific route or to determine the matched route in one of the matched handlers (we'll explain that in a later step). The target property references one or more targets from the section below that will be displayed when the route has been matched.
- `targets`
  A target defines the view that is displayed. It is associated with one or more routes or it can be displayed manually from within the app. Whenever a target is displayed, the corresponding view is loaded and added to the aggregation configured with the `controlAggregation` option of the control. This option is configured using `controlId`. Each target has a unique key (`home`). The `viewName` defines which view shall be loaded. In our little example, the absolute view path to be loaded for our `home` target is determined by the default `"viewPath": "sap.ui.demo.nav.view"` and `"viewName": "Home"`. This leads to `"sap.ui.demo.nav.view.Home"`. The `viewLevel` is especially relevant for `flip` and `slide` transitions. It helps the router to determine the direction of the transition from one page to another. (This will also be explained later.) A target can be assigned to a route, but it's not necessary. Targets can be displayed directly in the app without hitting a route.
  This basic routing configuration was easy enough. However, you can't see it in action until you have initialized the router.

> **i Note**
>
> As of SAPUI5 version 1.30, we recommend that you define the routing in the `manifest.json` descriptor file using routes and targets. In older versions of SAPUI5, the routing configuration had to be done directly in the metadata section of the component, and with different syntax.

webapp/Component.js

```
sap.ui.define([
    "sap/ui/core/UIComponent"
], function (UIComponent) {
    "use strict";
    return UIComponent.extend("sap.ui.demo.nav.Component", {
        metadata: {
            manifest: "json"
        },

        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);

            // create the views based on the url/hash
            this.getRouter().initialize();
        }
    });
});
```

We override the `init` function and call the parent's `init` function first. We get a reference to the router and call `initialize()` on it. The router is instantiated automatically with the configuration loaded in the descriptor. The routing events and our configuration in the descriptor are now automatically enabled in the app. Running the app at this point would lead to an error, because the home view is not implemented yet.

webapp/view/App.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    displayBlock="true">
    <Shell>
        <App id="app"/>
    </Shell>
</mvc:View>
```

In the `App` view, we remove the content of `App` control. The pages will be added dynamically the way we have configured it in the descriptor. The view configured with the property `rootView` is automatically instantiated when the app is called initially.

webapp/view/Home.view.xml (New)

```
<mvc:View
   controllerName="sap.ui.demo.nav.controller.Home"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Page title="{i18n>homePageTitle}" class="sapUiResponsiveContentPadding">
      <content>
         <Button text="{i18n>iWantToNavigate}" class="sapUiTinyMarginEnd"/>
      </content>
   </Page>
</mvc:View>
```

Create a file `Home.view.xml` in the `webapp/view` folder. The home view only contains a page control that displays a button. For illustration, we bind the title of the page to the `i18n>homePageTitle`, you can use data binding just the way you are used to it.

webapp/controller/Home.controller.js (New)

```
sap.ui.define([
   "sap/ui/core/mvc/Controller"
], function (Controller) {
   "use strict";

   return Controller.extend("sap.ui.demo.nav.controller.Home", {

   });
```

```
});
```

Create a file `Home.controller.js` in the `webapp/controller` folder. The controller for the home view does not contain any custom logic in this step, but we will add some features to it soon. Finally, run the app by calling the `webapp/index.html` file. This will be the entry point for our app in all the next steps. As you can see, the app is initially displaying the home view that we configured as the default pattern in the routing configuration. We have now successfully enabled routing in the app.

> **i** Note
>
> We think of routing as a set of features that dispatch hash-based URLs to an app's views and manage the views' states.
>
> Based on the routing configuration, you define the navigation between pages and pass parameters to the target views.

Conventions

- Configure the router in the `manifest.json` descriptor file
- Initialize the router exactly once
- Initialize the router in the component

Related Information

Routing and Navigation [page 1072]
API Reference: `sap.ui.core.routing`
API Reference: `sap.ui.core.routing.Route`
API Reference: `sap.ui.core.routing.Route`: Constructor Detail
API Reference: `sap.m.routing.Router`

Step 3: Catch Invalid Hashes

Sometimes it is important to display an indication that the requested resource was not found. To give you an example: If a user tries to access an invalid pattern which does not match any of the configured routes, the user is notified that something went wrong. You might also know this as a “404” or *Not Found Page* from traditional web pages. In this step, we will implement a feature that detects invalid hashes and visualizes this in a nice way.

Preview

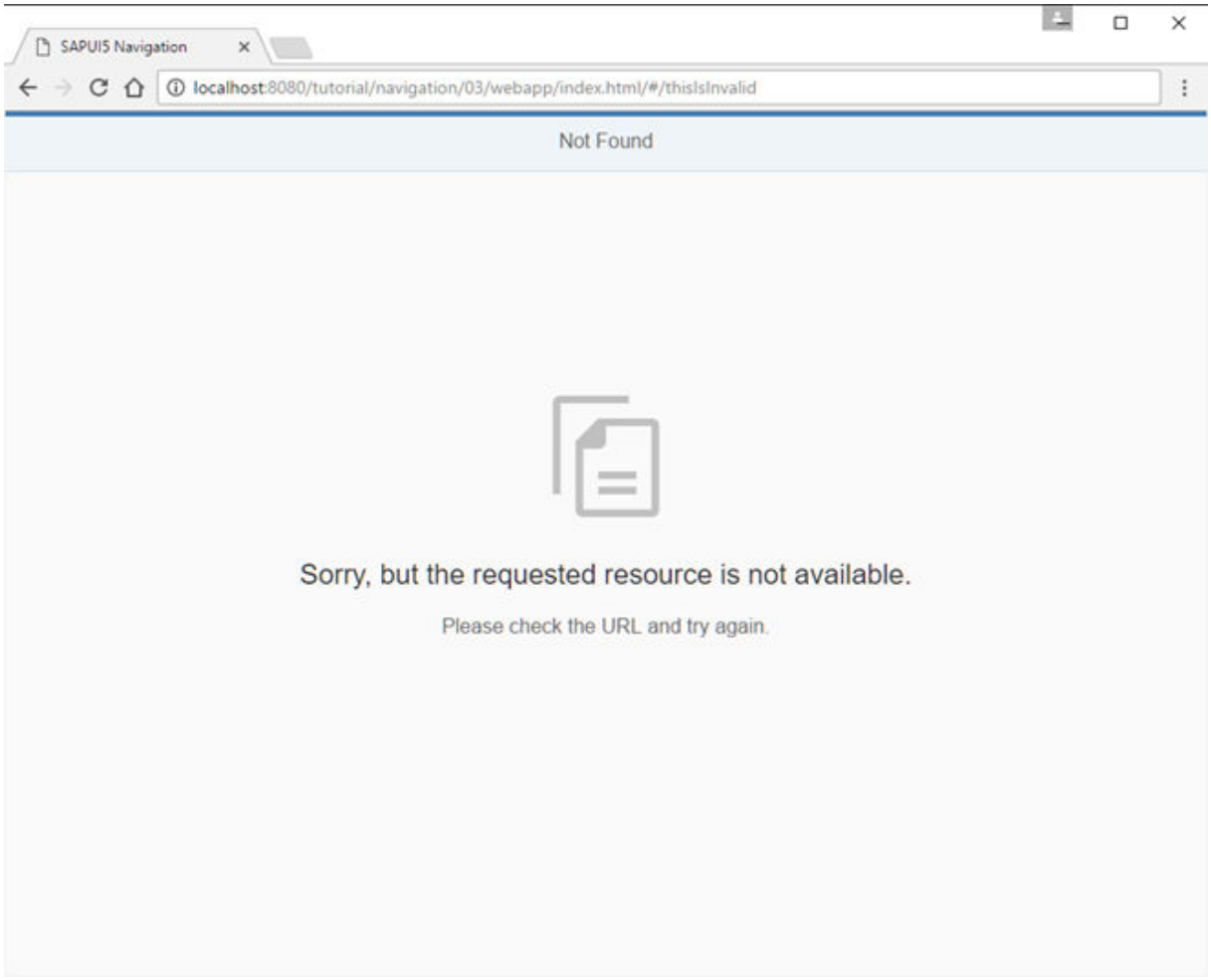


Figure 80: *Not Found* page

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 3.

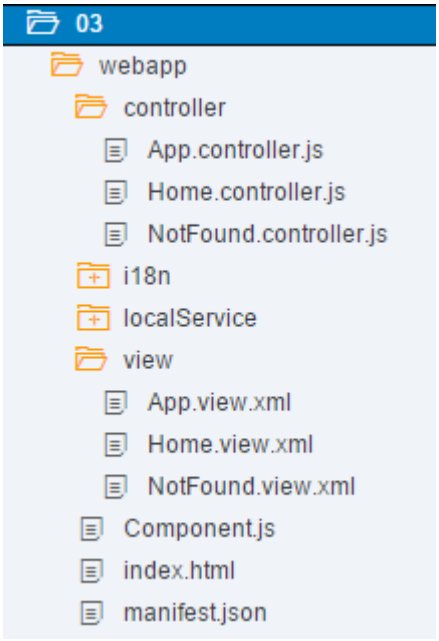


Figure 81: Folder structure for this step

webapp/manifest.json

```
{
   ...
   "sap.ui5": {
      ...
      "routing": {
         "config": {
            "routerClass": "sap.m.routing.Router",
            "viewType": "XML",
            "viewPath": "sap.ui.demo.nav.view",
            "controlId": "app",
            "controlAggregation": "pages",
            "transition": "slide",
            "bypassed": {
               "target": "notFound"
            },
            "async": true
         },
         "routes": [{
            "pattern": "",
            "name": "appHome",
            "target": "home"
         }],
         "targets": {
            "home": {
               "viewId": "home",
               "viewName": "Home",
               "viewLevel" : 1
            },
            "notFound": {
               "viewId": "notFound",
               "viewName": "NotFound",
               "transition": "show"
            }
         }
      }
   }
```

```
}
```

Let's extend the routing configuration in the descriptor by adding a `bypassed` property and setting its `target` to `notFound`. This configuration tells the router to display the `notFound` target in case no route was matched to the current hash. Next, we add a `notFound` target to the `bypassed` section. The `notFound` target simply configures a `notFound` view with a `show` transition.

webapp/view/NotFound.view.xml (New)

```
<mvc:View
   controllerName="sap.ui.demo.nav.controller.NotFound"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <MessagePage
      title="{i18n>NotFound}"
      text="{i18n>NotFound.text}"
      description="{i18n>NotFound.description}"/>
</mvc:View>
```

Now we create the view referenced above in a new file `NotFound.view.xml` in the `webapp/view folder`. It uses a `sap.m.MessagePage` control to display an error message to the user. In a real app you might use a dynamic message matching the current error situation. Here, we simply display a preconfigured text from our resource bundle.

webapp/controller/NotFound.controller.js (New)

```
sap.ui.define([
   "sap/ui/core/mvc/Controller"
], function (Controller) {
   "use strict";
   return Controller.extend("sap.ui.demo.nav.controller.NotFound", {
      onInit: function () {
      }
   });
});
```

Now we create the controller for the `NotFound` view and save it into the `webapp/controller` folder. This controller will be extended later.

webapp/i18n/i18n.properties

```
...
NotFound=Not Found
NotFound.text=Sorry, but the requested resource is not available.
NotFound.description=Please check the URL and try again.
```

Add the new properties to the `i18n.properties` file.

Open the URL `index.html#/thisIsInvalid` in your browser. From now on the user will see a nice *Not Found* page if a hash could not be matched to one of our routes.

Conventions

- Always configure the `bypassed` property and a corresponding target
- Use the `sap.m.MessagePage` control to display routing related error messages

Related Information

API Reference: `sap.m.MessagePage`
API Overview and Samples: `sap.m.MessagePage`

Step 4: Add a *Back* Button to *Not Found* Page

When we are on the *Not Found* page because of an invalid hash, we want to get back to our app to select another page. Therefore, we will add a *Back* button to the *Not Found* view and make sure that the user gets redirected to either the previous page or the overview page when the *Back* button is pressed.

Preview

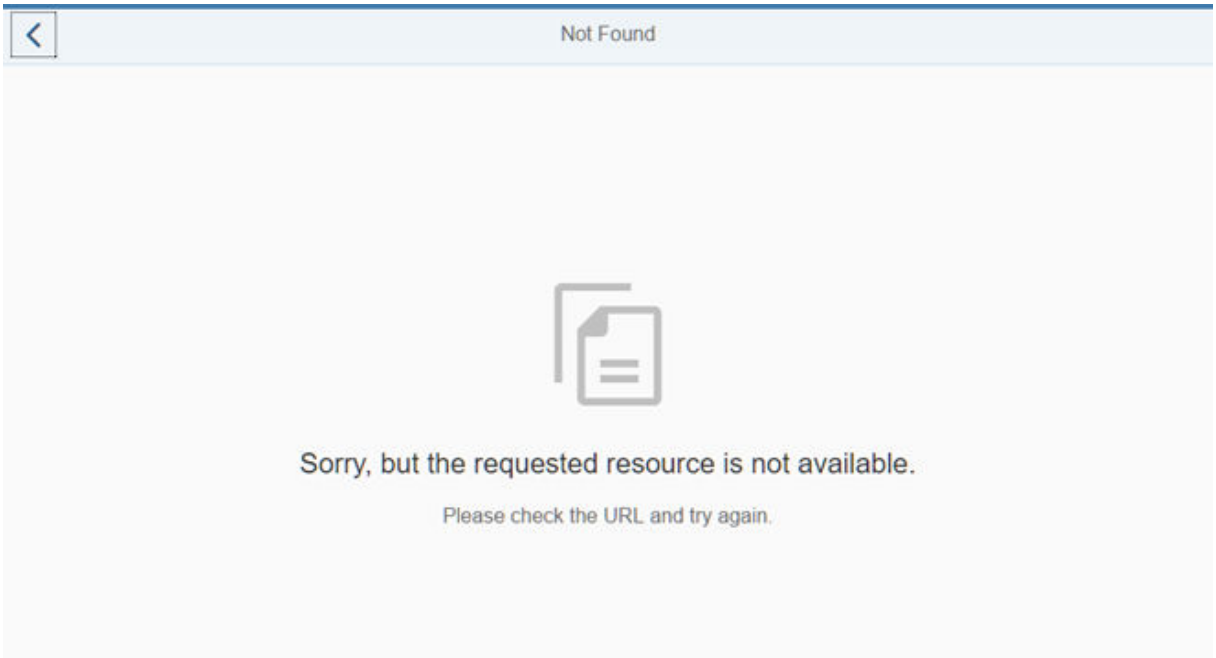


Figure 82: *Not Found* page with *Back* button

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 4.

webapp/view/NotFound.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.NotFound"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <MessagePage
        title="{i18n>NotFound}"
        text="{i18n>NotFound.text}"
        description="{i18n>NotFound.description}"
        showNavButton="true"
        navButtonPress="onNavBack"/>
</mvc:View>
```

In the `NotFound` view, we set the property `showNavButton` of the `MessagePage` control to `true` to automatically display the *Back* button. We also add an event handler function `onNavBack` to the `navButtonPress` event of the control. The `onNavBack` function will handle the actual back navigation. We could directly add this function to the view's controller. However, we are smart enough to anticipate that we might need the same handler function for different views. DRY ("Don't Repeat Yourself") is the right approach for us, so let's create a `BaseController` from which all other controllers will inherit.

webapp/controller/BaseController.js (New)

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/routing/History",
    "sap/ui/core/UIComponent"
], function(Controller, History, UIComponent) {
    "use strict";

    return Controller.extend("sap.ui.demo.nav.controller.BaseController", {

        getRouter : function () {
            return UIComponent.getRouterFor(this);
        },

        onNavBack: function () {
            var oHistory, sPreviousHash;

            oHistory = History.getInstance();
            sPreviousHash = oHistory.getPreviousHash();

            if (sPreviousHash !== undefined) {
                window.history.go(-1);
            } else {
                this.getRouter().navTo("appHome", {}, true /*no history*/);
            }
        }

    });

});
```

Create a new `BaseController.js` file in the `webapp/controller` folder. The base controller implements a set of functions that are reused by its subclasses. The `onNavBack` handler is a great example of code that we don't want to duplicate in our controllers for each page that has a back navigation.

The function checks if there is a previous hash value in the app history. If so, it redirects to the previous hash via the browser's native `History` API. In case there is no previous hash we simply use the router to navigate to the route `appHome` which is our home view.

The third parameter of `navTo("appHome", {}, true /*no history*/);` has the value `true` and makes sure that the hash is replaced. With the line `sap.ui.core.UIComponent.getRouterFor(this)` you can easily access your component's router throughout the app. To make it even more comfortable, we also add a handy shortcut `getRouter` to the base controller. This function is now available in each subclass as well. It is also used in the `onNavBack` handler to get a reference to the router before calling `navTo`. We now have to implement the reuse in all other controllers.

> **i Note**
>
> In SAPUI5 there are multiple options to reuse code. We recommend to use a base controller for such helper methods because this allows us to decoratively use the `onNavBack` handler directly in any XML view without adding additional code to the controller. Our base controller is an abstract controller that will not be instantiated in any view. Therefore, the naming convention `*.controller.js` does not apply, and we can just call the file `BaseController.js`. By not using the naming convention `*.controller.js` we can even prevent any usage in views.

webapp/controller/NotFound.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.NotFound", {
        onInit: function () {
        }
    });
});
```

In order to reuse the base controller implementation, we have to change the dependency from `sap/ui/core/mvc/Controller` to `sap/ui/demo/nav/controller/BaseController` and directly extend the base controller.

At this point you can open `index.html#/thisIsInvalid` in your browser and press the *Back* button to see what happens. You will be redirected to the app's home page that is matched by the route `appHome` as you opened the *Not Found* page with an invalid hash. If you change the hash to something invalid when you are on the home page of the app, you will also go to the *Not Found* page but with a history entry. When you press back, you will get to the home page again, but this time with a native history navigation.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.App", {
        onInit: function () {
        }
    });
});
```

To be consistent, we will now extend all of our controllers with the base controller. Change the app controller as described above.

webapp/controller/Home.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.Home", {
    });
});
```

The same applies to our home controller, we also extend it with the base controller now.

> **i** Note
>
> In this step we have added the *Back* button. The user can always use the browser's native *Back* button as well. Each app can freely configure the behavior of the *Back* button. However, there is no clean way to apply the same logic for the browser's *Back* button in single-page applications. Tweaking the browser history or using other quirks for cancelling backward or forward navigation is not recommended due to the implementation details of the browsers. The browser's *Back* button always uses the browser history while the *Back* button of the app can make use of the browser history **or** can implement its own navigation logic. Make sure to understand this difference and only control the *Back* button inside the app.

Conventions

- Implement a global `onNavBack` handler for back navigation in your app
- Query the history and go to the home page if there is no history available for the current app

Related Information

Routing and Navigation [page 1072]

Step 5: Display a Target Without Changing the Hash

In this step, you will learn more about targets and how to display a target from the routing configuration manually.

We will display the *Not Found* target from the previous step without changing the hash to illustrate this navigation pattern. We will also consider a side-effect that prevents us from navigating back in this case.

Fortunately, we can extend our app and offer an easy solution. There are some use cases that should not be persisted in the URL but just be triggered by the application logic if needed. A target is a navigation-related configuration for a view and we can display targets manually without referencing them in a navigation route. Good examples for this are temporary errors, switching to an edit page for a business object, or going to a *Settings* page. Sometimes you will also have to implement a way back manually.

Preview

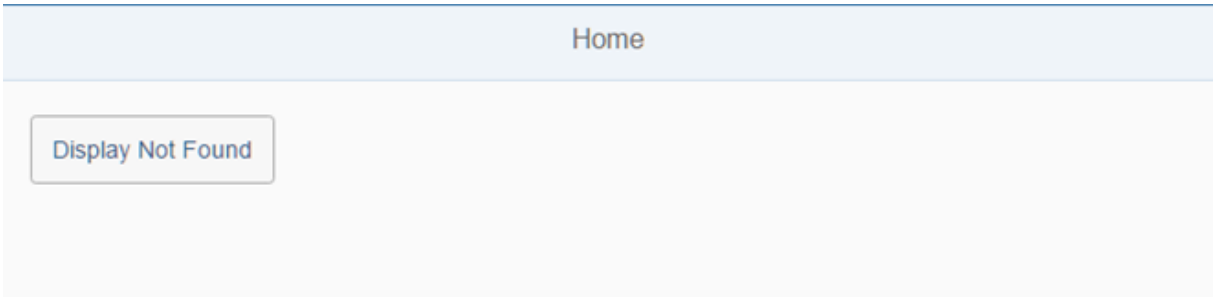


Figure 83: The new *Home* page with a navigation button

Coding

You can view and download all files in the *Samples* in the Demo -kit at Routing and Navigation - Step 5.

webapp/view/Home.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.Home"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}" class="sapUiResponsiveContentPadding">
        <content>
            <Button id="displayNotFoundBtn" text="{i18n>DisplayNotFound}"
press=".onDisplayNotFound" class="sapUiTinyMarginEnd"/>
        </content>
    </Page>
</mvc:View>
```

We start by changing the `Button` control from the home view. When the button is pressed, the `onDisplayNotFound` handler is called.

webapp/controller/Home.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.Home", {
        onDisplayNotFound : function () {
            //display the "notFound" target without changing the hash
            this.getRouter().getTargets().display("notFound");
        }
    });
});
```

Inside the `onDisplayNotFound` handler we get a reference to the `Targets` helper object of the router and simply call `display("notFound")`. The view associated to the target with the name `notFound` from the routing configuration will be displayed by the router without changing the hash.

The `sap.m.routing.Targets` object itself can be retrieved by calling `getTargets()` on the router. It provides a convenient way for placing views into the correct containers of your application. The main benefits of targets are structuring and lazy loading: you just configure the views in the routing configuration and you do not have to load the views until you really need them.

> **i Note**
>
> In the example code we get a reference to the `sap.m.routing.Targets` object by calling `getTargets()` on `this.getRouter()` from the base controller. However, you could also get a reference to the `sap.m.routing.Targets` object by calling `this.getOwnerComponent().getRouter().getTargets()` or `this.getOwnerComponent().getTargets()`.

If you now call the app and press the *Display Not Found* button you see that the `notFound` target is displayed without changing the URL. That was easy, but suddenly our app's *Back* button does not work anymore. The bug we have just introduced illustrates an interesting navigation trap. The application hash is still empty since we just display the target and did not hit a route.

When pressing the app's *Back* button, the `onNavBack` from the previous step is called. It detects that there is no previous hash and therefore tries to navigate to the `appHome` route again. The router is smart enough to detect that the current hash did not change and therefore skips the navigation to the route. Fortunately, there is an easy workaround for us. However, we need to touch the `Home` controller again.

webapp/controller/Home.controller.js (Changed Again)

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.Home", {
        onDisplayNotFound : function () {
            //display the "notFound" target without changing the hash
            this.getRouter().getTargets().display("notFound", {
                fromTarget : "home"
            });
        }
    });
});
```

This time we pass on a data object as the second parameter for the display method which contains the name of the current target; the one from which we navigate to the `notFound` target. We decide to choose the key `fromTarget` but since it is a custom configuration object any other key would be fine as well.

webapp/controller/NotFound.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.NotFound", {
        onInit: function () {
            var oRouter, oTarget;

            oRouter = this.getRouter();
            oTarget = oRouter.getTarget("notFound");
            oTarget.attachDisplay(function (oEvent) {
                this._oData = oEvent.getParameter("data");    // store the data
            }, this);
        },

        // override the parent's onNavBack (inherited from BaseController)
        onNavBack : function () {
            // in some cases we could display a certain target when the back
button is pressed
            if (this._oData && this._oData.fromTarget) {
                this.getRouter().getTargets().display(this._oData.fromTarget);
                delete this._oData.fromTarget;
                return;
            }

            // call the parent's onNavBack
            BaseController.prototype.onNavBack.apply(this, arguments);
        }
    });
});
```

Next, we have to register an event listener to the `display` event of the `notFound` target. The best place for us to register an event listener for this is inside the `init` function of our `NotFound` controller. There we can access and store the custom data that we are passing on when displaying the target manually.

From the router reference we can fetch a reference to the `notFound` target. Each target configuration will create a runtime object that can be accessed through the router.

Similar to SAPUI5 controls, targets define API methods and events that can be attached. We attach a display event handler and save the data that was received as the event parameter `data` in an internal controller variable `_oData`. This data also includes the `fromTarget` information in case the caller passed it on. However, we now have to override the base controller's `onNavBack` implementation to change the behavior a bit. We add a special case for our target back functionality in case the `fromTarget` property has been passed on. If specified, we simply display the target defined as `fromTarget` manually the same way we actually called the `notFound` target manually. Otherwise we just call the base controller's `onNavBack` implementation.

webapp/i18n/i18n.properties

```
...
DisplayNotFound=Display Not Found
```

Add the new property to the `i18n.properties` file.

When we now click the *Back* button, it works as expected and brings us back to the overview page, also when the *Not Found* view is displayed manually.

Conventions

- Display targets manually if you want to trigger a navigation without changing the hash
- Think carefully about all navigation patterns in your application, otherwise the user might get stuck

Related Information

API Reference: `sap.m.routing.Targets`
API Reference: `sap.ui.core.routing.Targets`
API Reference: `sap.ui.core.routing.Target`

Step 6: Navigate to Routes with Hard-Coded Patterns

In this step, we'll create a second button on the home page, with which we can navigate to a simple list of employees. This example illustrates how to navigate to a route that has a hard-coded pattern.

Preview

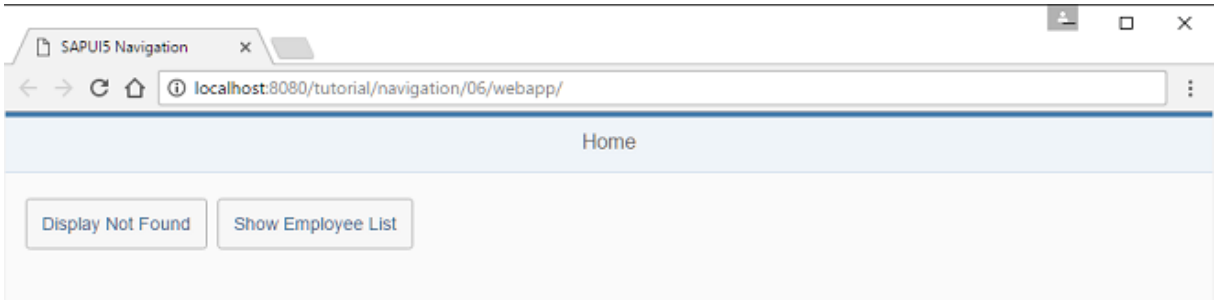


Figure 84: *Show Employee List* button on the *Home* page

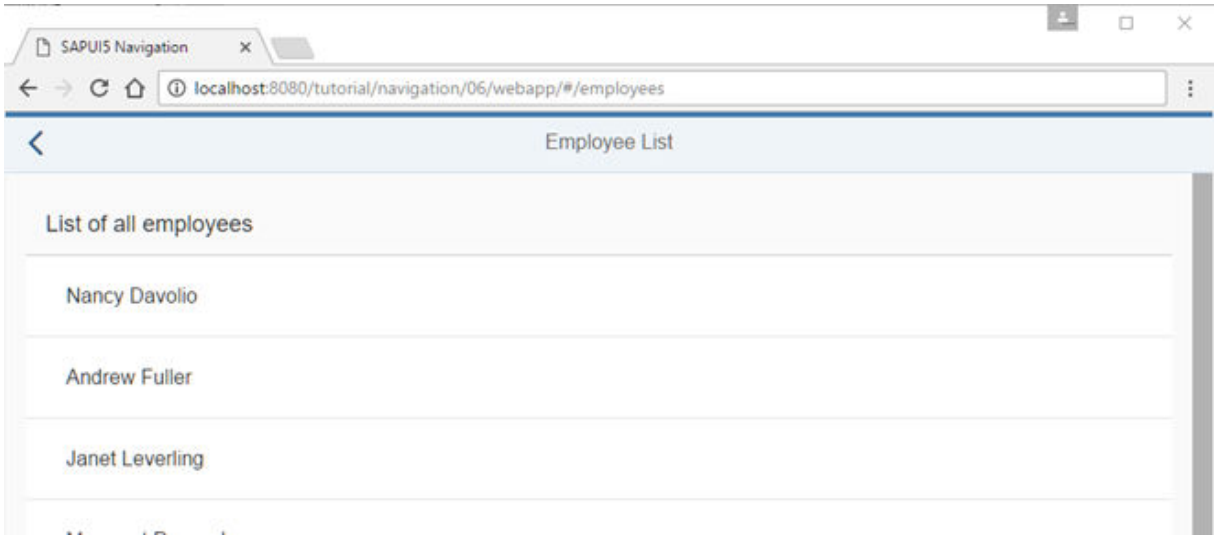


Figure 85: Employee list with *Back* button

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 6.

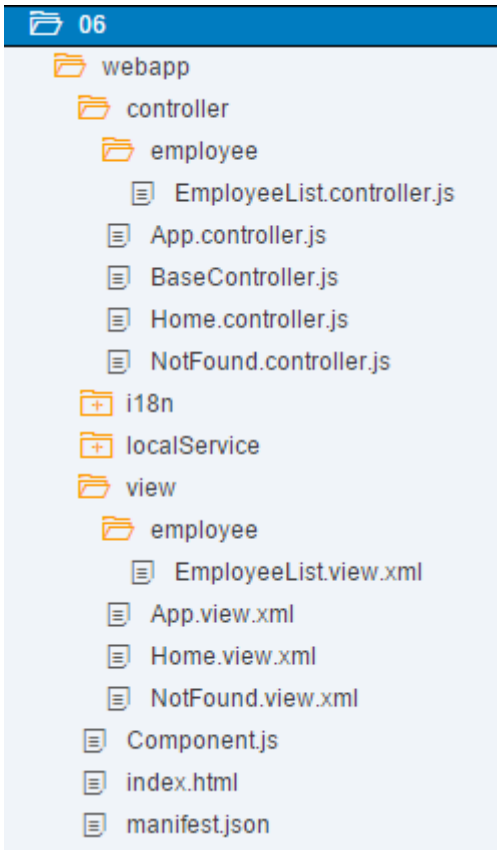


Figure 86: Folder structure for this step

webapp/view/Home.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.Home"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}" class="sapUiResponsiveContentPadding">
        <content>
            <Button id="displayNotFoundBtn" text="{i18n>DisplayNotFound}"
press=".onDisplayNotFound" class="sapUiTinyMarginEnd"/>
            <Button id="employeeListBtn" text="{i18n>ShowEmployeeList}"
press=".onNavToEmployees" class="sapUiTinyMarginEnd"/>
        </content>
    </Page>
```

First, we change the `Home` view by adding the *Show Employee List* button. We register an event handler `onNavToEmployees` for the press event.

webapp/controller/Home.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.Home", {
        onDisplayNotFound : function () {
            // display the "notFound" target without changing the hash
            this.getRouter().getTargets().display("notFound", {
                fromTarget : "home"
            });
        },
        onNavToEmployees : function (){
            this.getRouter().navTo("employeeList");
        }
    });
});
```

The new event handler `onNavToEmployees` calls `navTo("employeeList")` on the router instance. The parameter `employeeList` is the name of the route that we want to navigate to.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
        ...
    },
    "sap.ui": {
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
            "config": {
```

```
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
                "target": "home"
            }, {
                "pattern": "employees",
                "name": "employeeList",
                "target": "employees"
            }],
            "targets": {
                "home": {
                    "viewId": "home",
                    "viewName": "Home",
                    "viewLevel" : 1
                },
                "notFound": {
                    "viewId": "notFound",
                    "viewName": "NotFound",
                    "transition": "show"
                },
                "employees": {
                    "viewId": "employeeList",
                    "viewPath": "sap.ui.demo.nav.view.employee",
                    "viewName": "EmployeeList",
                    "viewLevel" : 2
                }
            }
        }
    }
}
```

To make the navigation work, we have to extend the routing configuration of the app in the descriptor file. We add a new pattern called `employeeList`; this is the name we used in the controller to trigger the navigation.

The pattern of the route is the hard-coded value `employees`, meaning the matching hash for this route is `/#/employees` in the address bar of the browser. The target `employees` should be displayed when this URL pattern is matched.

The `employees` entry in the `targets` section references the `sap.ui.demo.nav.view.employee.EmployeeList` view. As you can see, we added a new namespace `employee` for all views related to employees with the property `viewPath`. This overrides the default settings in the `config` section for the current target.

The view that we are about to create has to be placed in the `webapp/view/employee` folder accordingly. This approach helps to structure the views of the app according to business objects and to better understand the navigation patterns of the app in larger projects.

> **i Note**
>
> We could also have left out the `viewPath` property to use the default `viewPath` defined in the `config` section. In that case, we would have to change the `viewName` to `employee.EmployeeList` to achieve the same effect.

Setting the `viewLevel` to `2` helps the router to determine how to animate the (in our case) `slide` transition. For us, this means that a navigation from the home page to the `employees` target will be animated with a "Slide to Left" animation. In contrast to that, the back navigation from the `employees` target to the home page will be animated with a "Slide to Right" animation. This behavior is due to the fact that the home page has a lower `viewLevel` than the `employees` target.

webapp/view/employee/EmployeeList.view.xml (New)

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.EmployeeList"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        id="employeeListPage"
        title="{i18n>EmployeeList}"
        showNavButton="true"
        navButtonPress=".onNavBack"
        class="sapUiResponsiveContentPadding">
        <content>
            <List id="employeeList" headerText="{i18n>ListOfAllEmployees}"
items="{/Employees}">
                <items>
                    <StandardListItem
                        title="{FirstName} {LastName}"
                        iconDensityAware="false"
                        iconInset="false"/>
                </items>
            </List>
        </content>
    </Page>
</mvc:View>
```

We now create a subfolder `employee` below `webapp/view` and a file `EmployeeList.view.xml`.

We name the folder after the business object, to make it obvious from looking at the hash (included in the browser's address bar) where a view file for a certain business object is located. For example, we can determine from the URL `/#/employee` that the corresponding view must be somewhere in the folder `./employee` (in our case: `webapp/view/employee`) just by looking at the URL.

In the view, we use a `sap.m.List` control and bind its items to the data from our simulated OData service. Note that we have also registered the `onNavBack` handler from the base controller again to be able to navigate back to the overview.

This view can be referenced by `sap.ui.demo.nav.view.employee.EmployeeList`.

webapp/controller/employee/EmployeeList.controller.js (New)

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.EmployeeList", {
```

```
    });
});
```

Finally, we will add a new controller. Create a subfolder `employee` inside `webapp/controller` folder and place the file `EmployeeList.controller.js` there. As you can see, the folder structure of the controllers is in sync with the folder structure of the views.

webapp/i18n/i18n.properties

```
...
ShowEmployeeList=Show Employee List
EmployeeList=Employee List
ListOfAllEmployees=List of all employees
```

Add the new texts to the `i18n.properties` file.

Now you can open the app and press the *Show Employee List* button to navigate to the employee list. From there, you can press either the browser's or the app's *Back* button to get back to the home page.

Related Information

Methods and Events for Navigation [page 1078]
API Reference: `sap.ui.core.routing.Route`

Step 7: Navigate to Routes with Mandatory Parameters

In this step, we implement a feature that allows the user to click on an employee in the list to see additional details of the employee. A route pattern can have one or more mandatory parameters to identify objects in an app.

The detail page has to read the ID of the employee from the URL to fetch and display the employee data from the server. If the employee was not found, for example, because an invalid employee ID was passed on, we want to inform the user by displaying the `notFound` target. Of course, the back navigation has to work as well for this page.

Preview

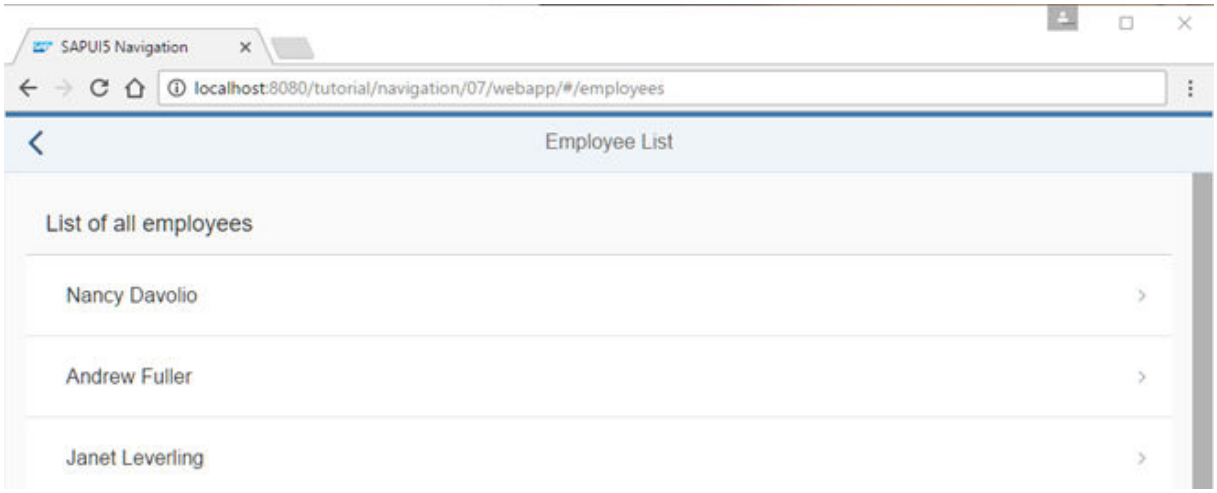


Figure 87: Employee list with navigation option for items

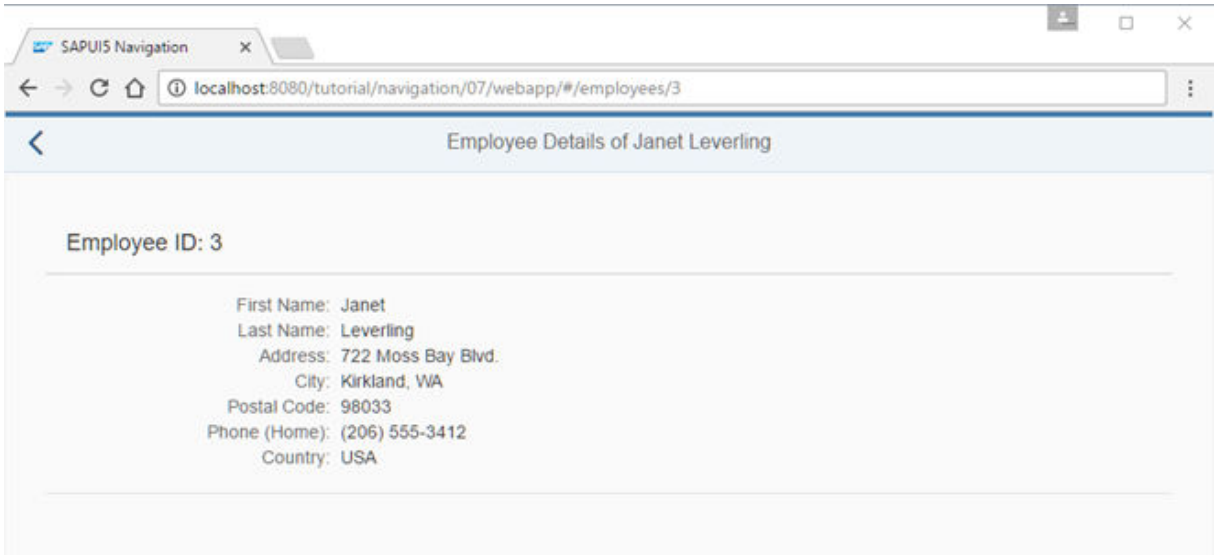


Figure 88: Detail Page for a selected employee

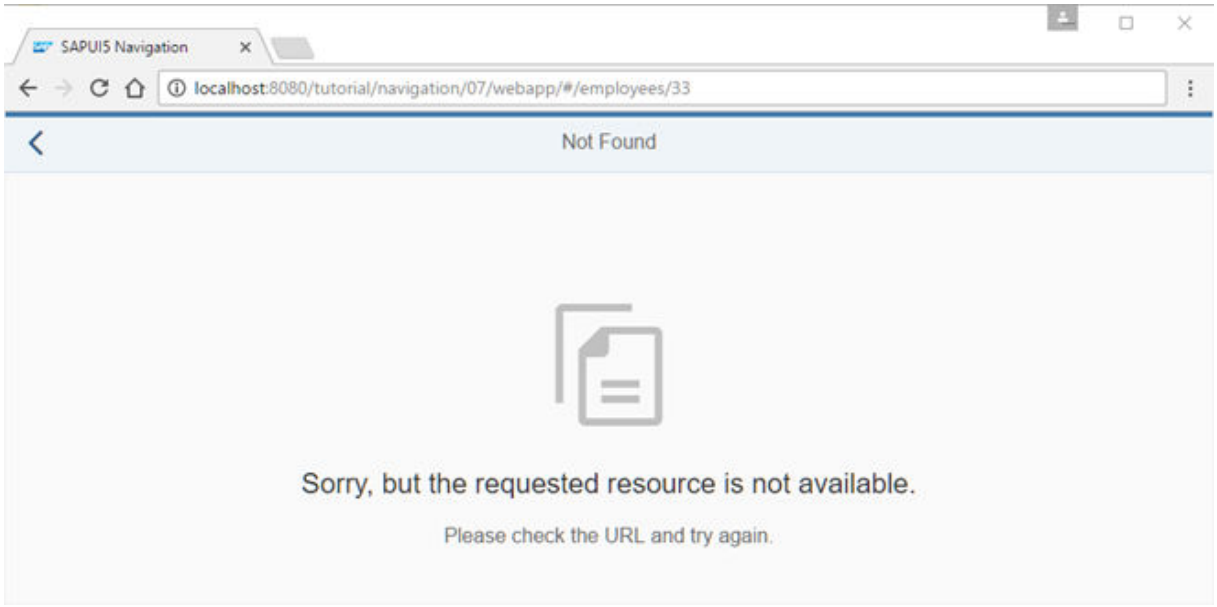


Figure 89: *Not Found* page for an invalid `EmployeeID`

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 7.

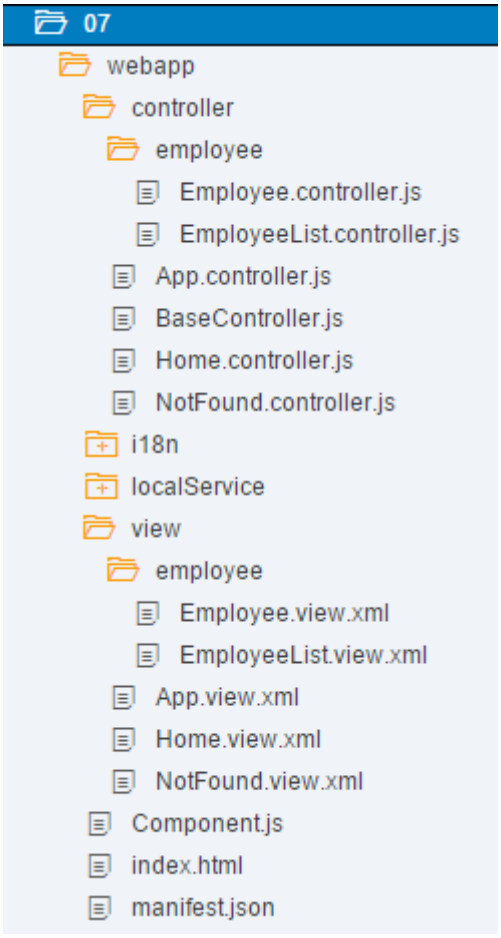


Figure 90: Folder structure for this step

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
        ...
    },
    "sap.ui": {
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
```

```
                "target": "home"
            }, {
                "pattern": "employees",
                "name": "employeeList",
                "target": "employees"
            }, {
                "pattern": "employees/{employeeId}",
                "name": "employee",
                "target": "employee"
            }],
            "targets": {
                "home": {
                    "viewId": "home",
                    "viewName": "Home",
                    "viewLevel" : 1
                },
                "notFound": {
                    "viewId": "notFound",
                    "viewName": "NotFound",
                    "transition": "show"
                },
                "employees": {
                    "viewId": "employeeList",
                    "viewPath": "sap.ui.demo.nav.view.employee",
                    "viewName": "EmployeeList",
                    "viewLevel" : 2
                },
                "employee": {
                    "viewId": "employee",
                    "viewName": "employee.Employee",
                    "viewLevel" : 3
                }
            }
        }
    }
}
```

From our data model (`webapp/localService/metadata.xml` or `webapp/localService/mockdata/Employees.json`), you can see that each employee entity is identified by an `EmployeeID`. We define a new route that expects a mandatory `employeeId` in its pattern to address an employee. Unlike the patterns we used before, this pattern has a dynamic part. We create a new route `employee` and use `employees/{employeeId}` as its pattern.

The `{employeeId}` part of the pattern is a mandatory parameter as indicated by the curly brackets. The hash that contains an actual employee ID is matched against that pattern at runtime.

The following hashes would match in our case: `employees/2`, `employees/7`, `employees/anInvalidId`, and so on. However, the hash `employees/` will not match as it does not contain an ID at all. The target of our route is `employee`. We create the target `employee` with `viewLevel 3`. With that, we make sure that we have the correct slide animation direction.

Next, we have to create the view `employees.Employee`; for better illustration the `viewPath` is not specified this time.

webapp/view/employee/Employee.view.xml (New)

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.Employee"
    xmlns="sap.m"
```

```
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:f="sap.ui.layout.form"
    busyIndicatorDelay="0">
    <Page
        id="employeePage"
        title="{i18n>EmployeeDetailsOf} {FirstName} {LastName}"
        showNavButton="true"
        navButtonPress=".onNavBack"
        class="sapUiResponsiveContentPadding">
        <content>
            <Panel
                id="employeePanel"
                width="auto"
                class="sapUiResponsiveMargin sapUiNoContentPadding">
                <headerToolbar>
                    <Toolbar>
                        <Title text="{i18n>EmployeeIDColon} {EmployeeID}"
level="H2"/>
                        <ToolbarSpacer />
                    </Toolbar>
                </headerToolbar>
                <content>
                    <f:SimpleForm
                        minWidth="1024"
                        editable="false"
                        layout="ResponsiveGridLayout"
                        labelSpanL="3" labelSpanM="3" emptySpanL="4"
emptySpanM="4"
                        columnsL="1" columnsM="1">
                        <f:content>
                            <Label text="{i18n>formFirstName}"/>
                            <Text text="{FirstName}"/>
                            <Label text="{i18n>formLastName}"/>
                            <Text text="{LastName}"/>
                            <Label text="{i18n>formAddress}"/>
                            <Text text="{Address}"/>
                            <Label text="{i18n>formCity}"/>
                            <Text text="{City}, {Region}"/>
                            <Label text="{i18n>formPostalCode}"/>
                            <Text text="{PostalCode}"/>
                            <Label text="{i18n>formPhoneHome}"/>
                            <Text text="{HomePhone}"/>
                            <Label text="{i18n>formCountry}"/>
                            <Text text="{Country}"/>
                        </f:content>
                    </f:SimpleForm>
                </content>
            </Panel>
        </content>
    </Page>
</mvc:View>
```

Create the file `Employee.view.xml` inside the `webapp/view/employee` folder. This employee view displays master data for an employee in a panel with a `SimpleForm` control: first name, last name and so on. The data comes from a relative data binding that is set on the view level as we can see in the controller later. As we are focusing on the navigation aspects in this tutorial, we won't go into detail on the controls of the view. Just copy the code.

webapp/controller/employee/Employee.controller.js (New)

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
```

```
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.employee.Employee",
{
        onInit: function () {
            var oRouter = this.getRouter();
            oRouter.getRoute("employee").attachMatched(this._onRouteMatched,
this);
            // Hint: we don't want to do it this way
            /*
            oRouter.attachRouteMatched(function (oEvent){
                var sRouteName, oArgs, oView;
                sRouteName = oEvent.getParameter("name");
                if (sRouteName === "employee"){
                    this._onRouteMatched(oEvent);
                }
            }, this);
            */
        },
        _onRouteMatched : function (oEvent) {
            var oArgs, oView;
            oArgs = oEvent.getParameter("arguments");
            oView = this.getView();

            oView.bindElement({
                path : "/Employees(" + oArgs.employeeId + ")",
                events : {
                    change: this._onBindingChange.bind(this),
                    dataRequested: function (oEvent) {
                        oView.setBusy(true);
                    },
                    dataReceived: function (oEvent) {
                        oView.setBusy(false);
                    }
                }
            });
        },
        _onBindingChange : function (oEvent) {
            // No data for the binding
            if (!this.getView().getBindingContext()) {
                this.getRouter().getTargets().display("notFound");
            }
        }
    });
});
```

Now we create the file `Employee.controller.js` in the `webapp/controller/employee` folder. In this controller file, we want to detect which employee shall be displayed in order to show the employee's data in the view. Therefore, we query the router for the route `employee` and attach a private event listener function `_onRouteMatched` to the matched event of this route.

In the event handler, we can access the `arguments` parameter from the `oEvent` parameter that contains all parameters of the pattern. Since this listener is only called when the route is matched, we can be sure that the mandatory parameter `employeeId` is always available as a key in `arguments`; otherwise the route would not have matched. The name of the mandatory parameter `employeeId` correlates to the `{employeeId}` from our pattern definition of the route `employee` and thus to the value in the URL.

In `_onRouteMatched` we call `bindElement()` on the view to make sure that the data of the specified employee is available in the view and it's controls. The `ODataModel` will handle the necessary data requests to the back end in the background. While the data is loading, it would be nice to show a busy indicator by simply setting the view to `busy`. Therefore, we pass an events object to `bindElement()` to listen to the events

`dataRequested` and `dataReceived`. The attached functions handle the busy state by calling `oView.setBusy(true)` and `oView.setBusy(false)` respectively.

We also add an event handler to the `change` event as a private function `_onBindingChange`. It checks if the data could be loaded by querying the binding context of the view. As seen in the previous steps, we will display the `notFound` target if the data could not be loaded.

> **i Note**
>
> Instead of calling `attachMatched(...)` on a route we could also call `attachRouteMatched(...)` directly on the router. However, the event for the latter is fired for every matched event of any route in the whole app. We don't use the latter because we would have to implement an additional check for making sure that current route is the route that has been matched. We want to avoid this extra overhead and register on the route instead.

webapp/view/employee/EmployeeList.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.EmployeeList"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        id="employeeListPage"
        title="{i18n>EmployeeList}"
        showNavButton="true"
        navButtonPress=".onNavBack"
        class="sapUiResponsiveContentPadding">
        <content>
            <List id="employeeList" headerText="{i18n>ListOfAllEmployees}"
items="{/Employees}">
                <items>
                    <StandardListItem
                        title="{FirstName} {LastName}"
                        iconDensityAware="false"
                        iconInset="false"
                        type="Navigation"
                        press=".onListItemPressed"/>
                </items>
            </List>
        </content>
    </Page>
</mvc:View>
```

It's time to change the `EmployeeList` view so that we can navigate to the new view. We set the attribute type of the `StandardListItem` template to `Navigation` to make the item clickable and indicate a navigation feature to the user. Additionally, we add an event handler for the `press` event that is called when the user clicks on an employee list item.

webapp/controller/employee/EmployeeList.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
```

```
], function (BaseController) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.EmployeeList", {
        onListItemPressed : function(oEvent){
            var oItem, oCtx;
            oItem = oEvent.getSource();
            oCtx = oItem.getBindingContext();
            this.getRouter().navTo("employee",{
                employeeId : oCtx.getProperty("EmployeeID")
            });
        }
    });
});
```

Finally, we add the handler `onListItemPressed` for the `press` event to the `EmployeeList` controller. In the handler, we determine the `EmployeeID` of the list item by querying the binding context and accessing the property `EmployeeID` from the data model.

Then we navigate to the `employee` route and pass a configuration object on to the `navTo` method with the mandatory parameter `employeeId` filled with the correct `EmployeeID`. The router always makes sure that mandatory parameters as specified in the route's pattern are set; otherwise an error is thrown.

webapp/i18n/i18n.properties

```
...
formEmployeeDetailsOf=Employee Details of
formEmployeeIDColon=Employee ID:
formFirstName=First Name
formLastName=Last Name
formAddress=Address
formCity=City
formPostalCode=Postal Code
formPhoneHome=Phone (Home)
formCountry=Country
```

Add the new texts to the `i18n.properties` file.

That's it. You can go to `webapp/index.html#/employees` and click on any list item to be redirected to corresponding employee's details. Check also what happens when you directly navigate to the following files:

- `webapp/index.html#/employees/3`
- `webapp/index.html#/employees/33`

Related Information

API Reference: `sap.ui.model.Binding`

Step 8: Navigate with Flip Transition

In this step, we want to illustrate how to navigate to a page with a custom transition animation. Both forward and backward navigation will use the "flip" transition but with a different direction. We will create a simple link on the *Employee* view that triggers a flip navigation to a page that displays the resume data of a certain employee. Pressing the *Back* button will navigate back to the *Employee* view with a reversed flip transition.

Preview

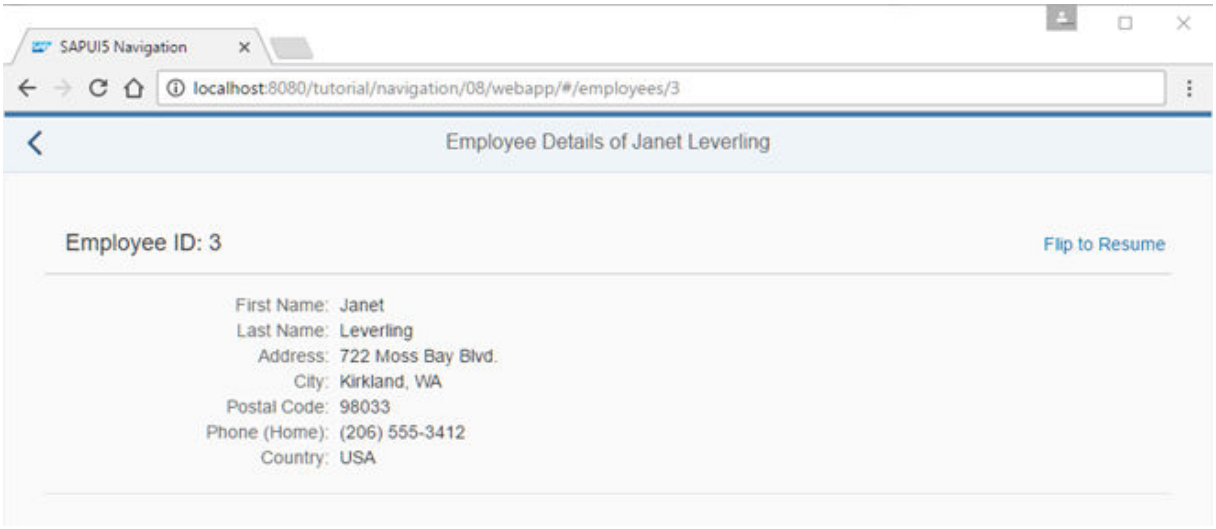


Figure 91: *Employee Details* page with *Flip to Resume* link

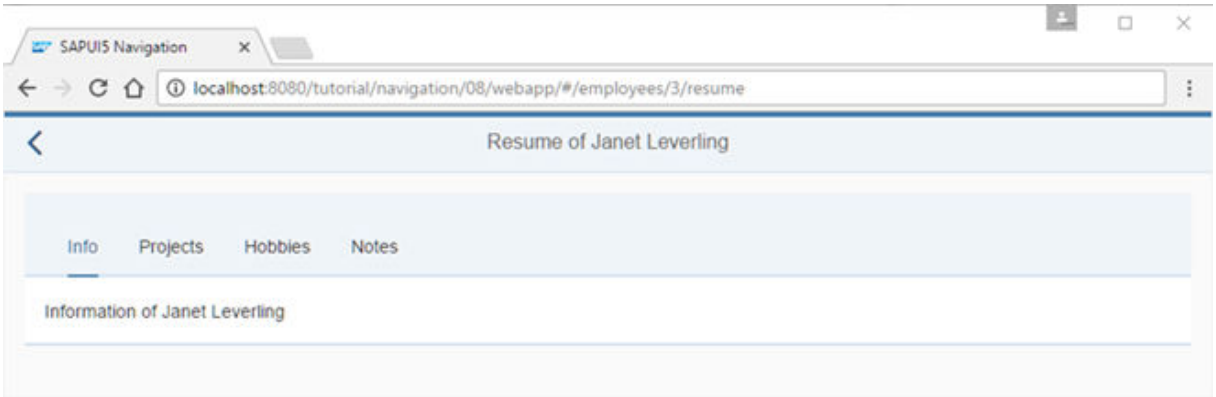


Figure 92: *Resume* page with multiple tabs

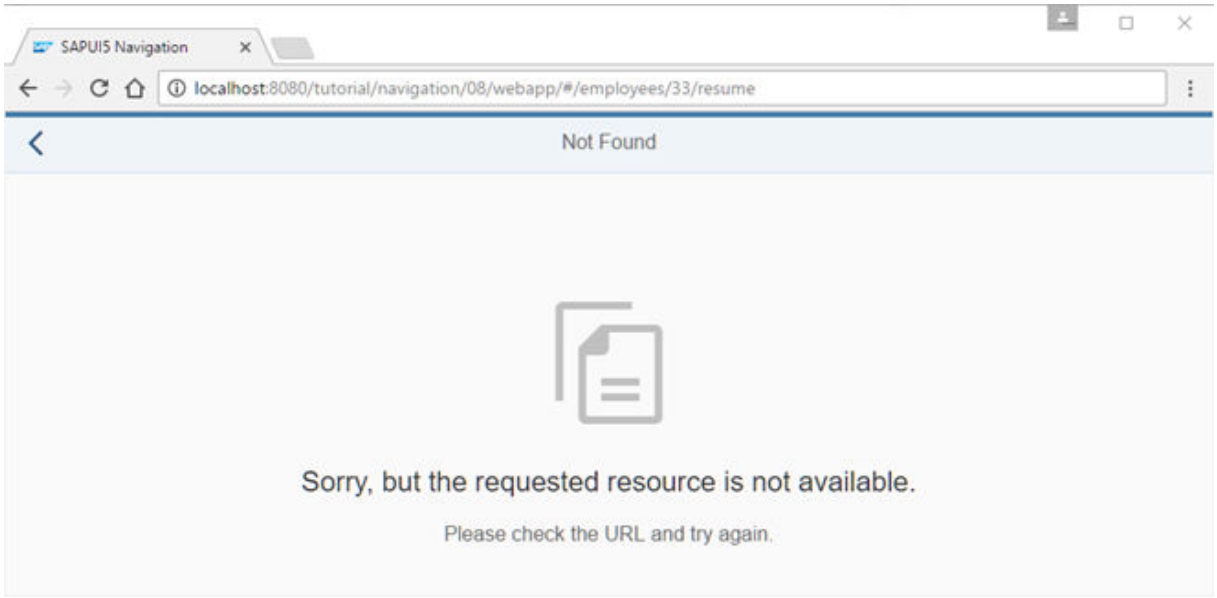


Figure 93: *Not Found* page for resume

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 8.

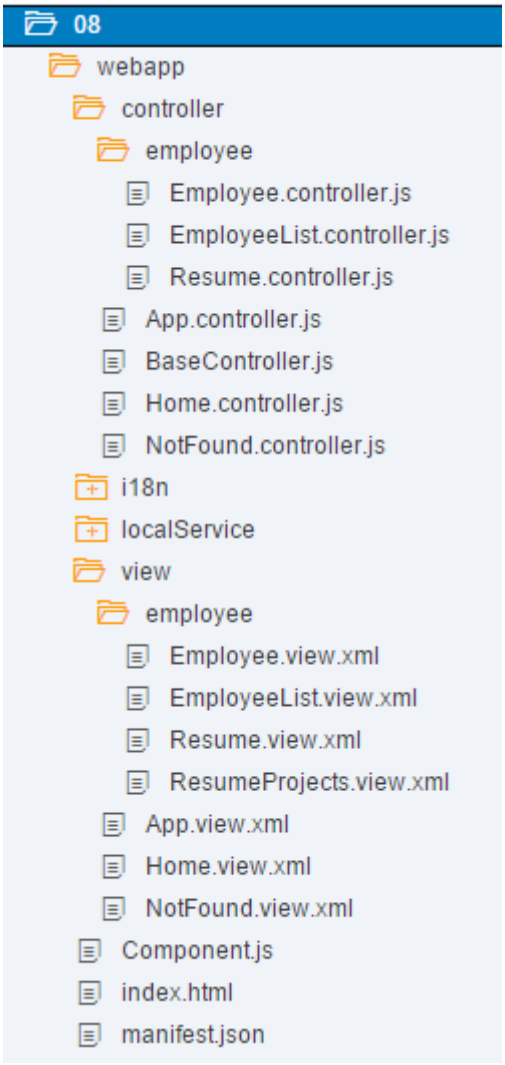


Figure 94: Folder structure for this step

webapp/view/employee/Employee.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.Employee"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:f="sap.ui.layout.form"
    busyIndicatorDelay="0">
    <Page
        id="employeePage"
        title="{i18n>EmployeeDetailsOf} {FirstName} {LastName}"
        showNavButton="true"
        navButtonPress=".onNavBack"
        class="sapUiResponsiveContentPadding">
        <content>
            <Panel
                id="employeePanel"
                width="auto"
                class="sapUiResponsiveMargin sapUiNoContentPadding">
                <headerToolbar>
                    <Toolbar>
```

```
                        <Title text="{i18n>EmployeeIDColon} {EmployeeID}"
level="H2"/>
                        <ToolbarSpacer />
                        <Link text="{i18n>FlipToResume}"
tooltip="{i18n>FlipToResume.tooltip}" press=".onShowResume"/>
                    </Toolbar>
                </headerToolbar>
                <content>
                    ...
                </content>
            </Panel>
        </content>
    </Page>
</mvc:View>
```

First we add the *Flip to Resume* link to the *Employee Details* view to trigger the navigation to the resume of the employee that is currently displayed.

webapp/controller/employee/Employee.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.employee.Employee",
{
        ...
        _onBindingChange : function (oEvent) {
            // No data for the binding
            if (!this.getView().getBindingContext()) {
                this.getRouter().getTargets().display("notFound");
            }
        }
            ...
        },
        onShowResume : function (oEvent) {
            var oCtx = this.getView().getElementBinding().getBoundContext();

            this.getRouter().navTo("employeeResume", {
                employeeId : oCtx.getProperty("EmployeeID")
            });
        }
    });
});
```

Then we change the `Employee.controller.js` file by adding the press handler `onShowResume` for the *Flip to Resume* link. The handler simply navigates to a new route `employeeResume` and fills the mandatory parameter `employeeId` with the property `EmployeeID` from the view's bound context. The route `employeeResume` is not available yet, so we will have to add it to our routing configuration.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
```

```
        ...
    },
    "sap.ui": {
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
                "target": "home"
            }, {
                "pattern": "employees",
                "name": "employeeList",
                "target": "employees"
            }, {
                "pattern": "employees/{employeeId}",
                "name": "employee",
                "target": "employee"
            }, {
                "pattern": "employees/{employeeId}/resume",
                "name": "employeeResume",
                "target": "employeeResume"
            }],
            "targets": {
                "home": {
                    "viewId": "home",
                    "viewName": "Home",
                    "viewLevel" : 1
                },
                "notFound": {
                    "viewId": "notFound",
                    "viewName": "NotFound",
                    "transition": "show"
                },
                "employees": {
                    "viewId": "employees",
                    "viewPath": "sap.ui.demo.nav.view.employee",
                    "viewName": "EmployeeList",
                    "viewLevel" : 2
                },
                "employee": {
                    "viewId": "employee",
                    "viewName": "employee.Employee",
                    "viewLevel" : 3
                },
                "employeeResume": {
                    "viewId": "resume",
                    "viewName": "employee.Resume",
                    "viewLevel" : 4,
                    "transition": "flip"
                }
            }
        }
    }
}
```

In the routing configuration, we add a new route `employeeResume` which references a target with the same name. The route's pattern expects an `{employeeId}` as a mandatory parameter and ends with the static string `/resume`.

The target `employeeResume` references the view `employee.Resume` that we are about to create. The target's `viewLevel` is `4`; compared to the employee target this is one level lower again. To configure a flip navigation, we simply set the transition of our target to `flip`. Together with the correct `viewLevel` configuration this will trigger the correct forward and backward flip navigation whenever the target is displayed.

> **i Note**
>
> Possible values for the `transition` parameter are:
>
> - `slide` (default)
> - `flip`
> - `show`
> - `fade`
>
> You can also implement your own transitions and add it to a control that extends `sap.m.NavContainer` (for example, `sap.m.App` or `sap.m.SplitApp`). For more information, see the *API Reference* for `NavContainer`.

webapp/view/employee/Resume.view.xml (New)

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.Resume"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        title="{i18n>ResumeOf} {FirstName} {LastName}"
        id="employeeResumePage"
        showNavButton="true"
        navButtonPress=".onNavBack">
        <content>
            <IconTabBar
                id="iconTabBar"
                headerBackgroundDesign="Transparent"
                class="sapUiResponsiveContentPadding"
                binding="{Resume}">
                <items>
                    <IconTabFilter id="infoTab" text="{i18n>tabInfo}" key="Info">
                        <Text text="{Information}"/>
                    </IconTabFilter>
                    <IconTabFilter id="projectsTab" text="{i18n>tabProjects}"
key="Projects">
                        <mvc:XMLView
viewName="sap.ui.demo.nav.view.employee.ResumeProjects"></mvc:XMLView>
                    </IconTabFilter>
                    <IconTabFilter id="hobbiesTab" text="{i18n>tabHobbies}"
key="Hobbies">
                        <Text text="{Hobbies}"/>
                    </IconTabFilter>
                    <IconTabFilter id="notesTab" text="{i18n>tabNotes}"
key="Notes">
                        <Text text="{Notes}"/>
                    </IconTabFilter>
                </items>
```

```
            </IconTabBar>
        </content>
    </Page>
</mvc:View>
```

Create a file `Resume.view.xml` inside the `webapp/view/employee` folder. The view uses an `IconTabBar` to display the resume data. Therefore, its binding attribute is set to `{Resume}`.

In the `IconTabBar` we display four tabs. Three of them simply use a `Text` control to display the data from the service. The *Projects* tab uses a nested XML view to display the projects of the employee. SAPUI5 takes care of loading the XML view automatically when the user navigates to the *Resume* page.

webapp/controller/employee/Resume.controller.js (New)

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.employee.Resume", {
        onInit: function () {
            var oRouter = this.getRouter();

oRouter.getRoute("employeeResume").attachMatched(this._onRouteMatched, this);
        },
        _onRouteMatched : function (oEvent) {
            var oArgs, oView;
            oArgs = oEvent.getParameter("arguments");
            oView = this.getView();
            oView.bindElement({
                path : "/Employees(" + oArgs.employeeId + ")",
                events : {
                    change: this._onBindingChange.bind(this),
                    dataRequested: function (oEvent) {
                        oView.setBusy(true);
                    },
                    dataReceived: function (oEvent) {
                        oView.setBusy(false);
                    }
                }
            });
        },
        _onBindingChange : function (oEvent) {
            // No data for the binding
            if (!this.getView().getBindingContext()) {
                this.getRouter().getTargets().display("notFound");
            }
        }
    });
});
```

Create a file `Resumee.controller.js` in the `webapp/controller/employee` folder. In this controller, we make sure to bind the view to the correct employee whenever the `employeeResume` route has matched. We have already used this approach in the previous step so you should be able to recognize the building blocks in the code above. Again, in case the user cannot be found we display the `notFound` target.

webapp/view/employee/ResumeProjects.view.xml (New)

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <Text text="{Projects}"/>
</mvc:View>
```

Create a file `ResumeProjects.view.xml` in the `webapp/view/employee` folder. This view does not have a controller as we don't need it. It just displays a `Text` control with the projects text of the selected employee. It illustrates that using nested views works just fine in combination with navigation and routing in SAPUI5.

> **i** Note
>
> For more complex applications, the performance is significantly increased if parts of the UI are only loaded when the user is actively selecting it. In this example, the view is always loaded even though the user never decided to display the project information. In the next steps, we will extend the UI so that the content is loaded "lazy" by SAPUI5 only when the filter item is clicked. The back-end service will fetch the data only on request and the UI will only have to be updated with the selected data instead of loading all data.

webapp/i18n/i18n.properties

```
...
ResumeOf=Resume of
tabInfo=Info
tabProjects=Projects
tabHobbies=Hobbies
tabNotes=Notes
FlipToResume=Flip to Resume
FlipToResume.tooltip=See the resume of this employee
```

Add the new texts to the `i18n.properties` file.

You can go to `webapp/index.html#/employees/3` and click on the *Flip to Resume* link to be redirected with a nice flip transition to the employee's resume. The back navigation uses a reverse flip navigation to get back to the *Employee Details* page. You can also directly navigate to `webapp/index.html#/employees/3/resume` or `webapp/index.html#/employees/33/resume` to see what happens.

Related Information

API Reference: `sap.m.NavContainer`

API Overview and Samples: `sap.m.NavContainer`

Step 9: Allow Bookmarkable Tabs with Optional Query Parameters

The `resume` view contains four tabs as we have seen in the previous steps. However, when the user navigates to the `resume` page, only the first tab is displayed initially. Navigating directly to a specific tab or bookmarking a tab is not yet supported in our current app.

In this step, we implement a bookmarking feature by enabling deep linking to tabs with optional query parameters. A deep link is basically a link that directly references a deeper structure and parameters of the app in the URL. It is often bookmarked or shared to have a convenient entry point into the app for a certain task or action. The selected tab should be reflected in the URL but the tab can also be omitted, for example, when we initially navigate to the resume page.

Preview

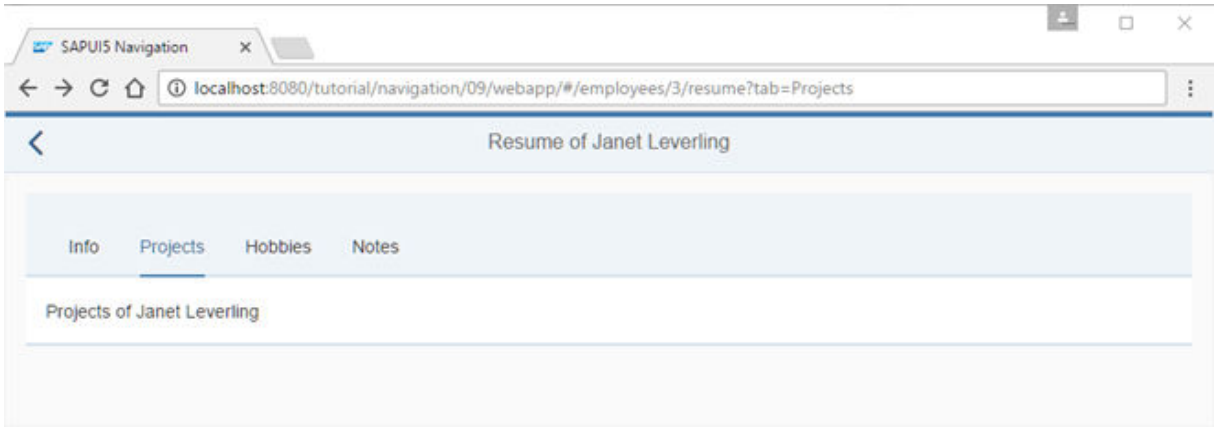


Figure 95: Deep link to allow bookmarkable tabs

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 9.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
        ...
    },
    "sap.ui": {
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
```

```
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
                "target": "home"
            }, {
                "pattern": "employees",
                "name": "employeeList",
                "target": "employees"
            }, {
                "pattern": "employees/{employeeId}",
                "name": "employee",
                "target": "employee"
            }, {
                "pattern": "employees/{employeeId}/resume:?query:",
                "name": "employeeResume",
                "target": "employeeResume"
            }],
            "targets": {
                ...
            }
        }
    }
}
```

Up until now, you could only navigate to an employee's resume with the deep link `webapp/index.html#/employees/3/resume`. This will always select the first tab as implemented by the `IconTabBar` control. In order to open the page directly with a specific tab selected and to make the tabs bookmarkable, we add the `?query` parameter to the URL pattern.

This allows URLs like `webapp/index.html#/employees/3/resume?tab=Projects` where the query parameter defines which tab shall be displayed. We change the pattern of the `employeeResume` route to `employees/{employeeId}/resume:?query:`. The new part `:?query:` allows to pass on queries with any parameters, for example, the hash `/#/employees/3/resume?tab=Projects` or `/#/employees/3/resume?tab=Projects&action=edit` matches the pattern and can be processed in the matched event.

The `:?query:` parameter starts and ends with ":", which means that it is optional. If you want to make it mandatory, you can use the `{?query}` syntax (everything in between `{}` is considered as being mandatory).

webapp/view/employee/Resume.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.Resume"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        title="{i18n>ResumeOf} {FirstName} {LastName}"
        id="employeeResumePage"
        showNavButton="true"
```

```
        navButtonPress=".onNavBack">
        <content>
            <IconTabBar
                id="iconTabBar"
                headerBackgroundDesign="Transparent"
                class="sapUiResponsiveContentPadding"
                binding="{Resume}"
                select=".onTabSelect"
                selectedKey="{view>/selectedTabKey}">
                <items>
                    <IconTabFilter id="infoTab" text="{i18n>tabInfo}" key="Info">
                        <Text text="{Information}"/>
                    </IconTabFilter>
                    <IconTabFilter id="projectsTab" text="{i18n>tabProjects}"
key="Projects">
                        <mvc:XMLView
viewName="sap.ui.demo.nav.view.employee.ResumeProjects"></mvc:XMLView>
                    </IconTabFilter>
                    <IconTabFilter id="hobbiesTab" text="{i18n>tabHobbies}"
key="Hobbies">
                        <Text text="{Hobbies}"/>
                    </IconTabFilter>
                    <IconTabFilter id="notesTab" text="{i18n>tabNotes}"
key="Notes">
                        <Text text="{Notes}"/>
                    </IconTabFilter>
                </items>
            </IconTabBar>
        </content>
    </Page>
</mvc:View>
```

To update the currently selected tab in the URL we listen to the select event of the `IconTabBar` by setting `select=".onTabSelect"` in the resume view. The `selectedKey` is bound to a view model. This allows to easily change the `selectedKey` according to the selected tab in the URL.

webapp/controller/employee/Resume.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/ui/model/json/JSONModel"
], function (BaseController, JSONModel) {
    "use strict";
    var _aValidTabKeys = ["Info", "Projects", "Hobbies", "Notes"];
    return BaseController.extend("sap.ui.demo.nav.controller.employee.Resume", {
        onInit: function () {
            var oRouter = this.getRouter();
            this.getView().setModel(new JSONModel(), "view");

oRouter.getRoute("employeeResume").attachMatched(this._onRouteMatched, this);
        },
        _onRouteMatched: function (oEvent) {
            var oArgs, oView, oQuery;
            oArgs = oEvent.getParameter("arguments");
            oView = this.getView();
            oView.bindElement({
                path: "/Employees(" + oArgs.employeeId + ")",
                events: {
                    change: this._onBindingChange.bind(this),
                    dataRequested: function (oEvent) {
                        oView.setBusy(true);
                    },
```

```
                    dataReceived: function (oEvent) {
                        oView.setBusy(false);
                    }
                }
            });
            oQuery = oArgs["?query"];
            if (oQuery && _aValidTabKeys.indexOf(oQuery.tab) > -1){
                oView.getModel("view").setProperty("/selectedTabKey",
oQuery.tab);
            } else {
                // the default query param should be visible at all time
                this.getRouter().navTo("employeeResume", {
                    employeeId: oArgs.employeeId,
                    "?query": {
                        tab: _aValidTabKeys[0]
                    }
                }, true /*no history*/);
            }
        },
        _onBindingChange: function (oEvent) {
            // No data for the binding
            if (!this.getView().getBindingContext()) {
                this.getRouter().getTargets().display("notFound");
            }
        },
        onTabSelect: function (oEvent){
            var oCtx = this.getView().getBindingContext();
            this.getRouter().navTo("employeeResume", {
                employeeId: oCtx.getProperty("EmployeeID"),
                "?query": {
                    tab: oEvent.getParameter("selectedKey")
                }
            }, true /*without history*/);
        }
    });
});
```

When a tab is selected manually, its select handler is called. Therefore, let's first have a look at the `onTabSelect` event handler that is added at the end of the `resume` controller. It detects the `selectedKey` of the tab and navigates to the `employeeResume` route to update the URL in the address bar. Additionally to the mandatory parameter `employeeId`, we pass on a custom `query` object with a parameter `tab` and fill it with the `selectedKey` value that we receive from the `select` event of the `IconTabBar`. By passing on `true` as the third argument we replace the current history to make sure that manually clicked tabs won't be added to the browser history.

A dependency to `sap/ui/model/json/JSONModel` is added to the controller. Now, we modify the `onInit` function to instantiate a JSONModel and use it as the `view` model. `_aValidTabKeys` is added to the controller. We want to make sure that only valid tabs can be selected. Therefore, the `array _aValidTabKeys` contains all allowed tab keys that we can check against to validate the tab parameter from the URL later. The keys are equal to the keys of our `IconTabFilters` in the `resume` view.

In the `_onRouteMatched` event handler, we add the `oQuery` variable to store a reference to the query object from the router. This allows a more comfortable access to the query object.

In case a query object is passed on and the `tab` parameter has a valid value, we display the specific tab by updating the property `selectedTabKey` in the view model. As the `selectedKey` property of the `IconTabBar` is bound to `{view>/selectedTabKey}` the corresponding tab is selected.

The `else` case is called when either no or an invalid tab parameter is specified. We navigate to the *Info* tab to make sure that the tab parameter is reflected in the URL at all times. The actual requirements of your app might differ, feel free to change it accordingly...

From now on our tabs are bookmarkable. Try to access the following (deep) links directly:

- `webapp/index.html#/employees/3/resume`
- `webapp/index.html#/employees/3/resume?tab=Info`
- `webapp/index.html#/employees/3/resume?tab=Projects`
- `webapp/index.html#/employees/3/resume?tab=Hobbies`
- `webapp/index.html#/employees/3/resume?tab=Notes`
- `webapp/index.html#/employees/3/resume?tab=SomethingInvalid`

When you click on any tab you will see that the hash in the URL changes immediately, and when you change the hash in the URL parameter manually, you can see that the UI is also updated accordingly.

Related Information

API Reference: `sap.m.IconTabBar`

Step 10: Implement “Lazy Loading”

In the previous steps, we have implemented a *Resume* view that uses tabs to display data. The complete content of the tabs is loaded once, no matter which tab is currently displayed. We can increase the performance of our app by avoiding to load content that is not visible. Therefore, we implement a “lazy loading” feature that only loads the view and data when requested by the user.

Preview

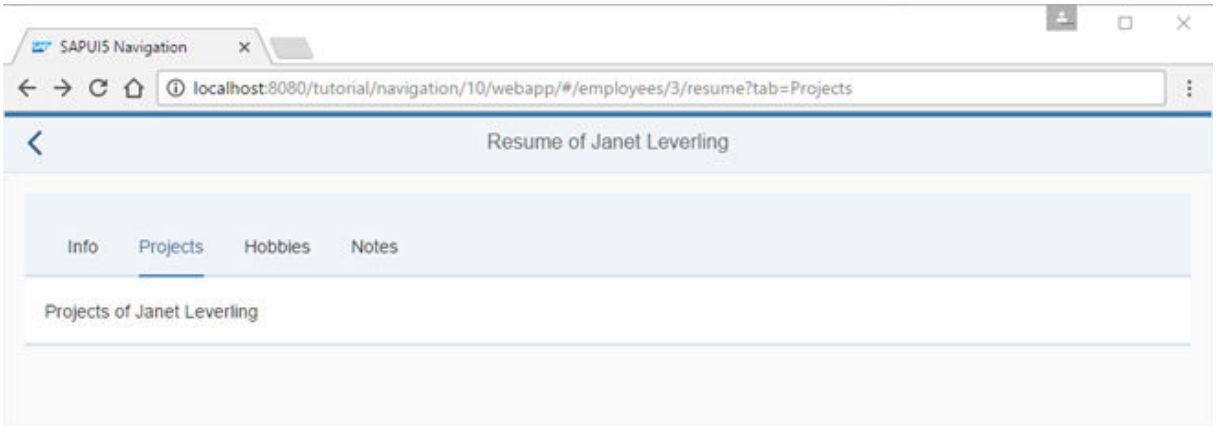


Figure 96: Tabs with lazy loading

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 10 .

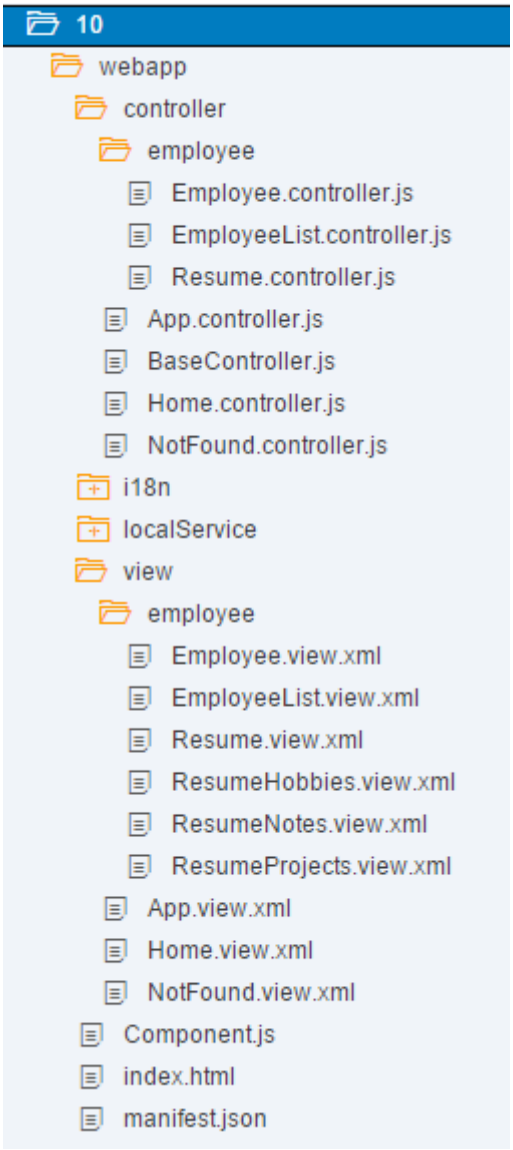


Figure 97: Folder Structure for this Step

webapp/view/employee/Resume.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.employee.Resume"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page
        title="{i18n>ResumeOf} {FirstName} {LastName}"
        id="employeeResumePage"
        showNavButton="true"
        navButtonPress=".onNavBack">
        <content>
            <IconTabBar
                id="iconTabBar"
                headerBackgroundDesign="Transparent"
                class="sapUiResponsiveContentPadding"
                binding="{Resume}"
                select=".onTabSelect"
```

```
                selectedKey="{view>/selectedTabKey}">
                <items>
                    <IconTabFilter id="infoTab" text="{i18n>tabInfo}" key="Info">
                        <Text text="{Information}"/>
                    </IconTabFilter>
                    <IconTabFilter id="projectsTab" text="{i18n>Projects}"
key="Projects">
                        <mvc:XMLView
viewName="sap.ui.demo.nav.view.employee.ResumeProjects"></mvc:XMLView>
                    </IconTabFilter>
                    <IconTabFilter id="hobbiesTab" text="{i18n>Hobbies}"
key="Hobbies">
                        <!-- place content via lazy loading -->
                    </IconTabFilter>
                    <IconTabFilter id="notesTab" text="{i18n>Notes}" key="Notes">
                        <!-- place content via lazy loading -->
                    </IconTabFilter>
                </items>
            </IconTabBar>
        </content>
    </Page>
</mvc:View>
```

To illustrate lazy loading, we implement that the content is loaded only when the user selects the corresponding tab for two of our tabs from the `IconTabBar`: *Hobbies* and *Notes*. The `IconTabFilter` controls each have a hard-coded ID so that we can address them later in our routing configuration. In real use cases, you would do this for tabs that contain a lot of content or trigger expensive service calls to a back-end service.

In the `resume` view we remove the content of the *Hobbies* and *Notes* tabs as we will now fill it dynamically with navigation features.

webapp/view/employee/ResumeHobbies.view.xml (New)

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <Text text="{Hobbies}"/>
</mvc:View>
```

Create the file `ResumeHobbies.view.xml` in the `webapp/view/employee` folder. Move the content for the tab that was previously in the `resume` view to that view. We don't need a controller for this view as there is no additional logic involved. This view will be lazy-loaded and placed into the content of the *Hobbies* tab with navigation features.

webapp/view/employee/ResumeNotes.view.xml (New)

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <Text text="{Notes}"/>
</mvc:View>
```

Create the file `ResumeNotes.view.xml` in the `webapp/view/employee` folder similar to the *Hobbies* view to transform this tab to a separate view as well.

webapp/controller/employee/Resume.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/ui/model/json/JSONModel"
], function (BaseController, JSONModel) {
    "use strict";
    var _aValidTabKeys = ["Info", "Projects", "Hobbies", "Notes"];
    return BaseController.extend("sap.ui.demo.nav.controller.employee.Resume", {
        ...
        _onRouteMatched : function (oEvent) {
            var oArgs, oView, oQuery;
            oArgs = oEvent.getParameter("arguments");
            oView = this.getView();
            oView.bindElement({
                ...
            });
            oQuery = oArgs["?query"];
            if (oQuery && _aValidTabKeys.indexOf(oQuery.tab) > -1){
                oView.getModel("view").setProperty("/selectedTabKey",
oQuery.tab);
                // support lazy loading for the hobbies and notes tab
                if (oQuery.tab === "Hobbies" || oQuery.tab === "Notes"){
                    // the target is either "resumeTabHobbies" or
"resumeTabNotes"
                    this.getRouter().getTargets().display("resumeTab" +
oQuery.tab);
                }
            } else {
                // the default query param should be visible at all time
                this.getRouter().navTo("employeeResume", {
                    employeeId : oArgs.employeeId,
                    "?query": {
                        tab : _aValidTabKeys[0]
                    }
                }, true /*no history*/);
            }
        },
        ...
    });
});
```

Now we extend the `resume` controller a little and add additional logic to the part of the `_onRouteMatched` function where a new tab has been selected and validated. In case the `selectedKey` matches `Hobbies` or `Notes` we call `this.getRouter().getTargets().display("resumeTab" + oQuery.tab)` to display the corresponding target manually. Here the valid targets are `resumeTabHobbies` and `resumeTabNotes` as we have changed the behavior for these two tabs by creating separate views.

These lines of code make sure that the targets are only loaded when they are needed (“lazy loading”). But the router does not know the new targets yet, so let’s create them in our routing configuration.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
        ...
    },
```

```
    "sap.ui": {
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                ...
            }, {
                "pattern": "employees/{employeeId}/resume:?query:",
                "name": "employeeResume",
                "target": "employeeResume"
            }],
            "targets": {
                ...
                "employeeResume": {
                    "viewId": "resume",
                    "viewName": "employee.Resume",
                    "viewLevel" : 4,
                    "transition": "flip"
                },
                "resumeTabHobbies": {
                    "viewId": "resumeHobbies",
                    "parent": "employeeResume",
                    "viewPath": "sap.ui.demo.nav.view.employee",
                    "viewName": "ResumeHobbies",
                    "controlId": "hobbiesTab",
                    "controlAggregation": "content"
                },
                "resumeTabNotes": {
                    "viewId": "resumeNotes",
                    "parent": "employeeResume",
                    "viewPath": "sap.ui.demo.nav.view.employee",
                    "viewName": "ResumeNotes",
                    "controlId": "notesTab",
                    "controlAggregation": "content"
                }
            }
        }
    }
}
```

We add the `resumeTabHobbies` and `resumeTabNotes` targets to the descriptor file with additional fields that override the default configuration as we now want to display the targets locally inside the `IconTabBar` control and not as pages of the app.

The `resumeTabHobbies` target sets the parent property to `employeeResume`. The parent property expects the name of another target. In our case, this makes sure that the view from the parent target `employeeResume` is loaded before the target `resumeTabHobbies` is displayed. This can be considered as a “view dependency”. By setting the `controlId` and `controlAggregation` properties the router places the view `ResumeHobbies` into the `content` aggregation of the `IconTabFilter` control with ID `hobbiesTab`. We also set a parameter `viewId` to a custom ID to illustrate how you could overrule a hard-coded ID inside a view.

> **i Note**
>
> Each target can define only one parent with its parent property. This is similar to the SAPUI5 control tree where each control can have only one parent control (accessed with the method `getParent()` of `sap.ui.base.ManagedObject`). The `controlId` property always references a control inside the parent view that is specified with the `parent` target.

Now we add the `resumeTabNotes` target similar to the `Hobbies` target. The `resumeTabNotes` target defines the parent target `employeeResume` as well, because they share the same parent view. We place the `ResumeNotes` view into the `content` aggregation of the `IconTabFilter` control with ID `notesTab`.

We have now implemented lazy loading for the tabs *Hobbies* and *Notes*. These two tabs are now managed by the routing configuration and only loaded when we click on them the first time.

Try it out yourself: Open the *Network* tab of your browser's developer tools and click on the tabs of your app. In the network traffic you will see that `ResumeHobbies.view.xml` file is only loaded when the *Hobbies* tab is displayed the first time. The same applies for the *Notes* tab. Mission accomplished!

Conventions

- Lazy-load content that is not initially displayed to the user

Related Information

API Reference: `ap.m.routing.Targets`

Step 11: Assign Multiple Targets

In this step, we will add a new button to the home page to illustrate the usage of multiple targets for a route. When the button is pressed, a new page opens that contains two parts: a header part at the top and a content part. The content part displays a table of employees that can be sorted and searched. We will use the array notation in the routing configuration to assign multiple targets to a route - a feature that we have not yet introduced.

Preview

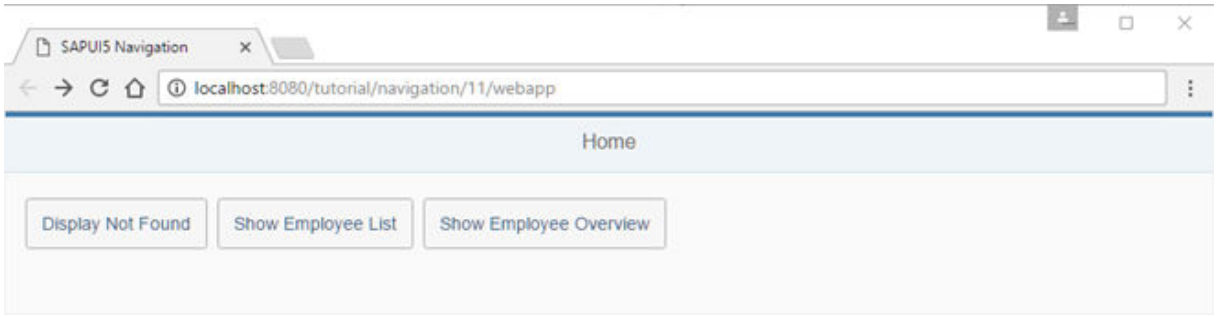


Figure 98: New button *Show Employee Overview*

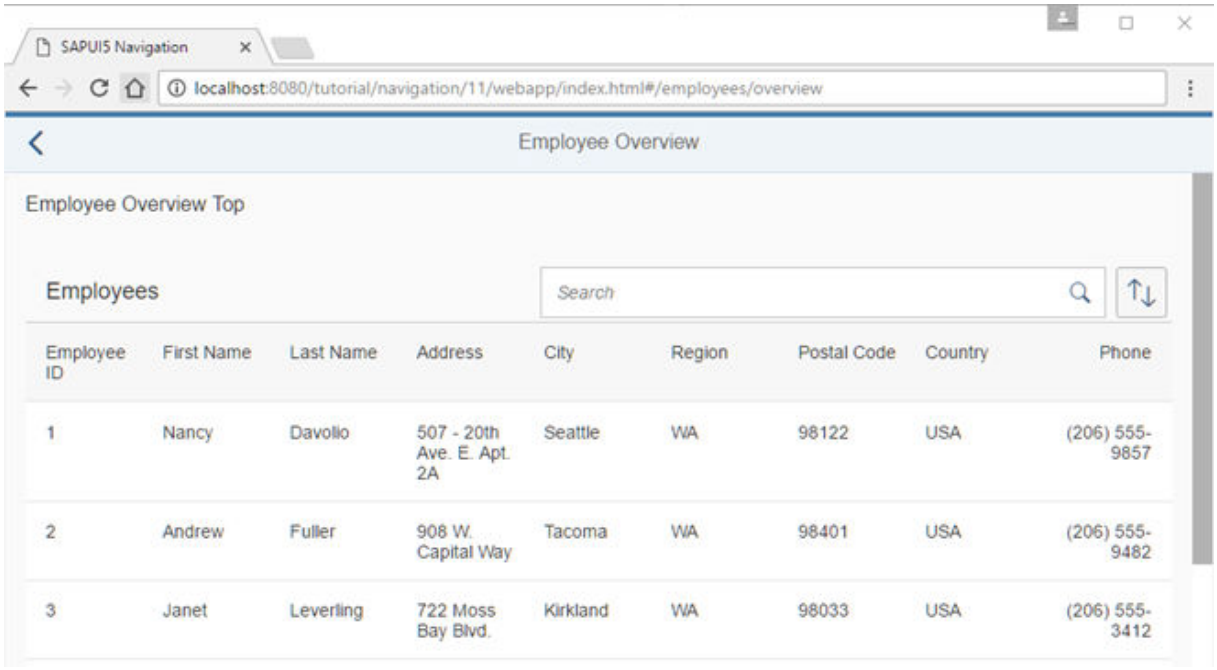


Figure 99: *Employee Overview* with search field

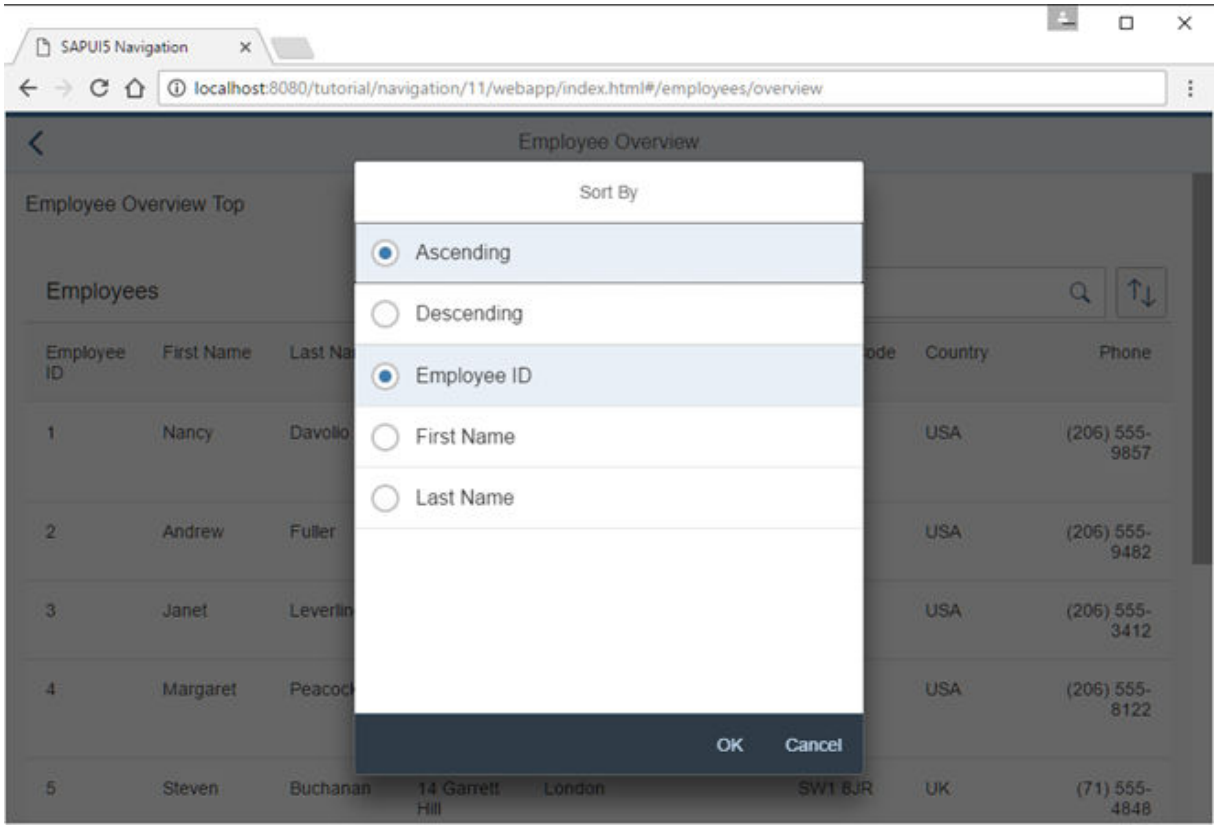


Figure 100: Sort options for the *Employee Overview*

Coding

You can view and download all files in the Demo Kit at Routing and Navigation - Step 11 .

Figure 101: Folder Structure for this Step

webapp/view/Home.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.nav.controller.Home"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}" class="sapUiResponsiveContentPadding">
        <content>
            <Button id="displayNotFoundBtn" text="{i18n>DisplayNotFound}"
press=".onDisplayNotFound" class="sapUiTinyMarginEnd"/>
            <Button id="employeeListBtn" text="{i18n>ShowEmployeeList}"
press=".onNavToEmployees" class="sapUiTinyMarginEnd"/>
            <Button id="employeeOverviewBtn" text="{i18n>ShowEmployeeOverview}"
press=".onNavToEmployeeOverview" class="sapUiTinyMarginEnd"/>
        </content>
    </Page>
</mvc:View>
```

First we add a new button to the `Home` view and add an event handler for the `press` event.

webapp/controller/Home.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.Home", {
        ...
        onNavToEmployees : function () {
            this.getRouter().navTo("employeeList");
        },
        onNavToEmployeeOverview : function ()  {
            this.getRouter().navTo("employeeOverview");
        }
    });
});
```

As you know already from the previous steps, we add the `press` event handler `onNavToEmployeeOverview`. It navigates to the route `employeeOverview` which does not exist yet, so let's create it.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
        ...
    },
    "sap.ui": {
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
                "target": "home"
            }, {
                "pattern": "employees",
                "name": "employeeList",
                "target": "employees"
            }, {
                "pattern": "employees/overview",
                "name": "employeeOverview",
                "target": ["employeeOverviewTop", "employeeOverviewContent"]
            }, {
                "pattern": "employees/{employeeId}",
                "name": "employee",
                "target": "employee"
```

```
            }, {
                "pattern": "employees/{employeeId}/resume:?query:",
                "name": "employeeResume",
                "target": "employeeResume"
            }],
            "targets": {
                ...
                "resumeTabNotes": {
                    "viewId": "resumeNotes",
                    "parent": "employeeResume",
                    "viewPath": "sap.ui.demo.nav.view.employee",
                    "viewName": "ResumeNotes",
                    "controlId": "notesTab",
                    "controlAggregation": "content"
                },
                "employeeOverview": {
                    "viewId": "employeeOverview",
                    "viewPath": "sap.ui.demo.nav.view.employee.overview",
                    "viewName": "EmployeeOverview",
                    "viewLevel" : 2
                },
                "employeeOverviewTop": {
                    "viewId": "employeeOverviewTop",
                    "parent": "employeeOverview",
                    "viewPath": "sap.ui.demo.nav.view.employee.overview",
                    "viewName": "EmployeeOverviewTop",
                    "controlId": "EmployeeOverviewParent",
                    "controlAggregation": "content"
                },
                "employeeOverviewContent": {
                    "viewId": "employeeOverviewContent",
                    "parent": "employeeOverview",
                    "viewPath": "sap.ui.demo.nav.view.employee.overview",
                    "viewName": "EmployeeOverviewContent",
                    "controlId": "EmployeeOverviewParent",
                    "controlAggregation": "content"
                }
            }
        }
    }
}
```

We extend our current routing configuration with a new route `employeeOverview`. Note that this route has to be configured before the `employee` route, else the `employee` route would be matched with a hash like `/#/employees/overview`. The new route `employeeOverview` references two targets at the same time with an array notation: `employeeOverviewTop` and `employeeOverviewContent`. As you can see here, a route can reference an arbitrary number of targets that will be displayed when the route is matched.

Both targets `employeeOverviewTop` and `employeeOverviewContent` reference the target `employeeOverview` as their parent target because we want to place them both inside the parent. Please also note that we also introduce a new layer `overview` in the `viewPath` property.

i Note

The order of the routing configuration matters here, because the router stops matching additional routes when the first match is found. You can override this behavior if you set parameter `greedy` to `true` on the route. Then the route will always be matched when the pattern matches the current URL, even if another route has been matched before. The `greedy` option comes from the underlying `Crossroads.js` library, a popular routing library. A common use case for using `greedy` is configuring targets without views and then listening for route-matched events.

Now we create both targets `employeeOverviewTop` and `employeeOverviewContent` as well as their parent target `employeeOverview`. On the parent target we set `viewLevel` to `2` to ensure a correct transition animation. In the targets, we also configure where the corresponding views of the children shall be displayed by setting the parameters `controlId` and `controlAggregation` to a control ID of a `sap.ui.layout.HorizontalLayout` that we are about to create in a new view. You should be familiar with this configuration from the last step.

The router makes sure that the parent view is loaded in addition to the target view when a corresponding route has been matched and the targets are displayed. The referenced views are displayed automatically at the configured place in the parent's view, in our case in the content aggregation of the page control. We have mentioned three different views that we still need to add to the app to make the configuration work:

- `EmployeeOverview`
- `EmployeeOverviewTop`
- `EmployeeOverviewContent`

webapp/view/employee/overview/EmployeeOverview.view.xml (New)

```
<mvc:View

controllerName="sap.ui.demo.nav.controller.employee.overview.EmployeeOverview"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page id="EmployeeOverviewParent" title="{i18n>EmployeeOverview}"
        showNavButton="true"
        navButtonPress=".onNavBack"
        class="sapUiResponsiveContentPadding">
        <content>
            <!-- inserted by routing -->
        </content>
    </Page>
</mvc:View>
```

First we create the parent view by creating the folder `overview` under `webapp/view/employee` and placing the file `EmployeeOverview.view.xml` into that folder. This view contains a `Page` control that is referenced from the targets in our `manifest.json` descriptor file. The content aggregation of the page will be filled by the router with the top and content part when the corresponding route has been hit.

webapp/controller/employee/overview/EmployeeOverview.controller.js (New)

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.overview.EmployeeOverv
iew", {
    });
});
```

The controller does not contain any logic yet, but we will add back navigation features here in the next steps.

webapp/view/employee/overview/EmployeeOverviewTop.view.xml (New)

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc"
class="sapUiMediumMarginBottom">
    <Title text="{i18n>EmployeeOverviewTop}"/>
</mvc:View>
```

Create the file `EmployeeOverviewTop.view.xml` and place it in the `webapp/view/employee/overview` folder. This view displays a static text for illustration purposes. Change it according to your own requirements. We don't need a controller for this view

webapp/view/employee/overview/EmployeeOverviewContent.view.xml (New)

```
<mvc:View

controllerName="sap.ui.demo.nav.controller.employee.overview.EmployeeOverviewCont
ent"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Table id="employeesTable"
        items="{/Employees}">
        <headerToolbar>
            <Toolbar>
                <Title text="{i18n>Employees}" level="H2"/>
                <ToolbarSpacer />
                <SearchField id="searchField" search=".onSearchEmployeesTable"
width="50%"/>
                <Button icon="sap-icon://sort" press=".onSortButtonPressed"/>
            </Toolbar>
        </headerToolbar>
        <columns>
            <Column id="employeeIDCol"><Text text="{i18n>EmployeeID}"/></Column>
            <Column id="firstNameCol" demandPopin="true"><Text
text="{i18n>FirstName}"/></Column>
            <Column id="lastNameCol" demandPopin="true"><Text
text="{i18n>LastName}"/></Column>
            <Column id="addressCol" minScreenWidth="Tablet"
demandPopin="true"><Text text="{i18n>Address}"/></Column>
            <Column id="cityCol" minScreenWidth="Tablet"
demandPopin="true"><Text text="{i18n>City}"/></Column>
            <Column id="regionCol" minScreenWidth="Tablet"
demandPopin="true"><Text text="{i18n>Region}"/></Column>
            <Column id="postalCodeCol" minScreenWidth="Tablet"
demandPopin="true"><Text text="{i18n>PostalCode}"/></Column>
            <Column id="countryCol" minScreenWidth="Tablet"
demandPopin="true"><Text text="{i18n>Country}"/></Column>
            <Column id="homePhoneCol" minScreenWidth="Tablet" demandPopin="true"
hAlign="Right"><Text text="{i18n>Phone}"/></Column>
        </columns>
        <items>
            <ColumnListItem>
                <cells>
                    <Text text="{EmployeeID}"/>
```

```
                    <Text text="{FirstName}"/>
                    <Text text="{LastName}"/>
                    <Text text="{Address}"/>
                    <Text text="{City}"/>
                    <Text text="{Region}"/>
                    <Text text="{PostalCode}"/>
                    <Text text="{Country}"/>
                    <Text text="{HomePhone}"/>
                </cells>
            </ColumnListItem>
        </items>
    </Table>
</mvc:View>
```

Create the file EmployeeOverviewContent.view.xml in the `webapp/view/employee/overview` folder. This view displays a responsive table with several columns containing employee data like *Employee ID*, *First Name*, *Last Name* and so on. In the `headerToolbar`, we add the `SearchField` and a `Button`. The `SearchField` in the header area allows to search in the table. The `Button` next to it opens a dialog to adjust the sorting of the table.

webapp/controller/employee/overview/ EmployeeOverviewContent.controller.js (New)

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/ui/model/Sorter",
    "sap/m/ViewSettingsDialog",
    "sap/m/ViewSettingsItem"
], function(
    BaseController,
    Filter,
    FilterOperator,
    Sorter,
    ViewSettingsDialog,
    ViewSettingsItem
) {
    "use strict";

    return
BaseController.extend("sap.ui.demo.nav.controller.employee.overview.EmployeeOverv
iewContent", {

        onInit: function () {
            this._oTable = this.byId("employeesTable");
            this._oVSD = null;
            this._sSortField = null;
            this._bSortDescending = false;
            this._aValidSortFields = ["EmployeeID", "FirstName", "LastName"];
            this._sSearchQuery = null;

            this._initViewSettingsDialog();
        },

        onSortButtonPressed : function () {
            this._oVSD.open();
        },

        onSearchEmployeesTable : function (oEvent) {
```

```
            this._applySearchFilter( oEvent.getSource().getValue() );
        },

        _initViewSettingsDialog : function () {
            this._oVSD = new ViewSettingsDialog("vsd", {
                confirm: function (oEvent) {
                    var oSortItem = oEvent.getParameter("sortItem");
                    this._applySorter(oSortItem.getKey(),
oEvent.getParameter("sortDescending"));
                }.bind(this)
            });

            // init sorting (with simple sorters as custom data for all fields)
            this._oVSD.addSortItem(new ViewSettingsItem({
                key: "EmployeeID",
                text: "Employee ID",
                selected: true            // by default the MockData is sorted
by EmployeeID
            }));

            this._oVSD.addSortItem(new ViewSettingsItem({
                key: "FirstName",
                text: "First Name",
                selected: false
            }));

            this._oVSD.addSortItem(new ViewSettingsItem({
                key: "LastName",
                text: "Last Name",
                selected: false
            }));
        },

        _applySearchFilter : function (sSearchQuery) {
            var aFilters, oFilter, oBinding;

            // first check if we already have this search value
            if (this._sSearchQuery === sSearchQuery) {
                return;
            }
            this._sSearchQuery = sSearchQuery;
            this.byId("searchField").setValue(sSearchQuery);

            // add filters for search
            aFilters = [];
            if (sSearchQuery && sSearchQuery.length > 0) {
                aFilters.push(new Filter("FirstName", FilterOperator.Contains,
sSearchQuery));
                aFilters.push(new Filter("LastName", FilterOperator.Contains,
sSearchQuery));
                oFilter = new Filter({ filters: aFilters, and: false });  // OR
filter
            } else {
                oFilter = null;
            }

            // update list binding
            oBinding = this._oTable.getBinding("items");
            oBinding.filter(oFilter, "Application");
        },

        /**
         * Applies sorting on our table control.
         * @param {string} sSortField        the name of the field used for
sorting
         * @param {string} sortDescending    true or false as a string or
boolean value to specify a descending sorting
         * @private
```

```
         */
        _applySorter : function (sSortField, sortDescending){
            var bSortDescending, oBinding, oSorter;

            // only continue if we have a valid sort field
            if (sSortField && this._aValidSortFields.indexOf(sSortField) > -1) {

                // convert  the sort order to a boolean value
                if (typeof sortDescending === "string") {
                    bSortDescending = sortDescending === "true";
                } else if (typeof sortDescending === "boolean") {
                    bSortDescending =  sortDescending;
                } else {
                    bSortDescending = false;
                }

                // sort only if the sorter has changed
                if (this._sSortField && this._sSortField === sSortField &&
this._bSortDescending === bSortDescending) {
                    return;
                }

                this._sSortField = sSortField;
                this._bSortDescending = bSortDescending;
                oSorter = new Sorter(sSortField, bSortDescending);

                // sync with View Settings Dialog
                this._syncViewSettingsDialogSorter(sSortField, bSortDescending);

                oBinding = this._oTable.getBinding("items");
                oBinding.sort(oSorter);
            }
        },

        _syncViewSettingsDialogSorter : function (sSortField, bSortDescending) {
            // the possible keys are: "EmployeeID" | "FirstName" | "LastName"
            // Note: no input validation is implemented here
            this._oVSD.setSelectedSortItem(sSortField);
            this._oVSD.setSortDescending(bSortDescending);
        }

    });

});
```

Finally create the controller for the *Employee Overview* page in the `webapp/controller/employee/overview` folder. It basically sets up a `ViewSettingsDialog` to sort and filter the table of employees and implements event handlers for the search field and for the sorting of the table.

There is nothing special about this implementation. If you are interested in how to set up a table with sorting and filtering you can check the corresponding steps of the *Walkthrough* tutorial or the examples in the Demo Kit. We will mainly make use of the UI and the functionality for showing additional navigation and routing features. Therefore, we suggest copying the code and trying it out.

Open `webapp/index.html#/employees/overview` and check the new views. As you can see, the three views are wired together automatically by the router based on our configuration in the descriptor. In the top area of the page, you see a static text and below you see the table filled with data from our test service. The whole routing functionality that we see in this example is implemented by referencing two targets from one route.

Of course, you can also search the table and change the sorting. When the sorting dialog opens, it creates a block layer so that the back button and other controls cannot be accessed. However, you can still use the back button of the browser. As you can see, the dialog is closed automatically by the router before navigating.

> **i Note**
>
> The default behavior of the `sap.m` router is that all dialogs are closed when the hash changes (i.e. when calling `navTo`, `display` or pressing the back button of the browser). You can change this default behavior by calling `getTargetHandler().setCloseDialogs(false)` on the router or on the `Targets` object.

However, we have one problem yet to solve: the search and table ordering are not bookmarkable. Fortunately, we have additional navigation features at hand and you will see how this works in the next steps

webapp/i18n/i18n.properties

```
...
EmployeeOverview=Employee Overview
ShowEmployeeOverview=Show Employee Overview

EmployeeOverviewTop=Employee Overview Top

Region=Region
EmployeeID=Employee ID
Phone=Phone
Employees=Employees
```

Add the new texts to the `properties` file.

Related Information

API Reference: `sap.m.routing.TargetHandler`

API Overview and Samples: `sap.ui.core.sample.PatternMatching`

Step 12: Make a Search Bookmarkable

In this step we will make the search bookmarkable. This allows users to search for employees in the *Employees* table and they can bookmark their search query or share the URL.

Preview

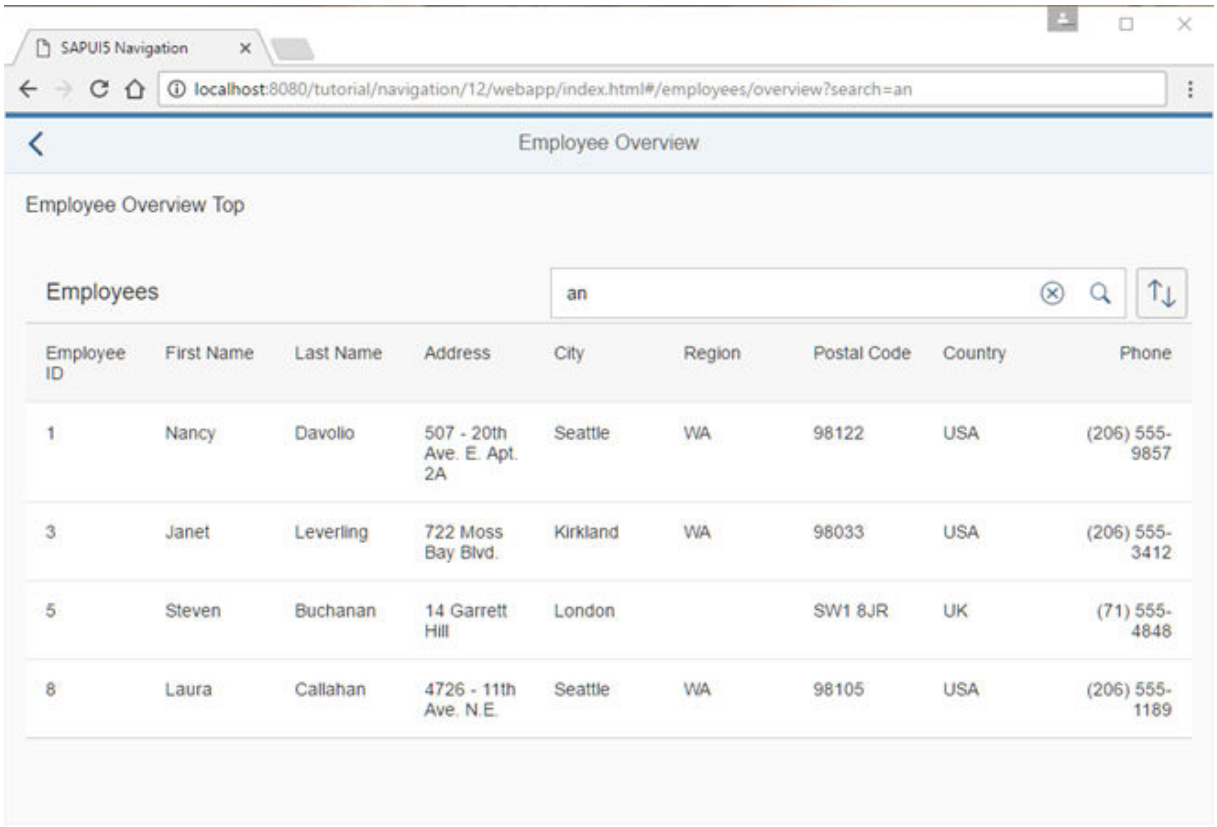


Figure 102: Search and sorting bookmarkable

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 12 .

webapp/manifest.json

```
{
    "_version": "1.12.0",
    "sap.app": {
        ...
    },
    "sap.ui": {
```

```
        ...
    },
    "sap.ui5": {
        ...
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.nav.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "transition": "slide",
                "bypassed": {
                    "target": "notFound"
                }
            },
            "routes": [{
                "pattern": "",
                "name": "appHome",
                "target": "home"
            }, {
                "pattern": "employees",
                "name": "employeeList",
                "target": "employees"
            }, {
                "pattern": "employees/overview:?query:",
                "name": "employeeOverview",
                "target": ["employeeOverviewTop", "employeeOverviewContent"]
            }, {
                "pattern": "employees/{employeeId}",
                "name": "employee",
                "target": "employee"
            }, {
                "pattern": "employees/{employeeId}/resume:?query:",
                "name": "employeeResume",
                "target": "employeeResume"
            }],
            "targets": {
                ...
            }
        }
    }
}
```

In order to make the search bookmarkable we have to think about how the pattern of the corresponding route should match the bookmark. We decide to allow `/#/employees/overview?search=mySearchQueryString` in order to bookmark a search. Therefore, we simply extend our routing configuration a little. We add the optional `:?query:` parameter to the route `employeeOverview`. We keep in mind that we want to use `search` as the URL parameter for the search term that was entered in the search field.

webapp/controller/employee/overview/EmployeeOverviewContent.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/ui/model/Sorter",
    "sap/m/ViewSettingsDialog",
```

```
    "sap/m/ViewSettingsItem"
], function(
    BaseController,
    Filter,
    FilterOperator,
    Sorter,
    ViewSettingsDialog,
    ViewSettingsItem
) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.overview.EmployeeOverv
iewContent", {
        onInit: function () {
            var oRouter = this.getRouter();
            this._oTable = this.byId("employeesTable");
            this._oVSD = null;
            this._sSortField = null;
            this._bSortDescending = false;
            this._aValidSortFields = ["EmployeeID", "FirstName", "LastName"];
            this._sSearchQuery = null;
            this._oRouterArgs = null;
            this._initViewSettingsDialog();
            // make the search bookmarkable

oRouter.getRoute("employeeOverview").attachMatched(this._onRouteMatched, this);
        },
        _onRouteMatched: function (oEvent) {
            // save the current query state
            this._oRouterArgs = oEvent.getParameter("arguments");
            this._oRouterArgs["?query"] = this._oRouterArgs["?query"] || {};

            // search/filter via URL hash
            this._applySearchFilter(this._oRouterArgs["?query"].search);
        },
        onSortButtonPressed : function (oEvent) {
            this._oVSD.open();
        },
        onSearchEmployeesTable : function (oEvent) {
            var oRouter = this.getRouter();
            // update the hash with the current search term
            this._oRouterArgs["?query"].search = oEvent.getSource().getValue();
            oRouter.navTo("employeeOverview", this._oRouterArgs, true /*no
history*/);
        },
        ...
    });
});
```

Now we handle the optional query parameter from the `employeeOverview` route in our `EmployeeOverviewContent` controller. First we change the `onInit` function by adding an event listener for the matched event of the `employeeOverview` route. Then we buffer the current router arguments as received from the event. If a query is available, the result from `oEvent.getParameter("arguments")` will contain a `?query` property with an object of all URL parameters specified, otherwise it is undefined. If no query parameter is defined, we always initialize the query and save it to `this._oRouterArgs["?query"]`. If we have a search term query at the `search` key we continue and call `this._applySearchFilter(this._oRouterArgs["?query"].search)` to trigger a search based on the search query parameter from the URL.

Storing the `arguments` objects internally in the controller is important, because we will use the current arguments when calling `navTo()` in the search event handler `onSearchEmployeesTable` and pass on the arguments with the updated search term. We keep the URL and the UI in sync by navigating to the current target again with the current value of the search field from the event's source. The search value is stored in

`this._oRouterArgs["?query"].search` together with the other query parameters and it is passed directly to the router again

That's it, now our search is bookmarkable and reflected in the URL. Try to access the following pages in your browser:

- `webapp/index.html#/employees/overview`
- `webapp/index.html#/employees/overview?search=`
- `webapp/index.html#/employees/overview?search=an`

When you change the value in the search field, you see that the hash updates accordingly.

Step 13: Make Table Sorting Bookmarkable

In this step, we will create a button at the top of the table which will change the sorting of the table. When the current sorting state of the table is changed, the sorting state will be reflected in the URL. This illustrates how to make the table sorting bookmarkable.

Preview

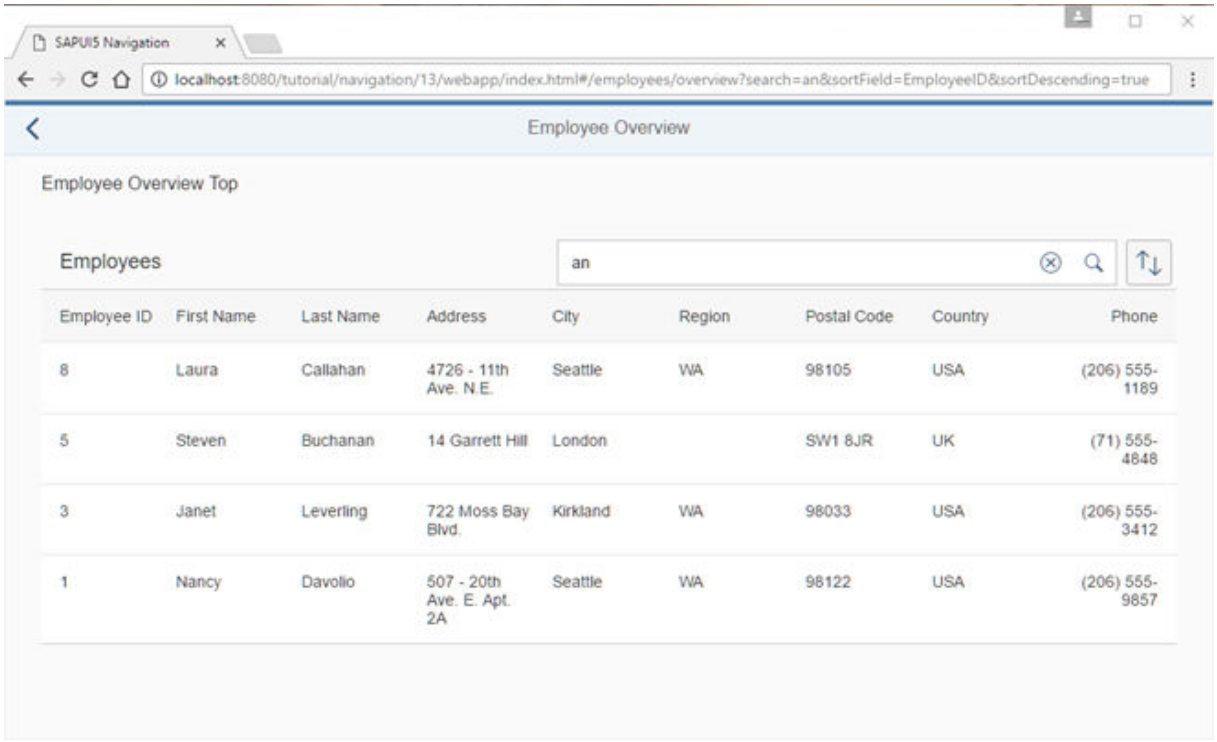


Figure 103: Bookmarkable search and sorting

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 13.

webapp/controller/employee/overview/EmployeeOverviewContent.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/ui/model/Sorter",
    "sap/m/ViewSettingsDialog",
    "sap/m/ViewSettingsItem"
], function(
    BaseController,
    Filter,
    FilterOperator,
    Sorter,
    ViewSettingsDialog,
    ViewSettingsItem
) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.overview.EmployeeOverv
iewContent", {
        onInit: function () {
            ...
        },
        _onRouteMatched: function (oEvent) {
            // save the current query state
            this._oRouterArgs = oEvent.getParameter("arguments");
            this._oRouterArgs["?query"] = this._oRouterArgs["?query"] || {};
            var oQueryParameter = this._oRouterArgs["?query"];
            // search/filter via URL hash
            this._applySearchFilter(oQueryParameter.search);
            // sorting via URL hash
            this._applySorter(oQueryParameter.sortField,
oQueryParameter.sortDescending);

        },
        ...
        _initViewSettingsDialog: function () {
            var oRouter = this.getRouter();
            this._oVSD = new sap.m.ViewSettingsDialog("vsd", {
                confirm: function (oEvent) {
                    var oSortItem = oEvent.getParameter("sortItem");
                    this._oRouterArgs["?query"].sortField = oSortItem.getKey();
                    this._oRouterArgs["?query"].sortDescending =
oEvent.getParameter("sortDescending");
                    oRouter.navTo("employeeOverview", this._oRouterArgs, true /
*without history*/);
                }.bind(this)
            });
            ...
        },
        ...
    });
});
```

We enhance the `EmployeeOverviewContent` controller further to add support for bookmarking the table's sorting options. We expect two query parameters `sortField` and `sortDescending` from the URL for configuring the sorting of the table. In the matched handler of the route `employeeOverview`, we store the query parameter in the `oQueryParameter` variable and add an additional call to `this._applySorter(oQueryParameter.sortField, oQueryParameter.sortDescending)`. This

triggers the sorting action based on the two query parameters `sortField` and `sortDescending` from the URL.

Next we change the `confirm` event handlers of our `ViewSettingsDialog`. The `confirm` handler updates the current router arguments with the parameters from the event accordingly. Then we call `oRouter.navTo("employeeOverview", this._oRouterArgs, true)` with the updated router arguments to persist the new sorting parameters in the URL. Both the previous arguments (i.e. `search`) and the new arguments for the sorting will then be handled by the matched event handler for the `employeeOverview` route.

Congratulations! Even the sorting options of the table can now be bookmarked. Try to access the following pages:

- `webapp/index.html#/employees/overview?sortField=EmployeeID&sortDescending=true`
- `webapp/index.html#/employees/overview?search=an&sortField=EmployeeID&sortDescending=true`

When changing the table's sorting options, you will see that the hash updates accordingly.

Step 14: Make Dialogs Bookmarkable

In this step, we want to allow bookmarking of the dialog box that is opened when the user clicks the *Sort* button. The dialog should automatically open when the URL contains the query parameter `showDialog`.

Preview

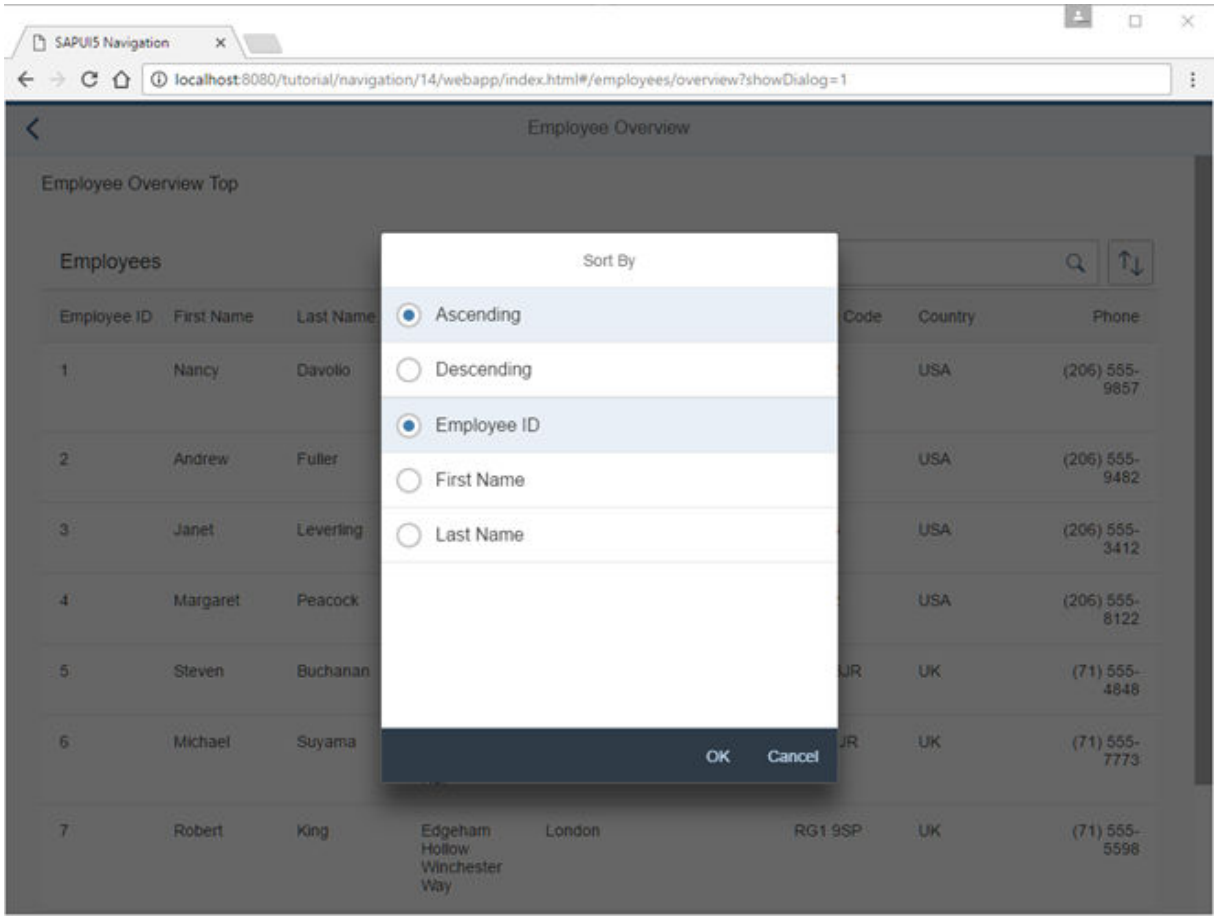


Figure 104: Bookmark for a dialog

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 14.

/controller/employee/overview/EmployeeOverviewContent.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
```

```
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/ui/model/Sorter",
    "sap/m/ViewSettingsDialog",
    "sap/m/ViewSettingsItem"
], function(
    BaseController,
    Filter,
    FilterOperator,
    Sorter,
    ViewSettingsDialog,
    ViewSettingsItem
) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.overview.EmployeeOverv
iewContent", {
        onInit: function () {
            ...
        },
        _onRouteMatched: function (oEvent) {
            // save the current query state
            this._oRouterArgs = oEvent.getParameter("arguments");
            this._oRouterArgs["?query"] = this._oRouterArgs["?query"] || {};
            var oQueryParameter = this._oRouterArgs["?query"];
            // search/filter via URL hash
            this._applySearchFilter(oQueryParameter.search);
            // sorting via URL hash
            this._applySorter(oQueryParameter.sortField,
oQueryParameter.sortDescending);
            // show dialog via URL hash
            if (oQueryParameter.showDialog) {
                this._oVSD.open();
            }

        },
        onSortButtonPressed: function (oEvent) {
            var oRouter = this.getRouter();
            this._oRouterArgs["?query"].showDialog = 1;
            oRouter.navTo("employeeOverview", this._oRouterArgs);
        },
        ...
        _initViewSettingsDialog: function () {
            var oRouter = this.getRouter();
            this._oVSD = new sap.m.ViewSettingsDialog("vsd", {
                confirm: function (oEvent) {
                    var oSortItem = oEvent.getParameter("sortItem");
                    this._oRouterArgs["?query"].sortField = oSortItem.getKey();
                    this._oRouterArgs["?query"].sortDescending =
oEvent.getParameter("sortDescending");
                    delete this._oRouterArgs["?query"].showDialog;
                    oRouter.navTo("employeeOverview", this._oRouterArgs, true /
*without history*/);
                }.bind(this),
                cancel: function (oEvent){
                    delete this._oRouterArgs.query.showDialog;
                    oRouter.navTo("employeeOverview", this._oRouterArgs, true /
*without history*/);
                }.bind(this)
            });
            ...
        },
        ...
    });
});
```

Once again we will update the `EmployeeOverviewContent` controller to add support for the bookmarking of our sorting dialog. We decide to choose a query parameter `showDialog` that controls if the dialog is opened directly when we navigate to the page with a deep link. Therefore, we extend the matched event handler for the `employeeOverview` route. If the query parameter `showDialog` is set to `1`, we open the dialog. We only have to make sure that the dialog does not get closed again by the router as this behavior is the default when navigating.

Next we change the `press` handler of the sort button. In the `onSortButtonPressed` function we set `this._oRouterArgs["?query"].showDialog = 1` and call `navTo()` to let the router do the job instead of directly opening the dialog. Finally, we delete `this._oRouterArgs["?query"].showDialog` before calling `navTo()` in the `confirm` and `cancel` event handlers of the `ViewSettingsDialog`. This is important to make sure that the dialog does not open again by the matched handler.

We are now done with this step. Try to access the following pages:

- `webapp/index.html#/employees/overview?showDialog=1`
- `webapp/index.html#/employees/overview?search=an&sortField=EmployeeID&sortDescending=true&showDialog=1`

As you can see, the dialog opens automatically if the parameter `showDialog=1` is added to the URL. That's exactly what we wanted.

Step 15: Reuse an Existing Route

The *Employees* table displays employee data. However, the resumes of the employees are not accessible from this view yet. We could create a new route and a new view to visualize the resume again, but we could also simply reuse an existing route to cross-link the resume of a certain employee. In this step, we will add a feature that allows users to directly navigate to the resume of a certain employee. We will reuse the *Resume* page that we have created in an earlier step. This example illustrates that there can be multiple navigation paths that direct to the same page.

Preview

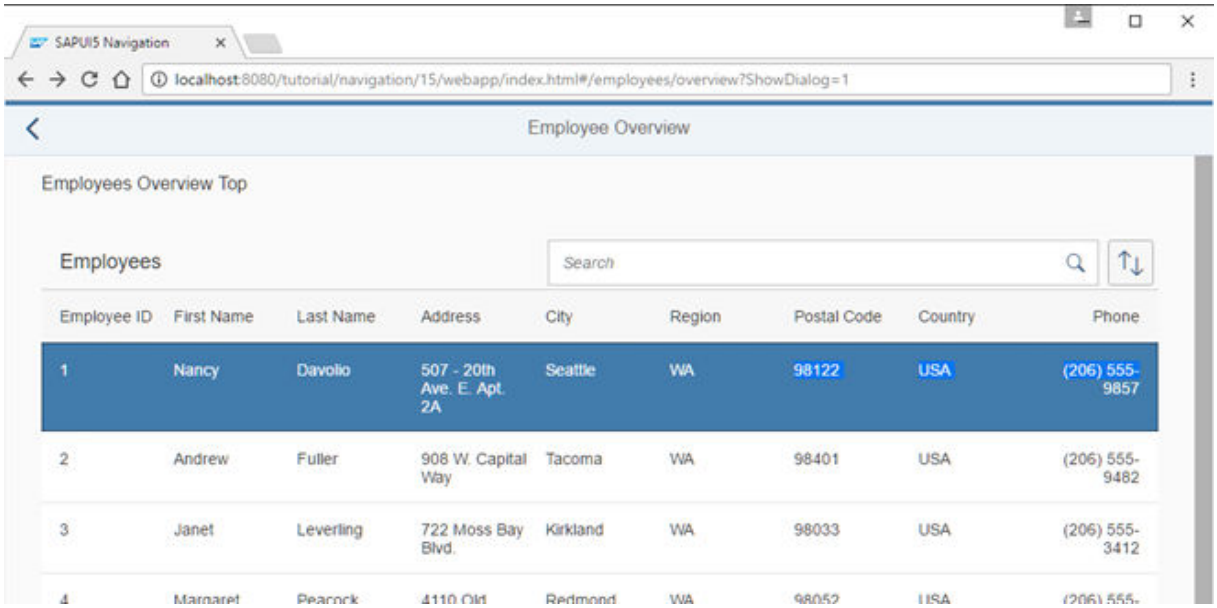


Figure 105: Navigation to an existing route from a table item

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 15.

webapp/view/employee/overview/EmployeeOverviewContent.view.xml

```
<mvc:View

controllerName="sap.ui.demo.nav.controller.employee.overview.EmployeeOverviewCont
ent"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Table id="employeesTable"
        items="{/Employees}"
            itemPress=".onItemPressed">
        <headerToolbar>
            ...
        </headerToolbar>
        <columns>
            ...
        </columns>
        <items>
            <ColumnListItem type="Active">
                <cells>
                    ...
                </cells>
            </ColumnListItem>
        </items>
    </Table>
</mvc:View>
```

In the `EmployeeOverviewContent` view we register an event handler for the `itemPress` event and set the type attribute of the `ColumnListItem` to `Active` so that we can choose an item and trigger the navigation.

webapp/controller/employee/overview/EmployeeOverviewContent.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/ui/model/Sorter",
    "sap/m/ViewSettingsDialog",
    "sap/m/ViewSettingsItem"
], function(
    BaseController,
    Filter,
    FilterOperator,
    Sorter,
    ViewSettingsDialog,
    ViewSettingsItem
) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.nav.controller.employee.overview.EmployeeOverv
iewContent", {
        ...
        _syncViewSettingsDialogSorter: function (sSortField, bSortDescending) {
            // the possible keys are: "EmployeeID" | "FirstName" | "LastName"
            // Note: no input validation is implemented here
            this._oVSD.setSelectedSortItem(sSortField);
            this._oVSD.setSortDescending(bSortDescending);
        },
        onItemPressed: function (oEvent) {
            var oItem, oCtx, oRouter;
            oItem = oEvent.getParameter("listItem");
            oCtx = oItem.getBindingContext();
            this.getRouter().navTo("employeeResume",{
                employeeId : oCtx.getProperty("EmployeeID"),
                "?query": {
                    tab: "Info"
                }
            });
        }
    });
});
```

Next we add the `itemPress` handler `.onItemPressed` to the `EmployeeOverviewContent` controller. It reads from the binding context which item has been chosen and navigates to the `employeeResume` route. We have already added this route and the corresponding target in a previous step and can now reuse it. From now on it is possible to navigate to the `employeeResume` route from our employee table as well as from the employee detail page created in an earlier step (the route name is `employee`).

Step 16: Handle Invalid Hashes by Listening to Bypassed Events

So far we have created many useful routes in our app. In the very early steps we have also made sure that a *Not Found* page is displayed in case the app was called with an invalid hash. Now, we proceed further and track invalid hashes to be able to detect and correct any invalid links or add new URL patterns that are often requested but not found. Therefore, we simply listen to the bypassed events

Preview

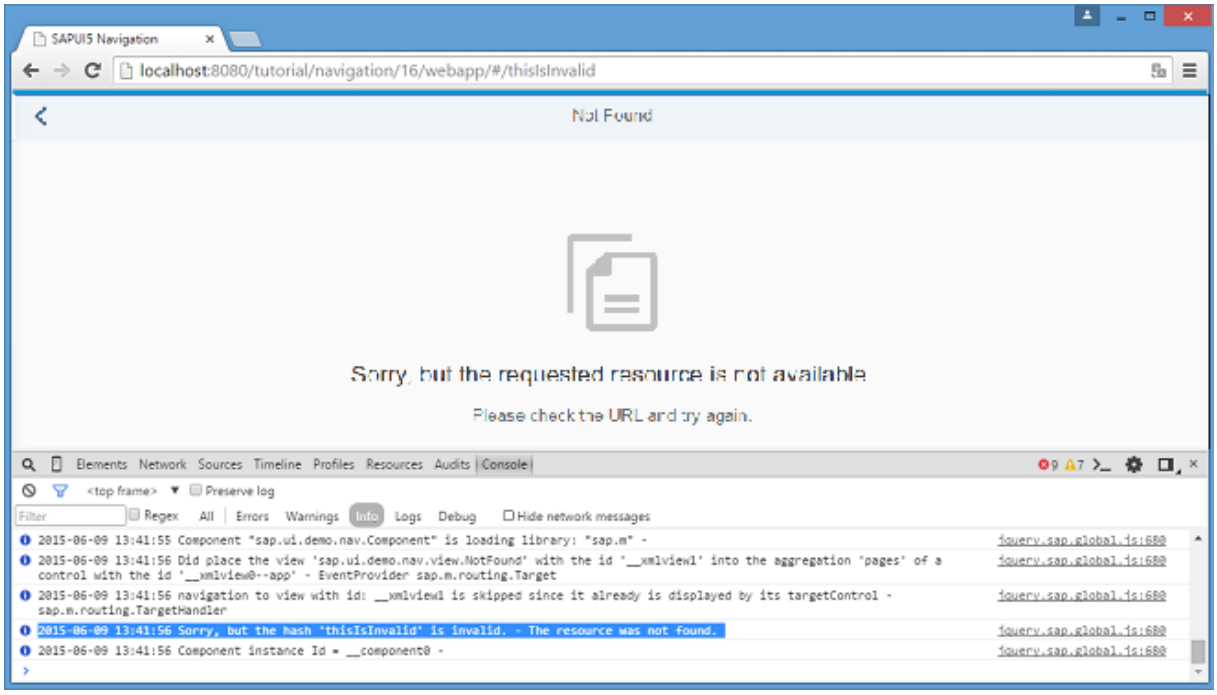


Figure 106: Console output for invalid hashes when listening to bypassed events

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 16.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController"
], function (BaseController) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.App", {
        onInit: function () {
```

```
            // This is ONLY for being used within the tutorial.
            // The default log level of the current running environment may be
higher than INFO,
            // in order to see the debug info in the console, the log level
needs to be explicitly
            // set to INFO here.
            // But for application development, the log level doesn't need to be
set again in the code.
            Log.setLevel(Log.Level.INFO);

            var oRouter = this.getRouter();

            oRouter.attachBypassed(function (oEvent) {
                var sHash = oEvent.getParameter("hash");
                // do something here, i.e. send logging data to the backend for
analysis
                // telling what resource the user tried to access...
                Log.info("Sorry, but the hash '" + sHash + "' is invalid.", "The
resource was not found.");
            });
        }
    });
});
```

All we need to do is listen to the bypassed event on the router. If the bypassed event is triggered, we simply get the current hash and log a message. In an actual app this is probably the right place to add some application analysis features, i.e. sending analytical logs to the back end for later evaluation and processing. This could be used to improve the app, for example, to find out why the user called the app with an invalid hash.

> **i Note**
>
> We have chosen to place this piece of code into the `App` controller because this is a global feature of the app. However, you could also place it anywhere else, for example in the `NotFound` controller file or in a helper module related to analysis.

Now try to access `webapp/index.html#/thisIsInvalid` while you have your browser console open. As you can see, there is a message that issues a faulty hash. Furthermore, our `NotFound` page is displayed.

Related Information

API Reference: `sap.m.routing.Router`

Step 17: Listen to Matched Events of Any Route

In the previous step, we have listened for bypassed events to detect possible technical issues with our app. In this step, we want to improve the analysis use case even more by listening to any matched event of the route. We could use this information to measure how the app is used and how frequently the pages are called. Many Web analytic tools track page hits this way. The collected information can be used, for example to improve our app and its usability.

Preview

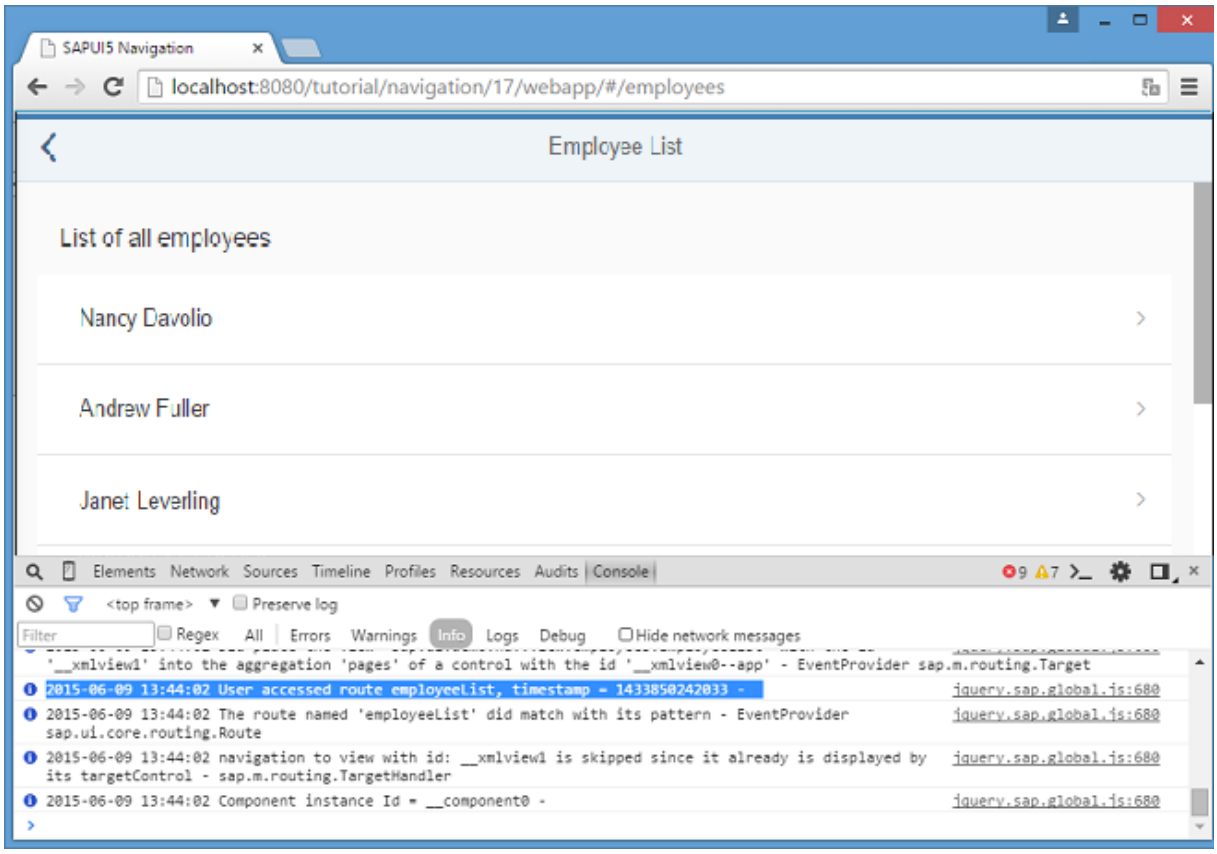


Figure 107: Console output for routes matched by listening to `routeMatched` events

Coding

You can view and download all files in the *Samples* in the Demo Kit at Routing and Navigation - Step 17.

webapp/controller/App.controller.js

```
sap.ui.define([
    "sap/ui/demo/nav/controller/BaseController",
    "sap/base/Log"
], function (BaseController, Log) {
    "use strict";
    return BaseController.extend("sap.ui.demo.nav.controller.App", {
        onInit: function () {
            var oRouter = this.getRouter();
            oRouter.attachBypassed(function (oEvent) {
                var sHash = oEvent.getParameter("hash");
                // do something here, i.e. send logging data to the back end for
analysis
                // telling what resource the user tried to access...
                Log.info("Sorry, but the hash '" + sHash + "' is invalid.", "The
resource was not found.");
```

```
            });
            oRouter.attachRouteMatched(function (oEvent){
                var sRouteName = oEvent.getParameter("name");
                // do something, i.e. send usage statistics to back end
                // in order to improve our app and the user experience (Build-
Measure-Learn cycle)
                Log.info("User accessed route " + sRouteName + ", timestamp = "
+ new Date().getTime());
            });
        }
    });
});
```

We extend the `App` controller again and listen to the `routeMatched` event. The `routeMatched` event is thrown for any route that matches to our route configuration in the descriptor file. In the event handler, we determine the name of the matched route from the event parameters and log it together with a time stamp. In an actual app, the information could be sent to a back-end system or an analytics server to find out more about the usage of your app.

Now you can access, for example, `webapp/index.html#/employees` while you have the console of the browser open. As you can see, there is a message logged for each navigation step that you do within the app.

Testing

In this tutorial we will test application functionality with the testing tools that are delivered with SAPUI5. At different steps of this tutorial you will write tests using QUnit, OPA5, and the mock server. Additionally, you will learn about testing strategies, Test Driven Development (TDD), and much more.

For the application features that we add, we focus on writing clean and testable code with the goal of having good test coverage and a high quality app. We will create a simple full screen app that we will extend with more tests and features throughout the tutorial.

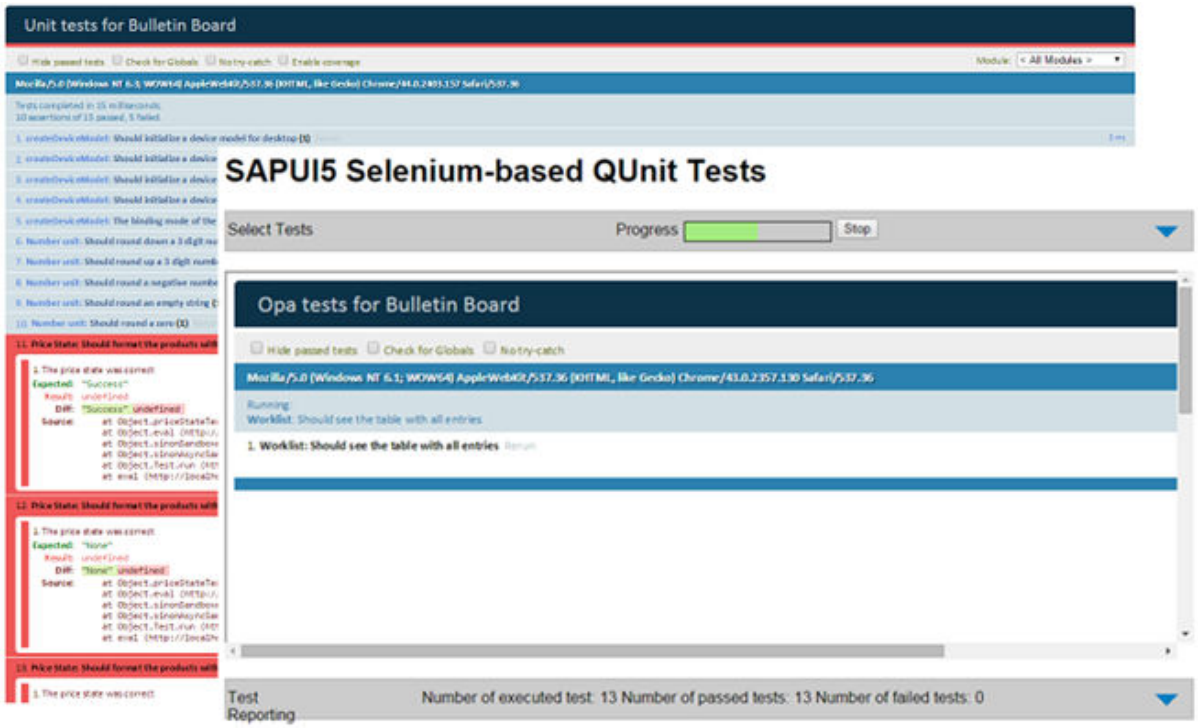
Imagine the following situation: You and your development team take over a bulletin board prototype that will be shipped as a product soon. A bulletin board typically consists of functionality to browse posts and add own offers to the board. However, the prototype only covers a minimum set of features and tests so far.

With this very minimalistic app as a starting point, we have a good foundation and we can inspect the most important testing functionality. Furthermore, we want to implement new features for the app that were requested by the product team using Test Driven Development and best practices for writing testable code and testing SAPUI5 apps.

So why do we do all this? Obviously, writing tests and testable code does not come without effort. Well, we want to ensure the implementation of a high quality app by having decent test coverage of our application logic. And we check that our code does not break by running the automated tests whenever we change something or when we upgrade to a newer version of the SAPUI5 framework or other external libraries. Additionally, we can find bugs proactively and do not need excessive manual testing anymore so the efforts definitely pay off. Also, when we decide to refactor something in the future, we can easily verify that the features of the app are still working as expected.

There are a lot more reasons and many small details that we will address throughout this tutorial. You can work yourself through the steps by applying the code deltas individually or by downloading the samples for each step and playing around with it.

Preview



Prerequisites

In addition to the prerequisites that are presupposed for all our tutorials (see Prerequisites [page 38]), you should also be familiar with the basics of JavaScript unit testing with QUnit. Have a look at the official QUnit documentation to make yourself familiar with basic testing knowledge. Steps 27 to 29 of the Walkthrough tutorial also cover the test setup in an app that is used throughout this tutorial.

If you want to automate the test execution using a test runner, you can set this up as described under Test Automation [page 1229].

> → Tip
>
> You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, and start there.
>
> You can view and download the files for all steps in the Demo Kit at Testing Apps. Copy the code to your workspace and make sure that the application runs by calling the `webapp/test/test.html` file. Depending on your development environment you might have to adjust resource paths and configuration entries.
>
> For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):
>
> - Downloading Code for a Tutorial Step [page 40]

- Adapting Code to Your Development Environment [page 40]

Related Information

Testing [page 1158]
QUnit Home Page

Step 1: Overview and Testing Strategy

In this step, we will take a look at the prototype and define the test strategy for our app. The prototype already contains the infrastructure for unit and integration testing and a minimum set of tests and features.

> **i Note**
>
> In this tutorial we will focus on writing clean unit and integration tests for apps. They build the foundation and are crucial for good application quality. We will also outline how to write testable code. Not all implementation patterns can be tested easily, but when writing the test code together with the implementation code as we have in this tutorial, testable code is a natural result.

Preview

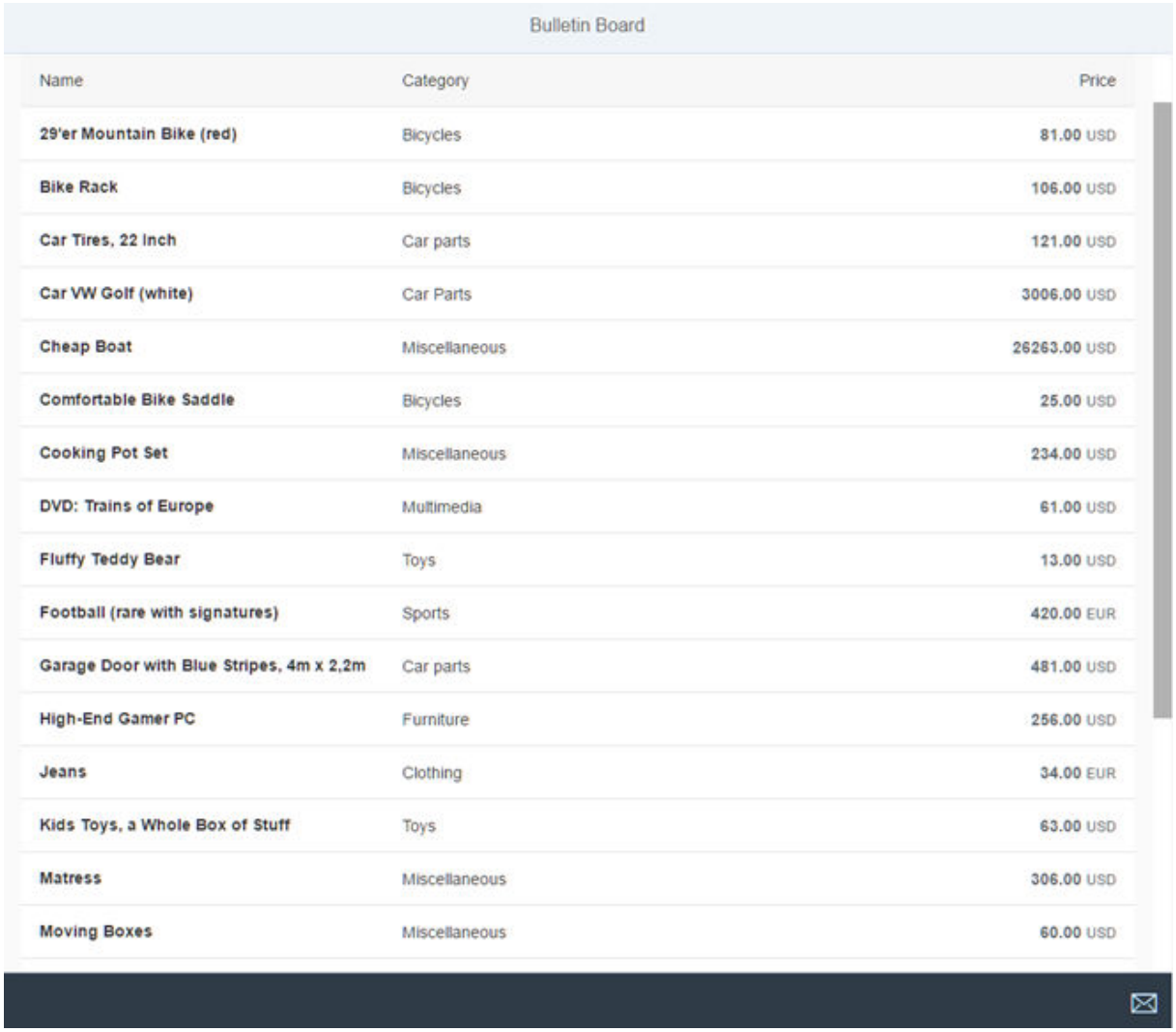


Figure 108: The prototype app

Coding

To set up your project for this tutorial, download the files for *Step 1* from the *Samples* in the Demo Kit at Testing - Step 1. Copy the code to your workspace and make sure that the application runs by calling the `webapp/test/mockServer.html` file.

Depending on your development environment, you might have to adjust resource paths and configuration entries. The project structure and the files provided with this tutorial are explained in detail in the Walkthrough [page 69] tutorial.

After downloading *Step 1*, you should have the following files:

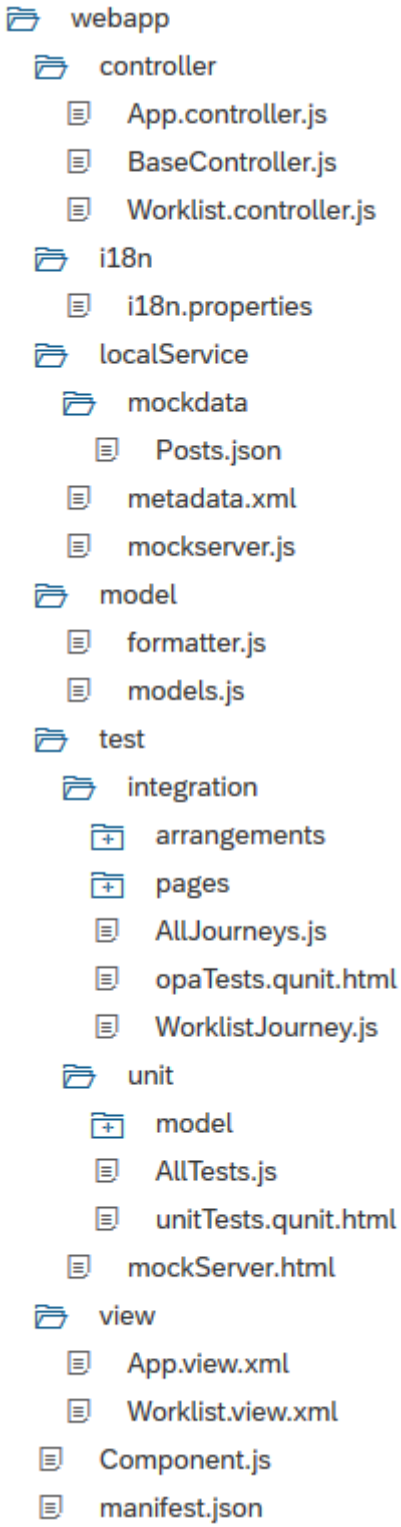


Figure 109: Folder structure with downloaded files

The Initial App

With the downloaded code, you now have the bulletin board prototype, set up according to the SAPUI5 best practices. The prototype provides the common features of an SAPUI5 app. If you have completed the Walkthrough tutorial, you should be familiar with most of the source code in this step. Additional features of the app are:

- **Entry Page**
  In this tutorial, we will often switch between testing application features manually, and running automated tests. The `webapp/test/test.html` file provides a list of entry points for the app so that you do not have to enter the URLs manually. From this page you can open the app with mock data, run the unit tests, run the integration tests, or run the app's test suite (this will be added later in the tutorial). Note that in a productive scenario we would have an additional entry point that calls the app with a real service. At this stage we are working with mock data and don't have a real service for our prototype yet, so we have left this step out.
- **Home Page**
  The home page of our bulletin board app is the `webapp/test/mockServer.html` file. On this page, we initialize SAPUI5, start the mock server, and instantiate our app component. It consists of a single view that displays a list of posts from a bulletin board with several attributes in a table.

  > **i** Note
  >
  > We do not yet have a real service for the bulletin board prototype so run the app with mock data and this test page throughout the tutorial. The mock server helps by mimicking a real service and it processes requests with a small delay, just as a real service would. This is perfect for realistic application testing and is also helpful for local development tests. It is a good practice to put all test pages in the test folder of the app, so that they are clearly separated from the productive coding.

- **Data**
  In the `webapp/localService/` folder, you can find the metadata and the mock data for the app. The `metadata.xml` file is used by the mock server to simulate real back-end service calls in the app. It describes our OData service and you can replace it later with a real service. The service we use has a single OData entity:
  - **Post**
    A post consists of typical properties like *Title*, *Description*, and *Price*. Each post is assigned to a *Category* and a *Contact*. The entity can be identified with its ID property: `PostID`. The corresponding `EntitySet` is `Posts`.
  - **Category**
    In our example, the category only has a `Name` property. Posts are sorted into a category by the category name. The corresponding `EntitySet` is `Categories`.
  - **Comment**
    A comment has an `Author`, a `Date`, and a `CommentText` property. The entity can be identified by the `CommentID` property and is linked to a `post` by the `ParentID`. The corresponding `EntitySet` is `Comments`.

  The actual test data containing several mock posts is located in the `webapp/test/service/posts.json` file.
- **Testing Functionality**
  The team that created the first prototype already took care of the basic test setup. Everything required for application testing is shipped with SAPUI5 and can simply be used within the app. The testing infrastructure is set up in the `test` folder that is located in the `webapp` folder of the app:

- **Mock Server**
  The mock server is set up in the `webapp/localService/mockserver.js` file. It loads the metadata and the mock data in the same folder. Using the mock server allows us to run the app easily and show realistic data for testing, even without a network connection and without the need of having a remote server for our application data.
  There is a configurable delay for each request that is processed by the mock server that allows you to mimick a slow back-end server.
- **Unit Tests**
  All unit tests are located in the `webapp/test/unit` folder and can be started by calling the `unitTests.qunit.html` file in the same folder. Initially, there are only a few tests for model instantiation and formatters that cover basic functionality in the prototype. We will give you more details about the unit test setup later in the tutorial.
- **Integration Tests**
  Integration tests are written in OPA5 – a tool for integration testing that is included in SAPUI5 – and can be found in the `webapp/test/integration` folder. You can start all OPA5 tests by calling the `opaTests.qunit.html` file in the same folder. OPA5 tests are organized in test journeys, and we have included a worklist journey that checks if the table of posts is displayed properly. We will give you more details about the integration test setup later in the tutorial.

- **Other quality-related features of the app**
  The app is set up according to best practices and already contains many helpful features.
  - **Separation of concerns (MVC)**
    All artifacts are located in either the `model`, `view`, or `controller` folder of the app. The app's component and its descriptor configure which of those MVC artifacts to load. This configuration controls the navigation flow of the app.
  - **Separation of productive and nonproductive code**
    All nonproductive code is located in the `test` subfolder. This includes the unit and integration tests, and the test page to call the app with mock data. All productive code is located in the `webapp` folder. This clearly separates the test artifacts from the application coding and makes it easy to remove all test-related artifacts before deploying the app for productive use.
  - **Busy handling**
    As a best practice, you should always give users instant feedback when triggering actions and navigating in the app. The app already includes functionality to display a busy indication when data is loaded or actions are triggered. To simulate a slow backend and show the behavior of the app, the mock server is configured with a delay of one second for each request.

Now that we have a running prototype, we can further extend it with additional tests and features. Make sure that the app is running by calling the test page, the unit tests, and the integration tests from the entry page `webapp/test/test.html`. The app should display a list of bulletin board posts as seen in the screenshot above and the tests should run without errors.

Test Strategy

Let's first take a look at best practices for testing apps written in SAPUI5. JavaScript is a dynamic programming language and only some issues can be detected by static code check tools and manual testing. Automated tests that execute the code regularly are beneficial for good quality and development productivity – especially when you're developing in short development cycles.

We expect our prototype to be released and shipped as a product soon, so we need a solid testing strategy. Fortunately the prototype team has already thought ahead and prepared an infrastructure for unit and

integration testing that is included in the app. This is a really good starting point for further enhancements of the app.

The mock server is also set up and allows us to test the app with local test data instead of a real back-end service. We can use the mock data for writing reliable integration tests that do not depend on another system which might be unavailable when the tests are run.

> **i Note**
>
> If you start developing an app from scratch, you should always consider testing from the very beginning of the software life cycle. Nobody wants to write tests for undocumented code and make assumptions about the logic. It is worth the effort to think about code checks, unit and integration testing, and a solid testing strategy from the very start.

Before you start implementing your first test, you should think about how to test the different aspects of your application. The image below shows the testing tools along the agile testing pyramid.

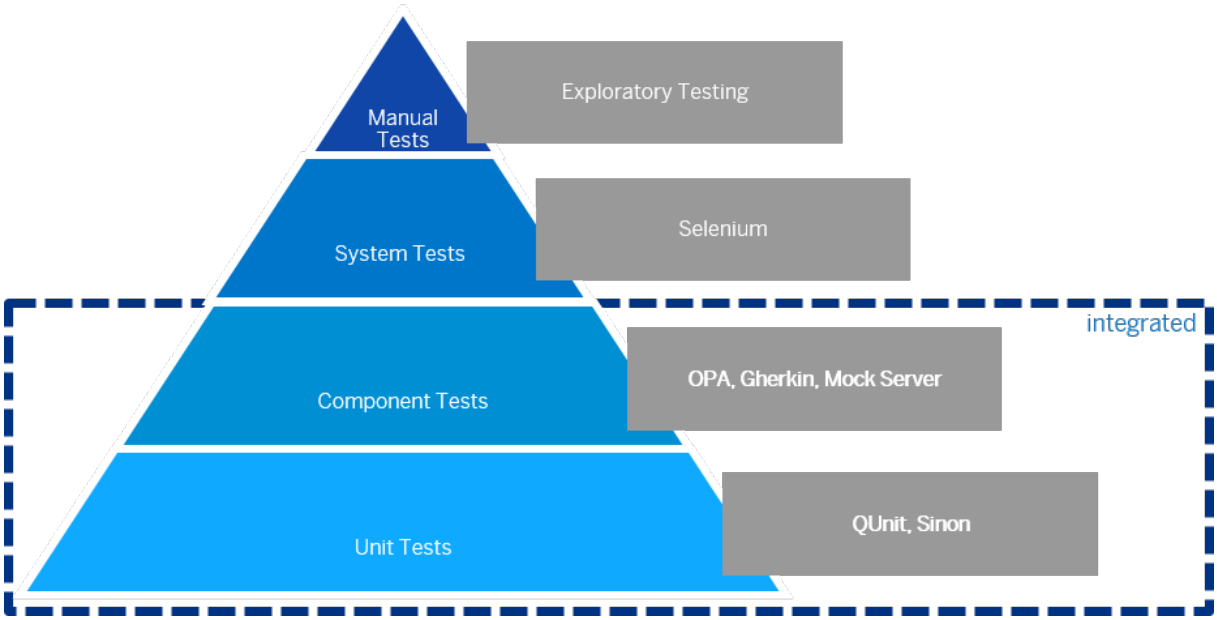


Figure 110: Testing pyramid

When you set up application testing, you should automate as many testing steps as possible. If you immediately write a test for all the features that we implement, then you can greatly reduce manual testing efforts that are time consuming and cumbersome. If you change something later, you can simply run the existing tests and see if the functionality is still working as expected.

SAPUI5 comes with two testing tools: QUnit for unit testing and OPA5 for integration testing. The unit tests are the foundation of our testing pyramid and they should validate the most important logic of our app. In addition, you can write integration tests for more interaction-related functionality, such as interacting with UI elements of the app.

There might still be features that are hard to test with these client-side testing frameworks. Certain features might require a more sophisticated system test, such as a screenshot comparison that can be implemented with additional testing frameworks. And of course, you should also schedule manual tests (for example, browser, performance, or security tests) to make sure that the app is behaving as expected.

Conventions

- Write unit tests in QUnit for more logic-related functionality
- Write integration tests in OPA5 for user interaction
- Separate productive and nonproductive code in the app (`webapp`, `test` folder)
- Provide a local test page that triggers the app in test mode with mock data (`test/mockServer.html`)

Related Information

App Templates: Kick Start Your App Development [page 1399]
Worklist Template [page 1400]
Testing [page 1158]
Unit Testing with QUnit [page 1159]
Integration Testing with One Page Acceptance Tests (OPA5) [page 1182]
Mock Server [page 1222]
Walkthrough [page 69]

Step 2: A First Unit Test

In this step we will analyze the unit testing infrastructure and write a first unit test.

The product team requested a feature to highlight the price with colors depending on the amount. This can be done using the standard semantic colors that are defined for states like *Success*, *Warning*, or *Error*.

The price values can be mapped to semantic states as follows:

- price < 50: Status is green (*Success*)
- price >= 50 and price < 250: Status is normal (*None*)
- price >= 250 and price < 2000: Status is orange (*Warning*)
- price >= 2000: Status is red (*Error*)

As we use Test Driven Development (TDD) we define the test case first, before we actually implement the feature. So we will now start by implementing a test for the *Price State* feature. Naturally the test will fail until the feature is implemented in the next step.

> **i Note**
>
> Test Driven Development (TDD) is a software development model that relies on a very short development cycle. When using TDD a developer first writes a failing automatic test case to describe the behavior of a new feature or functionality. As soon as the test fails (due to the still missing implementation) the role of the developer switches to the implementation. The code is added to make the test run successful and then the cycle starts over again.
>
> There might also be iterations where just the implementation or testing code is refactored to make it more elegant. TDD reduces complexity while maintaining high test coverage of the application coding at the same time.

Preview

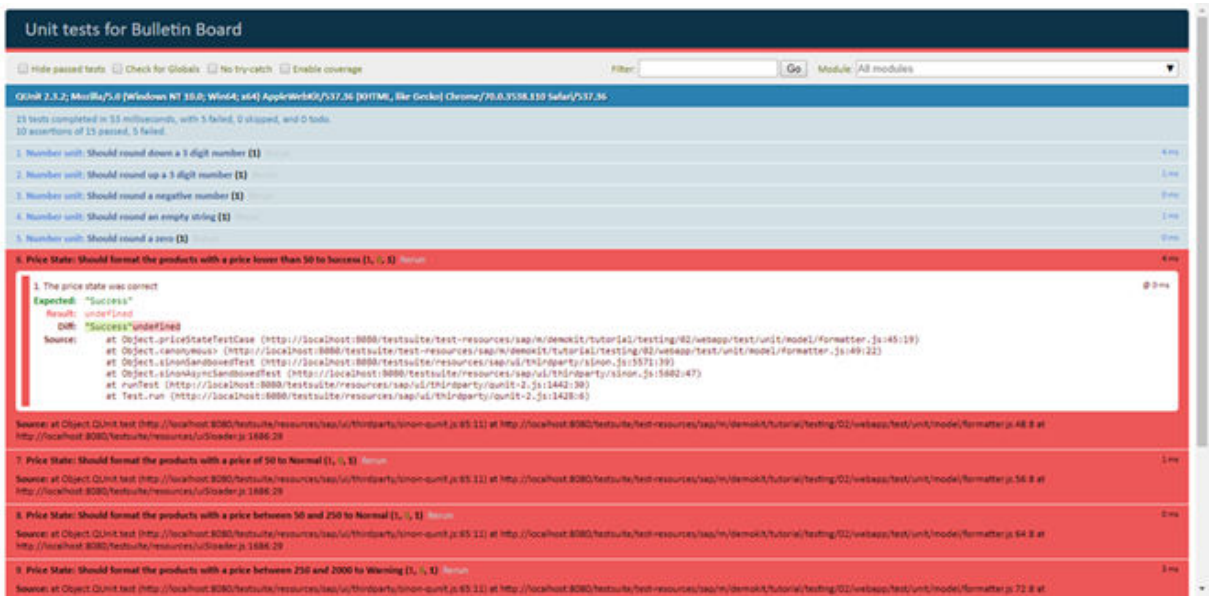


Figure 111: The unit test will initially fail as the implementation is not provided yet

Unit Test Setup

All unit tests are located in the `webapp/test/unit` folder and can be started manually by calling the `unitTests.qunit.html` file in the same folder or the entry page. This HTML page is a QUnit runner that calls all unit tests of the app and displays the test results in a readable format.

> ### i Note
>
> Some testrunners like Karma do not require an HTML page to invoke the tests but work with configuration files instead. They can directly invoke the `AllTests.js` file and log the test results in their own format. Therefore we make sure that the `AllTests.js` file does not contain any UI output and just calls the various test cases of the app.

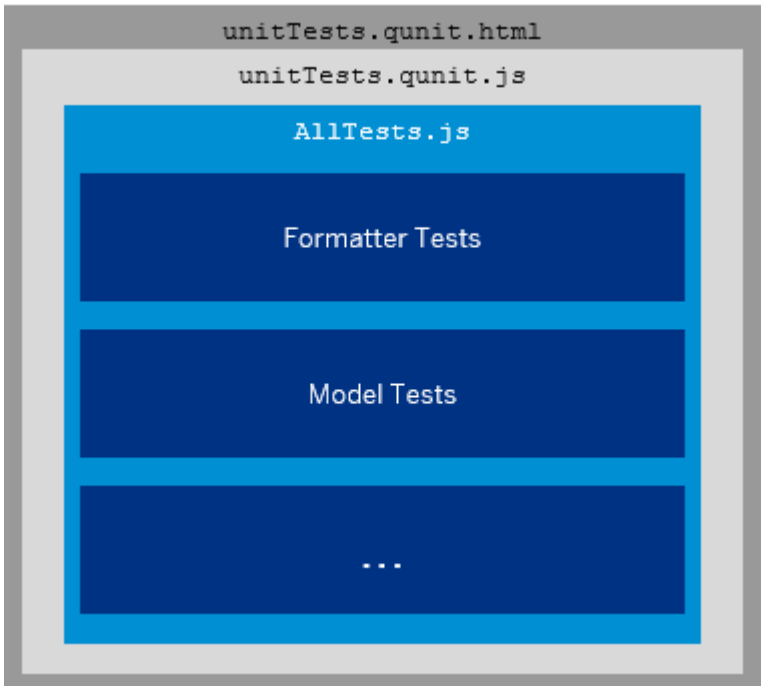


Figure 112: Unit test infrastructure in the application

Let's take a closer look at the `unitTests.qunit.html` file. The application root is stored in the `webapp` folder two levels above. In the `bootstrap` tag of the HTML page we define two namespaces to refer to the app and the unit tests. The namespace of the unit tests points to the current folder as all test artifacts are located below the current folder:

- `sap.ui.demo.bulletinboard: "../../"`
- `test.unit: "./"`

The namespace abstraction allows us to refer to all application and testing parts without having to use the full path. Furthermore, all unit tests are put in a similar folder structure and get the same name as the artifact that is tested. For example, the tests for the file `webapp/model/formatter.js` are located in the `webapp/test/unit/model/formatters.js` folder. For more details on the unit test setup please have a look at the coding of the prototype.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing Apps - Step 2.

webapp/model/formatter.js

```
sap.ui.define([
    "sap/m/Text"
], function (Text) {
    "use strict";
    return {
```

```
        numberUnit: function (sValue) {
            …
        },
        priceState: function () {
        }
    };
});
```

First we think about the feature that we want to implement. We want to introduce a new state for the price, and its value should depend on certain price ranges. SAPUI5 controls typically have semantic states like *Success*, *Warning*, or *Error*. We will need this formatter function to convert the numeric price value from the model to a state value for the control. But without caring too much about the actual implementation of this formatter we just add an empty function `priceState` to the formatter file for now and focus on the unit tests first.

webapp/test/unit/model/formatter.js

```
sap.ui.define([
    "./model/formatter"
], function (formatter) {
    "use strict";
    QUnit.module("Number unit");
        …
    QUnit.module("Price State");

    function priceStateTestCase(oOptions) {
        // Act
        var sState = formatter.priceState(oOptions.price);

        // Assert
        oOptions.assert.strictEqual(sState, oOptions.expected, "The price state
was correct");
    }

    QUnit.test("Should format the products with a price lower than 50 to
Success", function (assert) {
        priceStateTestCase.call(this, {
            assert: assert,
            price: 42,
            expected: "Success"
        });
    });

    QUnit.test("Should format the products with a price of 50 to Normal",
function (assert) {
        priceStateTestCase.call(this, {
            assert: assert,
            price: 50,
            expected: "None"
        });
    });

    QUnit.test("Should format the products with a price between 50 and 250 to
Normal", function (assert) {
        priceStateTestCase.call(this, {
            assert: assert,
            price: 112,
            expected: "None"
        });
    });

    QUnit.test("Should format the products with a price between 250 and 2000 to
```

```
Warning", function (assert) {
        priceStateTestCase.call(this, {
            assert: assert,
            price: 798,
            expected: "Warning"
        });
    });

    QUnit.test("Should format the products with a price higher than 2000 to
Error", function (assert) {
        priceStateTestCase.call(this, {
            assert: assert,
            price: 2001,
            expected: "Error"
        });
    });
});
```

Now we write tests that call the function we have just defined and check for the correct result when passing in various arguments.

By writing these tests, we actually implement the following specification in our tests that was defined by the product team.

- price < 50: Status is green (*Success*)
- price >= 50 and price < 250: Status is normal (*None*)
- price >= 250 and price < 2000: Status is orange (*Warning*)
- price >= 2000: Status is red (*Error*)

Whenever we run the tests, we will implicitly check that the feature is still working as it was designed. To keep it simple, we should only write a minimum set of tests that cover the most important cases, but also including edge cases like the value 50 or unexpected values.

Let's have a look at the implementation of the unit tests now: We add our unit tests to the `webapp/test/unit/model/formatter.js` file. The path below the app and the test folder is similar so it can easily associate the test with the tested functionality. There are already formatter functions for the number unit conversion defined in the code - you can have a quick look before we add our own tests.

We add a new QUnit module for our price state tests after the number unit conversion tests. We could write a test checking the result of the formatter for each of these cases but we do not want to repeat ourselves ("DRY") – neither in the tests nor in the application coding – so we create a reuse function called `priceStateTestCase`. In this function, we call the formatter with the arguments provided as `oOptions` and make a `strictEqual` assertion for the expected parameter.

> **i** Note
>
> There must be at least one assertion per QUnit test. If the actual value matches the expected value then the test is successful. However, if there are more assertions in a test case and a subsequent assertion fails, the whole test fails with the error message of the failed assertion.
>
> There are also other types of assertions, for example the `ok` assertion that does not check the type. For more details, have a look at the official QUnit documentation.

The assert object – a special object injected by QUnit – is passed on as a reference to the function. QUnit is loaded once for the whole unit testing part of the app.

> **i Note**
>
> The main page for calling the unit tests is `webapp/test/unit/unitTests.qunit.html`. In this file we load the QUnit runtime and an `AllTests.js` file that loads and directly executes all files with unit tests. The other content of this file is just HTML for displaying the QUnit test result page.

And now for the actual test cases: Whenever we want to start a new test we call `QUnit.test` with a test description and a callback function containing the test logic as an argument. The callback is invoked with a special assert object that is maintained by QUnit. We can simply call assertions as we saw above.

Inside each test we simply call our reuse function with different parameters for the price and the expected state that reflect our specification above. With five tests we can check the most important cases for our price state converter. There are four tests for the four different states and one edge case test with the value `50`, that makes sure that the correct state is chosen.

That's it, you just wrote your first unit test. When you call the `webapp/test/unit/unitTests.qunit.html` file in your browser, you can see that the first module for the number unit formatter is still green but our price state tests are red and failing. The error message tells us that the result of the empty formatter function is not as expected.

TDD methodology tells us to do the implementation as soon as the test fails and to come back to testing as soon as the tests are successful again. You run the unit tests after each code change, and you're done when the test does not fail anymore. We now switch to the implementation part and define the details of the formatter function in the next step.

Conventions

- Write unit tests for testing the logical correctness of your features

Related Information

Unit Testing with QUnit [page 1159]
QUnit Home Page

Step 3: Adding the Price Formatter

We will now take care of the implementation of the price formatter and make sure that the tests we wrote in the previous step run successfully.

If the tests are passed, we can be sure that the formatter is formally correct but it is still not visible in the app. So additionally, we will add the formatter to the UI to be able to verify and check that the price is shown properly.

Preview

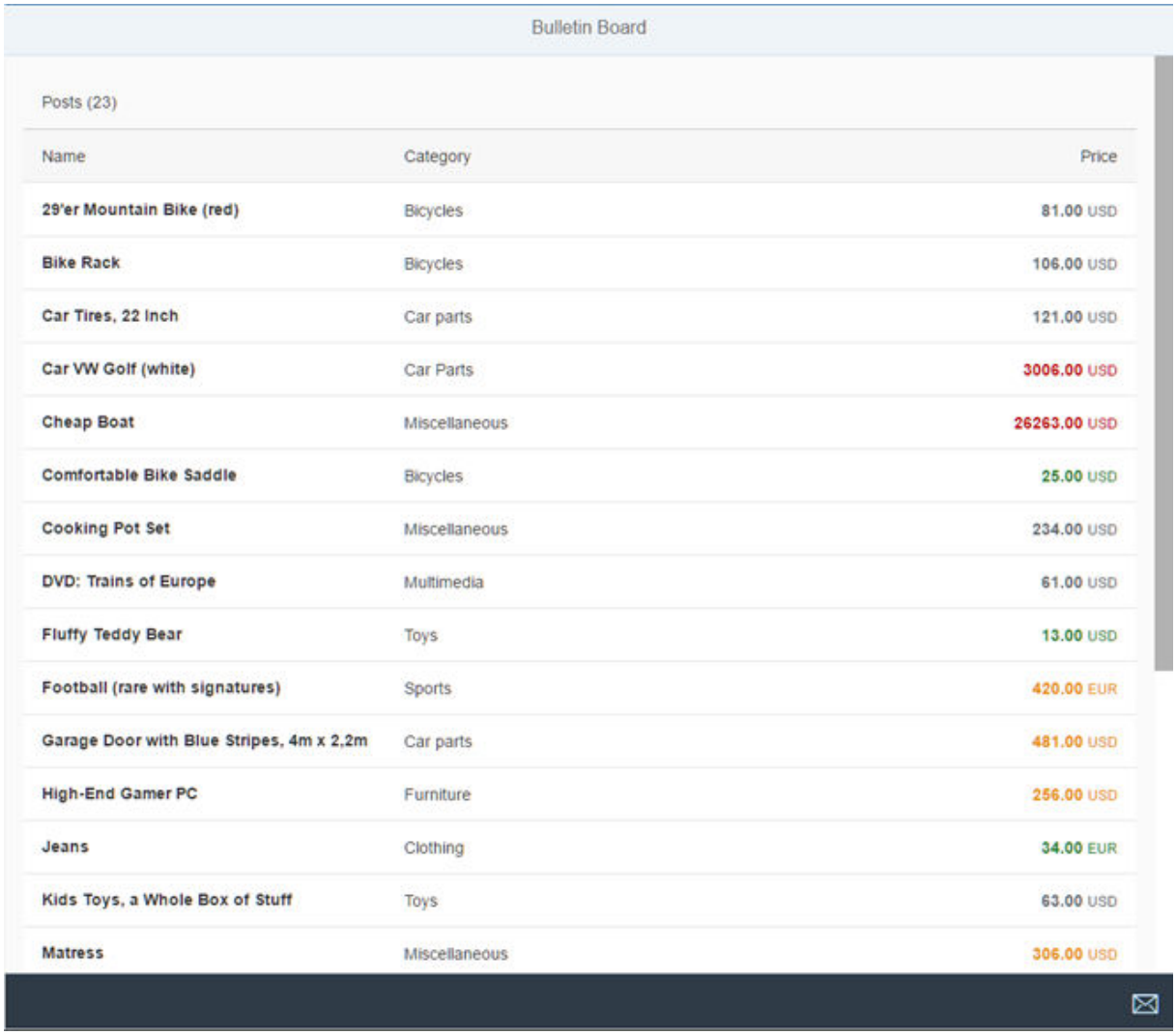


Figure 113: The price is now formatted with a semantic color

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 3.

webapp/model/formatter.js

```
sap.ui.define([
    "sap/m/Text"
], function (Text) {
    "use strict";
    return {
        numberUnit: function (sValue) {
            …
```

```
        },
        /**
         * Defines a value state based on the price
         *
         * @public
         * @param {number} iPrice the price of a post
         * @returns {string} sValue the state for the price
         */
        priceState: function (iPrice) {
            if (iPrice < 50) {
                return "Success";
            } else if (iPrice >= 50 && iPrice < 250) {
                return "None";
            } else if (iPrice >= 250 && iPrice < 2000) {
                return "Warning";
            } else {
                return "Error";
            }
        }
    };
});
```

We change the empty formatter function that we have added in the last step and add the implementation details to it. If the implementation matches the specification embedded in our tests we are done with implementing the formatter.

The input for the formatter is the price value from the model and the result is the state as a `string` value. The actual implementation logic is quite simple and returns a semantic state value based on the price as we have seen already in the test. There are four cases that are reflected in the `if/else` statements inside the formatter.

You can now run the file `webapp/test/unit/unitTests.qunit.html` and check if the unit tests run successfully. You should see your new test cases on the result page. If the overall result is successful then we have successfully implemented our first feature.

webapp/view/Worklist.view.xml

```
…
<ColumnListItem vAlign="Middle">
    <cells>
        …
        <ObjectNumber
            number="{
                path: 'Price',
                formatter: '.formatter.numberUnit'
            }"
            state="{
                path: 'Price',
                formatter: '.formatter.priceState'
            }"
            unit="{Currency}"/>
    </cells>
</ColumnListItem>
…
```

We still have to apply the changes to our UI so that we can actually see the formatted price in the app. Unit tests are typically testing the logic independent of the user interface. That is why the tests are running successfully even though we did not adapt the UI yet.

In our worklist view we simply add a state attribute to the `ObjectNumber` control in the `columns` aggregation. We define the same data binding path as for the number, but we use our new formatter function to determine the proper state. If you now run the `webapp/test/mockServer.html` file, you can see that some of the product prices are listed in green, black, orange, and red depending on their price.

Related Information

API Reference: `sap.ui.core.ValueState`
API Reference: `sap.m.ObjectNumber`

Step 4: Testing a New Module

In the first unit test we have just extended the formatters module with a new function. Now we will write a unit test that will test the functionality of an entirely new module.

A frequently used feature of a bulletin board is to flag interesting posts to mark them for later reading. The UI should contain a button to toggle the flagged state for each item. We will implement this feature with a custom type and again start writing the test case for it first and add the implementation later.

Preview

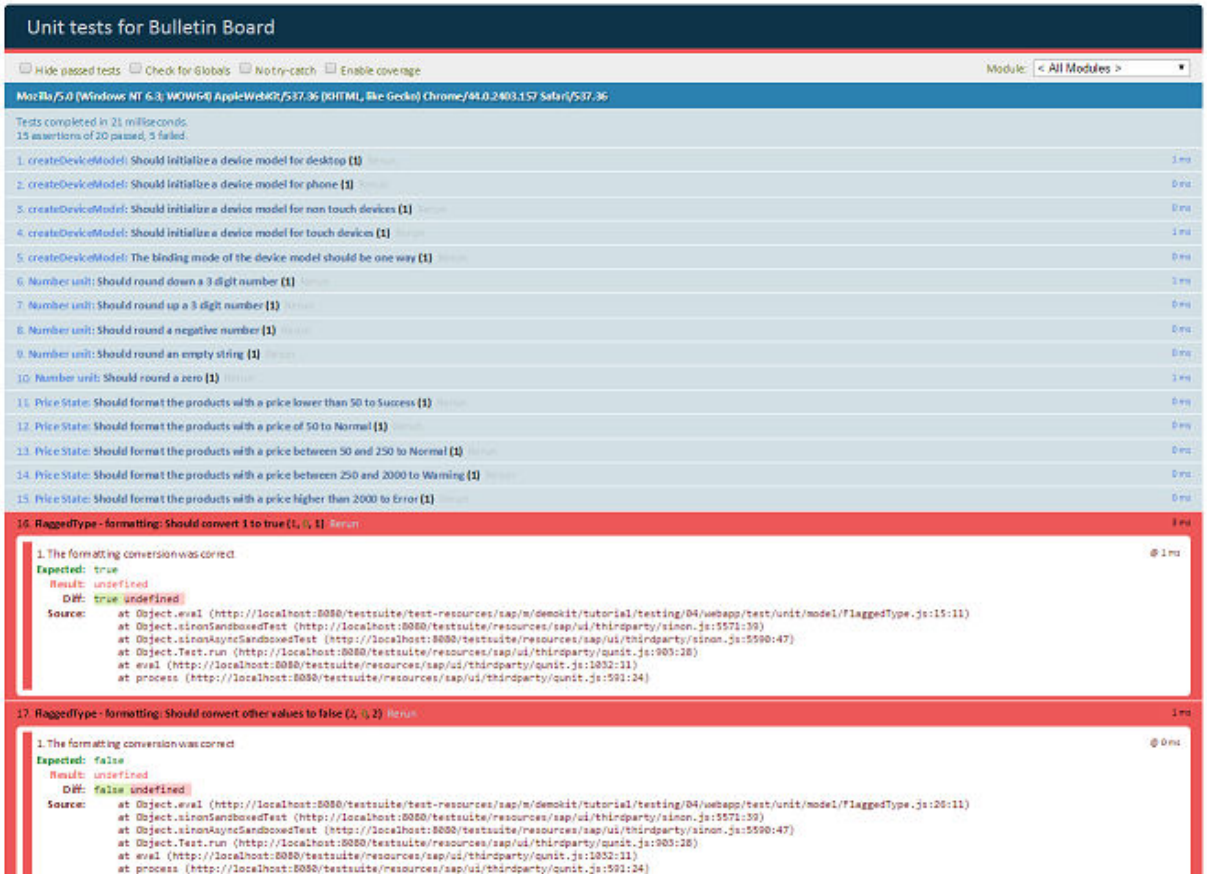


Figure 114: The unit test for the `Flagged` feature will fail until the feature is implemented

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 4.

webapp/model/FlaggedType.js (new)

```
sap.ui.define([
    "sap/ui/model/SimpleType"
], function (SimpleType) {
    "use strict";
    return SimpleType.extend("sap.ui.demo.bulletinboard.model.FlaggedType", {
        formatValue: function () {
        },
        parseValue: function () {
        },
        validateValue: function () {
        }
    });
});
```

We plan to control a button state based on the `Flagged` property in the model. The button expects a Boolean value for the pressed state. In the model, we have a binary integer representation, so we will again need conversion logic to format the model value. And we also need a back conversion to store a state change in the model when the user clicks the button.

A formatter function will only take care of one direction so this time we decide to implement a custom data type for the conversions. As with the previous test, we add an empty hull for our new data type in the model folder. The `FlaggedType` extends the `SimpleType`. Its interface provides two conversion functions and a validation function:

- `formatValue`: formats a model value to be displayed in the UI
- `parseValue`: parses a UI value to be stored in the model
- `validateValue`: checks a value for displaying validation errors

webapp/test/unit/model/FlaggedType.js (new)

```
sap.ui.require(
    [
        "sap/ui/demo/bulletinboard/model/FlaggedType"
    ],
    function (FlaggedType) {
        "use strict";
        QUnit.module("FlaggedType - formatting");
        QUnit.test("Should convert 1 to true", function (assert) {
            // Act
            var bFormattedValue = new FlaggedType().formatValue(1);
            // Assert
            assert.strictEqual(bFormattedValue , true, "The formatting
conversion was correct");
        });
        QUnit.test("Should convert other values to false", function (assert) {
            var oFlaggedType = new FlaggedType();
            // Act
            var bFormattedZero = oFlaggedType.formatValue(0);
            var bFormattedNegativeNumber = oFlaggedType.formatValue(-666);
            // Assert
            assert.strictEqual(bFormattedZero, false, "The formatting conversion
was correct");
            assert.strictEqual(bFormattedNegativeNumber, false, "The formatting
conversion was correct");
        });
        QUnit.module("FlaggedType - parsing");
        QUnit.test("Should parse false to 0", function (assert) {
            // Act
            var iParsedValue = new FlaggedType().parseValue(false);
            // Assert
            assert.strictEqual(iParsedValue, 0, "The parsing conversion matched
the input");
        });
        QUnit.test("Should parse true to 1", function (assert) {
            // Act
            var iParsedValue = new FlaggedType().parseValue(true);
            // Assert
            assert.strictEqual(iParsedValue, 1, "The parsing conversion matched
the input");
        });
    }
);
```

The new `FlaggedType.js` file matches the file name of the implementation and is put in the `model` subfolder of the `test/unit` folder similar to the implementation under the `webapp` folder. By keeping the same structure for tests and productive code, we can easily relate the tests to the implementation.

We define this testing module with `sap.ui.require` since we just want to load dependencies but do not want to declare a namespace for this testing module. We load the new and still empty `FlaggedType` implementation as the only dependency and declare two QUnit modules: one for formatting and one for parsing, to check both the to- and back-conversion of the flagged type.

> **i Note**
>
> We do not test the validation function of the data type as our conversion is so simple. There are no expected validation errors that we have to take care of.

In each QUnit module we define test cases for each condition. For a Boolean conversion there are just two cases, `true` and `false`. So we expect that the integer value `1` is converted to `true` and everything else to `false`.

Let's have a look at the first test case to see how the custom data type is invoked for testing.

As we have loaded the type as a dependency, we can just access it with the variable `FlaggedType` and create a new instance of it in each test case. This time we do not create a `reuse` function but simply create the instance inside the test case. On the type we manually call the function `formatValue` that we want to test and compare the result to the expected value in an assertion.

In the second test case, we check all other values, we expect it to be `0` but it could be also a negative value. So we check both cases in the same test case with a separate assertion each. Only when both assertions are fulfilled the test will be successful.

The other test cases in the parsing module are similar and check the back conversion from Boolean value to integer value.

webapp/test/unit/AllTests.js

```
sap.ui.define([
    "./model/models",
    "./model/formatter",
    "./model/FlaggedType"
], function() {
    "use strict";
});
```

In the `AllTests.js` file we just load the new testing module as a dependency so that it is executed automatically whenever we execute the unit tests.

You can now call the unit tests and check the result. As in the previous step, the tests should fail with an error message that the conversion is not correct. This is expected as we did not implement the conversion logic yet but just the tests for it.

Conventions

- Use data types if you need both formatting and parsing of a model value
- Organize the tests in the same file structure as the productive code

Related Information

API Reference: `sap.ui.model.SimpleType`

API Reference: `sap.ui.require`

Formatting, Parsing, and Validating Data [page 854]

Step 5: Adding a *Flag* Button

Now that we have implemented the conversion tests, we add the corresponding functionality and show the button to flag a post in the app. The design team has specified that the flag feature should be implemented with a toggle button that has a flag icon.

Preview

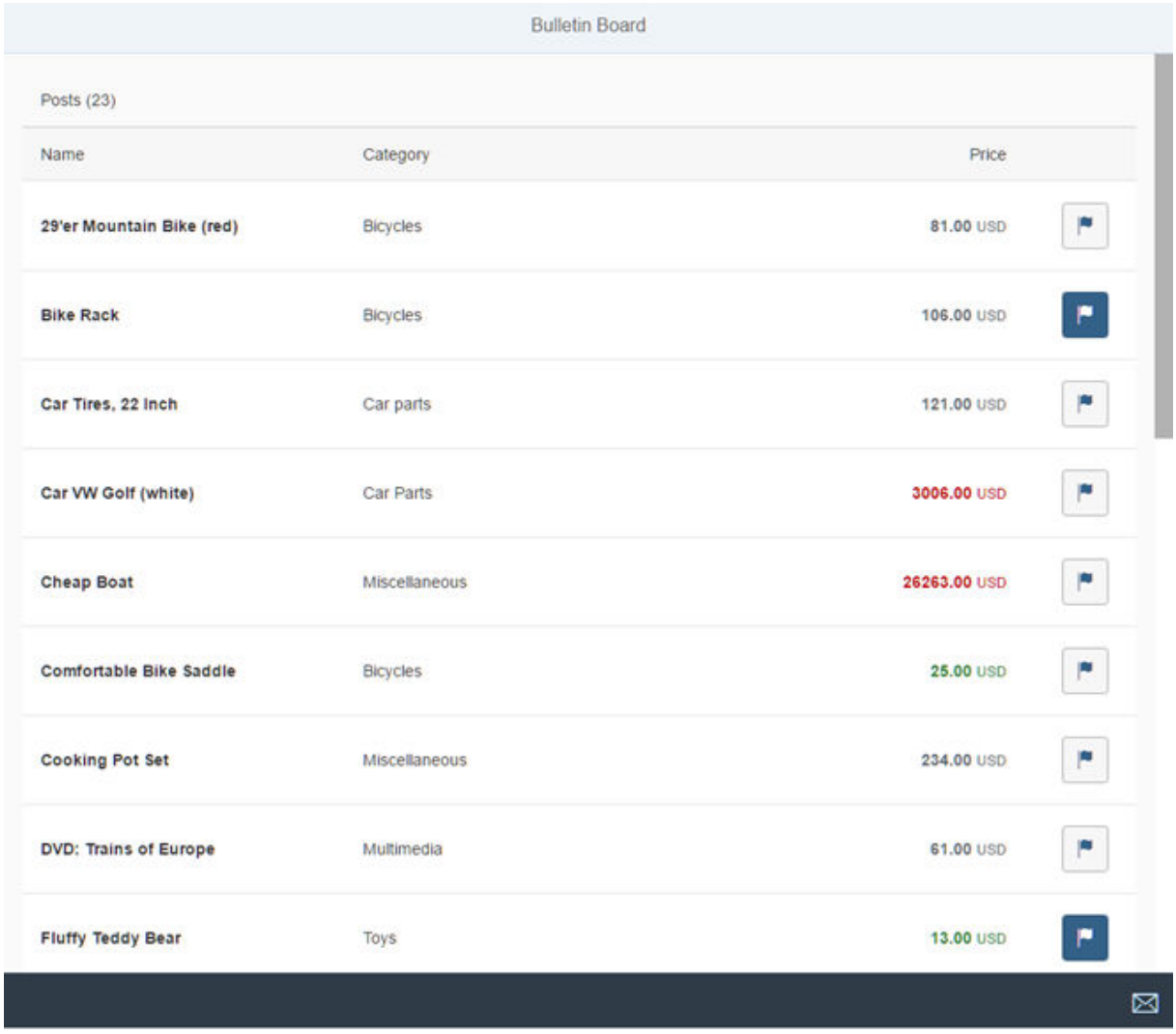


Figure 115: The *Flag* button is now added to the table

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 5.

webapp/model/FlaggedType.js

```
sap.ui.define([
    "sap/ui/model/SimpleType"
], function (SimpleType) {
    "use strict";
    return SimpleType.extend("sap.ui.demo.bulletinboard.model.FlaggedType", {
        /**
         * Formats the integer value from the model to a boolean for the pressed
state of the flagged button
         *
         * @public
         * @param {number} iFlagged the integer value of the formatted property
         * @returns {boolean} 1 means true, all other numbers means false
         */
        formatValue: function (iFlagged) {
            return iFlagged === 1;
        },
        /**
         * Parses a boolean value from the property to an integer
         *
         * @public
         * @param {boolean} bFlagged true means flagged, false means not flagged
         * @returns {number} true means 1 , false means 0
         */
        parseValue: function (bFlagged) {
            if (bFlagged) {
                return 1;
            }

            return 0;
        },
        /**
         * Validates the value to be parsed
         *
         * @public
         * Since there is only true and false, no client side validation is
required
         * @returns {boolean} true
         */
        validateValue: function () {
            return true;
        }
    });
});
```

Lets start with the implementation code for the `FlaggedType`. We now add the documentation in JSDoc format and the implementation of the three functions of the data type to the previously empty stub:

- The `formatValue` function takes care of the conversion from the model to the UI. As specified in the tests, a model value of `1` will be converted to `true`, everything else to `false`. In the implementation code, this equals to `"iFlagged === 1"`.
- Similarly, the `parseValue` function is called by SAPUI5 when the data is written back to the model. Here, we convert the Boolean value to an integer again.
- The validation function always returns `true` in this simple case, we do not expect any validation errors for this data type.

We call these functions of the data type in the unit tests directly. So if you now run your unit tests by calling the `webapp/test/unit/unitTests.qunit.html` page, the tests should already run successfully.

webapp/view/Worklist.view.xml

```
…
<Table …>
    …
    <columns>
        …
        <Column width="33%" id="unitNumberColumn" hAlign="End" vAlign="Middle">
            <Text text="{i18n>TableUnitNumberColumnTitle}"
id="unitNumberColumnTitle"/>
        </Column>
        <Column width="80px" id="flaggedColumn" demandPopin="true"
vAlign="Middle"/>
    </columns>
    <items>
        <ColumnListItem vAlign="Middle">
            <cells>
                …
                <ObjectNumber… />
                <ToggleButton
                    id="flaggedButton"
                    tooltip="{i18n>flaggedTooltip}"
                    icon="sap-icon://flag"
                    pressed="{
                        path: 'Flagged',
                        type: '.types.flagged'
                    }"
                    class="sapUiMediumMarginBeginEnd"/>
            </cells>
        </ColumnListItem>
    </items>
</Table>
…
```

In the view, we add a new column and a cell for the flag feature at the end of the table. We fill the cell with a `sap.m.ToggleButton` control that serves as our input control for the `Flagged` state. We define a `flag` icon in the button, a tooltip from the resource bundle, and a layouting class to make our example complete. The control's `pressed` property is bound to the `Flagged` field in the model. Here we also apply the custom data type that is part of the controller.

webapp/controller/Worklist.controller.js

```
sap.ui.define([
    './BaseController',
    'sap/ui/model/json/JSONModel',
    '../model/formatter',
    '../model/FlaggedType'
    'sap/m/library'
], function (BaseController, JSONModel, formatter, FlaggedType, mobileLibrary) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.bulletinboard.controller.Worklist", {
        types : {
            flagged: new FlaggedType()
        },
        formatter: formatter,
        …
    });
});
```

The controller loads the custom data type as a dependency similar to the formatters. It is then provided as a property of the internal variable `types` so that it can be accessed as `.types.flagged` in the view as we have seen above.

The conversion functions that are made available when we create an instance of the type are called automatically by SAPUI5 when needed. However, by default the back conversion to the model is not enabled, so we still need a small change in the component.

webapp/Component.js

```
sap.ui.define([
    …
], function (UIComponent, ResourceModel, models) {
    "use strict";
    return UIComponent.extend("sap.ui.demo.bulletinboard.Component", {
    …
        init: function () {
            // call the base component's init function
            UIComponent.prototype.init.apply(this, arguments);
            // allow saving values to the OData model
            this.getModel().setDefaultBindingMode("TwoWay");
            …
        }
    });
});
```

To enable the propagation of the bound view properties to the model, we need to set the model's default binding mode to `TwoWay`. For an OData model the default mode is `OneWay` which means that properties are not written back to the model automatically. We want to propagate the state of the button automatically to the model, when the button for a post is clicked.

webapp/i18n/i18n.properties

```
#~~~ Worklist View ~~~~~~~~~~~~~~~~~~~~~~~~~
…
#XTOL: tooltip for the flagged button
flaggedTooltip=Mark this post as flagged
…
```

Finally, add the new string for the button tooltip to the resource bundle file. Now we can also test the application manually by calling the `webapp/test/mockServer.html` page and making sure some of the buttons are pressed initially as reflected in the model. When we flag an item by choosing the button, the property is written back to the model transparently.

> **i** Note
>
> As this feature covers both conversion and interaction parts, we could also have written an integration test for it to test the interaction part also. Feel free to add an integration test for this feature if you like, we will skip it here to focus on unit testing in this step.

Related Information

Formatting, Parsing, and Validating Data [page 854]

Step 6: A First OPA Test

A bulletin board may contain many posts. We expect to have a high data load once it is officially released. Then, there might be performance issues and long loading times if we display all entries at the same time. Therefore we will introduce a feature that limits the initial display to 20 items. The user can then click on a more button to view more items. As with the unit test, we start by writing an integration test for this feature and then add the application functionality later.

Preview

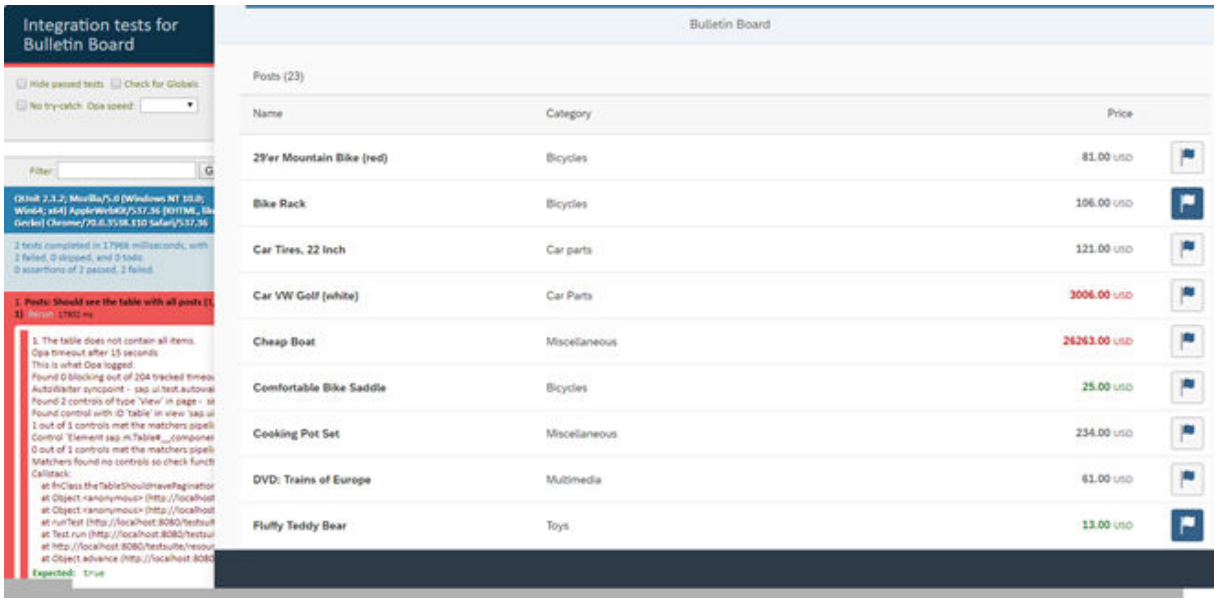


Figure 116: The OPA test page is waiting for more items to be loaded

Coding

You can view and download all files in the Demo Kit at Testing - Step 6.

Integration Test Setup

All integration tests are located in the `webapp/test/integration` folder and can be started manually by calling the `opaTests.qunit.html` file in the same folder or the entry page. Similar to the unit tests, the HTML

page is a QUnit runner that calls all integration tests of the app and displays the test results in a readable format. It also might be omitted by other testrunners. There are also two namespaces defined for the app and the integration test folder as you have seen in the unit test setup.

We write integration tests with OPA5 – a tool that is integrated and delivered with SAPUI5. It is the short name for One-Page Acceptance tests for SAPUI5. "One-Page" here means that OPA5 is designed for single-page Web applications, i.e. applications that consist only of one HTML file. OPA5 runs in the same browser window as the application to be tested.

> **i Note**
>
> There is also a stand-alone version of OPA5 called “OPA” available that can be used for testing any kind of single-page Web application and that does not provide any SAPUI5-specific functionality. In this tutorial, “OPA” always refers to OPA5. It includes functionality for easily finding and matching SAPUI5 controls as well as their properties and aggregations.

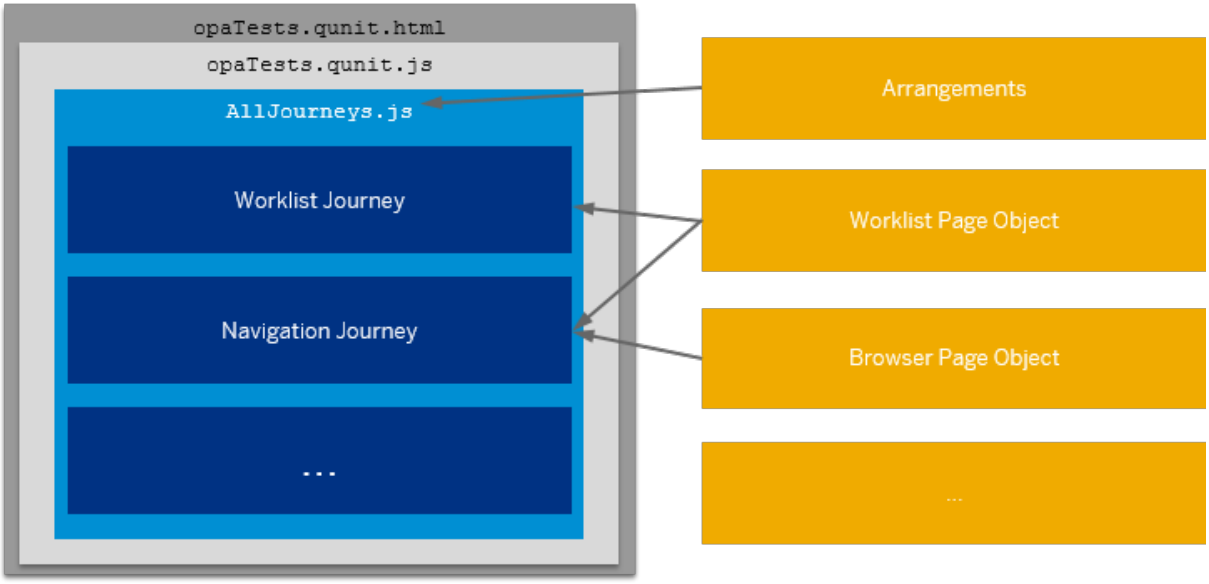


Figure 117: Integration test infrastructure in the project

For structuring integration tests with OPA we use “journeys”. A test journey contains all test cases for a specific view or use case, for example the navigation journey simulates user interaction with the app.

The journey uses another structuring element of OPA called “page object” that encapsulates actions and assertions needed to describe the journey. Typically those are related to a view in the app but there can also be stand-alone pages for browsers or common functionality.

> **i Note**
>
> When you first start writing tests, you may find it difficult to figure out the correct control locators. The *Test Recorder* tool can suggest a solution in the form of a code snippet. For most controls, it can find a combination of matchers that match a single control. Then, all you need to do is copy the code snippet to your OPA5 page object. For more information, see Test Recorder [page 1251].

webapp/test/integration/WorklistJourney.js

```
sap.ui.define([
    "sap/ui/test/opaQunit",
    "./pages/Worklist"
], function (opaTest) {
    "use strict";
    QUnit.module("Posts");
    opaTest("Should see the table with all posts", function (Given, When, Then) {
        // Arrangements
        Given.iStartMyApp();
        // Assertions
        Then.onTheWorklistPage.theTableShouldHavePagination().
            and.theTitleShouldDisplayTheTotalAmountOfItems();
    });

    opaTest("Should be able to load more items", function (Given, When, Then) {
        //Actions
        When.onTheWorklistPage.iPressOnMoreData();

        // Assertions
        Then.onTheWorklistPage.theTableShouldHaveAllEntries();
        // Cleanup
        Then.iTeardownMyApp();
    });
});
```

Let's add our first new OPA test to the `WorklistJourney.js` file. We describe all test cases related to the worklist logic. We can see that there is already a test `Should see the table with all posts` defined that checks if the table contains the expected number of items. There is a function `opaTest` that initiates a test description and receives a test description as the first argument as well as a callback function as the second argument. This format is similar to the unit test function `QUnit.test` except for the three arguments of the callback function that are specific to OPA.

The three objects `Given`, `When`, `Then` are filled by the OPA runtime when the test is executed and contain the `arrangements`, `actions`, and `assertions` for the test. The "Given-When-Then" pattern is a common style for writing tests in a readable format. To describe a test case, you basically write a user story. Test cases in this format are easy to understand, even by non-technical people.

Let's give it a try with our new feature that only displays 20 posts in the table initially and will load more posts when we press a trigger button or scroll down. Here is our user story "Should see the table with all posts" and its code representation:

- **Arrangements**
  Define possible initial states, e.g. the app is started, or specific data exists. For performance reasons, starting the app is usually done only in the first test case of a journey. `Given.iStartMyApp();`
- **Actions**
  Define possible events triggered by a user, e.g. entering some text, clicking a button, navigating to another page. `When.onTheWorklistPage.iPressOnMoreData();`
- **Assertions**
  Define possible verifications, e.g. do we have the correct amount of items displayed, does a label display the right data, is a list filled. At the end of the test case, the app is destroyed again. This is typically done only once in the last test case of the journey for performance reasons.
  `Then.onTheWorklistPage.theTableShouldHaveAllEntries ().and.iTeardownMyApp();`

Please also note that you have to move the `and.iTeardownMyApp()` concatenation from the previous `opaTest` function and put it at the end of the last test of a journey, in this case this is our new test. For

performance reasons, we only start and destroy the app once per journey, as it takes several seconds to load the app. You can concatenate actions and assertions with the OPA helper object and in an easily readable way. The functions will be executed one after another.

Now you might wonder where all those descriptive functions and the helper object `onTheWorklistPage` are coming from. The answer is simple, the `onTheWorklistPage` object is a structuring element of OPA and inside we will implement the actions and assertions used in this test.

webapp/test/integration/pages/Worklist.js

```
sap.ui.define([
        'sap/ui/test/Opa5',
        'sap/ui/test/matchers/AggregationLengthEquals',
        'sap/ui/test/matchers/I18NText',
        'sap/ui/test/actions/Press'
    ],
        function (Opa5,
              AggregationLengthEquals,
              I18NText,
              Press) {
        "use strict";
        var sViewName = "Worklist",
            sTableId = "table";
        Opa5.createPageObjects({
            onTheWorklistPage: {
                actions: {
                    iPressOnMoreData: function () {
                        // Press action hits the "more" trigger on a table
                        return this.waitFor({
                            id: sTableId,
                            viewName: sViewName,
                            actions: new Press(),
                            errorMessage: "The table does not have a trigger."
                        });
                    }
                },
                assertions: {
                    theTableShouldHavePagination: function () {
                        return this.waitFor({
                            id: sTableId,
                            viewName: sViewName,
                            matchers: new AggregationLengthEquals({
                                name: "items",
                                length: 20
                            }),
                            success: function () {
                                Opa5.assert.ok(true, "The table has 20 items on
the first page");
                            },
                            errorMessage: "The table does not contain all items."
                        });
                    },
                    theTableShouldHaveAllEntries: function () {
                        return this.waitFor({
                            id: sTableId,
                            viewName: sViewName,
                            matchers: new AggregationLengthEquals({
                                name: "items",
                                length: 23
                            }),
                            success: function () {
```

```
                                Opa5.assert.ok(true, "The table has 23 items");
                            },
                            errorMessage: "The table does not contain all items."
                        });
                    },
                    theTitleShouldDisplayTheTotalAmountOfItems: function () {
                        return this.waitFor({
                            id: "tableHeader",
                            viewName: sViewName,
                            matchers: new I18NText({
                                key: "worklistTableTitleCount",
                                propertyName: "text",
                                parameters: [23]
                            }),
                            success: function () {
                                Opa5.assert.ok(true, "The table header has 23
items");
                            },
                            errorMessage: "The table header does not contain the
number of items: 23"
                        });
                    }
                }
            }
        });
```

As you can see, the OPA page object is constructed with the call `Opa5.createPageObjects` and a `configuration` object that contains the actions and assertions properties.

For our test case we need to add an action `iPressOnMoreData` and an existing assertion `theTableShouldHaveAllEntries`. OPA tests are running asynchronously, so each action and assertion starts with a `waitFor` statement. The OPA run time will check and wait for the condition to be fulfilled every 400 ms by polling. If the condition is met, the `success` function of the configuration is called. If the condition is still not fulfilled after a certain amount of time (by default it is 15 seconds but this can be configured) the test will fail.

Let's start with the action `iPressOnMoreData`. We define a `waitFor` statement with the current view and the table. Those IDs are stored as internal variables in the `require` statement above and are available in all tests. OPA will now try to find the table based on IDs. As soon as the table is available on the screen and it can be interacted with (it is visible, not busy,...), the `Press` action is invoked, if not, the error message is displayed and the test fails. When executed on a table, the `Press` action will simulate that a users chooses the *More Data* button.

> **i Note**
>
> The `Press` action depends on the control that it is triggered on and has a default behavior for most UI controls. If you, for example, execute `Press` on a `sap.m.Page`, this will trigger the *Back* button's `Press` event. This behavior can be overridden by passing an ID as argument to the action. For more information, see the API Reference: `sap.ui.test.actions.Press`.

The assertion `theTableShouldHaveAllEntries` is structured similarly, but it does not trigger an action. Here, we use the `success` function of `waitFor` to assert if our application is in the expected state. This state is defined by the matchers (in our case we expect that the list contains 23 items by using the `AggregationLengthEquals`. The `success` function does not execute the additional checks that are needed for triggering an action. the liste does not have to be `interactable` to verify that the state of the application is correct..

With this helper object we can simply check the length of the table aggregation `items` to the expected number of items. We have 23 entries in our local mock data that we also use for this integration test. You can see that the number of items is actually hard-coded in the test. So only if the table has exactly 23 items, the matcher is evaluating to `true` and the assertion is passed successfully.

> **i** Note
>
> The items in our app are served from the mock server with a slight delay so that we can see how a real service on a backend system would behave. Even if we would have a real backend, we would purposely use the mock server for manual testing and for using them in our test cases as the test data remains stable and unchanged. This creates a more reliable test environment and easier tests. So we can write a test that checks exactly for 23 items here.

Now run the `webapp/test/integration/opaTests.qunit.html` file and make sure that the test is failing. When our new test is invoked, OPA will run into a timeout because the trigger area is not found yet. You can see more information, if you open the developer console of your browser and check the messages in the console.

Conventions

- Use OPA tests for UI-related integration tests
- Structure OPA tests with page objects
- Use the standard matchers provided by OPA5 if possible

Related Information

Integration Testing with One Page Acceptance Tests (OPA5) [page 1182]
Test Recorder [page 1251]
API Reference: `sap.ui.test.matchers`
API Reference: `sap.ui.test.Opa5`
Samples: `sap.ui.test.Opa5`

Step 7: Changing the Table to a Growing Table

Let's switch back to developing and add the missing feature for the test we implemented in the previous step. We will simply change the table to a growing table as this is a basic feature of the table. This will display a trigger at the end of the table that the user can click on to display more items.

Preview

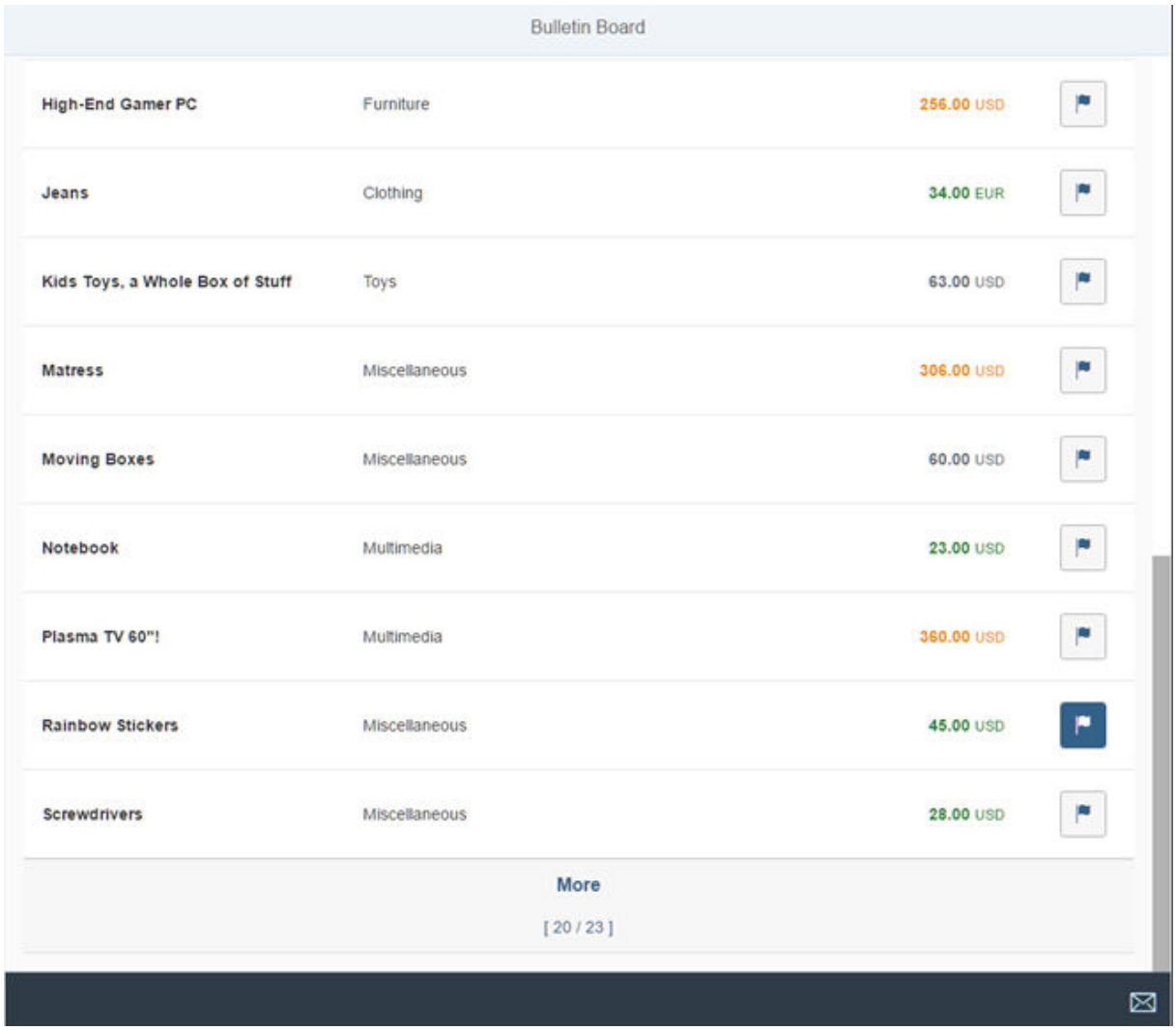


Figure 118: The List of posts is now dynamically loading more items when we scroll to the end of the page

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 7.

webapp/view/Worklist.view.xml

```
<mvc:View …
    <semantic:FullscreenPage
        id="page"
        title="{i18n>worklistViewTitle}">
        <semantic:content>
            <Table
                id="table"
                growing="true"
                width="auto"
                …
                >
            …
            </Table>
        </semantic:content>
        …
    </semantic:FullscreenPage>
</mvc:View>
```

We simply set the parameter `growing` to `true` to enable our feature. Now we can run the integration test that we just wrote in the previous step and it should not fail anymore. Similarly, if we run the app, we now see only 20 items initially. And if we choose the *More* button then three more items are loaded.

Conventions

- Use OPA tests for UI-related integration tests

Related Information

Growing Feature for Table and List [page 2342]
API Reference: `sap.m.Table`

Step 8: Testing Navigation

So far, we have a list of posts on the home page of the app. But typically, a post comes with more details that should be displayed on a separate detail page. We call it the post page because it displays details of a post. In this step we will introduce a new journey to test the post page. We write tests that trigger typical navigation events with OPA. Testing navigation greatly helps in reducing manual testing efforts as it covers a lot of testing paths. It is good practice to cover every view of your application with at least one test, since OPA will check if an exception is thrown. In this way you can detect critical errors very fast.

Preview

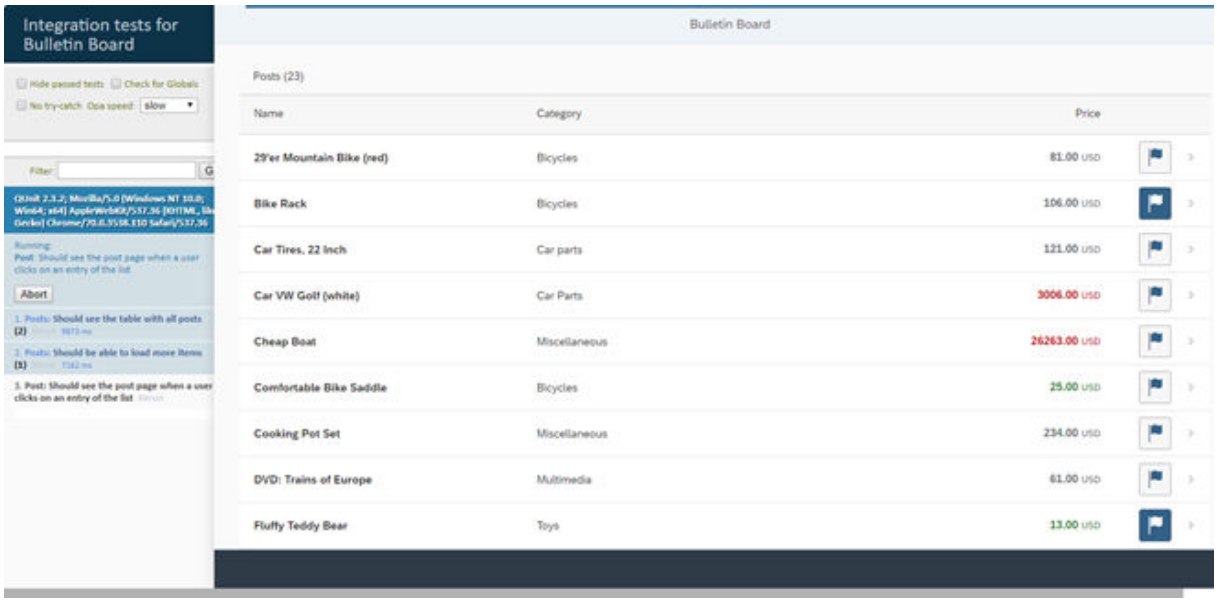


Figure 119: We add an OPA test that selects an item from the table and navigates to the post page

Coding

You can view and download all files in the Demo Kit at Testing - Step 8.

webapp/test/integration/PostJourney.js (New)

```
sap.ui.define([
    "sap/ui/test/opaQunit",
    "./pages/Worklist",
    "./pages/Browser",
    "./pages/Post"
], function (opaTest) {
    "use strict";

    QUnit.module("Post");

    opaTest("Should see the post page when a user clicks on an entry of the
list", function (Given, When, Then) {
        // Arrangements
        Given.iStartMyApp();

        //Actions
        When.onTheWorklistPage.iPressOnTheItemWithTheID("PostID_15");

        // Assertions
        Then.onThePostPage.theTitleShouldDisplayTheName("Jeans");
    });

    opaTest("Should go back to the TablePage", function (Given, When, Then) {
        // Actions
        When.onThePostPage.iPressTheBackButton();
```

```
        // Assertions
        Then.onTheWorklistPage.iShouldSeeTheTable();
    });

    opaTest("Should be on the post page again when the browser's forward button
is pressed", function (Given, When, Then) {
        // Actions
        When.onTheBrowser.iPressOnTheForwardButton();

        // Assertions
        Then.onThePostPage.theTitleShouldDisplayTheName("Jeans");

        // Cleanup
        Then.iTeardownMyApp();
    });
});
```

This new journey for the *Post* page introduces a test case that tests the navigation and also tests if the browser history is in the correct state, so that the user can navigate through our app with the back and forward button of the browser. This time, instead of adding a test we will add a new journey.

A journey represents a user's task in our app. Journeys start with the startup of our app and end with a teardown in the last test. We don't write isolated tests here, since starting up the app takes a lot of time and doing it too often slows down our test execution and feedback time considerably. If the execution speed of the tests is no problem, you may also write isolated tests.

Our new journey consists of three user interaction steps:

1. User chooses a *Post* to view the details
2. User chooses the *Back* button on the *Detail* page of the *Post* to see the list again
3. User chooses the *Forward* button to revisit the details of the post

webapp/test/integration/pages/Worklist.js – action object

```
sap.ui.define([
        'sap/ui/test/Opa5',
        'sap/ui/test/matchers/AggregationLengthEquals',
        'sap/ui/test/matchers/I18NText',
        'sap/ui/test/matchers/BindingPath',
        'sap/ui/test/actions/Press'
    ],
    function (Opa5,
              AggregationLengthEquals,
              I18NText,
              BindingPath,
              Press) {
        "use strict";
        var sViewName = "Worklist",
            sTableId = "table";
        Opa5.createPageObjects({
            onTheWorklistPage: {
                actions: {
…
                    ,
                iPressOnTheItemWithTheID: function (sId) {
                    return this.waitFor({
                        controlType: "sap.m.ColumnListItem",
                        viewName: sViewName,
                        matchers:  new BindingPath({
```

```
                                path: "/Posts('" + sId + "')"
                            }),
                            actions: new Press(),
                            errorMessage: "No list item with the id " + sId + "
was found."
                        });
                    }
```

Now that we have written our spec how the navigation to the *Post* page is planned, we first need to implement the "click" on a list item. To identify the item we are looking for, we use the `BindingPath` matcher. Doing so, we make sure that even if the order of the items changes, we always choose the same item. The `press` action simulates a user click on the item.

webapp/test/integration/pages/Post.js (New)

```
sap.ui.define([
    'sap/ui/test/Opa5',
    'sap/ui/test/matchers/Properties',
    'sap/ui/test/actions/Press'
], function (Opa5, Properties, Press) {
        "use strict";
        var sViewName = "Post";
        Opa5.createPageObjects({
            onThePostPage: {
                baseClass: Common,
                actions: {
                    iPressTheBackButton: function () {
                        return this.waitFor({
                            id: "page",
                            viewName: sViewName,
                            actions: new Press(),
                            errorMessage: "Did not find the nav button on object
page"
                        });
                    }
                },
                assertions: {
                    theTitleShouldDisplayTheName: function (sName) {
                        return this.waitFor({
                            success: function () {
                                return this.waitFor({
                                    id: "objectHeader",
                                    viewName: sViewName,
                                    matchers: new Properties({
                                        title: sName
                                    }),
                                    success: function (oPage) {
                                        Opa5.assert.ok(true, "was on the
remembered detail page");
                                    },
                                    errorMessage: "The Post " + sName + " is not
shown"
                                });
                            }
                        });
                    }
                }
            }
        });
    });
```

After navigating to the *Post* page, we need a new OPA5 `Page` object for the page to implement our actions and assertions.

An OPA5 `Page` object is used to group and reuse actions and assertions that are related to a specific part of the screen. For more information, see Cookbook for OPA5 [page 1188].

We implement a `press` event on the page's `nav` button and we assert that we are on the correct page by checking the title in the object header. The `nav` button is retrieved via DOM reference, because the page does not offer us an API here. Since the DOM ID is the most stable attribute, we are using this to retrieve the button.

webapp/test/integration/pages/Worklist.js – assertion object

```
…
,
                    iShouldSeeTheTable: function () {
                        return this.waitFor({
                            id: sTableId,
                            viewName: sViewName,
                            success: function () {
                                Opa5.assert.ok(true, "The table is visible");
                            },
                            errorMessage: "Was not able to see the table."
                        });
                    }
…
```

After going back, we want to move forwards again, but we need to check if the back navigation actually took place. So we assert that we are back on our table of posts again. We achieve this with a very simple `waitFor` statement just checking if the table is present.

webapp/test/integration/pages/Browser.js (New)

```
sap.ui.define([
    'sap/ui/test/Opa5'
], function (Opa5) {
    "use strict";
        Opa5.createPageObjects({
            onTheBrowser: {
                baseClass: Common,
                actions: {
                    iPressOnTheForwardButton: function () {
                        return this.waitFor({
                            success: function () {
                                Opa5.getWindow().history.forward();
                            }
                        });
                    }
                },
                assertions: {}
            }
        });
    });
```

We now implement an action that is triggered when the *Forward* button is chosen. Since it is not part of the browser's UI and it could be used on any page of our application, we just declare our browser's UI as an own

OPA page object. To simulate the *Forward* button, we use the `history` API of the browser. We have to wrap our action in a `waitFor` statement. Otherwise the action would be executed before our app is started.

webapp/test/integration/AllJourneys.js

```
sap.ui.define([
    "sap/ui/test/Opa5",
    "./arrangements/Startup",
    "./WorklistJourney",
    "./PostJourney"
], function (Opa5, Startup) {
    "use strict";
    Opa5.extendConfig({
        arrangements: new Startup(),
        viewNamespace: "sap.ui.demo.bulletinboard.view.",
        autoWait: true
    });
});
```

To make navigation tests complete, we add the new journey to the `AllJourneys` file that is invoked by the OPA test page.

If you execute the tests now, you can see in the logs of the developer tools that OPA is waiting for the object page to be displayed. Of course, this will not happen as it is not yet implemented. But we already have a pretty good idea on how we will implement the feature in the next step

Related Information

API Reference: `sap.ui.test.matchers.BindingPath`

Step 9: Adding the *Post* Page

Now that we have covered all kinds of tests for navigation, we introduce our *Post* page that shows details of a post in the bulletin board. To achieve this, we have to introduce a new view/controller pair and adjust the routing of the application.

Preview

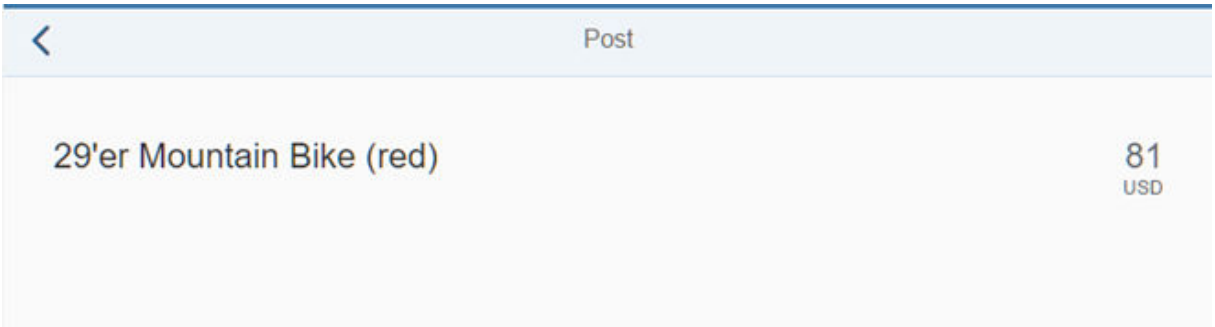


Figure 120: The *Post* page with more details about the post

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 9.

webapp/manifest.json

```
{
    "_version": "1.12.0",
    …
    "sap.ui5": {
    …
        "routing": {
            "config": {
                "routerClass": "sap.m.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.bulletinboard.view",
                "controlId": "app",
                "controlAggregation": "pages",
                "async": true
            },
            "routes": [
                {
                    "pattern": "",
                    "name": "worklist",
                    "target": "worklist"
                },
                {
                    "pattern": "Post/{postId}",
                    "name": "post",
```

```
                    "target": "post"
                }
            ],
            "targets": {
                "worklist": {
                    "viewName": "Worklist",
                    "viewId": "worklist",
                    "viewLevel": 1
                },
                "post": {
                    "viewName": "Post",
                    "viewId": "post",
                    "viewLevel": 2
                }
            }
        }
    }
}
```

We have already used the `#/Posts/{postId}` hash in our tests and a view called the *Post* page, so we will now add a route and a target to the routing configuration of the descriptor with these patterns. It is simply defining a mandatory routing parameter `postId` that we fill with the ID from the model when navigating. The target configuration references a view called `Post` with a view level deeper than the home page. For more information, see the Navigation and Routing [page 291] tutorial.

webapp/view/Worklist.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.bulletinboard.controller.Worklist"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:core="sap.ui.core"
    xmlns:semantic="sap.m.semantic">
    <semantic:FullscreenPage
        id="page"
        title="{i18n>worklistViewTitle}">
        <semantic:content>
            <Table …>
                …
                <items>
                    <ColumnListItem
                        vAlign="Middle"
                        type="Navigation"
                        press=".onPress">
                        …
                    </ColumnListItem>
                </items>
            </Table>
        </semantic:content>
        …
    </semantic:FullscreenPage>
</mvc:View>
```

We configure the table items to be of type `Navigation`, so a user can trigger the navigation by choosing an item. When a `press` event is triggered, the `onPress` handler is called to navigate to the *Post* page.

webapp/controller/Worklist.controller.js

```
sap.ui.define([
    './BaseController',
    'sap/ui/model/json/JSONModel',
    '../model/formatter',
    '../model/FlaggedType',
    'sap/m/library'
], function(BaseController, JSONModel, formatter, FlaggedType, mobileLibrary) {
    "use strict";
    return
BaseController.extend("sap.ui.demo.bulletinboard.controller.Worklist", {
        …
        /* =========================================================== */
        /* event handlers                                              */
        /* =========================================================== */
        …
                /**
         * Event handler when a table item gets pressed
         * @param {sap.ui.base.Event} oEvent the table selectionChange event
         * @public
         */
        onPress: function (oEvent) {
            this.getRouter().navTo("post", {
                // The source is the list item that got pressed
                postId:
oEvent.getSource().getBindingContext().getProperty("PostID")
            });

        },
        …
    });
});
```

The `press` handler function instructs the `router` to navigate to the `post` pattern with the `PostID` from the binding context of the currently selected item. This fills the mandatory URL parameter, navigates to the `post` page, and updates the hash automatically.

webapp/view/Post.view.xml (New)

```
<mvc:View
        controllerName="sap.ui.demo.bulletinboard.controller.Post"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc"
        xmlns:semantic="sap.m.semantic">
    <semantic:FullscreenPage
            id="page"
            busy="{postView>/busy}"
            busyIndicatorDelay="0"
            navButtonPress=".onNavBack"
            showNavButton="true"
            title="{i18n>objectTitle}">
        <semantic:content>
            <ObjectHeader
                    id="objectHeader"
                    title="{Title}"
                    number="{
                        path: 'Price',
                        formatter: '.formatter.numberUnit'
                    }"
```

```
                    numberUnit="{Currency}"
                    backgroundDesign="Translucent">
            </ObjectHeader>
        </semantic:content>
    </semantic:FullscreenPage>
</mvc:View>
```

We provide a minimalistic detail page showing only some fields of the selected post for now. In the test we use the following information:

- Control with the ID `page` on this view
- `title` of the post we navigate to
- *Back* button to navigate back to the home page

webapp/controller/Post.controller.js (New)

```
sap.ui.define([
    './BaseController',
    'sap/ui/model/json/JSONModel',
    '../model/formatter'
], function (BaseController, JSONModel, formatter) {
    "use strict";
    return BaseController.extend("sap.ui.demo.bulletinboard.controller.Post", {
        formatter: formatter,
        /* =========================================================== */
        /* lifecycle methods                                           */
        /* =========================================================== */
        /**
         * Called when the worklist controller is instantiated.
         * @public
         */
        onInit: function () {
            // Model used to manipulate control states. The chosen values make
sure,
            // detail page is busy indication immediately so there is no break in
            // between the busy indication for loading the view's meta data
            var oViewModel = new JSONModel({
                    busy: false
                });

this.getRouter().getRoute("post").attachPatternMatched(this._onPostMatched,
this);
            this.setModel(oViewModel, "postView");
        },
        /* =========================================================== */
        /* event handlers                                              */
        /* =========================================================== */
        /**
         * Navigates back to the worklist
         * @function
         */
        onNavBack: function () {
            this.myNavBack("worklist");
        },
        /* =========================================================== */
        /* internal methods                                            */
        /* =========================================================== */
        /**
         * Binds the view to the post path.
         *
         * @function
```

```
         * @param {sap.ui.base.Event} oEvent pattern match event in route
'object'
         * @private
         */
        _onPostMatched: function (oEvent) {
            var oViewModel = this.getModel("postView"),
                oDataModel = this.getModel();
            this.getView().bindElement({
                path: "/Posts('" + oEvent.getParameter("arguments").postId +
"')",
                events: {
                    dataRequested: function () {
                        oDataModel.metadataLoaded().then(function () {
                            // Busy indicator on view should only be set if
metadata is loaded,
                            // otherwise there may be two busy indications next
to each other on the
                            // screen. This happens because route matched
handler already calls '_bindView'
                            // while metadata is loaded.
                            oViewModel.setProperty("/busy", true);
                        });
                    },
                    dataReceived: function () {
                        oViewModel.setProperty("/busy", false);
                    }
                }
            });
        }
    });
});
```

The controller of the `Post` page needs to take care of the data binding when a navigation event has happened. In the `init` function of the controller we define a local view model and attach to the `routing` event. When the `routing` event is triggered, we bind the view to the post with the specified ID.

Step 10: Test Suite and Automated Testing

In this step, we will step back from our tests and application features that we have implemented so far and add another important piece of test code: The test suite page. A test suite can execute multiple tests and collect the results. This comes in handy for automatic tools in a continuous integration process.

Preview

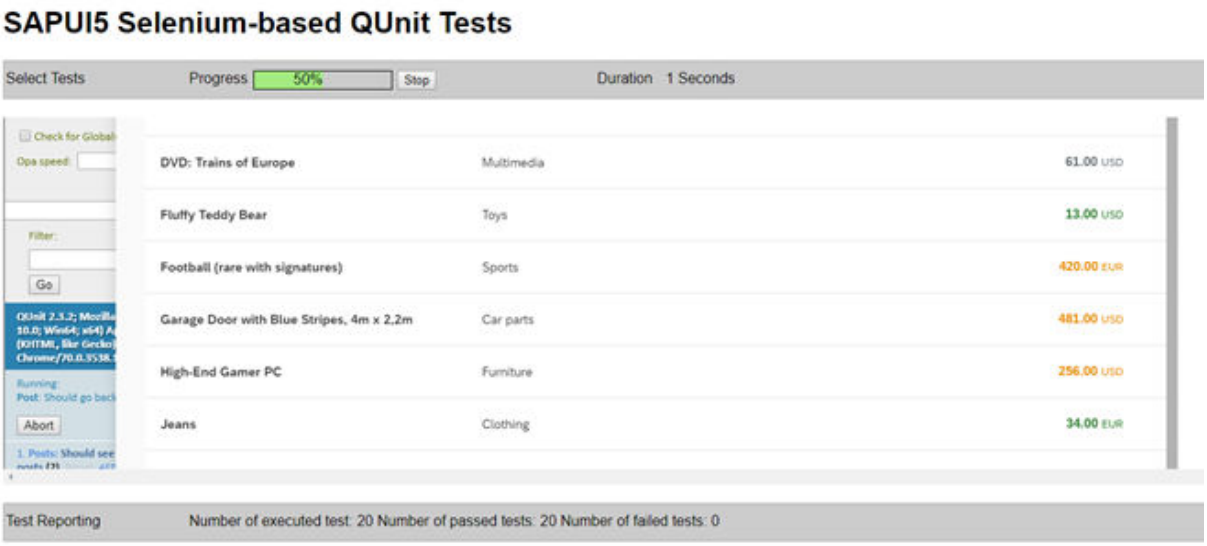


Figure 121: A Selenium runner for the test suite of the bulletin board

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 10.

webapp/test/testsuite.qunit.html (New)

```
<!DOCTYPE HTML>
<html>
<head>
    <title>QUnit test suite for Bulletin Board</title>
    <script src="resources/sap/ui/qunit/qunit-redirect.js"></script>
    <script src="testsuite.qunit.js" data-sap-ui-testsuite></script>
</head>
<body>
</body>
</html>
```

Create a new `testsuite.qunit.html` file. Here, you add the `testsuite.qunit.js` script, which we will define next, as a source.

webapp/test/testsuite.qunit.js (New)

```
window.suite = function() {
    "use strict";

    var oSuite = new parent.jsUnitTestSuite(),
        sContextPath = location.pathname.substring(0,
location.pathname.lastIndexOf("/") + 1);

    oSuite.addTestPage(sContextPath + "unit/unitTests.qunit.html");
    oSuite.addTestPage(sContextPath + "integration/opaTests.qunit.html");

    return oSuite;
};
```

This new `testsuite.qunit.js` file contains the logic for the QUnit tests. The coding is quite straightforward: We require the relevant QUnit files for redirecting to the central test suite and provide a configuration function `suite()` that is called automatically by the testrunner.

Inside this function, we add the QUnit pages for the app's unit and integration tests. For technical reasons, we have to provide an absolute path to the HTML pages so that the testrunner can execute them centrally. You can now run the `webapp/test/testsuite.qunit.html` file to check if all unit and integration tests are running fine with one URL.

> **i Note**
>
> A similar test suite can be configured as a pre-commit hook in local build environments or as a pre-submit hook in a continuous integration scenario on the central build server. Only when all tests run successfully, a new change is accepted and may be merged.
>
> Alternatively you can use a local test runner, such as Selenium or Karma, that automatically executes all tests whenever a file in the app project has been changed. All of these configurations run the tests and collect the resulting messages for further analysis. Therefore, it is very important to define meaningful test descriptions and success as well as error messages as you write your application tests.

Conventions

- Create a test suite app that triggers all your tests at once
- Run the test suite whenever you change the code of the app

Related Information

Karma Home Page
Selenium Home Page
Test Automation [page 1229]

Step 11: Testing User Input

In this step, we will write a test that simulates a user search. We will enter the search string into the search field and check if the correct results are shown in worklist table.

Preview

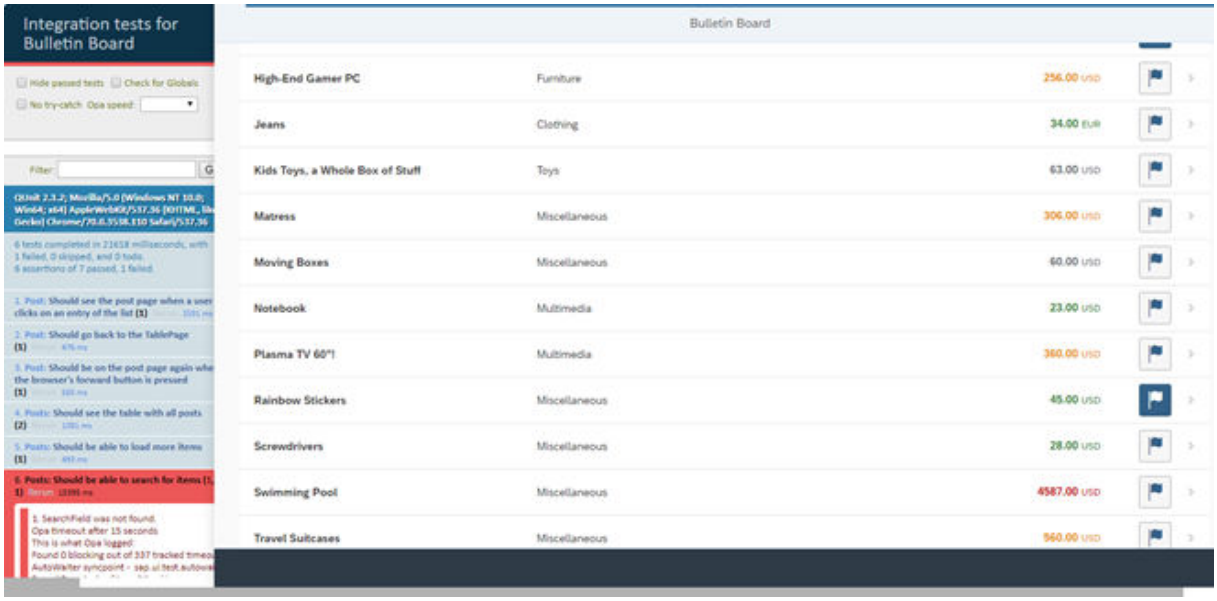


Figure 122: Testing user input in a search field

Coding

You can view and download all files in the Demo Kit at Testing - Step 11.

test/integration/WorklistJourney.js

```
sap.ui.define([
    "sap/ui/test/opaQunit",
    "./pages/Worklist"
], function (opaTest) {
    "use strict";
    QUnit.module("Posts");
    opaTest("Should see the table with all posts", function (Given, When, Then) {
        // Arrangements
        Given.iStartMyApp();
        // Assertions
        Then.onTheWorklistPage.theTableShouldHavePagination().
            and.theTitleShouldDisplayTheTotalAmountOfItems();
    });
    opaTest("Should be able to load more items", function (Given, When, Then) {
        //Actions
```

```
        When.onTheWorklistPage.iPressOnMoreData();
        // Assertions
        Then.onTheWorklistPage.theTableShouldHaveAllEntries();
    });
    opaTest("Should be able to search for items", function (Given, When, Then) {
        //Actions
        When.onTheWorklistPage.iSearchFor("Bear");

        // Assertions
        Then.onTheWorklistPage.theTableHasOneItem();

        // Cleanup
        Then.iTeardownMyApp();
    });
    }
);
```

In this example, we extend the `WorklistJourney.js` file with a new test `"Should be able to enter text into the search field"`. The action within this test simulates a user entering text into a search field, so we pass a search string `"Bear"` to this action. It is important to move the `Teardown` step to the last test, otherwise our app would be destroyed and the test would not be able to find the *Statistics* tab.

Delete `.and.iTeardownMyApp();` from the previous test in the file and add the new test case.

test/integration/pages/Worklist.js

```
sap.ui.require([
        'sap/ui/test/Opa5',
        'sap/ui/test/matchers/AggregationLengthEquals',
        'sap/ui/test/matchers/I18NText',
        'sap/ui/test/matchers/BindingPath',
        'sap/ui/demo/bulletinboard/test/integration/pages/Common',
        'sap/ui/test/actions/Press',
        'sap/ui/test/actions/EnterText'
    ],
    function (Opa5,
              AggregationLengthEquals,
              I18NText,
              BindingPath,
              Common,
              Press,
              EnterText) {
        "use strict";
        var sViewName = "Worklist",
            sTableId = "table";
        Opa5.createPageObjects({
            onTheWorklistPage: {
                baseClass: Common,
                actions: {
...
                    },

                    iSearchFor: function (sSearchString) {
                        return this.waitFor({
                            id: "searchField",
                            viewName: sViewName,
                            actions: new EnterText({
                                text: sSearchString
                            }),
                            errorMessage: "SearchField was not found."
                        });
                    }
```

```
                },
                assertions: {
                    theTableHasOneItem: function () {
                        return this.waitFor({
                            id: sTableId,
                            viewName: sViewName,
                            matchers: new AggregationLengthEquals({
                                name: "items",
                                length: 1
                            }),
                            success: function () {
                                Opa5.assert.ok(true, "The table contains one
corresponding entry");
                            },
                            errorMessage: "The table does not contain one item."
                        });
                    },
...
```

For the new test case we add an action `iEnterSearchStringIntoSearchField` and a new assertion `theTableShouldHaveCorrespondingEntries`.

In `iEnterSearchStringIntoSearchField`, we use the `EnterText` action and load the dependency `sap/ui/test/actions/EnterText`.

We define a `waitFor` statement with the current view and with the ID of our `SearchField`, which is stored as an internal variable. This is done in the same way as in the `iPressOnMoreData` action that we implemented in our first OPA test. But now we don't use the `EnterText` action. As soon as the `SearchField` is visible on the screen and can be interacted with, the `EnterText` action is invoked. If is is not invoked, an error message is displayed and the test fails.

The `assert` part is implemented in the same way as in our first OPA test. Again, we use the matchers to check the state. Here we check the number of items in the table resulting from the simulated search. According to our mock data, there should be only one item visible.

Conventions

Actions in OPA never contain a QUnit assertion.

Related Information

API Reference: `sap.ui.test.actions.EnterText`

Step 12: Adding a Search

We now add a search field to our bulletin board and define a filter that represents the search term. This is done similarly as in step 24 of the Walkthrough tutorial.

Preview

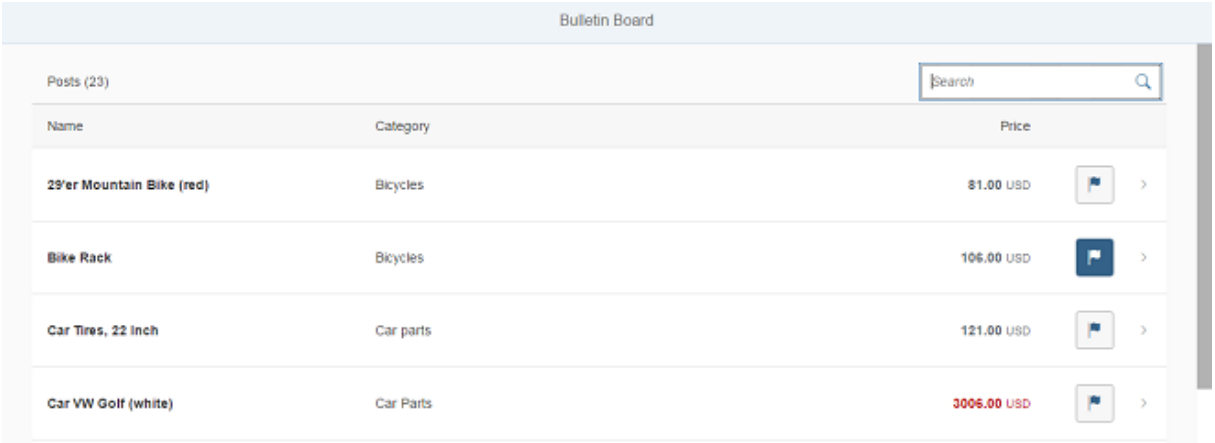


Figure 123: Search field

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 12.

webapp/view/Worklist.view.xml

```
...
         <Table
               id="table"
               width="auto"
               class="sapUiResponsiveMargin"
               growing="true"
               items="{
               path: '/Posts',
               sorter: {
                  path: 'Title',
                  descending: false
                  }
               }"
               busyIndicatorDelay="{worklistView>/tableBusyDelay}"
               updateFinished=".onUpdateFinished">
            <headerToolbar>
               <Toolbar>
                  <Label id="tableHeader" text="{worklistView>/
worklistTableTitle}"/>
```

```
                    <ToolbarSpacer />
                    <SearchField id="searchField" width="auto"
search=".onFilterPosts" />
                </Toolbar>
            </headerToolbar>
...
```

We add a `ToolbarSpacer` and a `SearchField` to the `headerToolbar` of our table.

webapp/controller/Worklist.controller.js

```
sap.ui.define([
    './BaseController',
    'sap/ui/model/json/JSONModel',
    '../model/formatter',
    '../model/FlaggedType',
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    'sap/m/library'
], function (BaseController, JSONModel, formatter, FlaggedType, Filter,
FilterOperator, mobileLibrary) {
    "use strict";
...
        onUpdateFinished: function (oEvent) {
            // update the worklist's object counter after the table update
            var sTitle,
                oTable = oEvent.getSource(),
                iTotalItems = oEvent.getParameter("total");
            // only update the counter if the length is final and
            // the table is not empty
            if (iTotalItems && oTable.getBinding("items").isLengthFinal()) {
                sTitle =
this.getResourceBundle().getText("worklistTableTitleCount", [iTotalItems]);
            } else {
                sTitle = this.getResourceBundle().getText("worklistTableTitle");
            }
            this.getModel("worklistView").setProperty("/worklistTableTitle",
sTitle);
        },
        onFilterPosts: function (oEvent) {

            // build filter array
            var aFilter = [];
            var sQuery = oEvent.getParameter("query");
            if (sQuery) {
                aFilter.push(new Filter("Title", FilterOperator.Contains,
sQuery));
            }

            // filter binding
            var oTable = this.byId("table");
            var oBinding = oTable.getBinding("items");
            oBinding.filter(aFilter);
        },
...
```

To enable filtering, we extend the controller with a method that applies the search term entered in the search field to the list binding, similarly as we did for `InvoiceList.controller.js` in step 24 of the Walkthrough tutorial.

Related Information

Step 42 of Walkthrough: Filtering [page 131]

Step 13: Testing User Interaction

In this step we want to write a test that simulates user interaction with an icon tab bar. We want to change the tab and check if the correct content is shown.

Preview

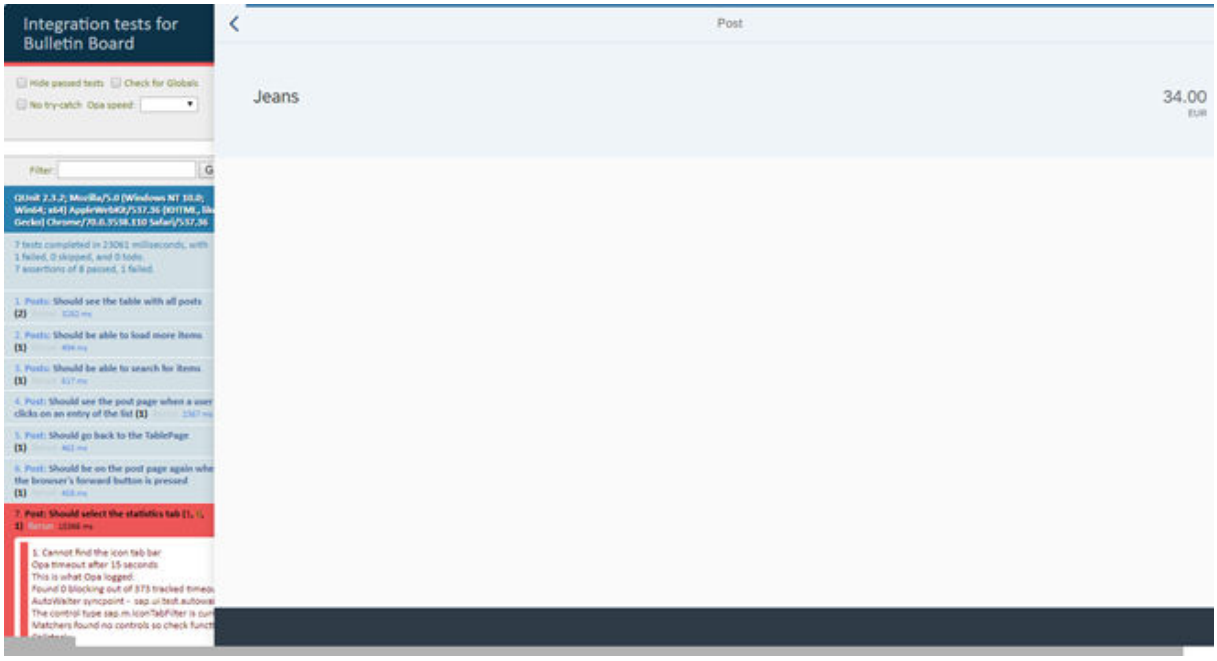


Figure 124: Test interacting with an icon tab bar

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 13.

test/integration/journeys/PostJourney.js

```
sap.ui.define([
    "sap/ui/test/opaQunit",
    "./pages/Worklist",
    "./pages/Browser",
```

```
    "./pages/Post"
], function (opaTest) {
    "use strict";
    …
    opaTest(…) {
        // Actions
            When.onTheBrowser.iPressOnTheForwardButton();

            // Assertions
            Then.onThePostPage.theTitleShouldDisplayTheName("Jeans");
        });
        opaTest("Should select the statistics tab", function (Given, When, Then)
{
            // Actions
            When.onThePostPage.iPressOnTheTabWithTheKey("statistics");
            // Assertions
            Then.onThePostPage.iShouldSeeTheViewCounter()
                .and.iTeardownMyApp();
        });
```

We extend the `PostJourney.js` file with a new test. It is important to move the `Teardown` to the last test, otherwise our app would be removed and the test would not be able to find the *Statistics* tab.

Delete `.and.iTeardownMyApp();` from the last test in the file and add the new test case

test/integration/pages/Post.js

```
sap.ui.define([
    'sap/ui/test/Opa5',
    'sap/ui/test/matchers/Properties',
    'sap/ui/test/actions/Press'
], function (Opa5, Properties, Press) {
    "use strict";

        var sViewName = "Post";

        Opa5.createPageObjects({
            onThePostPage: {
                actions: {
                    iPressTheBackButton: function () {
                        return this.waitFor({
                            id: "page",
                            viewName: sViewName,
                            actions: new Press(),
                            errorMessage: "Did not find the nav button on object
page"
                        });
                    },
                    iPressOnTheTabWithTheKey: function (sKey) {
                        return this.waitFor({
                            controlType: "sap.m.IconTabFilter",
                            viewName : sViewName,
                            matchers: new Properties({
                                key: sKey
                            }),
                            actions: new Press(),
                            errorMessage: "Cannot find the icon tab bar"
                        });

                    }
                },
                assertions: {
```

```
                    theTitleShouldDisplayTheName: function (sName) {
                        return this.waitFor({
                            id: "objectHeader",
                            viewName: sViewName,
                            matchers: new Properties({
                                title: sName
                            }),
                            success: function (oPage) {
                                Opa5.assert.ok(true, "was on the remembered
detail page");
                            },
                            errorMessage: "The Post " + sName + " is not shown"
                        });
                    },

                    iShouldSeeTheViewCounter: function () {
                        return this.waitFor({
                            id: "viewCounter",
                            viewName: sViewName,
                            success: function () {
                                Opa5.assert.ok(true, "The view counter was
visible");
                            },
                            errorMessage: "The view counter could not be found"
                        });
                    }
                }
            }
        });
    });
```

To change the selected tab, you set the selected key of the `sap.m.IconTabBar`. We have looked up the `setSelectedKey` function in the API documentation of the control, so we know that we can write a `waitFor` statement that makes use of it.

There is no uniform way of triggering user interactions with OPA. In most cases it is sufficient to use the API of the controls, e.g. setting a value. Note however, that calling the API methods of a control might not trigger the same events as when pressing the control.

In the `assert` part we add a new assertion for checking the visibility of a control with the ID `viewCounter`. Each `waitFor` statement checks if the control is rendered and visible unless you set `visible: false`. Therefore, we only use the `ok(true)` assertion in the success handler of the `waitFor` statement since QUnit expects at least one assertion for a test.

Conventions

- Actions in OPA never contain a QUnit assertion

Related Information

API Reference: `sap.m.IconTabBar`

API Overview and Samples: `sap.m.IconTabBar`

Step 14: Adding Tabs

We want to display statistics for posts, for example, how many times it was viewed. To achieve this, we implement an icon tab bar with an *Info* tab and a *Statistics* tab. The existing content should be placed on the *Info* tab and the view count on the *Statistics* tab.

Preview

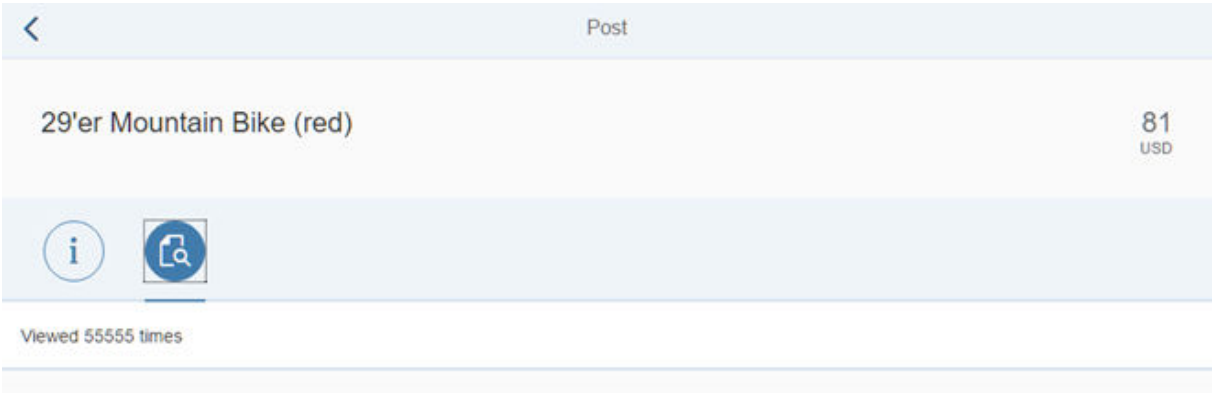


Figure 125: An icon tab bar with statistics

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 14.

view/Post.view.xml

```
<mvc:View
      controllerName="sap.ui.demo.bulletinboard.controller.Post"
      xmlns="sap.m"
      xmlns:form="sap.ui.layout.form"
      xmlns:mvc="sap.ui.core.mvc"
      xmlns:semantic="sap.m.semantic">
   <semantic:FullscreenPage
         id="page"
         busy="{postView>/busy}"
         busyIndicatorDelay="0"
         navButtonPress=".onNavBack"
         showNavButton="true"
         title="{i18n>objectTitle}">
      <semantic:content>
         <ObjectHeader
               id="objectHeader"
               title="{Title}"
               number="{
                  path: 'Price',
                  formatter: '.formatter.numberUnit'
               }"
```

```
                numberUnit="{Currency}"
                backgroundDesign="Translucent">
            </ObjectHeader>
            <IconTabBar id="iconTabBar"
                     expanded="{device>/isNoPhone}"
                     class="sapUiNoContentPadding">
               <items>
                  <IconTabFilter icon="sap-icon://hint" key="info" >
                     <form:SimpleForm>
                        <form:content>
                           <Label text="{i18n>postDateLabel}"/>
                           <Text text="{Timestamp}"/>
                           <Label text="{i18n>postDescriptionLabel}"/>
                           <Text text="{Description}"/>
                        </form:content>
                     </form:SimpleForm>
                  </IconTabFilter>
                  <IconTabFilter icon="sap-icon://inspection" key="statistics">
                     <Text text="Viewed 55555 times" id="viewCounter"/>
                  </IconTabFilter>
               </items>
            </IconTabBar>
         </semantic:content>
      </semantic:FullscreenPage>
</mvc:View>
```

We add a `sap.m.IconTabBar` with the two tabs `info` and `statistics`. The `statistics` tab we have already referred to in our test case.

Inside the first tab there is a `sap.ui.layout.form.SimpleForm` with a date and a description field that are mapped to the model data. In the second tab we place a new `Text` control.

In this very simple example, we just put a static text in the tab. In a real application, we would bind the value to the model.

webapp/i18n/i18n.properties

```
#~~~ Object View ~~~~~~~~~~~~~~~~~~~~~~~~~
#XTIT: Object view title
objectTitle=Post

#XTIT: Post view date label
postDateLabel=Posted At

#XTIT: Post view description label
postDescriptionLabel=Description
#~~~ Footer Options ~~~~~~~~~~~~~~~~~~~~~~
```

We add the missing texts to the `i18n.properties` file.

Step 15: Writing a Short Date Formatter Using TDD

It's now time to improve the content of the *Info* tab. We want to see the *Posted At* date in a formatted way. Based on the age of the post, we either display the time, a textural representation of the day, or the date only.

Preview

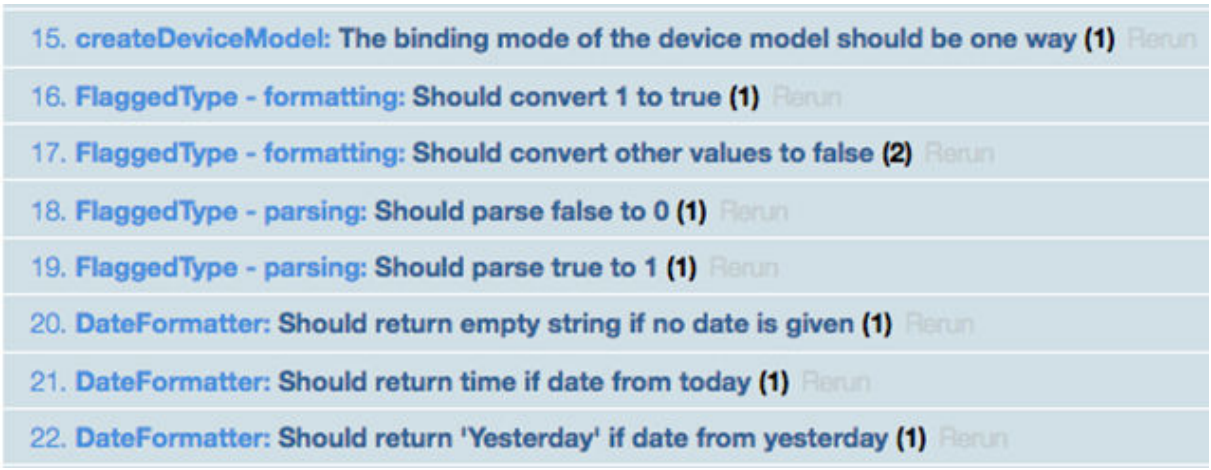


Figure 126: Unit tests of the formatter

Depending on the current date, we distinguish four different formatting categories, as shown in the table below:

Table 6: Formatting Categories

Category	Sample Input	Expected Output (for `en-US`)
Today	`2013/02/13 12:05:20`	`12:05 PM`
Yesterday	`2013/02/12 12:05:20`	`Yesterday`
Last 7 days	`2013/02/08 12:05:20`	`Friday`
Others	`2011/02/05 12:05:20`	`Dec 5, 2011`

As you can see, we have many different cases, and our formatter contains real logic.

We test this in a unit test. In this step we will follow an iterative approach. We first write a failing test and immediately fix it by adding the production code to make the test pass. Then the next iteration starts. We do not write more than one failing unit test at once.

> **i Note**
>
> There are many benefits of consequently applying the test-driven development (TDD) methodology, for example, very fast feedback, you can execute your tests after each change and get immediate feedback if the tests run green. You also spend less time debugging and for analysis. We recommend that you get familiar with TDD and clean code practices. In this step you get a first impression how TDD results in better

> separation of concerns, APIs, handling of dependencies, code reuse, and a test suite growing together with the code.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 15.

webapp/test/unit/AllTests.js

```
sap.ui.define([
    "./model/models",
    "./model/formatter",
    "./model/FlaggedType",
    "./model/DateFormatter"
], function() {
    "use strict";
});
```

First, we add the new test file we are about to create to the `AllTests.js` file.

webapp/model/DateFormatter.js (New)

```
sap.ui.define([
    "sap/ui/base/Object"
], function(Object) {
        return Object.extend("sap.ui.demo.bulletinboard.model.DateFormatter", {
    });
});
```

We create an empty hull for our formatter implementation first so that we can include it in our test. It does not contain any logic yet but simply extends an SAPUI5 base object.

webapp/test/unit/model/DateFormatter.js (New)

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter"
], function(DateFormatter) {
    QUnit.module("DateFormatter");
    QUnit.test("initial", function(assert) {
        assert.ok(new DateFormatter());
    });
});
```

And we create our test that checks if there is a `DateFormatter` object. Now we can execute our unit tests. We see that this test is failing as the object does not exist in our code yet.

webapp/test/unit/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter"
], function(DateFormatter) {
    QUnit.module("DateFormatter");
    QUnit.test("Should return empty string if no date is given",
function(assert) {
        var oFormatter = new DateFormatter();
        var sFormattedDate = oFormatter.format(null);
        assert.strictEqual(sFormattedDate, "");
    });
});
```

Now we implement a test for the API of the format function. We assume it will have a `Date` object as input parameter. In the first step, the test verifies that the format function returns an empty string if we pass `null`.

webapp/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/base/Object"
], function(Object) {
        return Object.extend("sap.ui.demo.bulletinboard.model.DateFormatter", {
            format: function() {
                return "";
            }
        });
    }
);
```

Now we fix our test again by returning the expected string.

Dependency Injection:

webapp/test/unit/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter",
    "sap/ui/core/Locale"
], function(DateFormatter, Locale) {
    QUnit.module("DateFormatter");
    QUnit.test("Should return empty string if no date is given",
function(assert) {
        var oFormatter = new DateFormatter({
            locale : new Locale("en-US")
        });
        var sFormattedDate = oFormatter.format(null);
        assert.strictEqual(sFormattedDate, "");
    });
    QUnit.test("Should return time if date from today", function(assert) {
        var oFormatter = new DateFormatter({
            locale : new Locale("en-US")
```

```
        });
        var oDate = new Date(2015, 2, 14, 12, 5, 0, 0);
        var sFormattedDate = oFormatter.format(oDate);
        assert.strictEqual(sFormattedDate, "12:05 PM");
    });
});
```

Here our test expects that the date is displayed as time when the post is from today. If we rely on the browser language the test would be fragile. It will fail in some languages. To avoid this, we pass the locale settings to the formatter's constructor. The test will use a fixed locale `en-US` in order to remain stable. This mechanism is called **Dependency Injection**.

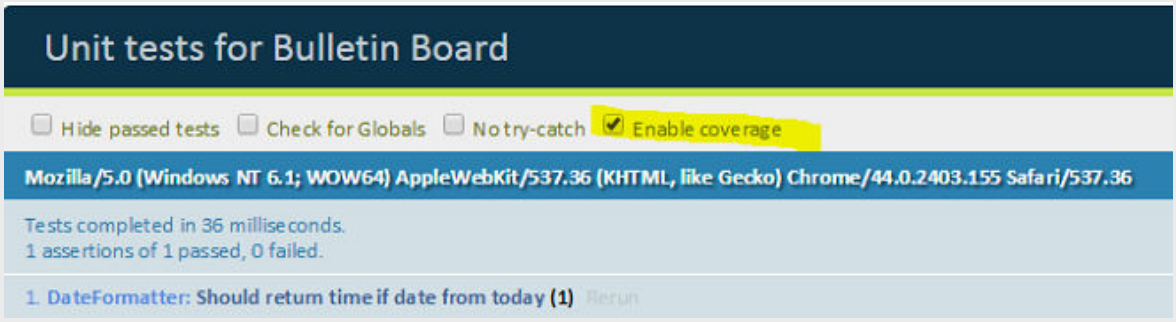
webapp/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/base/Object",
    "sap/ui/core/format/DateFormat"
], function(Object, DateFormat) {
    return Object.extend("sap.ui.demo.bulletinboard.model.DateFormatter", {
        constructor: function(oProperties) {
            this.timeFormat = DateFormat.getTimeInstance({
                style: "short"
            }, oProperties.locale);
            },
        format: function(oDate) {
            if (!oDate) {
                return "";
            }
                return this.timeFormat.format(oDate);
            }
        });
    }
);
```

In the implementation we use the `DateFormat` of SAPUI5 to create a short date. The locale is passed on to the `getTimeInstance` function.

i Note

The implementation should not do more than the current tests covers. This makes sure you cover all the code paths. You can enable the code coverage by selecting the *Enable coverage* checkbox.



It will show the lines covered by your tests (white) and the ones that were not covered (red). For the single test above the coverage looks like this. The red line is already covered by the previous test so in total we have a test coverage of 100%.

Refactoring:

webapp/test/unit/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter",
    "sap/ui/core/Locale"
], function(DateFormatter, Locale) {
    var oFormatter = null;
    QUnit.module("DateFormatter", {
        beforeEach: function() {
            oFormatter = new DateFormatter({
                locale: new Locale("en-US")
            });
        }
    });
    QUnit.test("Should return empty string if no date is given",
function(assert) {
        /*Delete in your code: var oFormatter = new DateFormatter();
        ...
    });
    QUnit.test("Should return time if date from today", function(assert) {
        /*Delete in your code: var oFormatter = new DateFormatter({
        /*Delete in your code:     locale: new Locale("en-US")
        /*Delete in your code: });
        ...
    });
});
```

Our tests are running so we can start refactoring our code. Since we need the `DateFormatter` object in every test case we will move it to the QUnit module's `beforeEach` function. As the name suggests, the function is invoked before each test so we may use it to save some code we need in every test.

Dependency Injection to Get Independent from System Time:

webapp/test/unit/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter"
    "sap/ui/core/Locale"
], function(DateFormatter, Locale) {
    var oFormatter = null;
    QUnit.module("DateFormatter", {
        beforeEach: function() {
            oFormatter = new DateFormatter({
                now : function() {
                    return new Date(2015, 2, 14, 14, 0, 0, 0).getTime();
                },
                locale : new Locale("en-US")
            });
        }
    });
    ...
    QUnit.test("Should return 'Yesterday' if date from yesterday",
function(assert) {
        var oDate = new Date(2015, 2, 13);
```

```
        var sFormattedDate = oFormatter.format(oDate);
        assert.strictEqual(sFormattedDate, "Yesterday");
    });
});
```

The next test verifies that `Yesterday` is returned for yesterday's date. To keep the test independent of the system time, we pass on a stable date to the formatter.

webapp/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/base/Object",
    "sap/ui/core/format/DateFormat"
], function(Object, DateFormat) {
    return Object.extend("sap.ui.demo.bulletinboard.model.DateFormatter", {
        constructor : function(oProperties) {
            this.timeFormat = DateFormat.getTimeInstance({
                style : "short"
            }, oProperties.locale);
            this.now = oProperties.now;
        },
        format : function(oDate) {
            if (!oDate) {
                return "";
            }
            var iElapsedDays = this._getElapsedDays(oDate);
            if (iElapsedDays === 0) {
                return this.timeFormat.format(oDate);
            } else if (iElapsedDays === 1) {
                return "Yesterday";
            }
            return this.dateFormat.format(oDate);
        },
        _getElapsedDays : function(oDate) {
            var iElapsedMilliseconds = this.now() - oDate.getTime();
            var fElapsedDays = iElapsedMilliseconds / 1000 / 60 / 60 / 24;
            return Math.floor(fElapsedDays);
        }
    });
});
```

In the implementation we add a calculation for determining how many days passed. If zero days passed, the format function is called, and if one day passed `Yesterday` is returned. Currently we skip reading "Yesterday" from the i18n model to keep the example simple.

Boundary Testing:

webapp/test/unit/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter",
    "sap/ui/core/Locale"
], function(DateFormatter, Locale) {
    var oFormatter = null;
```

```
    ...
    QUnit.test("Should return day of the week if date < 7 days ago",
function(assert) {
        var oDate = new Date(2015, 2, 8);
        var sFormattedDate = oFormatter.format(oDate);
        assert.strictEqual(sFormattedDate, "Sunday");
    });
});
```

The next test verifies that the day of the week is returned. As test input we take a value at the boundary: Sunday is one day before a different formatting pattern should be applied.

webapp/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/base/Object",
    "sap/ui/core/format/DateFormat"
], function(Object, DateFormat) {
    return Object.extend("sap.ui.demo.bulletinboard.model.DateFormatter", {
        constructor: function(oProperties) {
            this.timeFormat = DateFormat.getTimeInstance({
                style: "short"
            }, oProperties.locale);
            this.weekdayFormat = DateFormat.getDateInstance({
                pattern: "EEEE"
            }, oProperties.locale);
            this.now = oProperties.now;
        },
        format: function(oDate) {
            if (!oDate) {
                return "";
            }
            var iElapsedDays = this._getElapsedDays(oDate);
            if (iElapsedDays === 0) {
                return this.timeFormat.format(oDate);
            } else if (iElapsedDays === 1) {
                return "Yesterday";
            } else if (iElapsedDays < 7) {
                return this.weekdayFormat.format(oDate);
            }
        }
    …
```

Now we define a new format in our constructor, the `weekdayFormat`. In the format function we apply the format if the elapsed days are smaller than 7.

webapp/test/unit/model/DateFormatter.js

```
sap.ui.define([
    "sap/ui/demo/bulletinboard/model/DateFormatter",
    "sap/ui/core/Locale"
], function(DateFormatter, Locale) {
    var oFormatter = null;
    ...
    QUnit.test("Should return date w/o time if date > 7 days ago",
function(assert) {
```

```
        var oDate = new Date(2015, 2, 7);
        var sFormattedDate = oFormatter.format(oDate);
        assert.strictEqual(sFormattedDate, "Mar 7, 2015");
    });
});
```

In the next test we verify that the date is formatted as date without time. Again, we take a value at the boundary.

webapp/model/DateFormatter.js

```
…
        constructor: function(oProperties) {
            this.timeFormat = DateFormat.getTimeInstance({
                style : "short"
            }, oProperties.locale);
            this.weekdayFormat = DateFormat.getDateInstance({
                pattern : "EEEE"
            }, oProperties.locale);
            this.dateFormat = DateFormat.getDateInstance({
                style : "medium"
            }, oProperties.locale);
            this.now = oProperties.now;
        },
        format: function(oDate) {
            if (!oDate) {
                return "";
            }
            var iElapsedDays = this._getElapsedDays(oDate);
            if (iElapsedDays === 0) {
                return this.timeFormat.format(oDate);
            } else if (iElapsedDays === 1) {
                return "Yesterday";
            } else if (iElapsedDays < 7) {
                return this.weekdayFormat.format(oDate);
            } else {
                return this.dateFormat.format(oDate);
            }
        },
    …
```

In the implementation, we use a different `style` property for instantiating the `dateFormat` property. We call the format of this instance for dates that are more than 6 days in the past.

Although our formatter depends on system time and locale settings, our tests are very easy to read and maintain. We wrote blackbox tests, providing only the input and expecting a certain output without knowing the implementation details. The `DateFormatter` does not actively resolve the dependencies to the system time and locale settings. Instead, it asks its creator to pass the dependencies along in the constructor. In the next step, we have to bring the pieces together.

Step 16: Adding the Date Formatter

Our formatter does its job, but it is not yet used. In this step we will use it.

Preview

Figure 127: Date formatter in action

Coding

You can view and download all files in the *Samples* in the Demo Kit at Testing - Step 16.

webapp/view/Post.view.xml

```
…
<IconTabBar id="iconTabBar"
            expanded="{device>/isNoPhone}"
            class="sapUiResponsiveContentPadding">
    <items>
        <IconTabFilter icon="sap-icon://hint" key="info">
            <form:SimpleForm>
                <form:content>
                    <Label text="{i18n>postDateLabel}"/>
                    <Text text="{
                        path: 'Timestamp',
                        formatter: '.formatter.date'
                    }"/>
                    <Label text="{i18n>postDescriptionLabel}"/>
                    <Text text="{Description}"/>
                </form:content>
            </form:SimpleForm>
        </IconTabFilter>
        …
    </items>
</IconTabBar>
…
```

On the *Info* tab we bind the date field to a format method `.formatter.date` of the controller of the view. The leading `.` indicates that the function is defined on the controller instance.

webapp/model/formatter.js

```
sap.ui.define([ "sap/ui/demo/bulletinboard/model/DateFormatter" ], function
(DateFormatter) {
    ...
    return {
        ...
        numberUnit: function(sValue) {
            ...
        },
        date: function(date) {
            return new DateFormatter({now: Date.now}).format(date);
        }
    };
});
```

In the `formatter.js file`, create an instance of the previously implemented `DateFormatter` and provide the necessary dependencies.

Now run the app again to see that the formatter is applied on the post date of the detail page.

> **i** Note
>
> The files that create objects with dependencies should be kept simple. They do not have multiple code paths caused by if-else statements or loops. To test these components, just a few simple integration tests, or merely smoke tests, are sufficient. We already know that the `DateFormatter` does the job right for all the different cases.

Summary

You should now be familiar with the major development paradigms and concepts of SAPUI5 and have created a very simple first app. You are now ready to build a proper app based on what you've learned.

Mock Server

In this tutorial, we will explore some advanced features of the mock server.

If no OData service is available or you simply don't want to depend on the OData backend connectivity for your development and tests, the mock server can mimic the OData back-end calls. It is designed to simulate an OData provider by intercepting the HTTP communication made to the server, and providing a fake output. All this is transparent to the data binding and usage of OData model.

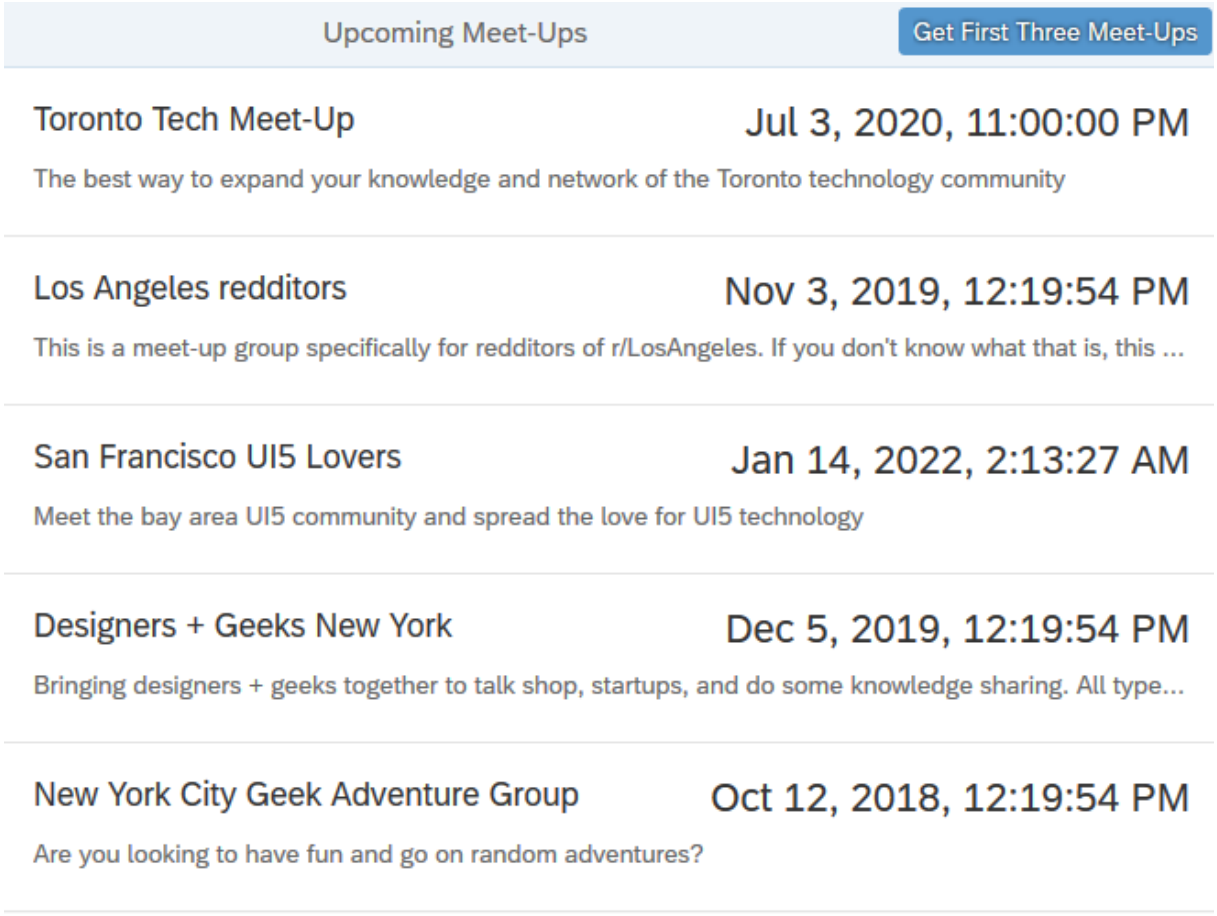
In certain scenarios, using only the built-in OData simulation of the mock server is insufficient for completely server-independent tests. For example, if your application is using an OData feature that is not supported by

the mock server, or if your application invokes a function import that depends on server specific implementation (and thus is also not simulated generically). We will demonstrate how to use function callbacks in order to change existing mock requests.

Additionally, we will demonstrate how to mock an additional request that is not simulated out of the box by the SAPUI5 mock server.

> ⚠ Caution
>
> The tutorial describes how to use some advanced features of the mock server, disregarding the legal aspects of shipping mock data. Usually the mock data and mock server invocation is done in a test folder that is not shipped to customers. Be very careful that you don't ship mock data!

Preview



> → Tip
>
> You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, and start there.

You can view and download the files for all steps in the Demo Kit at Mock Server. Copy the code to your workspace and make sure that the application runs by calling the `webapp/index.html` file. Depending on your development environment you might have to adjust resource paths and configuration entries.

For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Prerequisites

This tutorial assumes you have access to the SAP Web IDE either by having a trial account or a customer account. For more information, see App Development Using SAP Web IDE [page 44].

You should also be familiar with the concepts explained in the Walkthrough [page 69] tutorial and with the OData specification.

Related Information

Mock Server [page 1222]

Step 1: Initial App Without Data

We start with a simple app scenario with a list of items bound to an OData service. Since the OData service is not available yet on a real server, we will use the mock server to simulate both data and data calls.

For this very simple tutorial app we will use an OData service called `NerdMeetups` that lists meet-up groups according to location, date, topic, etc. The app will display a simple list populated by a function import call to display only upcoming meet-ups (meet-ups with an event date greater to the current date).

Additionally, a button will fetch the first three meet-ups (using a custom URL parameter called `first`). This exercise simply shows an app with no data retrieved from the back end. This can happen when the back end is down, or when the service is not implemented yet.

Usually you start the development of an app with local mock data first. This way you can continue implementing the application logic without depending on the back end readiness and connectivity.

Preview

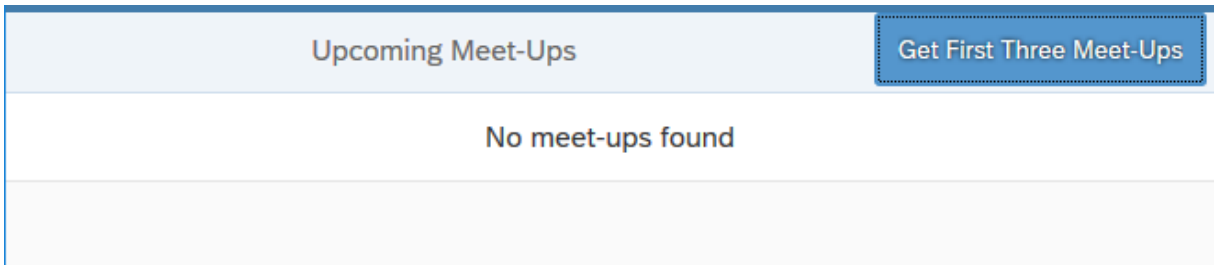


Figure 128: The initial app

Coding

To set up your project for this tutorial, download the files for *Step 1* in the Demo Kit at Mock Server - Step 1. Copy the code to your workspace and make sure that the application runs by calling the `webapp/test/mockServer.html` file.

Depending on your development environment you might have to adjust resource paths and configuration entries. The project structure and the files coming with this tutorial are explained in detail in the Walkthrough [page 69] tutorial.

You should have the same files as displayed in the following figure:

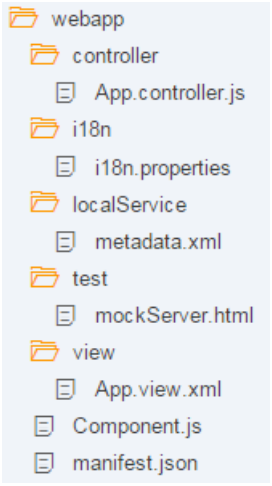


Figure 129: Folder structure with downloaded files

Step 2: Creating a Mock Server to Simulate Data

In this step, we use the mock server to add data to our app without dependency to any remote server or system.

Preview

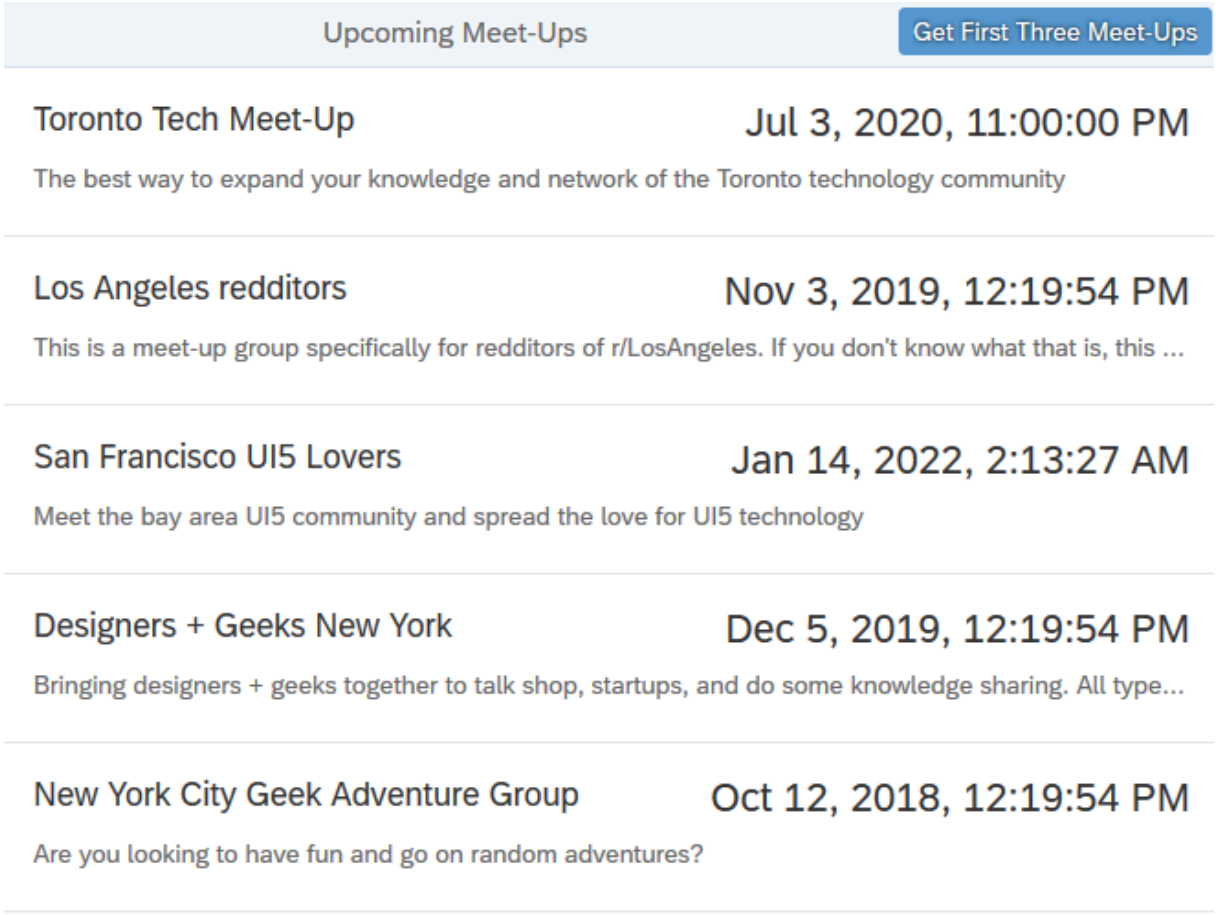


Figure 130: The app now contains data

Coding

You can view and download all files in the Demo Kit at Mock Server - Step 2.

webapp/test/mockServer.html

```
<!DOCTYPE HTML>
<html>
```

```
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Mock Server Tutorial</title>
    <script id="sap-ui-bootstrap"
        src="resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.MockServer": "../"
        }'
        data-sap-ui-oninit="module:sap/ui/demo/MockServer/test/initMockServer"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true">
    </script>
</head>
<body class="sapUiBody">
    <div data-sap-ui-component data-name="sap.ui.demo.MockServer" data-
id="container" data-settings='{"id" : "MockServer"}'></div>
</body>
</html>
```

We use this file to run our app in test mode with mock data. The new artifact `initMockServer` performs the required set up steps before the application component is instantiated. By doing so, we can catch all requests that would go to the real service and process it locally with our mock server when the app is launched with the `webapp/test/mockServer.html`.

> **i Note**
>
> A productive application does not contain the mock server code and thus connects to the real service instead. The HTML page above is defined only for local testing and to be called in automated tests. The application coding itself is unchanged and does not know the difference between the real and the mocked back-end service.
>
> The mock server does not need to be called from anywhere else in our code so we use `sap.ui.require` to load dependencies asynchronously without defining a global namespace.

webapp/test/initMockServer.js

```
sap.ui.define([
    "sap/ui/demo/MockServer/localService/mockserver"
], function (mockserver) {
    "use strict";
    // initialize the mock server
    mockserver.init();
    // initialize the embedded component on the HTML page
    sap.ui.require(["sap/ui/core/ComponentSupport"]);
});
```

We load a dependency to a file called `mockserver.js` that is located in the `webapp/localService` folder. This file contains our local mock server. It is immediately called with the `init` method before we initialize the application component.

webapp/localService/metadata.xml

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
    <edmx:DataServices
        xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
m:DataServiceVersion="1.0">
        <Schema Namespace="NerdMeetup.Models"
            xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
            xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/
metadata"
            xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
            <EntityType Name="Meetup">
                <Key>
                    <PropertyRef Name="MeetupID" />
                </Key>
                <Property Name="MeetupID" Type="Edm.Int32" Nullable="false" />
                <Property Name="Title" Type="Edm.String" Nullable="true" />
                <Property Name="EventDate" Type="Edm.DateTime"
Nullable="false" />
                <Property Name="Description" Type="Edm.String" Nullable="true" />
                <Property Name="HostedBy" Type="Edm.String" Nullable="true" />
                <Property Name="ContactPhone" Type="Edm.String"
Nullable="true" />
                <Property Name="Address" Type="Edm.String" Nullable="true" />
                <Property Name="Country" Type="Edm.String" Nullable="true" />
                <Property Name="Latitude" Type="Edm.Double" Nullable="false" />
                <Property Name="Longitude" Type="Edm.Double" Nullable="false" />
                <Property Name="HostedById" Type="Edm.String" Nullable="true" />
                <Property Name="Location"
Type="NerdMeetup.Models.LocationDetail" Nullable="false" />
            </EntityType>
            <ComplexType Name="LocationDetail" />
            <EntityContainer Name="NerdMeetups"
m:IsDefaultEntityContainer="true">
                <EntitySet Name="Meetups"
EntityType="NerdMeetup.Models.Meetup" />
                <FunctionImport Name="FindUpcomingMeetups" EntitySet="Meetups"
ReturnType="Collection(NerdMeetup.Models.Meetup)" m:HttpMethod="GET" />
            </EntityContainer>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

The `metadata` file contains information about the service interface and does not need to be written manually. It defines a `Meetup` entity, a `Meetups` entity set and a function import definition.

webapp/localService/mockdata/Meetups.json (New)

```
[{
    "MeetupID": 1,
    "Title": "Toronto Tech Meet-Up",
    "EventDate": "/Date(1593810000000)/",
    "Description": "The best way to expand your knowledge and network of the
Toronto technology community"
},
{
    "MeetupID": 2,
    "Title": "Los Angeles redditors",
```

```
    "EventDate": "/Date(1572779994000)/",
    "Description": "This is a meet-up group specifically for redditors of r/
LosAngeles. If you don't know what that is, this isn't the meet-up you're
looking for"
}, {
    "MeetupID": 3,
    "Title": "San Francisco UI5 Lovers",
    "EventDate": "/Date(1642122807784)/",
    "Description": "Meet the bay area UI5 community and spread the love for UI5
technology"
}, {
    "MeetupID": 4,
    "Title": "Designers + Geeks New York",
    "EventDate": "/Date(1575544794000)/",
    "Description": "Bringing designers + geeks together to talk shop, startups,
and do some knowledge sharing. All types of designers + geeks welcome"
}, {
    "MeetupID": 5,
    "Title": "New York City Geek Adventure Group",
    "EventDate": "/Date(1539339594000)/",
    "Description": "Are you looking to have fun and go on random adventures?"
}]
```

The `Meetups.json` file is automatically read by the mock server later in this step. It represents a flat array of `Meetup` items.

webapp/localService/mockserver.js (New)

```
sap.ui.define([
    "sap/ui/core/util/MockServer",
    "sap/base/Log"
], function(MockServer, Log) {
    "use strict";

    return {
        /**
         * Initializes the mock server.
         * You can configure the delay with the URL parameter "serverDelay".
         * The local mock data in this folder is returned instead of the real
data for testing.
         * @public
         */
        init: function() {
            // create
            var oMockServer = new MockServer({
                rootUri: "/"
            });

            // simulate against the metadata and mock data
            oMockServer.simulate("../localService/metadata.xml", {
                sMockdataBaseUrl: "../localService/mockdata",
                bGenerateMissingMockData: true
            });

            // start
            oMockServer.start();

            Log.info("Running the app with mock data");
        }

    };

});
```

Now we can write the code to initialize the mock server that will simulate the requests instead of the real server. We load the `MockServer` module as a dependency and create a helper object that defines an `init` method to start the server. This method is called before the Component initialization in the `mockServer.html` file above. The `init` method creates a `MockServer` instance with the same URL as the real service. The URL in configuration parameter `rootURI` is now served by our test server instead of the real service.

Next, we set two global configuration settings for all `MockServer` instances that tell the server to respond automatically and introduce a delay of one second to imitate a typical server response time.

In order to simulate a manual back-end call we can simply call the `simulate` method on the `MockServer` instance with the path to our newly created `metadata.xml` file. This will read the test data from our local file system and set up the URL patterns that will mimic the real service. The first parameter is the path to the service `metadata.xml` document. The second parameter is an object with the following properties:

- `sMockdataBaseUrl`: path where to look for mock data files in JSON format
- `bGenerateMissingMockData`: Boolean property to tell the `MockServer` to use auto-generated mock data in case no JSON files are found.

We call the function `start` on the mock server instance. From this point on, each request matching the URL pattern `rootURI` will be processed by the `MockServer`.

Finally, we add a message toast to indicate for the user that the app runs with mock data.

This approach is perfect for local and automated testing, even without any network connection. Your development does not depend on the availability of a remote server, i.e. to run your tests independently from the back-end service. You can control the mocked data so the requests will return reliable and predictable results.

If the real service connection cannot be established, for example, when there is no network connection, you can always fall back to the local test page and run the app with mock data.

Just run the app now again with the `mockServer.html` file.. The list should now be populated with meet-ups from our mock data. You can also choose the button and see data.

Related Information

Mock Server [page 1222]
API Reference: `sap.ui.core.util.MockServer`

Step 3: Handling Custom URL Parameters

In this step, we add the functionality to interpret URL parameters in our local mock server configuration.

We know that the OData provider of this service implements a URL parameter that returns only the first three entries of a set. So, for example, calling the URL with parameter `/Meetups?first=3` should return only the first 3 meet-up entries instead of all available entries.

Preview

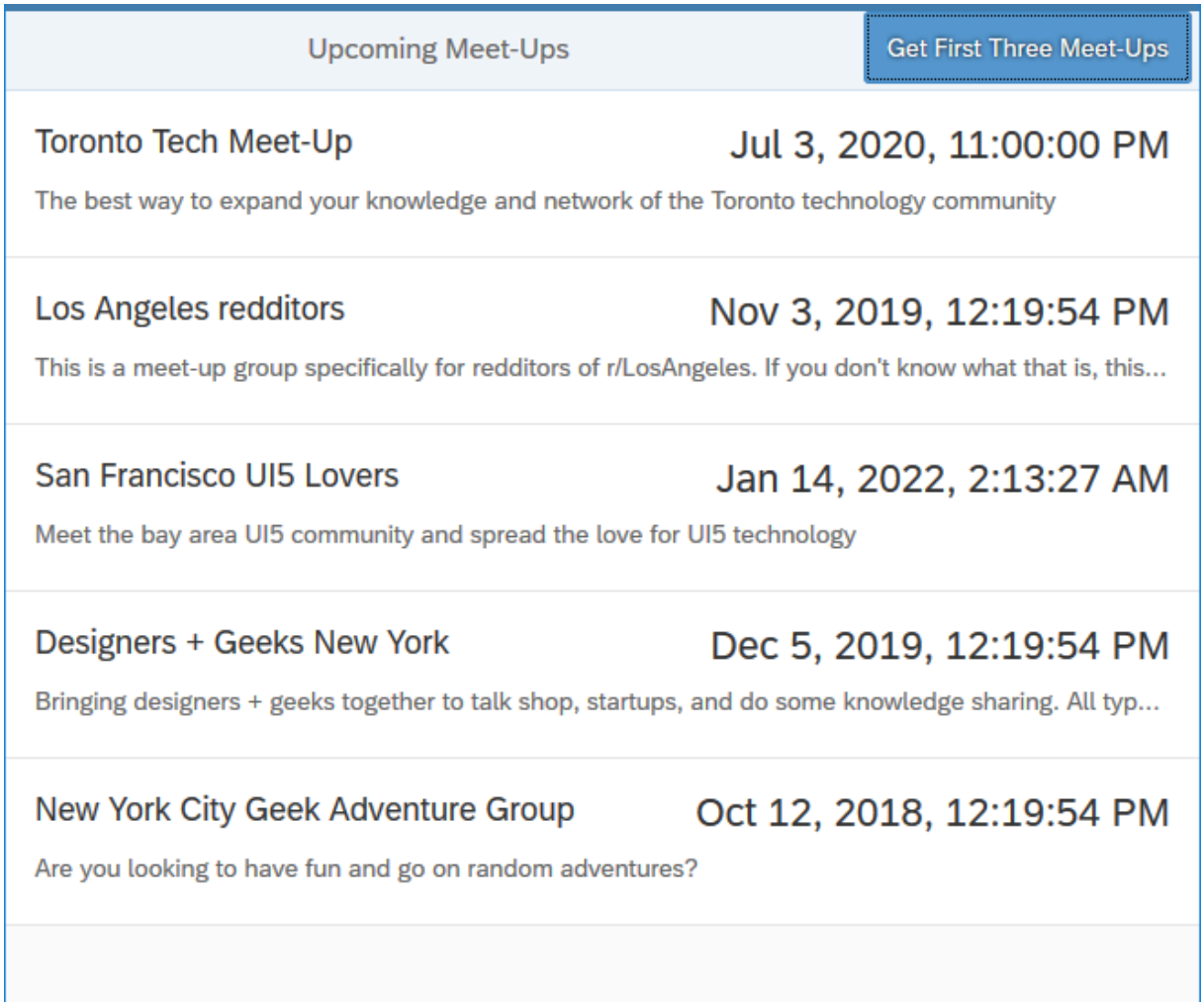


Figure 131: Only the next three meet-ups are shown

Coding

You can view and download all files in the Demo Kit at Mock Server - Step 3.

webapp/localService/mockserver.js

```
sap.ui.define([
    "sap/ui/core/util/MockServer",
    "sap/base/Log"
], function(MockServer, Log) {
    "use strict";
    return {
        /**
         * Initializes the mock server.
         * You can configure the delay with the URL parameter "serverDelay".
```

```
         * The local mock data in this folder is returned instead of the real
data for testing.
         * @public
         */
        init: function() {
            // create
            var oMockServer = new MockServer({
                rootUri: "/"
            });
            oMockServer.simulate("../localService/metadata.xml", {
                sMockdataBaseUrl: "../localService/mockdata",
                bGenerateMissingMockData: true
            });
            // handling custom URL parameter step
            var fnCustom = function(oEvent) {
                var oXhr = oEvent.getParameter("oXhr");
                if (oXhr && oXhr.url.indexOf("first") > -1) {
                    oEvent.getParameter("oFilteredData").results.splice(3, 100);
                }
            };
            oMockServer.attachAfter("GET", fnCustom, "Meetups");
            // start
            oMockServer.start();
            Log.info("Running the app with mock data");
        }
    };
});
```

In some scenarios, a server-specific implementation is used to calculate the returned data. For example, you can use a custom URL parameter that is typically interpreted by the server. The mock server ignores it, thus still returning the entire set of meet-ups.

In this tutorial, we use the URL parameter `first=3` to fetch the first three entries. So, for example, calling to `/Meetups?first=3` should return at most three meet-up entries.

However, since this is a custom parameter that is not part of the standard official OData query options, it will not get processed correctly by the mock server. Moreover, the mock server simply ignores it and return the entire set of meet-ups.

We now enable the functionality when running in mock mode. As its functionality corresponds to the OData `$top` system query, we simply evaluate it to `$top` at runtime.

First, we create a callback function that we later attach to every `GET` request made to the `Meetups` entity set of the service. Note that we choose the `attachAfter` event that is fired after the built-in request processing of the mock server. The event contains the actual `XHR` object and the mock data to be returned to the application. Inside the callback function we remove all results starting from third entry: The `oFilteredData` parameter comes with the event `attachAfter` and contains the mock data entries that are about to be returned in the response.

Second, we attach the callback to every `GET` request to the specific `Meetups` entity set.

Step 4: Calling a Function Import

We only want to display the upcoming meetings and hide the meetings happened in the past in our app. By using a function import that calculates these items on the back end we do not need to do the calculation on the client. The mock server will be instructed to do the calculation locally for testing purposes.

Preview

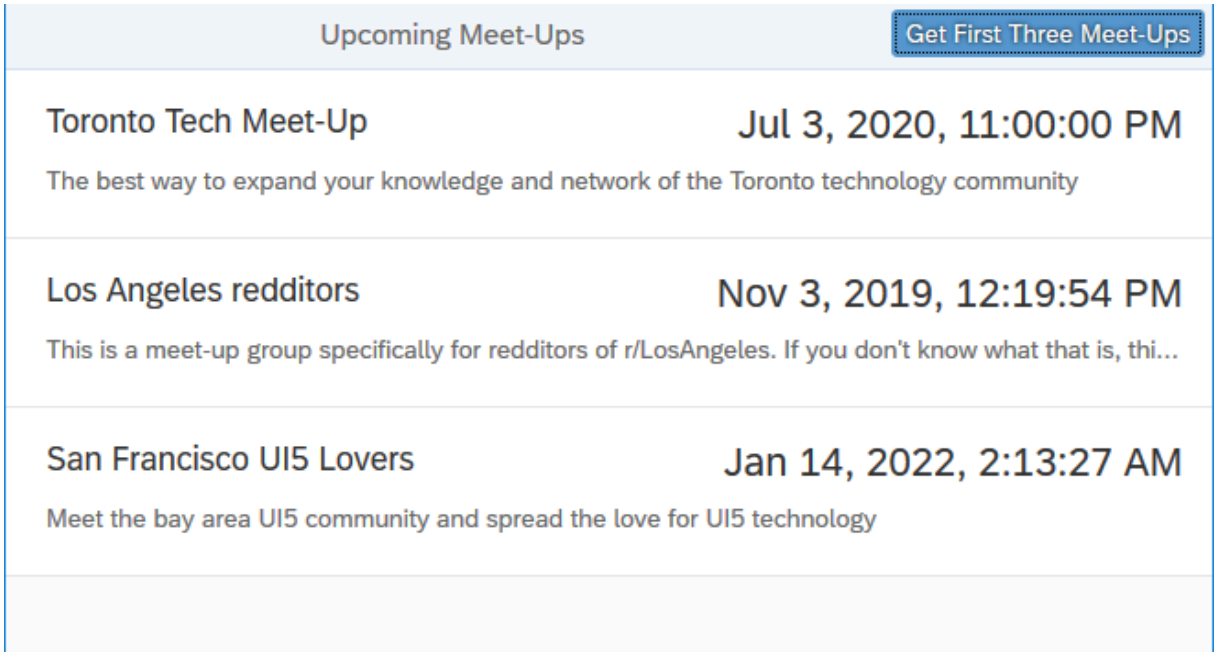


Figure 132: Only the upcoming meet-ups are shown

Coding

You can view and download all files in the Demo Kit at Mock Server - Step 4.

webapp/localService/metadata.xml

```
...
      <EntityContainer Name="NerdMeetups" m:IsDefaultEntityContainer="true">
        <EntitySet Name="Meetups" EntityType="NerdMeetup.Models.Meetup" />
        <FunctionImport Name="FindUpcomingMeetups" EntitySet="Meetups"
ReturnType="Collection(NerdMeetup.Models.Meetup)" m:HttpMethod="GET" />
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

The function import we are going to use is declared in the `metadata.xml` file.

webapp/view/App.view.xml

```
...
  //Delete items="{/Meetups}"
    <List id="list" items="{/FindUpcomingMeetups}"
noDataText="{i18n>noDataText}">
...
```

We change the binding of the list to a function import call that returns only upcoming meet-ups, instead of the call to the entire meet-ups collection.

After saving and running the app again, we should get the following result:

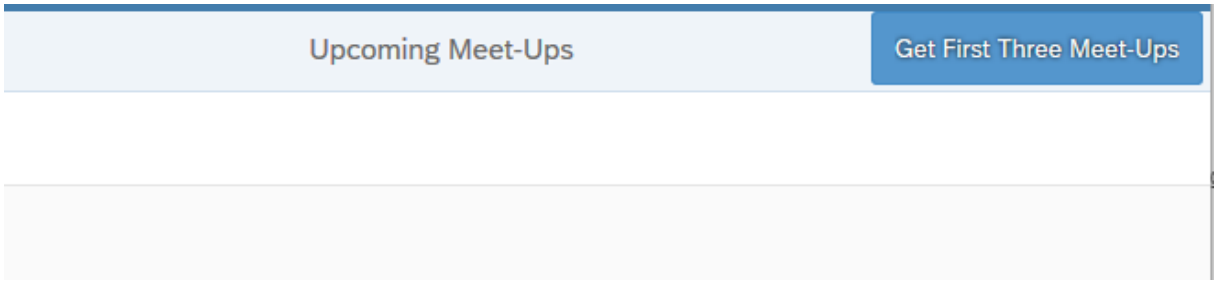


Figure 133: No data visible

Since the function import call is not simulated automatically by the mock server, we do not see any data in list, and a failed network call is issued in the developer tools of the browser.

> → Tip
>
> In Google Chrome, mocked requests will appear in a debug level log of the console (both request and response) and not on the *Network* tab. If you do see them in the *Network* tab, they are not mocked and you need to check your code.

In order to simulate the function import call, we write our own (mocked) implementation, and add to the internal list of requests.

webapp/localService/mockserver.js

```
sap.ui.define([
    "sap/ui/thirdparty/jquery",
    "sap/ui/core/util/MockServer",
    "sap/base/Log"
], function(jQuery, MockServer, Log) {
    "use strict";
    return {
        /**
         * Initializes the mock server.
         * You can configure the delay with the URL parameter "serverDelay".
         * The local mock data in this folder is returned instead of the real
data for testing.
         * @public
         */
        init: function() {
            // create
```

```
            var oMockServer = new MockServer({
                rootUri: "/"
            });
            oMockServer.simulate("../localService/metadata.xml", {
                sMockdataBaseUrl: "../localService/mockdata",
                bGenerateMissingMockData: true
            });
            // handling mocking a function import call step
            var aRequests = oMockServer.getRequests();
            aRequests.push({
                method: "GET",
                path: new RegExp("FindUpcomingMeetups(.*)"),
                response: function(oXhr) {
                    Log.debug("Incoming request for FindUpcomingMeetups");
                    var today = new Date();
                    today.setHours(0); // or today.toUTCString(0) due to
timezone differences
                    today.setMinutes(0);
                    today.setSeconds(0);
                    jQuery.ajax({
                        url: "/Meetups?$filter=EventDate ge " + "/Date(" +
today.getTime() + ")/",
                        dataType : 'json',
                        async: false,
                        success : function(oData) {
                            oXhr.respondJSON(200, {}, JSON.stringify(oData));
                        }
                    });
                    return true;
                }
            });
            oMockServer.setRequests(aRequests);
            // handling custom URL parameter step
            var fnCustom = function(oEvent) {
                var oXhr = oEvent.getParameter("oXhr");
                if (oXhr && oXhr.url.indexOf("first") > -1) {
                    oEvent.getParameter("oFilteredData").results.splice(3, 100);
                }
            };
            oMockServer.attachAfter("GET", fnCustom, "Meetups");
            // start
            oMockServer.start();
            Log.info("Running the app with mock data");
        }
    };
});
```

We push a new request handler to mock the function import call as follows:

1. Fetch the array of requests from the `MockServer`. The mock server holds an internal list of requests that you have to get and set if you want to modify.
2. Push a new request handler to handle the function import
3. Set the updated request array

The request handler has the following structure:

- `method`: The HTTP method of the mock request
- `path`: The relative path (appended to the `rootUri`) of the request.
  We can define the path as a regular expression, for example, to handle URL parameters.
- `response`: A response function that simulates the answer from the server

The `response` function executes a request to the `Meetups` entity set with an OData `$filter` query that actually returns all meet-ups with `EventDate` that is greater than or equals today. We compose a date for the filter and send it to the `server` manually as a synchronous request.

It is o.k. to use `jQuery.sap.sjax` here, because the call will not actually leave the client. It triggers a new request that again is intercepted and processed by the mock server.

We finally respond on the XHR object by calling the `respondJSON` API. It will add the proper content type header for the JSON format and send the result. We provide the HTTP status code `200` (success) and the 'stringified' response data as the arguments. Returning `true` at the end of the function indicates that we have completed the processing of the request in this handler (no additional request handlers should be checked for that request).

When you now start the app again you will see a list of upcoming meet-ups.

Creating and Editing Mock Data in SAP Web IDE (Optional)

webapp/localService/mockserver.js

```
...
oMockServer.simulate("localService/metadata.xml", {
    sMockdataBaseUrl : "localService/mockdata",
    bGenerateMissingMockData : true
});
...
```

The path we gave in the simulate function for mock data is where we want to store the .json file(s).

- Save it (in JSON format) from a real service
- Create it manually
- Generate it in SAP Web IDE by choosing *Edit Mock Data* in the context menu of the `medatdata.xml` file. For more information about SAP Web IDE, see the documentation for SAP Web IDE on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_Web_IDE.

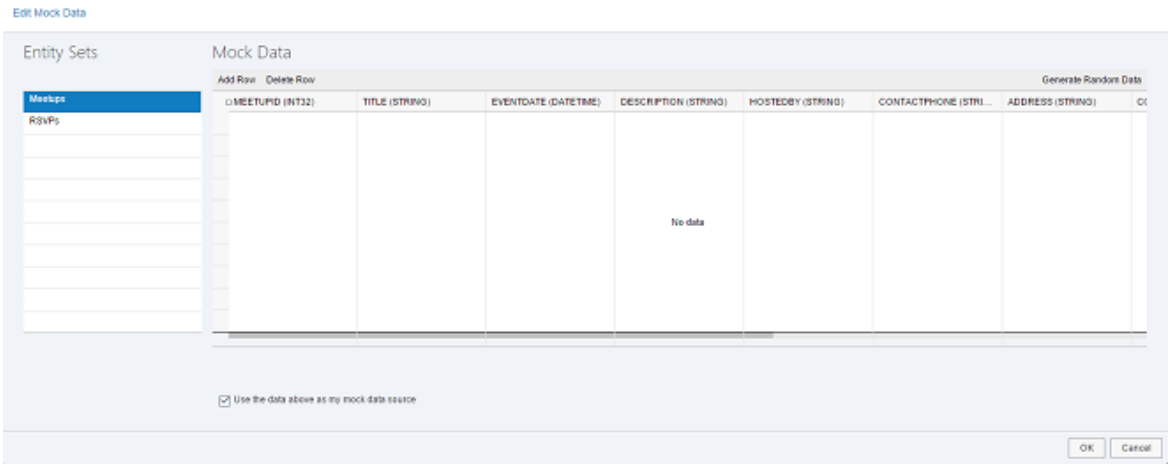


Figure 134: Editing mock data in SAP Web IDE

Worklist App

In this tutorial we will build an app using SAPUI5 that, for example, a shop owner can use to manage his product stock levels.

The app provides the following features:

- Overview of all products
- Track products with shortages or products that are completely out of stock
- Reorder products that are low in stock
- View product details and add comments

We will use the worklist template as a starting point for this tutorial and add additional features to the app as we go through the steps. The template implements a typical "Worklist" floorplan, one of the patterns that are specified by the SAP Fiori design guidelines, but you can also use it as a starting point for easily creating any kind of list-based apps. For more information about worklist floorplans, see the *Related Information* section at the bottom of this topic.

Preview

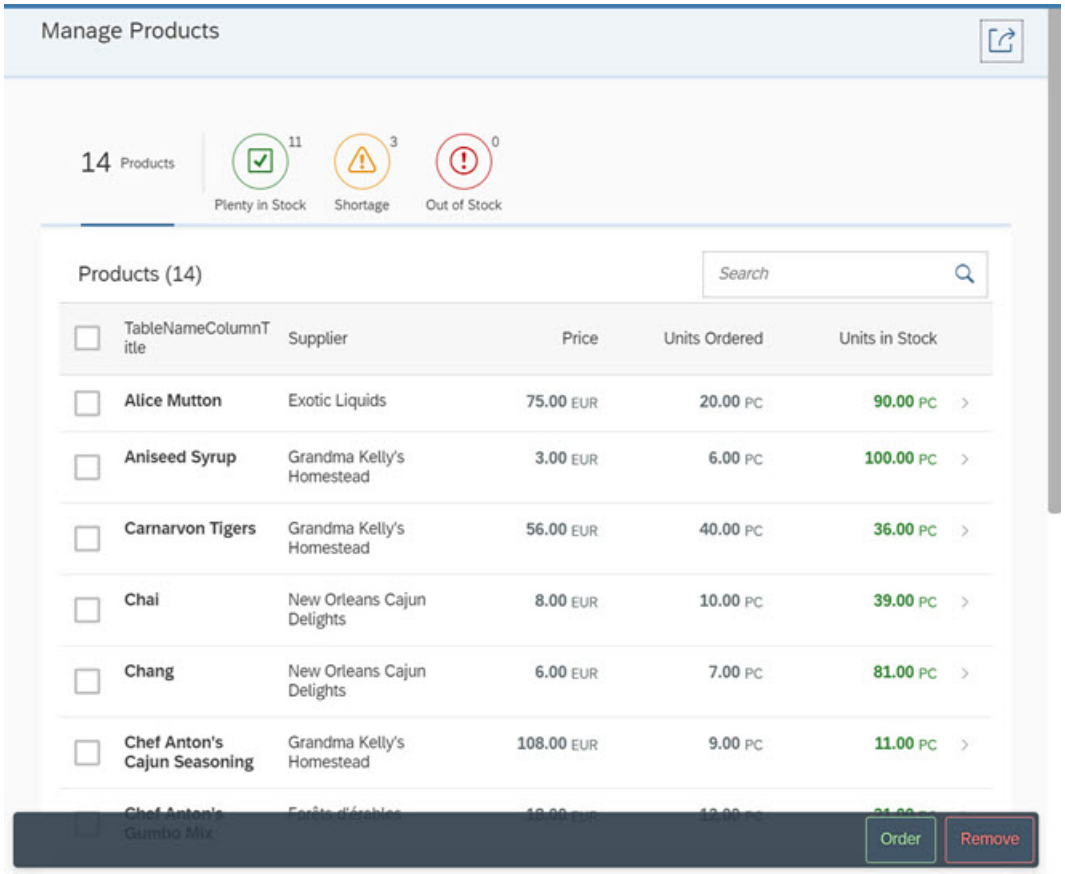


Figure 135: Start page of the app with list of products and actions

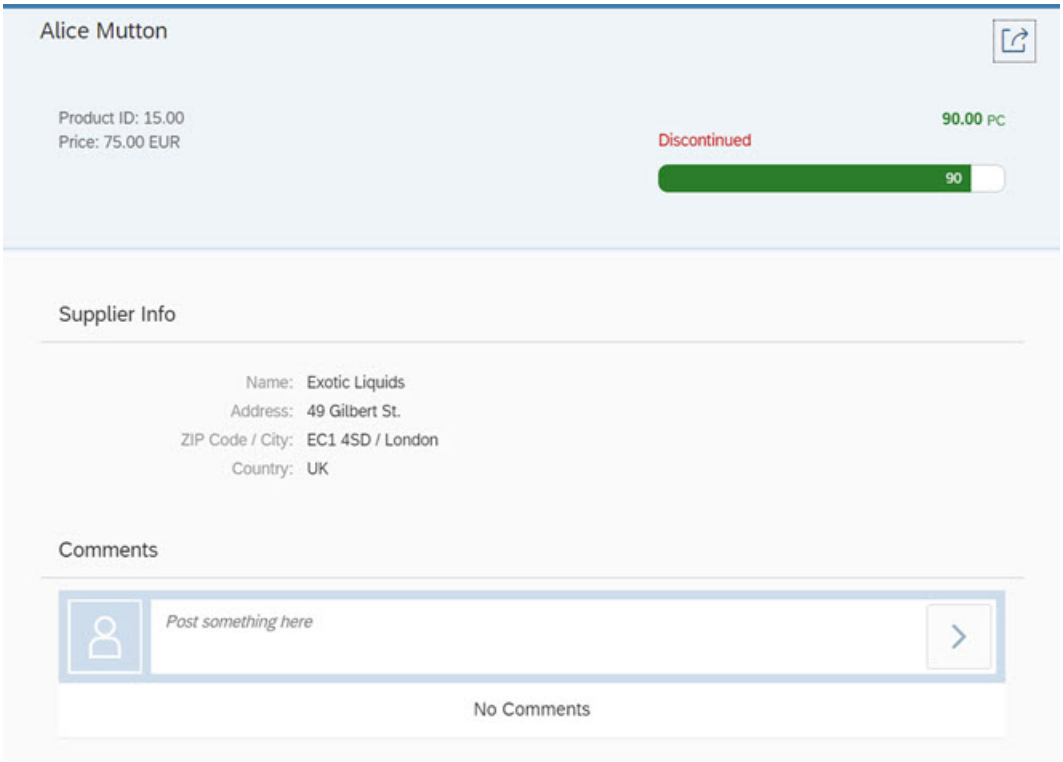


Figure 136: Product detail page of the app

Choose your development environment

You can do this tutorial either with SAP Web IDE or choose your own development environment:

- If you use SAP Web IDE, you don't need to set up a development environment, a server and so on. All you need is a browser and an account for the SAP Cloud Platform. If you don't have an account yet, you can easily get a trial account. We recommend to continue with the SAP Web IDE as it has out-of-the-box support for SAPUI5 and because there is no setup overhead at all.
  In this case, you start with the template that is available in SAP Web IDE as described in option 1 for step 1 of this tutorial (see Step 1 (Option 1): Creating the Initial App with an App Template in SAP Web IDE [page 449]).
- If you want to use your local development environment and deploy to any Web server of your choice, you download the code for step 1 from the Demo Kit at Worklist App. In this case, start with option 2 for step 1 of this tutorial (see Step 1 (Option 2): Downloading the Code [page 454]).

There might be some small differences between the worklist app code generated by the SAP Web IDE template and the code downloaded from the Demo Kit. However, the differences are small and not relevant for the purpose of this tutorial

> → Tip
>
> You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, copy it to your workspace and make sure that the application runs by calling the `webapp/test.html` file.
>
> For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Related Information

SAP Fiori design guidelines: Worklist Floorplans
Worklist Template [page 1400]
Get a SAP Cloud Platform trial account

Step 1 (Option 1): Creating the Initial App with an App Template in SAP Web IDE

This first step is only relevant if you decided to use the SAP Web IDE. In this step, we will set up the worklist app using a template and configure the service to display products in the app. The template includes generic app functionality and tests that can be easily extended with custom functionality for our use case.

Prerequisites

Set up your SAP Web IDE and define a destination to the Northwind OData service as described under App Development Using SAP Web IDE [page 44].

Preview

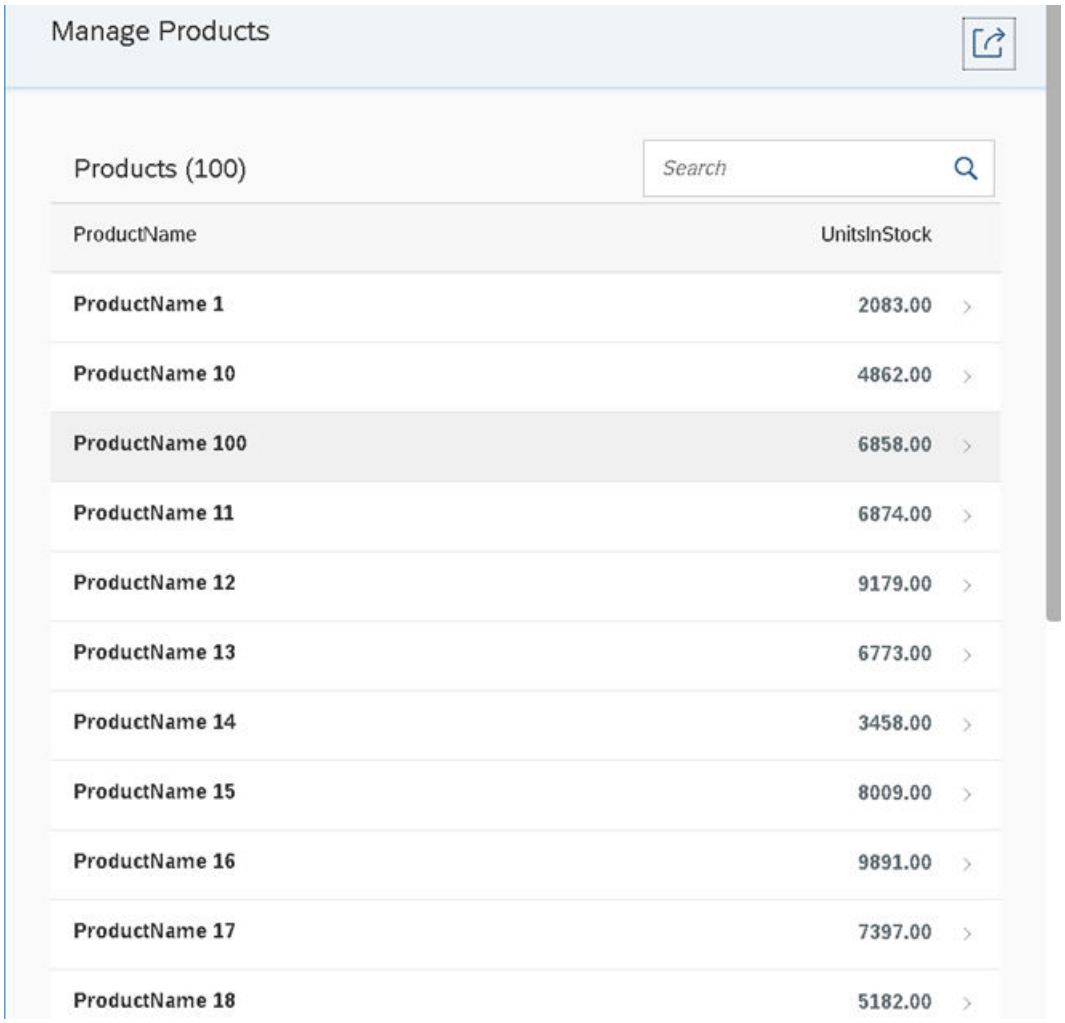


Figure 137: The worklist app

Create the Initial App Using the Template Wizard

1. Launch SAP Web IDE.
2. Choose *File* > *New* > *Project from Template*
3. Select the *SAP Fiori Worklist Application* template, and choose the latest SAPUI5 version from the *SAPUI5 Version* dropdown box. Choose *Next*.
4. On the *Basic Information* screen, enter `MyWorklistApp` as project name. Enter the following data:

Table 7: *App Descriptor Data*

Field	Value	Description
Title	`Manage Products`	Title of the app, which will be displayed as header.

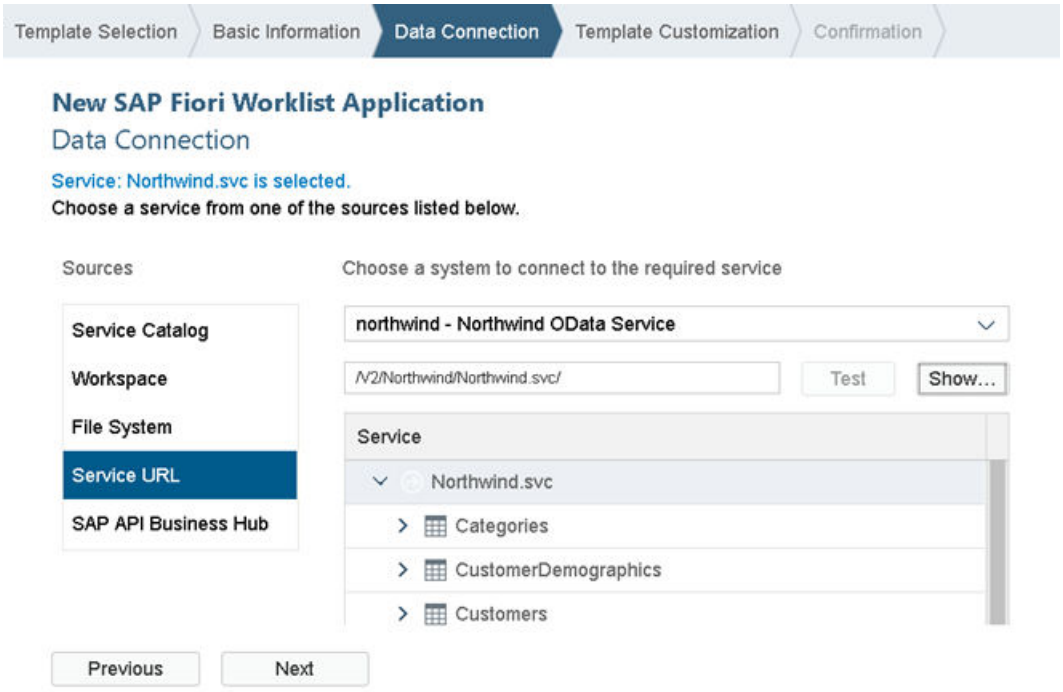
Field	Value	Description
Namespace	`mycompany.myapp`	The application namespace is a unique identifier for your application resources.
Description	`My Worklist App`	Short description of your app.

Choose *Next*.

5. On the *Data Connection* screen, select *Service URL* in the *Sources* area.
Choose *northwind - Northwind OData Service* and enter the URL `/V2/Northwind/Northwind.svc/`.

> **i Note**
>
> If you cannot find the Northwind service, create the destination as described under Create a Northwind Destination [page 49].

Validate the URL by choosing *Test* next to the URL. You should now see the service entities as displayed in the following screenshot:



> **i Note**
>
> At runtime, the relative URL `/V2/Northwind/Northwind.svc/` is prefixed with `/destinations/northwind`. As a result, all our Northwind OData requests will be proxied via the *Northwind OData Service* destination that is defined in the SAP Cloud Platform Cockpit. The destination contains the URL to the resource `http://services.odata.org` and has the proxy type *Internet*. From this configuration the proxy knows where the requests are directed.

Choose *Next*.

6. On the *Template Customization* screen, select the following data:

Table 8: *Application Scenario*

Field	Value	Description
App Type	**`Standalone App`**	We create a standalone app that can be run without SAP Fiori launchpad (FLP). If you choose to build an *SAP Fiori Launchpad Component*, you automatically get test HTML files with the FLP Sandbox, and the app automatically includes additional features like *Save as Tile*.

Table 9: *Data Binding - Object*

Field	Value	Description
Object Collection	**`Products`**	This is the main entity set that will be displayed in the app. Some of the other fields below are automatically selected depending on this field.
Object Collection ID	**`ProductID`**	The unique key that is used to identify the object collection.
Object Title	**`ProductName`**	The display name of the main entity.
Object Numeric Attribute	**`UnitsInStock`**	The number displayed next to the product name. In this scenario we pick the `UnitsInStock`. This represents the stock quantity of the product.

7. Choose *Next* and *Finish*

A new folder `MyWorklistApp` is now available in your local workspace. It contains the following files and folders of the initial app:

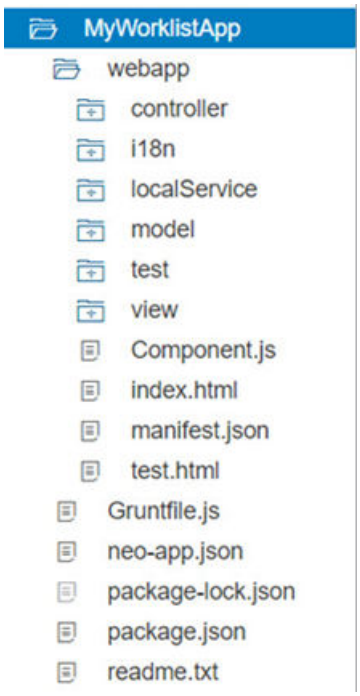


Figure 138: Folder structure of the initial project

> **i Note**
>
> The auto-generated files `Gruntfile.js`, `package-lock.json`, and `package.json` are only necessary for working with SAP Web IDE or a local IDE. They are not included in case you download the example code from the *Samples*.
>
> These three files will not be changed throughout this tutorial, so we will ignore them in the following steps.

8. Run the app.
   There are several HTML files available in the `webapp/test` folder, which enable you to run the app with a Mock Server. You can also use them to run unit tests and OPA tests.

   > **i Note**
   >
   > The template comes with two run configurations for SAP Web IDE: You can either run the app with data from a real back end service (*Run webapp/index.html*) or with local mock data (*Run webapp/test/mockServer.html (Mock Server)*).
   >
   > We choose the mock server option, because then the app will still be able to run even if the back end is unavailable or the service is not implemented yet. We could even configure a delay to make local testing more realistic.

   You should see the screen, which contains generated mock data.

From now on you can quickly run the app by selecting the root folder `MyWorklistApp` of your project in SAP Web IDE and pressing the *Run* button. The system will automatically use the option from the *Run* menu that you chose last (in this case, the *Run with MockServer* option).

> **i Note**
>
> The texts in the `i18n.properties` file are automatically generated based on the template Customizing (OData entity set, entities, properties, and texts). The result can be incorrect texts like "Enter an <Products> name or a part of it." You should therefore revise the generated texts in the `i18n.properties` file.

Related Information

App Development Using SAP Web IDE [page 44]

Step 1 (Option 2): Downloading the Code

In this step, we set up the initial app without SAP Web IDE.

If you are working with another IDE or development environment than the SAP Web IDE, you can simply download the code from the *Samples* in the Demo Kit at Worklist App - Step 1 and skip the wizard steps described in the previous step of this tutorial. The code contains a preconfigured application project that can be used as a starting point to develop the worklist app. You can deploy the downloaded application to a (local) Web server and call the `webapp/test/mockServer.html` file in your browser manually to start the app.

To access the real service, you would need to set up a proxy service that connects your app project deployed on a Web server to the remote service. Due to the so called same-origin policy browsers deny AJAX requests to service endpoints in case the domain/subdomain, protocol, or port differ from the app's domain/subdomain, protocol, or port. Cross-origin resource sharing (CORS) makes it possible to break out of these restrictions derived from the same-origin policy. With CORS the server and browser agree which cross-origin requests are allowed. Another way to bypass the same-origin policy is using a proxy on the same host of the app. To keep it simple, our app contains a test page to run the app with local mock data instead of retrieving the data from a real server hosted somewhere else. This way we won't have any issues related to the same-origin policy of the browsers, as long as we run the app with our mock server.

> **i Note**
>
> The texts in the `i18n.properties` file are automatically generated based on the template Customizing (OData entity set, entities, properties, and texts). The result can be incorrect texts like "Enter an <Products> name or a part of it." You should therefore revise the generated texts in the `i18n.properties` file.

Related Information

Development Environment [page 41]

Step 1 (Result): The Initial App

With the generated code (SAP Web IDE) or the downloaded code from the Demo Kit, you have an initial app structure with the following content inside the `webapp` folder.

- **Home Page** (`webapp/view/Worklist.view.xml` file)
  The home page of the app shows a table of products including the corresponding number of units in the stock. The title of the table shows how many items are available. A search field in the header toolbar of the table allows you to search for a product by name. Pressing a table row navigates the user to a new page that shows the details of the pressed product.
- **Data**
  You can run the app with the real service or with the mock server serving mock data. In the `webapp/localService/mockserver.js` file, the mock server is configured. Using the mock server in this tutorial allows us to easily run the code even without network connection and without the need of having a remote server for our application data. To run the app with the mock server and its corresponding mock data the `/webapp/test/mockServer.html` file has to be called in the browser.
  The `webapp/localService/metadata.xml` file is used by the mock server to describe our OData service. In this step, the mock server will generate mock data based on this file. In a subsequent step the mock server will use our own custom mock data.
- **Configuration of the App**
  In the `webapp/manifest.json` descriptor file, we configure our app. The descriptor file contains the following relevant sections:
  - `sap.app`
    In this section we reference an `i18n.properties` file and use a special syntax to bind the texts for the title and description properties. In the `dataSources` section, we tell our app where to find our `mainService` OData service. As you might guess, the URI correlates to the `rootUri` of our mock server instance, which can be found in `webapp/localService/mockserver.js`. It is important that these two paths match to allow our mock server to provide the test data we defined above.
  - `sap.ui5`
    In the `sap.ui5` section, we declare with the `rootView` parameter that our `mycompany.myapp.MyWorklistApp.view.App` view shall be loaded and used as the `rootView` for our app.
    Furthermore, we define two models to be automatically instantiated and bound to the component: an `i18n` model and a default model `""`. The latter references our `mainService dataSource`, which is declared in our `sap.app` section as an OData 2.0 data source. The `i18n` file can be found at `webapp/i18n/i18n.properties`. The `mainService` data source will be mocked by our mock server.

> **i Note**
>
> There is a `test.html` file in the `webapp` folder. This file serves as an easy entry point for developers to run and test the app in various ways during development. It contains links to the relevant files inside the `test` folder, which you can use to run with the Mock Server or to run unit tests and OPA tests.

Related Information

App Templates: Kick Start Your App Development [page 1399]

Folder Structure: Where to Put Your Files [page 1428]

Step 2: Custom Mock Data

In this step, we want to change the mock data of the initial app. The service metadata only contains a description of the service entities. The mock server that is part of the app will auto-generate random mock data based on the data types defined in the metadata file. To have a more realistic development environment we will now add additional sample data.

Preview

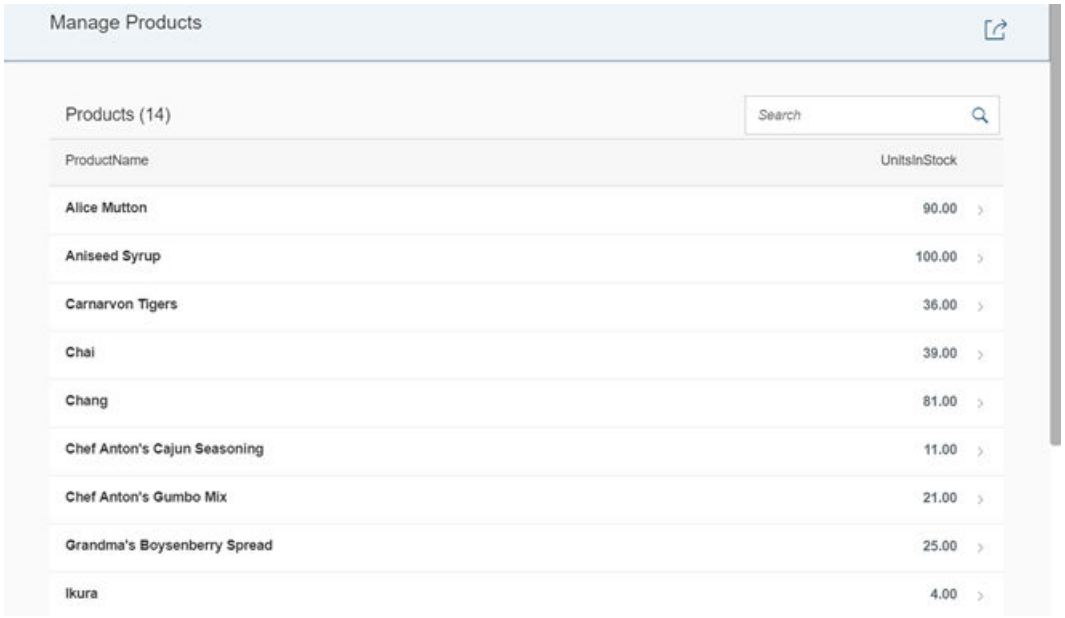


Figure 139: The product list of the initial app with custom mock data

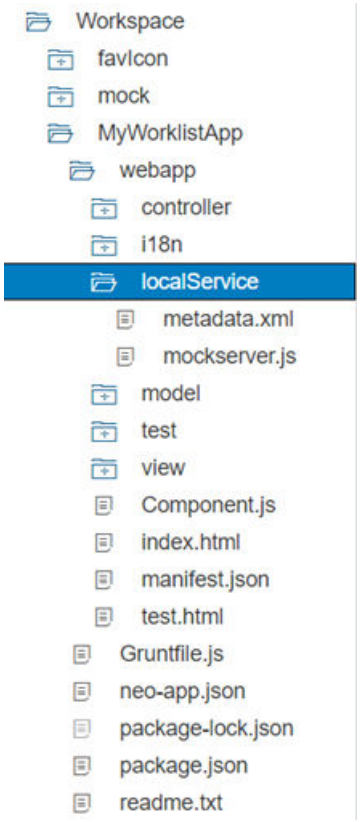


Figure 140: Folder structure for this step including custom mock data

The `webapp/localService/metadata.xml` file used by the mock server describes our OData service. The service only has two OData entities, and the data for these two entities is located in the folder `webapp/localService/mockdata`:

- Products
  A product has typical properties like `ProductName` and `UnitsInStock` as well as a navigation property to a supplier entity referenced by a `SupplierID`. Of course, the entity has an ID property `ProductID`. The corresponding `EntitySet` is `Products`. The actual test data containing several products is located in the `webapp/localService/mockdata/Products.json` file.
- Suppliers
  Later in this tutorial, we will display some information about the supplier of a product. The properties are `CompanyName`, `Address`, `City`, `PostalCode`, `Country`, and so on; all of them contain textual information of type `Edm.String`. The entity has an ID property `SupplierID` and the corresponding `EntitySet` is `Suppliers`. The supplier data for products is located in the file `webapp/localService/mockdata/Suppliers.json`.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Worklist App - Step 2.

webapp/localService/mockdata/Products.json (New)

```
[
  {
```

```
    "ProductID": 1,
    "ProductName": "Chai",
    "UnitsInStock": 39,
    "UnitsOnOrder": 10,
    "UnitPrice": 8,
    "SupplierID": 1,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(1)/
Supplier"
      }
    }
  },
  {
    "ProductID": 2,
    "ProductName": "Chang",
    "UnitsInStock": 81,
    "UnitsOnOrder": 7,
    "UnitPrice": 6,
    "SupplierID": 1,
    "Discontinued": true,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(2)/
Supplier"
      }
    }
  },
  {
    "ProductID": 3,
    "ProductName": "Aniseed Syrup",
    "UnitsInStock": 100,
    "UnitsOnOrder": 6,
    "UnitPrice": 3,
    "SupplierID": 3,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(3)/
Supplier"
      }
    }
  },
  {
    "ProductID": 4,
    "ProductName": "Schwarzwälder Kirschtorte",
    "UnitsInStock": 2,
    "UnitsOnOrder": 3,
    "UnitPrice": 19,
    "SupplierID": 3,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(4)/
Supplier"
      }
    }
  },
  {
    "ProductID": 5,
    "ProductName": "Chef Anton's Cajun Seasoning",
    "UnitsInStock": 11,
    "UnitsOnOrder": 9,
    "UnitPrice": 108,
    "SupplierID": 3,
    "Discontinued": false,
    "Supplier": {
```

```
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(5)/
Supplier"
      }
    }
  },
  {
    "ProductID": 6,
    "ProductName": "Chef Anton's Gumbo Mix",
    "UnitsInStock": 21,
    "UnitsOnOrder": 12,
    "UnitPrice": 18,
    "SupplierID": 4,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(6)/
Supplier"
      }
    }
  },
  {
    "ProductID": 7,
    "ProductName": "Grandma's Boysenberry Spread",
    "UnitsInStock": 25,
    "UnitsOnOrder": 25,
    "UnitPrice": 18,
    "SupplierID": 5,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(7)/
Supplier"
      }
    }
  },
  {
    "ProductID": 8,
    "ProductName": "Uncle Bob's Organic Dried Pears",
    "UnitsInStock": 29,
    "UnitsOnOrder": 7,
    "UnitPrice": 35,
    "SupplierID": 6,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(8)/
Supplier"
      }
    }
  },
  {
    "ProductID": 9,
    "ProductName": "Northwoods Cranberry Sauce",
    "UnitsInStock": 4,
    "UnitsOnOrder": 32,
    "UnitPrice": 35,
    "SupplierID": 6,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(9)/
Supplier"
      }
    }
  },
  {
    "ProductID": 10,
```

```
    "ProductName": "Mishi Kobe Niku",
    "UnitsInStock": 40,
    "UnitsOnOrder": 5,
    "UnitPrice": 130,
    "SupplierID": 5,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(10)/
Supplier"
      }
    }
  },
  {
    "ProductID": 11,
    "ProductName": "Ikura",
    "UnitsInStock": 4,
    "UnitsOnOrder": 10,
    "UnitPrice": 13,
    "SupplierID": 4,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(11)/
Supplier"
      }
    }
  },
  {
    "ProductID": 13,
    "ProductName": "Carnarvon Tigers",
    "UnitsInStock": 36,
    "UnitsOnOrder": 40,
    "UnitPrice": 56,
    "SupplierID": 3,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(13)/
Supplier"
      }
    }
  },
  {
    "ProductID": 14,
    "ProductName": "Teatime Chocolate Biscuits",
    "UnitsInStock": 21,
    "UnitsOnOrder": 40,
    "UnitPrice": 7,
    "SupplierID": 2,
    "Discontinued": false,
    "Supplier": {
      "__deferred": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(14)/
Supplier"
      }
    }
  },
  {
    "ProductID": 15,
    "ProductName": "Alice Mutton",
    "UnitsInStock": 90,
    "UnitsOnOrder": 20,
    "UnitPrice": 75,
    "SupplierID": 2,
    "Discontinued": true,
    "Supplier": {
      "__deferred": {
```

```
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/Products(15)/
Supplier"
      }
    }
  }
]
```

First create a new `mockdata` folder inside `webapp/localService`. Create a `Products.json` file, and copy and paste the code.

webapp/localService/mockdata/Suppliers.json (New)

```
[
  {
    "SupplierID": 1,
    "CompanyName": "New Orleans Cajun Delights",
    "ContactName": "Shelley Burke",
    "ContactTitle": "Order Administrator",
    "Address": "P.O. Box 78934",
    "City": "New Orleans",
    "Region": "LA",
    "PostalCode": "70117",
    "Country": "USA"
  },
  {
    "SupplierID": 2,
    "CompanyName": "Exotic Liquids",
    "ContactName": "Charlotte Cooper",
    "ContactTitle": "Purchasing Manager",
    "Address": "49 Gilbert St.",
    "City": "London",
    "Region": "UK",
    "PostalCode": "EC1 4SD",
    "Country": "UK"
  },
  {
    "SupplierID": 3,
    "CompanyName": "Grandma Kelly's Homestead",
    "ContactName": "Regina Murphy",
    "ContactTitle": "Sales Representative",
    "Address": "707 Oxford Rd.",
    "City": "Ann Arbor",
    "Region": "MI",
    "PostalCode": "48104",
    "Country": "USA"
  },
  {
    "SupplierID": 4,
    "CompanyName": "Forêts d'érables",
    "ContactName": "Chantal Goulet",
    "ContactTitle": "Accounting Manager",
    "Address": "148 rue Chasseur",
    "City": "Ste-Hyacinthe",
    "Region": "Québec",
    "PostalCode": "J2S 7S8",
    "Country": "Canada"
  },
  {
    "SupplierID": 5,
    "CompanyName": "Plutzer Lebensmittelgroßmärkte AG",
    "ContactName": "Martin Bein",
    "ContactTitle": "International Marketing Mgr.",
    "Address": "Bogenallee 51",
    "City": "Frankfurt",
    "Region": "DE",
    "PostalCode": "60439",
    "Country": "Germany"
```

```
  },
  {
    "SupplierID": 6,
    "CompanyName": "Lyngbysild",
    "ContactName": "Niels Petersen",
    "ContactTitle": "Sales Manager",
    "Address": "Lyngbysild Fiskebakken 10",
    "City": "Lyngby",
    "Region": "NL",
    "PostalCode": "2800",
    "Country": "Denmark"
  },
  {
    "SupplierID": 7,
    "CompanyName": "Formaggi Fortini s.r.l.",
    "ContactName": "Elio Rossi",
    "ContactTitle": "Sales Representative",
    "Address": "Viale Dante, 75",
    "City": "Ravenna",
    "Region": "IL",
    "PostalCode": "48100",
    "Country": "Italy"
  }
]
```

Create a `Suppliers.json` file, and copy and paste the code.

You can now run the app again and see the mock data in your app.

> **i Note**
>
> In order to get realistic mock data you can call a real OData service directly in your browser to receive the real data of a given `Entity` or `EntitySet`. Make sure that you call the service with the system option `$format=json`, that is `http://services.odata.org/V2/Northwind/Northwind.svc/Products?$format=json`. This will return the data in JSON format, which is the format required for our mock data. This data is put into a local file in your application's `webapp/localService/mockdata` folder. The file name is expected to be the name of the corresponding `EntitySet` ends with `.json`, for example `Products.json`. The obtained data from the OData service can serve as a first set of mock data, which you can change to your needs if necessary. SAP Web IDE also offers a dedicated editor for mock data that makes the maintenance of the data even easier.

Step 3: Extending the Worklist Table

In this step, we will edit the worklist table to include additional columns for our manage product stocks scenario. We display the supplier, the product price, and the number of units on order for each product and format the values accordingly.

Preview

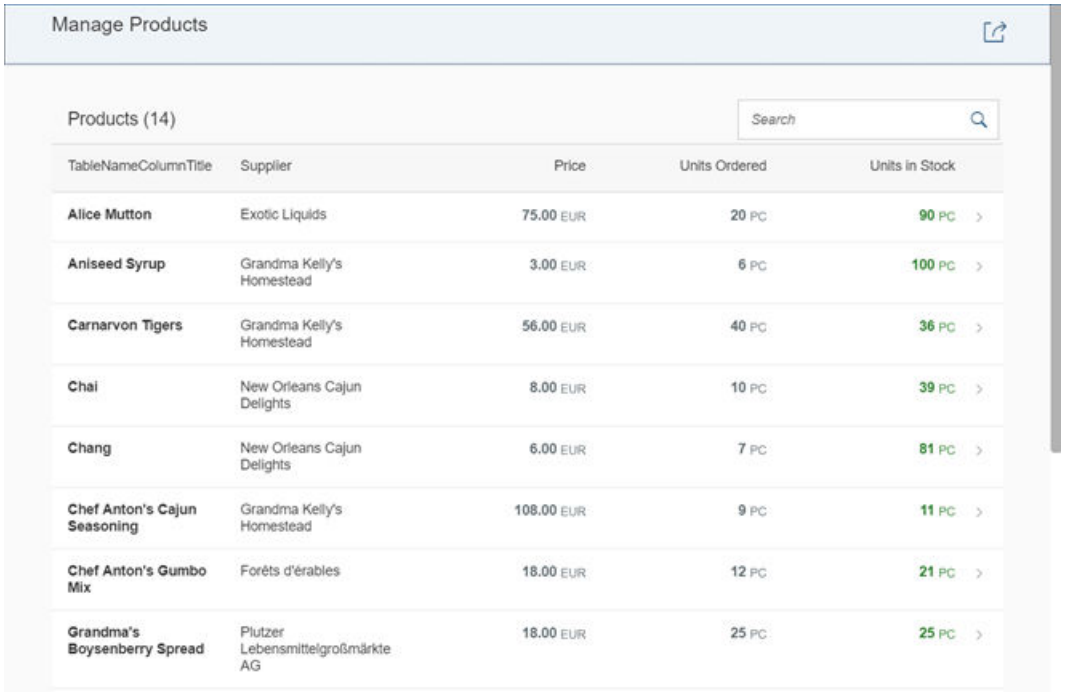


Figure 141: The improved worklist table with new columns and formatting

Coding

You can view and download all files in the Demo Kit at Worklist App - Step 3.

webapp/view/Worklist.view.xml

```
…
<Table
    id="table"
    width="auto"
    items="{
        path: '/Products',
        sorter: {
            path: 'ProductName',
            descending: false
        },
        parameters: {
            'expand': 'Supplier'
        }
    }"
```

```
    noDataText="{worklistView>/tableNoDataText}"
    busyIndicatorDelay="{worklistView>/tableBusyDelay}"
    growing="true"
    growingScrollToLoad="true"
    updateFinished=".onUpdateFinished">
    <headerToolbar>
        <Toolbar>
            <Title id="tableHeader" text="{worklistView>/worklistTableTitle}"/>
            <ToolbarSpacer />
            <SearchField
                id="searchField"
                tooltip="{i18n>worklistSearchTooltip}"
                search=".onSearch"
                width="auto">
            </SearchField>
        </Toolbar>
…
```

We want to display the supplier's company name in a separate column in the table for each product. Therefore, we extend the `items` aggregation of the table with an `expand` parameter for the `Supplier` entity. With this, the supplier data will be already included in the service request for the products.

We expand the supplier because we want to avoid sending one additional request for each product to get the supplier. Furthermore, this allows us to bind directly to `{Supplier/CompanyName}` later.

> **i** Note
>
> **OData's "expand" Mechanism:**
>
> OData `$expand` is very helpful when combining data from different service entities. Instead of having to send an additional service request for the second entity, we simply expand the service call to include the second entity as well – similar to a join in a relational database. Have a look at the local service metadata definition file `webapp/localService/metadata.xml` that represents the interface of our service. In the `metadata` you can see a list of entities that are available in this service, for example `Products` and `Suppliers`. Each entity lists a number of fields that we can bind to the properties of our view.

webapp/localService/metadata.xml

```
<EntityType Name="Product">
    <Key>
        <PropertyRef Name="ProductID"/>
    </Key>
    <Property
        xmlns:p8="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
Name="ProductID" Type="Edm.Int32" Nullable="false"
p8:StoreGeneratedPattern="Identity"/>
        <Property Name="ProductName" Type="Edm.String" Nullable="false"
MaxLength="40" Unicode="true" FixedLength="false"/>
        <Property Name="SupplierID" Type="Edm.Int32" Nullable="true"/>
        <Property Name="CategoryID" Type="Edm.Int32" Nullable="true"/>
        <Property Name="QuantityPerUnit" Type="Edm.String" Nullable="true"
MaxLength="20" Unicode="true" FixedLength="false"/>
        <Property Name="UnitPrice" Type="Edm.Decimal" Nullable="true"
Precision="19" Scale="4"/>
        <Property Name="UnitsInStock" Type="Edm.Int16" Nullable="true"/>
        <Property Name="UnitsOnOrder" Type="Edm.Int16" Nullable="true"/>
        <Property Name="ReorderLevel" Type="Edm.Int16" Nullable="true"/>
        <Property Name="Discontinued" Type="Edm.Boolean" Nullable="false"/>
        <NavigationProperty Name="Category"
Relationship="NorthwindModel.FK_Products_Categories" FromRole="Products"
ToRole="Categories"/>
```

```
        <NavigationProperty Name="Order_Details"
Relationship="NorthwindModel.FK_Order_Details_Products" FromRole="Products"
ToRole="Order_Details"/>
        <NavigationProperty Name="Supplier"
Relationship="NorthwindModel.FK_Products_Suppliers" FromRole="Products"
ToRole="Suppliers"/>
    </EntityType>
```

In the entity `Products`, you can see that an additional relation to the `Supplier` is available as a `NavigationProperty`. A navigation property links two entities of an OData service and assigns the supplier to the product here.

When using a real OData service, the interface would be available by calling the service URL directly in a browser (e.g. `http://services.odata.org/V3/Northwind/Northwind.svc/$metadata` for the Northwind OData test service). In our app project we use local mock data and serve the data with the mock server instead.

webapp/view/Worklist.view.xml

```
…
<columns>
    <Column id="nameColumn">
        <Text
            id="nameColumnTitle"
            text="{i18n>TableNameColumnTitle}"/>
    </Column>
    <Column
        id="supplierNameColumn"
        demandPopin="false"
        minScreenWidth="Tablet">
        <Text text="{i18n>TableSupplierColumnTitle}"/>
    </Column>
    <Column
        id="unitPriceColumn"
        hAlign="End"
        demandPopin="true"
        minScreenWidth="Tablet">
        <Text text="{i18n>TablePriceColumnTitle}"/>
    </Column>
    <Column
        id="unitsOnOrderColumn"
        demandPopin="true"
        minScreenWidth="Tablet"
        hAlign="End">
        <Text text="{i18n>TableUnitsOrderedColumnTitle}"/>
    </Column>
    <Column
        id="unitsInStockColumn"
        hAlign="End">
        <Text text="{i18n>TableUnitsInStockColumnTitle}"/>
    </Column>
</columns>
…
```

Next, we change the column definitions of the table. We define the new columns and update the existing ones in the columns aggregation of the table according to the code above (i.e. just copy and paste the highlighted content into your columns aggregation).

The column definitions include a text that we will later define in the resource bundle (`i18n` model – a short name for internationalization) so that the column titles can be translated to other languages. And we will define additional settings for text alignment and making the table responsive. Some columns are not as important as others and can be displayed below the main columns (`popin`) on devices with small or medium-sized screens.

Let's have a detailed look at the columns:

- Product Name
  The product name is the first column and it is always visible on any device.
- Supplier
  Each product has a supplier. This column contains the company name of the supplier supplying the product. On small screen devices like smart phones we hide this column as we do not have much screen space for a table.
- Price
  The currency of the product's unit price is Euro (EUR). We are talking about stock levels in this app, so the number of units is most interesting for us - not their price. Price is still good to know, so it is not entirely removed. However, this field is not as important as the unit fields and will `popin` on smart phones.
- Units on Order
  This column shows the units that have been ordered already for this product and will be added to the stock shortly. In other words, this is the number of items ordered, but not yet received. A shortage for a product can easily be resolved by reordering the product in advance (we add this feature later). This field will `popin` on smart phone devices.
- Units in Stock
  The column contains the product's stock units currently available for sale. This field is the most important column for our manage product stocks app. Therefore, this column is visible for all devices and it's visible without a `popin`. Later, we will use this column to visualize a stock status for the specific products so that attention will be drawn to any stock issues with the products.

webapp/model/formatter.js

```
sap.ui.define([
    "sap/ui/core/library"
] , function (coreLibrary) {
    "use strict";
    // shortcut for sap.ui.core.ValueState
    var ValueState = coreLibrary.ValueState;
    return {
        ...
        },

        /**
         * Defines a value state based on the stock level
         *
         * @public
         * @param {number} iValue the stock level of a product
         * @returns {string} sValue the state for the stock level
         */
        quantityState: function(iValue) {
            if (iValue === 0) {
                return ValueState.Error;
            } else if (iValue <= 10) {
                return ValueState.Warning;
            } else {
                return ValueState.Success;
            }
        }
    };
});
```

Our table has a column that will contain the units in stock for each product. It would be nice to visualize the corresponding numbers so that we can point out important information to the users, such as a shortage. We want to visualize the numbers by using a specific `ValueState` depending on the units in stock. This can be achieved by a simple formatter, which we will use later.

We add a new formatter function `quantityState` to the `webapp/model/formatter.js` file. The `ValueState` type is loaded as an additional dependency. The formatter implements the following logic with a simple `if/else` statement:

- A totally depleted stock (0 pieces remaining) will return a semantic `Error` state that will color the text in the units in stock field red.
- Very low stock (10 or less pieces remaining) will lead to a `Warning` state (orange).
- A stock of more than 10 items will convert to `Success` (green)

webapp/view/Worklist.view.xml

```
…
<items>
    <ColumnListItem
        type="Navigation"
        press="onPress">
        <cells>
         <ObjectIdentifier
            title="{ProductName}"/>
         <Text text = "{Supplier/CompanyName}"/>
         <ObjectNumber
            unit="EUR"
            number="{
               path: 'UnitPrice',
               formatter: '.formatter.numberUnit'
            }"/>
         <ObjectNumber
            number="{UnitsOnOrder}"
            unit="PC"/>
         <ObjectNumber
            number="{UnitsInStock}"
            unit="PC"
            state="{
               path: 'UnitsInStock',
               formatter: '.formatter.quantityState'
            }"/>
      </cells>
   </ColumnListItem>
</items>
…
```

The next task is to define the cells to appear in each row of the table. For each column, we define a control in the `cells` aggregation of the table and configure the data binding as well as the formatting of the data.

- The first cell simply displays the `ProductName` property of the corresponding entity by using an `ObjectIdentifier` control.
- The *Supplier* cell of each row is a simple `sap.m.Text` control. Its text property is bound to `Supplier/CompanyName`. This references the property `CompanyName` of the entity's `NavigationProperty Supplier`. This `NavigationProperty` will be expanded automatically; we configured this earlier in this step.
- The *Price* cell uses an `sap.m.ObjectNumber` control and a custom formatter. You can find the formatter's implementation in the `webapp/model/formatter.js` file. The unit property is not bound and hard coded to "EUR" as the currency is not part of the model for our app. The units on order are displayed with a sap.m.ObjectNumber control as well, but without additional formatting. Its `unit` property is hard coded to `PC`, which is the short form for "pieces".
- The last cell shows the units in stock and was already specified in the previous step. We would like to use this field to show an additional status based on the stock level so we change the binding syntax to an object

notation and add an additional formatter `quantityState`. We implemented this formatter in the previous code block above.

> **i Note**
>
> The formatter functions used in this XML view are loaded by the controller and thus can be accessed relatively to the controller through the property name `.formatter`. This logic is already part of the initial app.

webapp/i18n/i18n.properties

```
#XTIT: The title of the column containing Product name
TableProductColumnTitle=Product
#XTIT: The title of the column containing Supplier name
TableSupplierColumnTitle=Supplier

#XTIT: The title of the column containing Price
TablePriceColumnTitle=Price

#XTIT: The title of the column containing Ordered Units
TableUnitsOrderedColumnTitle=Units Ordered

#XTIT: The title of the column containing Units in Stock
TableUnitsInStockColumnTitle=Units in Stock
#XBLI: Text for a table with no data
tableNoDataText=No products are currently available
...
```

Finally, we modify the existing column names in the resource bundle file `webapp/i18n/i18n.properties` to match our scenario and add the new texts for the column titles.

> **i Note**
>
> The `webapp/i18n/i18n.properties` file contains some annotations for each key in the file. These annotations offer some more context, which can help translators to better interpret the semantics of the text belonging to the keys. An example for such an annotation is XTIT in the `i18n.properties` file above, which tells that the corresponding key is supposed to be used as a title. The guidelines at https://github.com/SAP/openui5/blob/master/docs/guidelines/translationfiles.md give you a better idea of how this can be used. Be aware that this is how SAP uses the annotations internally. In case you want to use this approach to work with your own translators make sure that you agree on a common set of allowed annotations that everybody understands.

> **→ Tip**
>
> **Testing the Responsiveness of the App**
>
> In the previous code blocks of this step we made sure that our table is responsive. Depending on the device type columns are hidden, displayed as a popin, or displayed without a popin. Now, we want to test the responsiveness without the having different devices. This can be done in different ways, we will cover two options:
>
> - **Testing the responsiveness with the SAP Web IDE**
>   SAP Web IDE can simulate different screen sizes. You just have to make sure that the run configuration is set up correctly:
>
>   1. In SAP Web IDE, choose *Run* > *New Configuration*

2. Select the *Run with MockServer* configuration.
3. In the *Frame* screen area, select the *Open with frame* checkbox.
   If you create a new project this option is switched on by default. For apps created based on the Worklist template, however, this option is switched off to make it easier to debug the application coding.
4. Save the configuration and close the dialog.

If you now run the application again, you will see the surrounding frame, in which you can easily choose between different screen sizes and change the device orientation.
Switch, for example from *Medium* to *Small*, and you will see that the table behaves as expected.

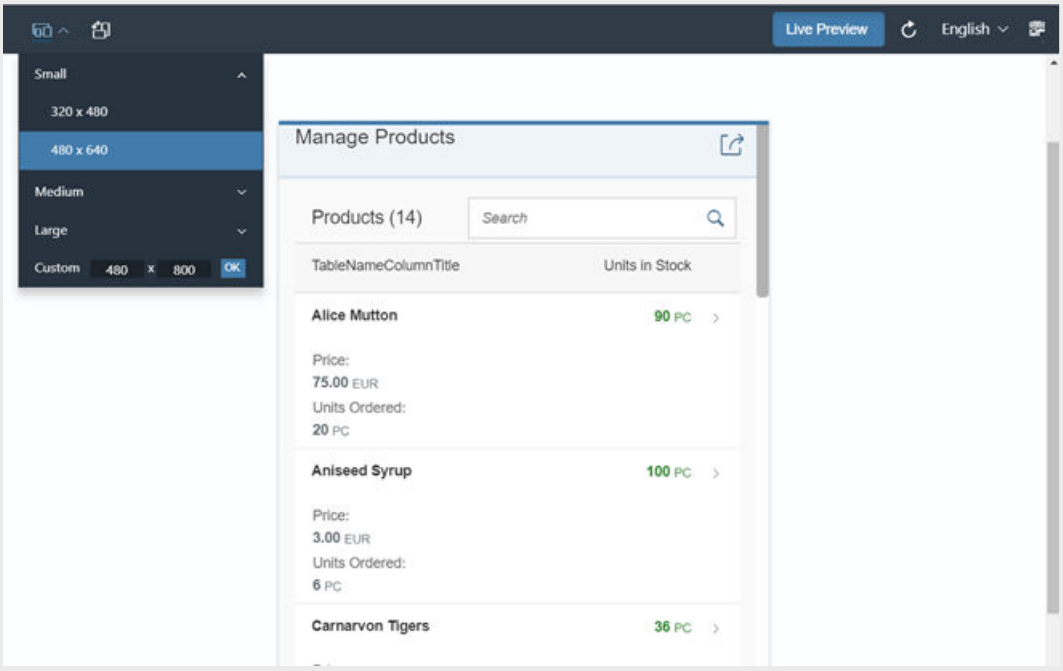


Figure 142: Testing the responsiveness in SAP Web IDE

- **Testing the responsiveness using the Developer Tools of Google Chrome**
  If you use the Google Chrome browser, you can also use its great developer tools to test the responsiveness of your app.
  1. Call the app and open the developer tools in Chrome with `F12`
  2. Choose the *Toggle device mode* icon.
  3. Now choose from the different devices in the *Models* field, and observe the behavior of your app.

Step 4: Quick Filter for the Worklist

For easily detecting and managing product shortages in our app, we will add a quick filter for the worklist table. Users can press the filter tabs to display the products according to whether they are in stock, have low stock or no stock. The table will update accordingly and show only the products matching the criteria.

Preview

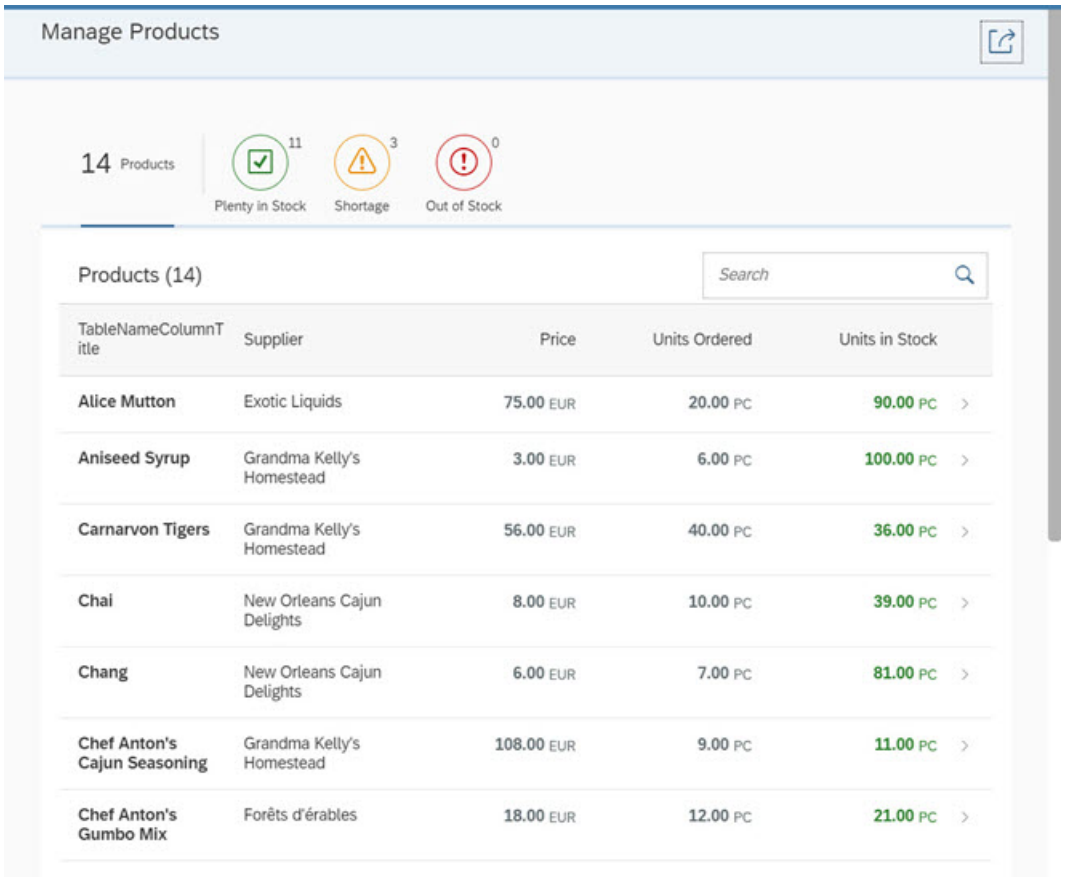


Figure 143: A quick filter allows filtering the product table

Coding

You can view and download all files in the Demo Kit at Worklist App - Step 4.

webapp/view/Worklist.view.xml

```
<mvc:View
    controllerName="myCompany.myApp.controller.Worklist"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:semantic="sap.m.semantic"
    xmlns="sap.m">
    <semantic:FullscreenPage
        id="page"
```

```
        navButtonPress="onNavBack"
        showNavButton="true"
        title="{i18n>worklistViewTitle}">
        <semantic:content>
            <IconTabBar
                id="iconTabBar"
                select=".onQuickFilter"
                expandable="false"
                headerBackgroundDesign="Transparent">
                <items>
                    <IconTabFilter
                        key="all"
                        showAll="true"
                        count="{worklistView>/countAll}"
                        text="{i18n>WorklistFilterProductsAll}"/>
                    <IconTabSeparator/>
                    <IconTabFilter
                        key="inStock"
                        icon="sap-icon://message-success"
                        iconColor="Positive"
                        count="{worklistView>/inStock}"
                        text="{i18n>WorklistFilterInStock}"/>
                    <IconTabFilter
                        key="shortage"
                        icon="sap-icon://message-warning"
                        iconColor="Critical"
                        count="{worklistView>/shortage}"
                        text="{i18n>WorklistFilterShortage}"/>
                    <IconTabFilter
                        key="outOfStock"
                        icon="sap-icon://message-error"
                        iconColor="Negative"
                        count="{worklistView>/outOfStock}"
                        text="{i18n>WorklistFilterOutOfStock}"/>
                </items>
                <content>
                    <Table
...
                    </Table>
                </content>
            </IconTabBar>
        </semantic:content>
        <semantic:sendEmailAction>
            <semantic:SendEmailAction
                id="shareEmail"
                press="onShareEmailPress"/>
        </semantic:sendEmailAction>
    </semantic:FullscreenPage>
</mvc:View>
```

We now update the view and add the new UI for the quick filter to the content aggregation of the `sap.m.SemanticPage` control just before the `table`. It is modeled using a `sap.m.IconTabBar` control and a `sap.m.IconTabFilter` for each of the following filter options:

- *Total Stock*
  This tab will simply show the overall number of products that has been returned by the data service. The `count` property is bound to a local view model and the number will be updated in the controller later in this step. This tab will show a larger number only (optional) and no icon by using the `showAll` property.
- *Out of Stock*
  This tab will show all the products that are out of stock. We choose a matching icon from the icon font and set the icon color to the semantic `Negative` state so that it will appear in red.
- *Shortage*

This tab will show products that have less than 10 pieces remaining with a semantic `Critical` state that will make the icon appear in orange. The count of the number of low stock products will be displayed on the tab and the icon will appear in orange.

- *Plenty in Stock*
  This tab will show products that have more than 10 pieces in stock. The semantic `Positive` state will let the icon appear in green. As usual the UI texts for the tabs are linked to the resource bundle file and will be added later. Do not forget to set the standard CSS class `sapUiNoMarginTop` on the table to remove the spacing between the `IconTabBar` and the `table` and make the UI look nicer.

> **i Note**
>
> Each `IconTabFilter` element has a `key` property that is used to identify the tab that was pressed in the event handler `onQuickFilter` that is registered on the `IconTabBar` control directly. The event handler implementation does the actual filtering on the table and is defined in the controller.

webapp/controller/Worklist.controller.js

```
...
onInit: function() {
    var oViewModel,
        iOriginalBusyDelay,
        oTable = this.byId("table");
    // Put down worklist table's original value for busy indicator delay,
    // so it can be restored later on. Busy handling on the table is
    // taken care of by the table itself.
    iOriginalBusyDelay = oTable.getBusyIndicatorDelay();
    this._oTable = oTable;
    // keeps the search state
    this._oTableSearchState = [];
    // Model used to manipulate control states
    oViewModel = new JSONModel({
        worklistTableTitle:
this.getResourceBundle().getText("worklistTableTitle"),
        shareOnJamTitle: this.getResourceBundle().getText("worklistTitle"),
        shareSendEmailSubject:
this.getResourceBundle().getText("shareSendEmailWorklistSubject"),
        shareSendEmailMessage:
this.getResourceBundle().getText("shareSendEmailWorklistMessage",
[location.href]),
        tableNoDataText: this.getResourceBundle().getText("tableNoDataText"),
        tableBusyDelay: 0,
        inStock: 0,
        shortage: 0,
        outOfStock: 0,
        countAll: 0
    });
    this.setModel(oViewModel, "worklistView");
    // Create an object of filters
    this._mFilters = {
        "inStock": [new Filter("UnitsInStock", "GT", 10)],
        "outOfStock": [new Filter("UnitsInStock", "LE", 0)],
        "shortage": [new Filter("UnitsInStock", "BT", 1, 10)],
        "all": []
    };
    // Make sure, busy indication is showing immediately so there is no
    // break after the busy indication for loading the view's meta data is
    // ended (see promise 'oWhenMetadataIsLoaded' in AppController)
    oTable.attachEventOnce("updateFinished", function() {
        // Restore original busy indicator delay for worklist's table
        oViewModel.setProperty("/tableBusyDelay", iOriginalBusyDelay);
    });
},
```

```
...
```

As a preparation step for the filter tabs we add properties for the counters into the local view model of the worklist controller. We initialize the four values with `0` each. Furthermore, we create an object `_mFilters` that contains a filter for each tab. We will use the filters for filtering the table below the tabs. The properties in `_mFilters` correlate to the keys of the `IconTabFilter` controls we defined above in the `Worklist.view.xml` file. This way we can easily access a filter for a given tab based on the key of the corresponding tab.

Creating a simple filter requires a binding path as first parameter of the filter constructor (e.g. `"UnitsInStock"`), a filter operator (e.g. `"GT"`) as second argument, and a value to compare (e.g. `10`) as the third argument. We create such filters for all three tabs with different filter operators as described in the view part above. Additionally, we create an `all` filter, which is an empty array for clearing the binding again (when the user chooses the *All* tab).

webapp/controller/Worklist.controller.js

```
...
onUpdateFinished: function(oEvent) {
    // update the worklist's object counter after the table update
    var sTitle,
        oTable = oEvent.getSource(),
        oViewModel = this.getModel("worklistView"),
        iTotalItems = oEvent.getParameter("total");
    // only update the counter if the length is final and
    // the table is not empty
    if (iTotalItems && oTable.getBinding("items").isLengthFinal()) {
        sTitle = this.getResourceBundle().getText("worklistTableTitleCount",
[iTotalItems]);
        // Get the count for all the products and set the value to 'countAll'
property
        this.getModel().read("/Products/$count", {
            success: function (oData) {
                oViewModel.setProperty("/countAll", oData);
            }
        });
        // read the count for the unitsInStock filter
        this.getModel().read("/Products/$count", {
            success: function (oData) {
                oViewModel.setProperty("/inStock", oData);
            },
            filters: this._mFilters.inStock
        });
        // read the count for the outOfStock filter
        this.getModel().read("/Products/$count", {
            success: function(oData){
                oViewModel.setProperty("/outOfStock", oData);
            },
            filters: this._mFilters.outOfStock
        });
        // read the count for the shortage filter
        this.getModel().read("/Products/$count", {
            success: function(oData){
                oViewModel.setProperty("/shortage", oData);
            },
            filters: this._mFilters.shortage
        });
    } else {
        sTitle = this.getResourceBundle().getText("worklistTableTitle");
    }
    this.getModel("worklistView").setProperty("/worklistTableTitle", sTitle);
},
...
```

In the `onUpdateFinished` function, we get the count of all products by triggering a read operation on the model with the appropriate filter. The filter is a helper object of SAPUI5 that defines the condition for each tab on the data binding level. We already created the filters in the `onInit` function.

> **i** Note
>
> The `v2.ODataModel` will automatically bundle these `read` requests to one batch request to the server (if batch mode is enabled).

In the `success` handler of each `read` operation we update the corresponding property in the view model with the real count of the matching items that were returned by the service.

webapp/controller/Worklist.controller.js

```
...
_applySearch: function(oTableSearchState) {
    ...
},
/**
 * Event handler when a filter tab gets pressed
 * @param {sap.ui.base.Event} oEvent the filter tab event
 * @public
 */
onQuickFilter: function(oEvent) {
    var oBinding = this._oTable.getBinding("items"),
        sKey = oEvent.getParameter("selectedKey");
    oBinding.filter(this._mFilters[sKey]);
}
...
```

Next, we implement the handler for the `select` event of the `IconTabBar`. In this event handler we get a reference to the binding for the `items` aggregation of our `table` and store it in the variable `oBinding`. Then we read the parameter `selectedKey` from the `event` object to find out which tab has been selected. This `selectedKey` is used to get the correct filter for the selected tab. Next, we simply call the filter method on `oBinding` and pass the correct filter of the selected tab.

The filters are always applied as an array on the binding level, so you don't need to take care of managing the data, the data binding features of SAPUI5 will automatically take care.

webapp/i18n/i18n.properties

```
...

#XTIT: The title of the products quick filter
WorklistFilterProductsAll=Products

#XTIT: The title of the out of stock products filter
WorklistFilterOutOfStock=Out of Stock

#XTIT: The title of the low stock products filter
WorklistFilterShortage=Shortage

#XTIT: The title of the products in stock filter
WorklistFilterInStock=Plenty in Stock
#~~~ Object View ~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

We finally add the texts for the tab filters to the resource bundle. Copy the text definitions from the code section above to the end of the `Worklistn View` section in the i18n file.

Now run the app again and click the filter icons on top of the table. The products should be filtered according to the selection in the filter bar and the count should match the number of items displayed.

Related Information

API Reference: `sap.ui.model.ListBinding.filter`

Step 5: Adding Actions to the Worklist

Now we can easily spot shortages on our stock, but we would also like to take action and resolve it. Either we can decide to remove the product until the shortage is resolved or order new items of the product. In this step, we will add these actions to the footer of the worklist table.

Preview

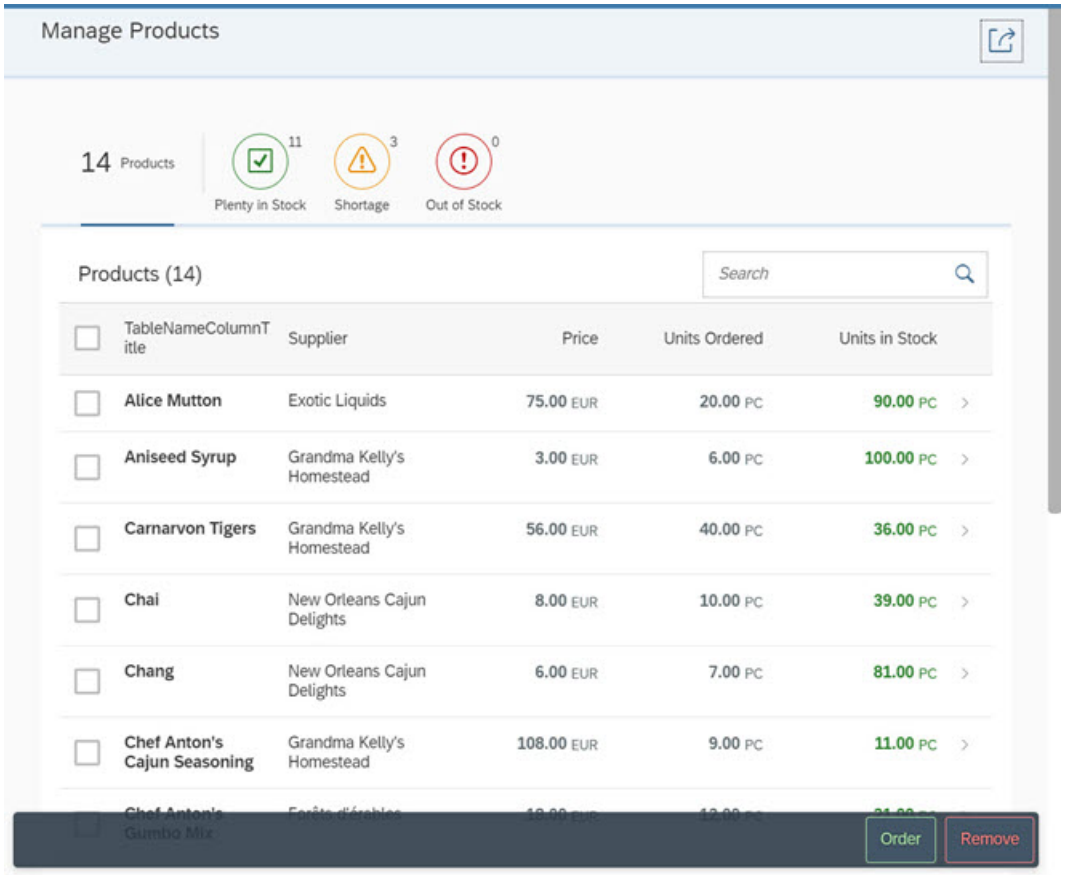


Figure 144: Actions are now available in the footer bar

Coding

You can view and download all files in the *Samples* in the Demo Kit at Worklist App - Step 5.

webapp/view/Worklist.view.xml

```
...
<Table
    id="table"
    busyIndicatorDelay="{worklistView>/tableBusyDelay}"
    growing="true"
    growingScrollToLoad="true"
    noDataText="{worklistView>/tableNoDataText}"
    updateFinished=".onUpdateFinished"
    width="auto"
    mode="MultiSelect"
    items="{
        path: '/Products',
        sorter: {
            path: 'ProductName',
            descending: false
        },
        parameters: {
            'expand': 'Supplier'
        }
    }">
...
```

We change the table `mode` to `MultiSelect`. This allows you to select multiple items in the table. Below, we will add two buttons to the footer bar of the screen. The first button will add to the `UnitsInStock` property, and the second will remove the selected products.

webapp/view/Worklist.view.xml

```
<mvc:View
     controllerName="mycompany.myapp.MyWorklistApp.controller.Worklist"
     xmlns="sap.m"
     xmlns:mvc="sap.ui.core.mvc"
     xmlns:semantic="sap.f.semantic">
     <semantic:SemanticPage
          id="page"
          headerPinnable="false"
          toggleHeaderOnTitleClick="false"
          showFooter="true">
          <semantic:titleHeading>
                <Title text="{i18n>worklistTitle}"/>
          </semantic:titleHeading>
          ...
     </semantic:content>
     <semantic:sendEmailAction>
          <semantic:SendEmailAction id="shareEmail" press=".onShareEmailPress"/>
     </semantic:sendEmailAction>
     <semantic:positiveAction>
          <semantic:PositiveAction text="{i18n>TableProductsReorder}"
press=".onUpdateStockObjects"/>
     </semantic:positiveAction>
     <semantic:negativeAction>
          <semantic:NegativeAction text="{i18n>TablePorductsUnlist}"
press=".onUnlistObjects"/>
     </semantic:negativeAction>
</semantic:SemanticPage>
...
```

Now we add the buttons to the footer bar of the page. The two semantic actions `Negative` and `Positive` will automatically be positioned in the footer bar. The first button will order new items of the selected products and the second one will remove them. The corresponding event handlers will be implemented in the controller.

webapp/controller/Worklist.controller.js

```
sap.ui.define([
    "./BaseController",
    "sap/ui/model/json/JSONModel",
    "myCompany/myApp/model/formatter",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    "sap/m/MessageToast",
    "sap/m/MessageBox"
], function(BaseController, JSONModel, formatter, Filter, FilterOperator,
MessageToast, MessageBox) {
    "use strict";
    return BaseController.extend("myCompany.myApp.controller.Worklist", {
        formatter: formatter,
...
        /**
         * Displays an error message dialog. The displayed dialog is content
density aware.
         * @param {string} sMsg The error message to be displayed
         * @private
         */
        _showErrorMessage: function(sMsg) {
            MessageBox.error(sMsg, {
                styleClass: this.getOwnerComponent().getContentDensityClass()
            });
        },
        /**
         * Event handler when a filter tab gets pressed
         * @param {sap.ui.base.Event} oEvent the filter tab event
         * @public
         */
        onQuickFilter: function(oEvent) {
            var oBinding = this._oTable.getBinding("items"),
                sKey = oEvent.getParameter("selectedKey");
            oBinding.filter(this._mFilters[sKey]);
        },
        /**
         * Error and success handler for the unlist action.
         * @param {string} sProductId the product ID for which this handler is
called
         * @param {boolean} bSuccess true in case of a success handler, else
false (for error handler)
         * @param {number} iRequestNumber the counter which specifies the
position of this request
         * @param {number} iTotalRequests the number of all requests sent
         * @private
         */
        _handleUnlistActionResult : function (sProductId, bSuccess,
iRequestNumber, iTotalRequests){
            // we could create a counter for successful and one for failed
requests
            // however, we just assume that every single request was successful
and display a success message once
            if (iRequestNumber === iTotalRequests) {

MessageToast.show(this.getModel("i18n").getResourceBundle().getText("StockRemoved
SuccessMsg", [iTotalRequests]));
            }
        },

        /**
         * Error and success handler for the reorder action.
```

```
         * @param {string} sProductId the product ID for which this handler is
called
         * @param {boolean} bSuccess true in case of a success handler, else
false (for error handler)
         * @param {number} iRequestNumber the counter which specifies the
position of this request
         * @param {number} iTotalRequests the number of all requests sent
         * @private
         */
        _handleReorderActionResult : function (sProductId, bSuccess,
iRequestNumber, iTotalRequests){
            // we could create a counter for successful and one for failed
requests
            // however, we just assume that every single request was successful
and display a success message once
            if (iRequestNumber === iTotalRequests) {

MessageToast.show(this.getModel("i18n").getResourceBundle().getText("StockUpdated
SuccessMsg", [iTotalRequests]));
            }
        },

        /**
         * Event handler for the unlist button. Will delete the
         * product from the (local) model.
         * @public
         */
        onUnlistObjects: function() {
            var aSelectedProducts, i, sPath, oProduct, oProductId;

            aSelectedProducts = this.byId("table").getSelectedItems();
            if (aSelectedProducts.length) {
                for (i = 0; i < aSelectedProducts.length; i++) {
                    oProduct = aSelectedProducts[i];
                    oProductId =
oProduct.getBindingContext().getProperty("ProductID");
                    sPath = oProduct.getBindingContext().getPath();
                    this.getModel().remove(sPath, {
                        success : this._handleUnlistActionResult.bind(this,
oProductId, true, i+1, aSelectedProducts.length),
                        error : this._handleUnlistActionResult.bind(this,
oProductId, false, i+1, aSelectedProducts.length)
                    });
                }
            } else {

this._showErrorMessage(this.getModel("i18n").getResourceBundle().getText("TableSe
lectProduct"));
            }
        },


        /**
         * Event handler for the reorder button. Will reorder the
         * product by updating the (local) model
         * @public
         */
        onUpdateStockObjects: function() {
            var aSelectedProducts, i, sPath, oProductObject;

            aSelectedProducts = this.byId("table").getSelectedItems();
            if (aSelectedProducts.length) {
                for (i = 0; i < aSelectedProducts.length; i++) {
                    sPath = aSelectedProducts[i].getBindingContext().getPath();
                    oProductObject =
aSelectedProducts[i].getBindingContext().getObject();
                    oProductObject.UnitsInStock += 10;
                    this.getModel().update(sPath, oProductObject, {
```

```
                        success : this._handleReorderActionResult.bind(this,
oProductObject.ProductID, true, i+1, aSelectedProducts.length),
                        error : this._handleReorderActionResult.bind(this,
oProductObject.ProductID, false, i+1, aSelectedProducts.length)
                    });
                }
            } else {

this._showErrorMessage(this.getModel("i18n").getResourceBundle().getText("TableSe
lectProduct"));
            }
        }
    });
});
```

Let's have a look at the implementation of the event handlers for the new actions. We first load the `sap.m.MessageToast` control as a new dependency to display a success message for the `unlist` and `reorder` actions.

Both actions are similar from an implementation perspective and the details are described below. They both loop over the selected items in the table and trigger a model update or deletion on the selected path. After that, a success message with the number of products processed is displayed. The table is updated automatically by the model change.

- *Order*
  For each of the selected items the binding path in the model is retrieved by calling the helper method `getBindingContextPath` on the selected item. Additionally, the data object from the model is fetched by calling `getBindingContext().getObject()` on the item. We update the data object and simply add `10` items to the stock to keep things simple in this example. Then we call the update function on the model with the product path and the new object. This will trigger an OData update request to the back end and a refresh of the model afterwards (multiple requests are handled together in batch mode). When the model refreshes, the table will be updated as well because of its binding.
- *Remove*
  For each of the selected items the binding path in the model is retrieved by calling the helper method `getBindingContextPath` on the selected item. Then, we call the `remove` function on the model with the product path. This triggers an OData `delete` request to the back end and a refresh of the OData model afterwards. Again, when the model is refreshed, the table will be updated as well because of its binding. The ODataModel v2 collects all these requests and only sends one batch request (this default behavior can be changed).

For each action we register both a `success` handler and an `error` handler. The `success` handler and `error` handler for each action is the same, but the function is called with different parameters. This allows us to use the same handler function for both the error and success case. Inside the corresponding handlers we simply display a success message once by comparing the current request number with the total number of requests. Furthermore, we assume that all of our requests always succeed.

In a real scenario, you could have a counter for error responses, and one for success responses. Finally, you could implement you own business logic for error and success cases, like displaying the number of failed and succeeded requests together with the corresponding product identified by the product ID parameter of the handlers. We don't do this to keep things simple.

> **i Note**
>
> In our example, the remove or order actions are only applied to items that are visible in the table, even if the *Select All* checkbox of the table is selected. Keep in mind that there may be more data on the back end that is currently not loaded, and therefore it is neither displayed and nor can it be selected by the user.

If you want to change this behavior, you might need to change both back-end and front-end code.

webapp/i18n/i18n.properties

```
...
#text of the button for Products reordering
TableProductsReorder=Order

#text for the button for Products unlisting
TablePorductsUnlist=Remove

#Text for no product selected
TableNoProductsSelected=No product selected

#Product successfully deleted
StockRemovedSuccessMsg=Product removed

#Product successfully updated
StockUpdatedSuccessMsg=Product stock level updated
#~~~ Object View ~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

Add the missing texts for the buttons and the message toast.

Save the changes and run the application again. Try the *Order* and *Remove* buttons with one or more products selected. The stock value will be increased or the product will be (temporarily) removed from the worklist table. Since all of our changes happen on a local mock server, we can simply reload the app to reset the data again.

Step 6: Extending the Detail Page

In this step, we will extend the detail page of our app to show more information of a given product with various UI controls. We will enrich the header area and display further attributes in an info panel for information about the supplier.

Preview

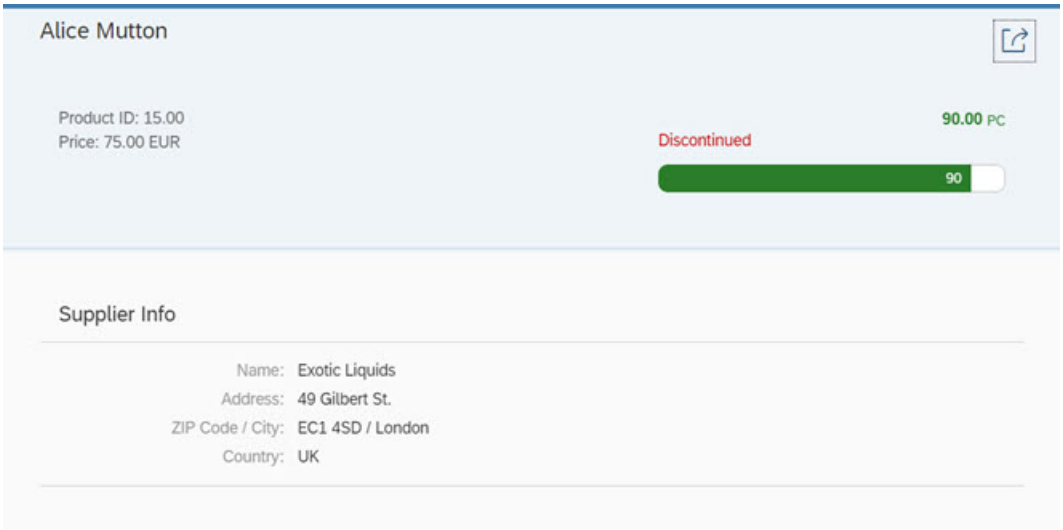


Figure 145: Detail page with more product information

Coding

You can view and download all files in the *Samples* in the Demo Kit at Worklist App - Step 6 .

webapp/view/Object.view.xml

```
<mvc:View
controllerName="mycompany.myapp.MyWorklistApp.controller.Object"
xmlns="sap.m"
xmlns:mvc="sap.ui.core.mvc"
xmlns:semantic="sap.f.semantic"
xmlns:form="sap.ui.layout.form">
<semantic:SemanticPage
    id="page"
    headerPinnable="false"
    toggleHeaderOnTitleClick="false"
    busy="{objectView>/busy}"
    busyIndicatorDelay="{objectView>/delay}">
    <semantic:titleHeading>
    <Title text="{ProductName}" />
    </semantic:titleHeading>
    <semantic:headerContent>
            <FlexBox
                alignItems="Start"
                justifyContent="SpaceBetween">
                <Panel backgroundDesign="Transparent">
                    <ObjectAttribute
```

```
                        title="{i18n>ObjectProductIdText}"
                        text="{
                        path: 'ProductID',
                        formatter: '.formatter.numberUnit'}"/>
                    <ObjectAttribute
                        title="{i18n>ObjectPriceTitle}"
                        text="{
                        path: 'UnitPrice',
                        formatter: '.formatter.numberUnit'} EUR"/>
                </Panel>
                <Panel backgroundDesign="Transparent">
                    <ObjectNumber
                        id="objectHeader"
                        unit="PC"
                        textAlign="End"
                        state="{
                        path: 'UnitsInStock',
                        formatter: '.formatter.quantityState'}"
                        number="{
                        path: 'UnitsInStock',
                        formatter: '.formatter.numberUnit'}">
                    </ObjectNumber>
                    <ObjectStatus
                        text="{i18n>ObjectDiscontinuedStatusText}"
                        state="Error"
                        visible="{path:'Discontinued'}"/>
                    <ProgressIndicator
                        width="300px"
                        percentValue="{UnitsInStock}"
                        displayValue="{UnitsInStock}"
                        showValue="true"
                        state="{
                        path: 'UnitsInStock',
                        formatter: '.formatter.quantityState'}"/>
                </Panel>
            </FlexBox>
    </semantic:headerContent>
    <semantic:content>
        <Panel
            class="sapUiNoContentPadding"
            headerText="{i18n>ObjectSupplierTabTitle}">
            <content>
                <form:SimpleForm
                    minWidth="1024"
                    maxContainerCols="2"
                    editable="false"
                    layout="ResponsiveGridLayout"
                    labelSpanL="3"
                    labelSpanM="3"
                    emptySpanL="4"
                    emptySpanM="4"
                    columnsL="1"
                    columnsM="1">
                    <form:content>
                        <Label text="{i18n>ObjectSupplierName}"/>
                        <Text text="{Supplier/CompanyName}"/>
                        <Label text="{i18n>ObjectSupplierAddress}"/>
                        <Text text="{Supplier/Address}"/>
                        <Label text="{i18n>ObjectSupplierZipcode} /
{i18n>ObjectSupplierCity}"/>
                        <Text text="{Supplier/PostalCode} / {Supplier/City}"/>
                        <Label text="{i18n>ObjectSupplierCountry}"/>
                        <Text text="{Supplier/Country}"/>
                    </form:content>
                </form:SimpleForm>
            </content>
        </Panel>
    </semantic:content>
```

```
    <semantic:sendEmailAction>
        <semantic:SendEmailAction id="shareEmail" press="onShareEmailPress"/>
    </semantic:sendEmailAction>
</semantic:SemanticPage>
</mvc:View>
```

We define a new `headerContent` section as well as some additional attributes for the product with two `sap.m.ObjectAttribute` controls, one for the `Price` and one for the `ProductID`. These are important product attributes for us, so we want to include them in our header area.

To get a better visual representation of the current stock of the shown product, we use the `ObjectStatus` and `ProgressIndicator` control statuses. If our product will not be produced anymore, the `ObjectStatus` shows up as *Discontinued*. The `ProgressIndicator` uses the same formatter function as our `UnitsInStock` (in the `state` of the `ObjectNumber`).

Below the object header we can use `sap.m.Panel` to display some additional information in a nice layout on the page. Inside the panel we use `sap.ui.layout.form.SimpleForm` to align the labels and texts we want to display.

webapp/i18n/i18n.properties

```
...
#Price per unit text
ObjectPriceTitle=Price

#Discontinued text
ObjectDiscontinuedStatusText=Discontinued

#Supplier tab title
ObjectSupplierTabTitle=Supplier Info

#Supplier company name
ObjectSupplierName=Name

#Supplier contact person name
ObjectSupplierContact=Contact

#Supplier contact address
ObjectSupplierAddress=Address

#Supplier zip code
ObjectSupplierZipcode=ZIP Code

#Supplier city name
ObjectSupplierCity=City

#Supplier country
ObjectSupplierCountry=Country

#Object Product ID text
ObjectProductIdText=Product ID
#~~~ Footer Options ~~~~~~~~~~~~~~~~~~~~~~
...
```

As before, we add new i18n texts to the resource bundle.

Save all the changes and run the application. Click on any product and see the product details displayed on the detail page.

Step 7: Adding a Comments Section

In this step, we extend the product detail view by adding a feature allowing to add comments to the product.

Preview

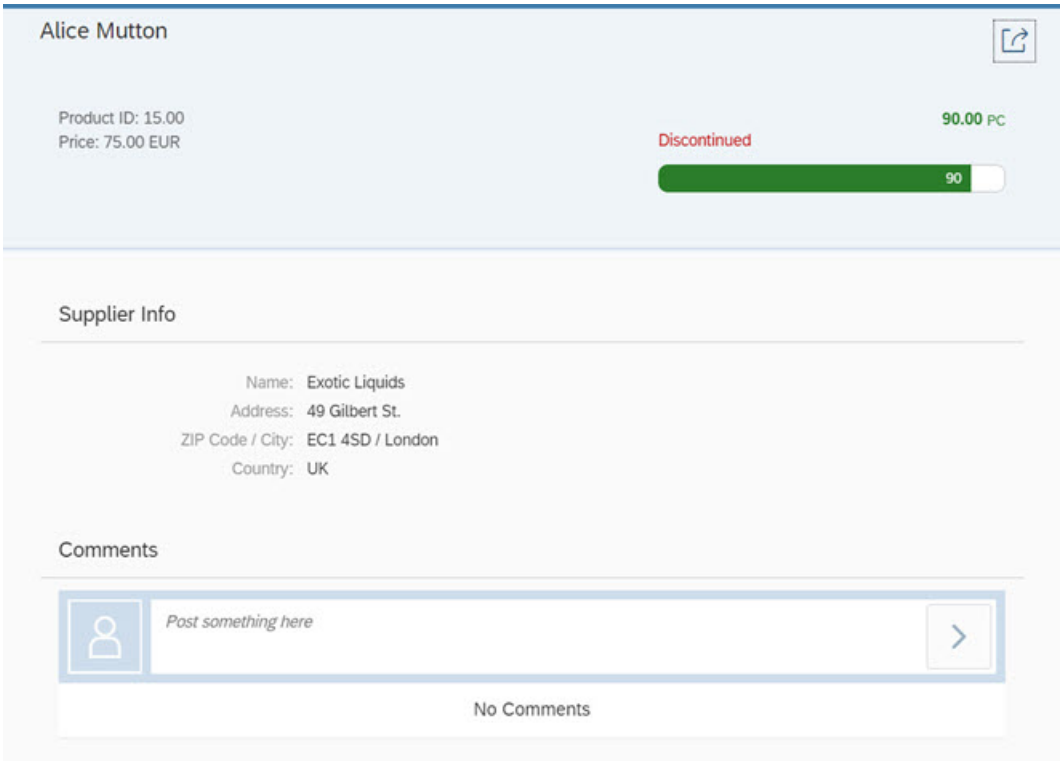


Figure 146: Comments section added to the detail page

Coding

You can view and download all files in the *Samples* in the Demo Kit at Worklist App - Step 7 .

webapp/view/Object.view.xml

```
<mvc:View
    controllerName="mycompany.myapp.MyWorklistApp.controller.Object"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:semantic="sap.f.semantic"
    xmlns:form="sap.ui.layout.form"
    xmlns:l="sap.ui.layout">
...
        <semantic:content>
            <l:VerticalLayout width="100%">
            <Panel
                backgroundDesign="Transparent"
                headerText="{i18n>ObjectSupplierTabTitle}">
...
```

```
            </Panel>
                <Panel
                    backgroundDesign="Transparent"
                    headerText="{i18n>ObjectCommentsTabTitle}">
                    <content>
                        <FeedInput post=".onPost"/>
                        <List
                            id="idCommentsList"
                            noDataText="{i18n>ObjectCommentNoData}"
                            showSeparators="Inner"
                            items="{
                                path: 'productFeedback>/productComments',
                                sorter: {
                                    path: 'date',
                                    descending: true
                                }
                            }">
                            <FeedListItem
                                info="{productFeedback>type}"
                                text="{productFeedback>comment}"
                                timestamp="{productFeedback>date}"/>
                        </List>
                    </content>
                </Panel>
            </l:VerticalLayout>
        </semantic:content>
```

Below the already existing panel, we add another panel that will serve as a container for our comments section. We put both panels inside a vertical layout, because `sap.f.semanticPage` allows only one control for content aggregation. Within the new panel, we add a `sap.m.FeedInput` control and attach an event handler `onPost` for the `post` event. This control will render an input field and a button, which allow users to post comments. The event handler we registered will be implemented below.

Below the `FeedInput` control, we add a list with all existing comments. The `items` aggregation of the list is bound to the `/productComments` property of the named model `productFeedback` that we will create below. All comments shall be displayed in descending order based on their publishing date. Therefore, we also configure a sorter for our items in the list.

The template for each row is a `FeedListItem` control. We configure the `FeedListItem` to simply display the date of the post, the text of the post itself, and the type of the post.

webapp/controller/Object.controller.js

```
...
/*global location*/
sap.ui.define([
   "myCompany/myApp/controller/BaseController",
   "sap/ui/model/json/JSONModel",
   "sap/ui/core/routing/History",
   "myCompany/myApp/model/formatter",
   "sap/ui/core/format/DateFormat",
   "sap/ui/model/Filter",
   "sap/ui/model/FilterOperator"
], function(BaseController, JSONModel, History, formatter, DateFormat, Filter,
FilterOperator) {
   "use strict";
   return BaseController.extend("myCompany.myApp.controller.Object", {
      formatter: formatter,
      ...
      _onBindingChange: function(oEvent) {
         ...
         // Update the comments in the list
         var oList = this.byId("idCommentsList");
         var oBinding = oList.getBinding("items");
```

```
            oBinding.filter(new Filter("productID", FilterOperator.EQ, sObjectId));
        },
        /**
        * Updates the model with the user comments on Products.
        * @function
        * @param {sap.ui.base.Event} oEvent object of the user input
        */
        onPost: function (oEvent) {
            var oFormat = DateFormat.getDateTimeInstance({style: "medium"});
            var sDate = oFormat.format(new Date());
            var oObject = this.getView().getBindingContext().getObject();
            var sValue = oEvent.getParameter("value");
            var oEntry = {
                productID: oObject.ProductID,
                type: "Comment",
                date: sDate,
                comment: sValue
            };
            // update model
            var oFeedbackModel = this.getModel("productFeedback");
            var aEntries = oFeedbackModel.getData().productComments;
            aEntries.push(oEntry);
            oFeedbackModel.setData({
               productComments : aEntries
            });
        }
    });
});
```

First, we add three new dependencies to the controller. We need these dependencies because we want to create a filter for the list and because we format the date and time of each post.

Whenever the binding of the detail view changes, we want to make sure that the comments for the current product are displayed. Therefore, we change the private function `_onBindingChange` and update the filter of the list that displays the comments by getting a reference to the binding of the `items` aggregation of our list and calling the `filter()` API afterwards. The filter is passed on to the `filter()` API. We use the `productID` as filter criterion, because we only want comments for a specific product.

Next, the event handler for the `post` event of the `FeedInput` is implemented. In the `onPost` handler, we create a new `entry` object that contains all data we want to store in our model. This data is the `productId`, the `type` of the post (hard-coded in our example), the current `date` in a medium date format, and the `comment` itself. The comment is retrieved from the event object. The `productId` is determined by calling `getObject()` on the view's binding context.

Finally, the new entry is added to the named model called `productFeedback`. This model does not exist yet, so let's create it next.

webapp/model/models.js

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/Device"
], function(JSONModel, Device) {
    "use strict";
    return {
        createDeviceModel: function() {
            var oModel = new JSONModel(Device);
            oModel.setDefaultBindingMode("OneWay");
            return oModel;
        },
        createCommentsModel: function() {
            return new JSONModel({ productComments : [] });
        }
```

```
    };
});
```

In both the object view (detail page) as well as in the corresponding controller we used a named model called `productFeedback`. In our example this model is a simple `JSONModel`. It is created in the function `createCommentsModel()` in the `model.js` file. As you can see above, the function simply returns a new instance of a `JSONModel` with a simple data object. The property `productComments` is an empty array and it will be updated every time someone posts a new comment.

However, this model is not yet accessible throughout our app. Let's fix this next.

webapp/Component.js

```
sap.ui.define([
   "sap/ui/core/UIComponent",
   "sap/ui/Device",
   "./model/models",
   "./controller/ErrorHandler"
], function(UIComponent, Device, models, ErrorHandler) {
   "use strict";
   return UIComponent.extend("myCompany.myApp.Component", {
      ...
      init: function() {
         // call the base component's init function
         UIComponent.prototype.init.apply(this, arguments);
         // initialize the error handler with the component
         this._oErrorHandler = new ErrorHandler(this);
         // set the device model
         this.setModel(models.createDeviceModel(), "device");

         // set the product feedback model
         this.setModel(models.createCommentsModel(), "productFeedback");
         // create the views based on the url/hash
         this.getRouter().initialize();
      },
      ...
   });
});
```

Now it's time to make the named model `productFeedback` available to our app. Therefore, just change the `init` function of our `Component.js` file by calling our `createCommentsModel()` method and setting the returned model on the component. After this, our model is accessible in our app.

webapp/i18n/i18n.properties

```
...
#Comments tab title
ObjectCommentsTabTitle=Comments

#No comments text
ObjectCommentNoData=No Comments
#~~~ Footer Options ~~~~~~~~~~~~~~~~~~~~~
...
```

Now add the new texts to our `i18n.properties` file and you're done.

You can test the new features by navigating to the details page of any given product. After that, just create a new comment for that product and post it.

Summary

You have learned how to use SAP Web IDE to create a simple worklist app from a template and you know where to find the code in the *Samples*. Based on the initial app you have seen how easy it can be to generate or download initial code and to extend it according to your own requirements. This tutorial also illustrated how to communicate easily with an OData back end using the OData V2 model. Furthermore, it illustrated how to use the mock server with both generated mock data and more realistic data.

SAP Fiori 2.0 App

In this tutorial, we showcase how to structure your SAPUI5 app using the layout patterns that comply with the latest SAP Fiori design guidelines.

The app provides the following features:

- An up-to-three-column layout based on the `sap.f.FlexibleColumnLayout` control. This layout has predefined layout types and defined routing between them that enables smooth navigation between the master-detail and master-detail-detail patterns of the app.
- A master page based on the `sap.f.DynamicPage` control that lists the available products and has filtering and sorting options.
- A detail page based on the `sap.uxap.ObjectPageLayout` control containing detailed information about the selected object from the master page:
  - It implements the dynamic header of the `ObjectPageLayout` control.
  - The `sap.f.Avatar` control is used in the title area to display an image of the selected product.
  - The header title area can be collapsed (snapped to the title) by scrolling down the content of the page or by clicking/tapping the title area. The header area can also be pinned so that it remains visible when the user scrolls down the content of the page.
  - The title area has a set of actions on the right. The title area can display specific content when the header is snapped.
  - The floating footer is positioned at the bottom of the page, on top of the page content. It holds finalizing actions on the right.
- A detail-detail page based on `sap.f.DynamicPage` to display further details of the selected object from the detail page.
- A simple about page based on `sap.f.DynamicPage` to display further details of the selected object from the detail-detail page.

Preview

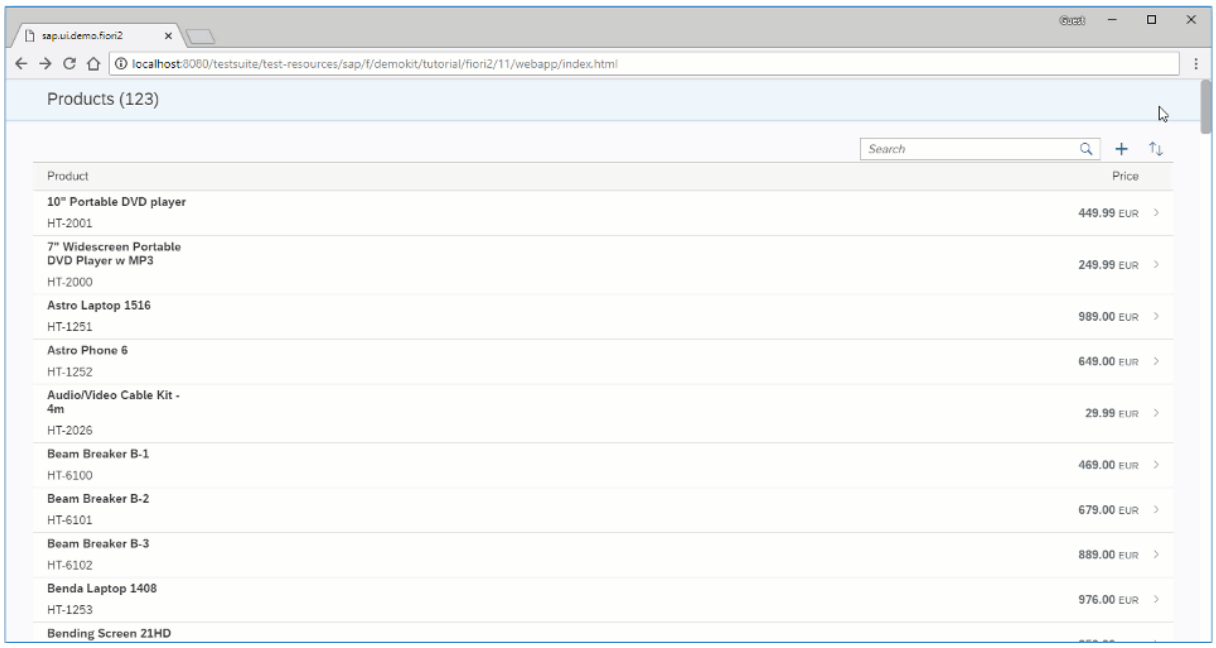


Figure 147: Master-detail-detail pattern with `sap.f.FlexibleColumnLayout`, `sap.f.DynamicPage` and `sap.uxap.ObjectPageLayout`

Choose your development environment

You can do this tutorial either with SAP Web IDE or choose your own development environment:

- If you use SAP Web IDE, you don't need to set up a development environment, a server and so on. All you need is a browser and an account for the SAP Cloud Platform. If you don't have an account yet, you can easily get a trial account. We recommend to continue with the SAP Web IDE as it has out-of-the-box support for SAPUI5 and because there is no setup overhead at all.
  In this case, you start with the template that is available in SAP Web IDE as described in step 1 of this tutorial.
- If you want to use your local development environment and deploy to any Web server of your choice, you can download the code for step 1 from the *Samples* in the Demo Kit at SAP Fiori 2.0 App - Step 1.

> → Tip
>
> You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, copy it to your workspace and make sure that the app runs by calling the `webapp/index.html` file.
>
> For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):
>
> - Downloading Code for a Tutorial Step [page 40]
> - Adapting Code to Your Development Environment [page 40]

Step 1: Setting Up the Initial App

We start by setting up a basic SAPUI5 app for this tutorial.

The structure in this step will be used throughout the rest of this tutorial. The basic SAPUI5 app created in this step will be extended in the subsequent steps to illustrate the main features of an SAP Fiori 2.0 app.

Prerequisites

You have set up your SAP Web IDE as described under App Development Using SAP Web IDE [page 44].

> **i Note**
>
> If your preferred development environment is not SAP Web IDE, you can skip the instructions below and set up your project for this tutorial by directly downloading the files for the initial app from the *Samples* in the Demo Kit at SAP Fiori 2.0 App - Step 1. Copy the code to your workspace and make sure that the app runs by calling the `webapp/index.html` file.

Preview

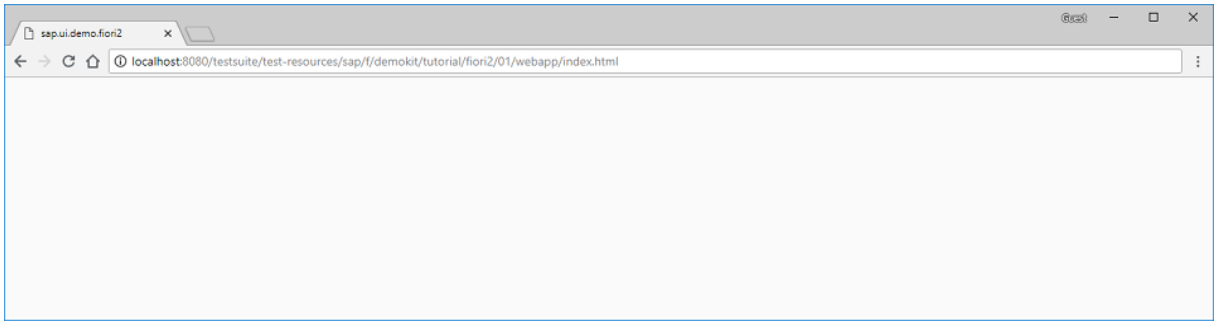
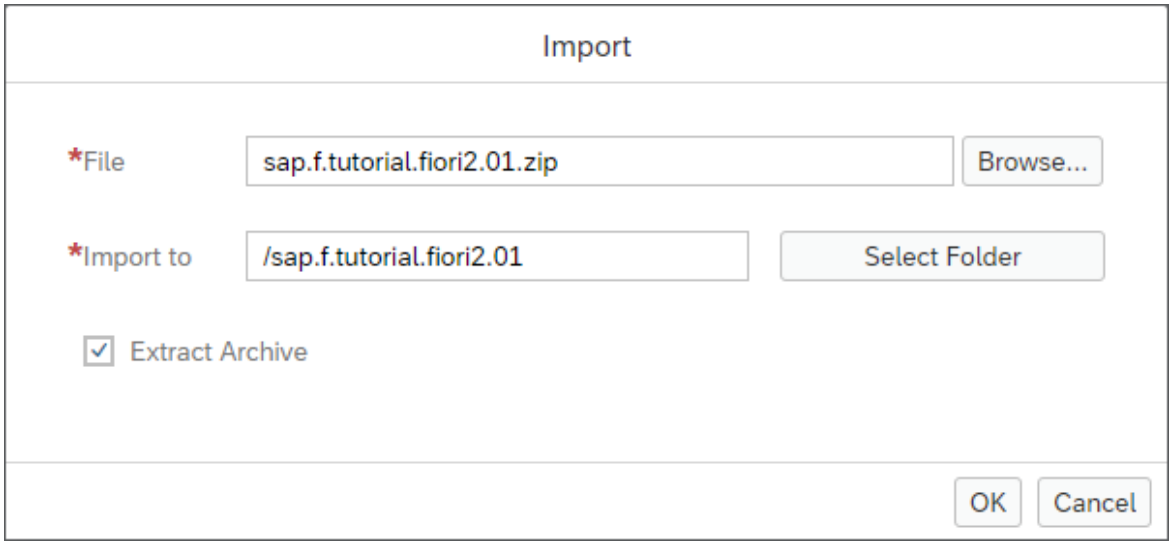


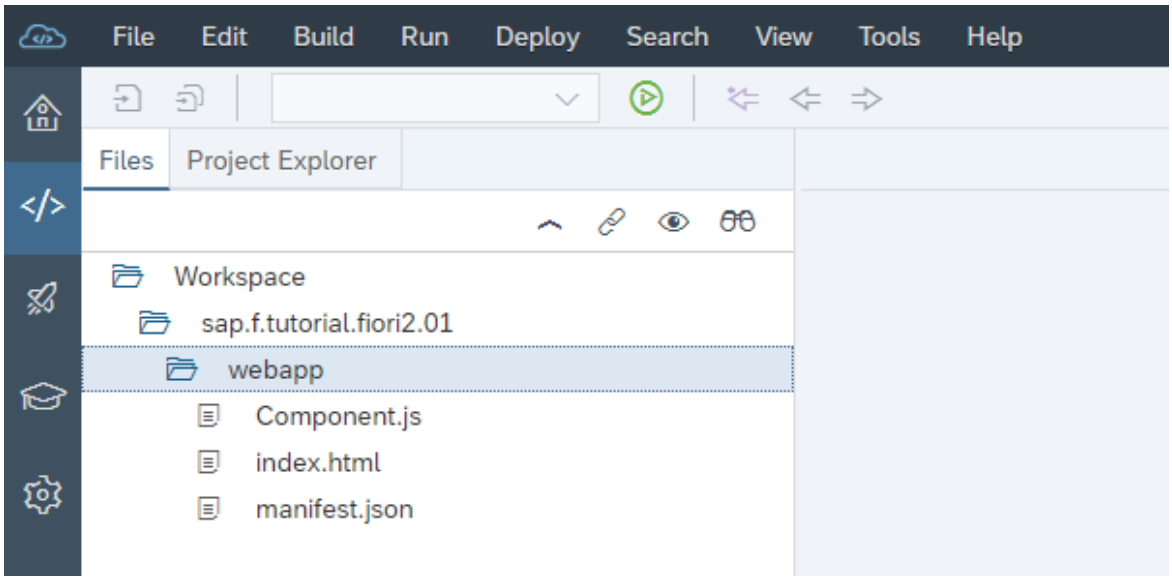
Figure 148: The basic SAPUI5 app

Create the Initial App Using the SAP Web IDE Import Functionality

1. Download the code for the initial app from the *Samples* in the Demo Kit at SAP Fiori 2.0 App - Step 1.
2. Launch SAP Web IDE.
3. Choose *File* > *Import* > *File or Project*
4. Choose *Browse* and select the downloaded `sap.f.tutorial.fiori2.01.zip` file, then choose *Open*.



5. Make sure *Extract Archive* is checked and choose *OK*.
   A new folder `sap.ui.demo.fiori2.01` is now available in your local workspace. It contains the following files and folders of the initial app:



6. Run the app by selecting the *webapp* folder and then *Run* > *Run as* > *Web Application*. Keep in mind that there is no content yet and the app appears as an empty page.

From now on, you can quickly run the app by selecting the root folder `sap.ui.demo.fiori2` of your project in SAP Web IDE and pressing the *Run* button. The system will automatically use the option from the *Run* menu that you chose last (in this case, the *Run index.html* option).

Step 2: Creating an Empty Flexible Column Layout

In this step, we add an instance of the `sap.f.FlexibleColumnLayout` in the main view of the app.

Preview

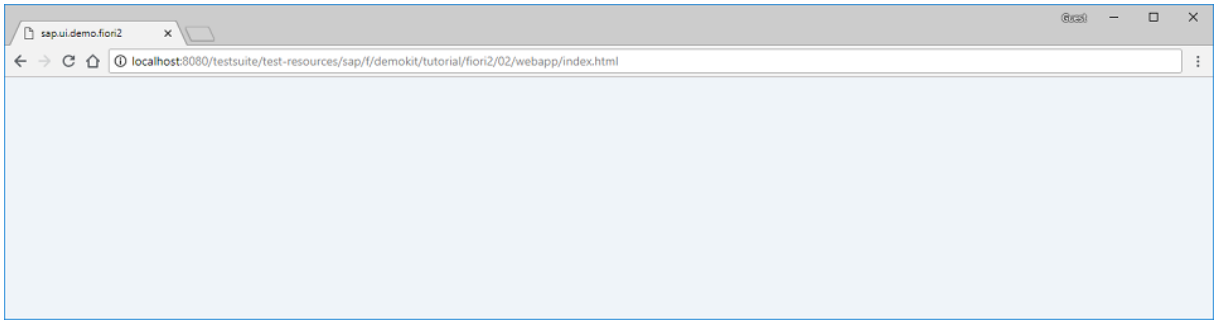


Figure 149: An empty instance of the **`sap.f.FlexibleColumnLayout`** control

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 2.

webapp/manifest.json [MODIFY]

```
{
    "_version": "1.12.0",
    "sap.app": {
        "id": "sap.ui.demo.fiori2",
        "type": "application",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui5": {
        "rootView": {
            "viewName": "sap.ui.demo.fiori2.view.App",
            "type": "XML",
            "async": true,
            "id": "fcl"
        },
        "dependencies": {
            "minUI5Version": "1.60.0",
            "libs": {
                "sap.ui.core": {},
                "sap.f": {}
            }
        },
        "config": {
            "fullWidth": true
        }
    }
```

```
}
```

First, we add the `sap.f` library as a dependency in the `manifest.json` file as most of the SAP Fiori 2.0 controls are in this library.

webapp/view/App.view.xml [NEW]

```
<mvc:View
    displayBlock="true"
    height="100%"
    xmlns="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
        <FlexibleColumnLayout id="flexibleColumnLayout"
backgroundDesign="Solid"></FlexibleColumnLayout>
</mvc:View>
```

We create a new `App.view.xml` that contains an instance of the `sap.f.FlexibleColumnLayout` control. Keep in mind that there is no content yet and the app appears as an empty page.

webapp/manifest.json [MODIFY]

```
{
    "_version": "1.12.0",
    "sap.app": {
        "id": "sap.ui.demo.fiori2",
        "type": "application",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui5": {
        "rootView": {
            "viewName": "sap.ui.demo.fiori2.view.App",
            "type": "XML",
            "async": false,
            "id": "fcl"
        },
        "dependencies": {
            "minUI5Version": "1.60.0",
            "libs": {
                "sap.ui.core": {},
                "sap.f": {}
            }
        },
        "config": {
            "fullWidth": true
        }
    }
}
```

We set the `rootView` to point to the created `App.view.xml`.

Step 3: Using Dynamic Page for the Master View

In this step, we create the master view of the app using `sap.f.DynamicPage` control.

Preview

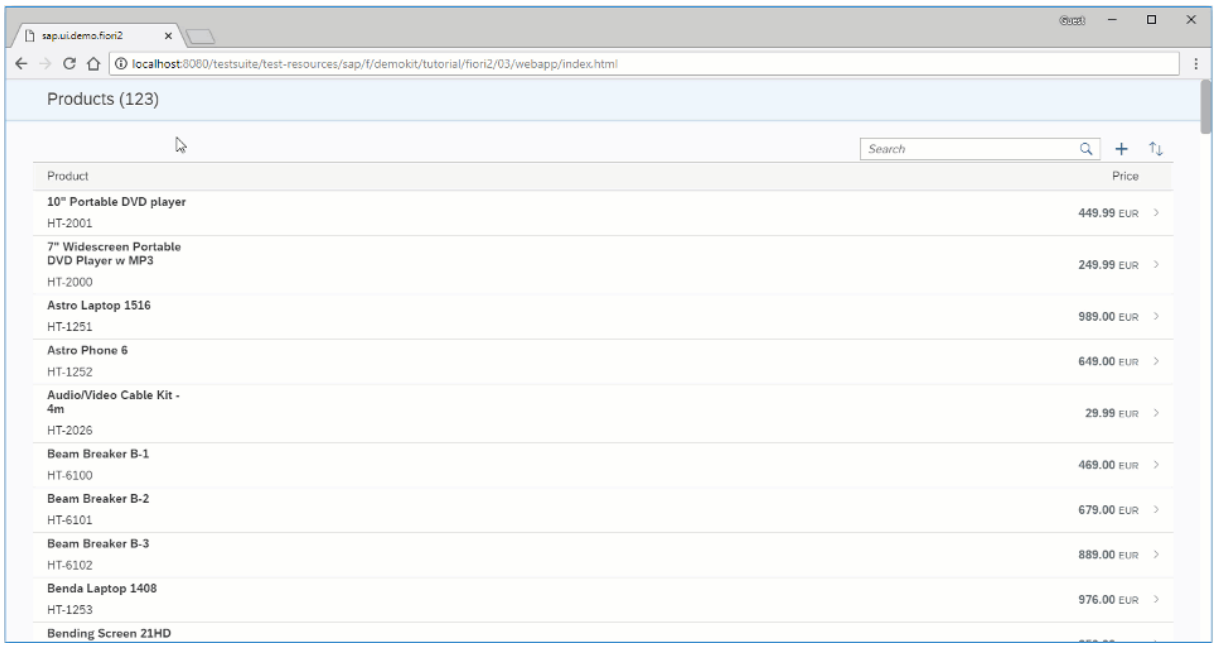


Figure 150: Master page with `sap.f.DynamicPage`

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 3.

webapp/manifest.json [MODIFY]

```
{
    "_version": "1.12.0",
    "sap.app": {
        "id": "sap.ui.demo.fiori2",
        "type": "application",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui5": {
        "rootView": {
            "viewName": "sap.ui.demo.fiori2.view.App",
            "type": "XML",
            "async": true,
```

```
            "id": "fcl"
        },
        "dependencies": {
            "minUI5Version": "1.60.0",
            "libs": {
                "sap.ui.core": {},
                "sap.m": {},
                "sap.f": {}
            }
        },
        "config": {
            "fullWidth": true
        }
    }
}
```

First, we add the `sap.m` library as a dependency in the `manifest.json`.

webapp/index.html [MODIFY]

```
    ...
    <script id="sap-ui-bootstrap"
        src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-resourceroots='{
            "sap.ui.demo.fiori2": "./",
            "sap.ui.demo.mock": "https://openui5.hana.ondemand.com/test-
resources/sap/ui/documentation/sdk/"
        }'
        data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
        data-sap-ui-compatVersion="edge"
        data-sap-ui-async="true"
        data-sap-ui-frameOptions="trusted">
    </script>
    ...
```

We add the link to the mock data that is used in the app.

webapp/Component.js [MODIFY]

```
sap.ui.define([
    'sap/ui/core/UIComponent',
    'sap/ui/model/json/JSONModel'
], function(UIComponent, JSONModel) {
    'use strict';
    return UIComponent.extend('sap.ui.demo.fiori2.Component', {
        metadata: {
            manifest: 'json'
        },

        init: function () {
            var oProductsModel;

            UIComponent.prototype.init.apply(this, arguments);

            // set products demo model on this sample
            oProductsModel = new JSONModel(sap.ui.require.toUrl('sap/ui/demo/
```

```
mock') + '/products.json');
            oProductsModel.setSizeLimit(1000);
            this.setModel(oProductsModel, 'products');
        }
    });
});
```

We create the `init` method in the `Component.js` to set the model.

webapp/view/Master.view.xml [NEW]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.Master"
    xmlns="sap.m"
    xmlns:f="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
    <f:DynamicPage id="dynamicPageId" toggleHeaderOnTitleClick="false">
        <!-- DynamicPage Title -->
        <f:title>
            <f:DynamicPageTitle>
                <f:heading>
                    <Title text="Products ({products>/ProductCollectionStats/
Counts/Total})"/>
                </f:heading>
            </f:DynamicPageTitle>
        </f:title>

        <!-- DynamicPage Content -->
        <f:content>
            <VBox fitContainer="true">
                <OverflowToolbar class="sapFDynamicPageAlignContent">
                    <ToolbarSpacer/>
                    <SearchField search=".onSearch" width="17.5rem"/>
                    <OverflowToolbarButton icon="sap-icon://add" text="Add"
type="Transparent" press=".onAdd"/>
                    <OverflowToolbarButton icon="sap-icon://sort" text="Sort"
type="Transparent" press=".onSort"/>
                </OverflowToolbar>
                <Table
                    id="productsTable"
                    inset="false"
                    items="{
                        path: 'products>/ProductCollection',
                        sorter: {
                            path: 'Name'
                        }
                    }"
                    class="sapFDynamicPageAlignContent"
                    width="auto">
                    <columns>
                        <Column width="12em">
                            <Text text="Product"/>
                        </Column>
                        <Column    hAlign="End">
                            <Text text="Price"/>
                        </Column>
                    </columns>
                    <items>
                        <ColumnListItem type="Navigation">
                            <cells>
                                <ObjectIdentifier title="{products>Name}"
text="{products>ProductId}"/>
                                <ObjectNumber
```

```
                                    number="{
                                        parts:[
                                            {path:'products>Price'},
                                            {path:'products>CurrencyCode'}
                                        ],
                                        type: 'sap.ui.model.type.Currency',
                                        formatOptions: {showMeasure: false}
                                    }"
                                    unit="{products>CurrencyCode}"/>
                            </cells>
                        </ColumnListItem>
                    </items>
                </Table>
            </VBox>
        </f:content>

        <!-- DynamicPage Footer -->
        <f:footer>
            <OverflowToolbar>
                <ToolbarSpacer/>
                <Button type="Accept" text="Accept"/>
                <Button type="Reject" text="Reject"/>
            </OverflowToolbar>
        </f:footer>
    </f:DynamicPage>
</mvc:View>
```

We create the master view using `sap.f.DynamicPage`. The page consists of a list with all products.

webapp/view/App.view.xml [MODIFY]

```
<mvc:View
    displayBlock="true"
    height="100%"
    xmlns="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
    <FlexibleColumnLayout id="flexibleColumnLayout" backgroundDesign="Solid">
        <beginColumnPages>
            <mvc:XMLView id="beginView"
viewName="sap.ui.demo.fiori2.view.Master"/>
        </beginColumnPages>
    </FlexibleColumnLayout>
</mvc:View>
```

We add the master view in `FlexibleColumnLayout's beginColumnPages` aggregation.

webapp/controller/Master.controller.js [NEW]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    'sap/ui/model/Sorter',
    'sap/m/MessageBox'
], function (JSONModel, Controller, Filter, FilterOperator, Sorter, MessageBox) {
    "use strict";
```

```
    return Controller.extend("sap.ui.demo.fiori2.controller.Master", {
        onInit: function () {
            this.oView = this.getView();
            this._bDescendingSort = false;
            this.oProductsTable = this.oView.byId("productsTable");
        },

        onSearch: function (oEvent) {
            var oTableSearchState = [],
                sQuery = oEvent.getParameter("query");

            if (sQuery && sQuery.length > 0) {
                oTableSearchState = [new Filter("Name", FilterOperator.Contains,
sQuery)];
            }

            this.oProductsTable.getBinding("items").filter(oTableSearchState,
"Application");
        },

        onAdd: function () {
            MessageBox.information("This functionality is not ready yet.",
{title: "Aw, Snap!"});
        },

        onSort: function () {
            this._bDescendingSort = !this._bDescendingSort;
            var oBinding = this.oProductsTable.getBinding("items"),
                oSorter = new Sorter("Name", this._bDescendingSort);

            oBinding.sort(oSorter);
        }
    });
});
```

We create the master controller that provides a basic search and sort functionality for the products listed in the master page.

Step 4: Adding a Detail Page

In this step, we add an empty detail page.

Preview

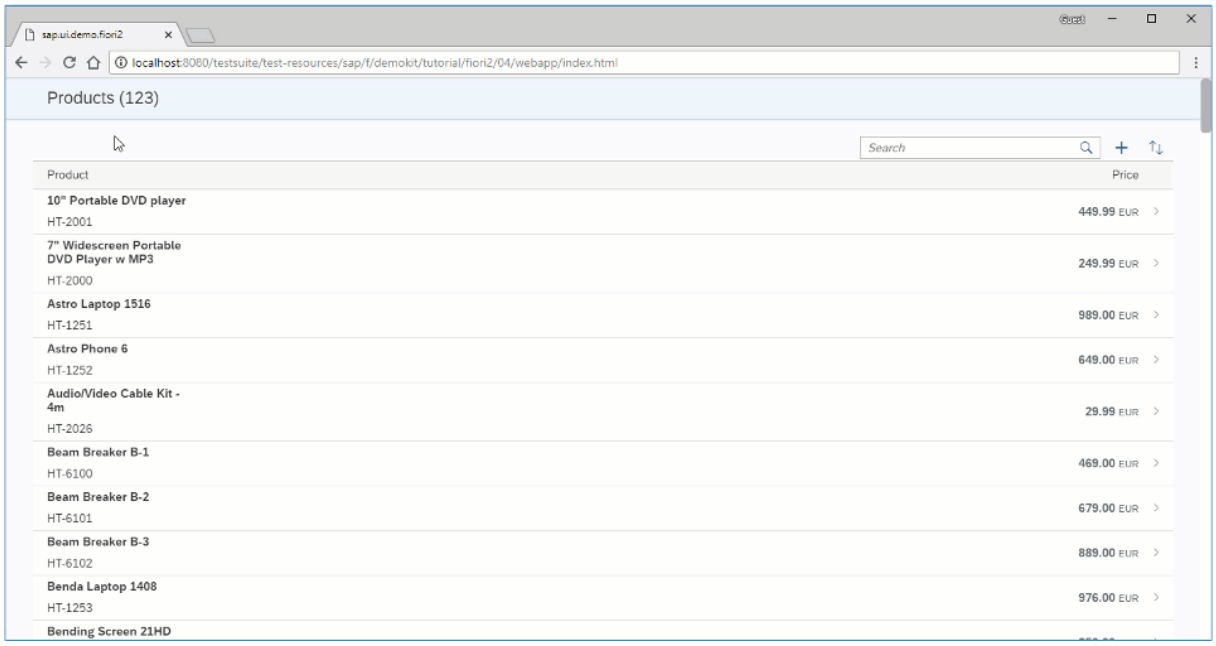


Figure 151: Master Page with Empty Detail Page

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 4.

webapp/view/Detail.view.xml [NEW]

```
<mvc:View
    xmlns:mvc="sap.ui.core.mvc">
</mvc:View>
```

First, we create a blank detail page.

webapp/view/App.view.xml [MODIFY]

```
<mvc:View
```

```
    displayBlock="true"
    height="100%"
    xmlns="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
    <FlexibleColumnLayout id="flexibleColumnLayout" backgroundDesign="Solid">
        <beginColumnPages>
            <mvc:XMLView id="beginView"
viewName="sap.ui.demo.fiori2.view.Master"/>
        </beginColumnPages>
        <midColumnPages>
            <mvc:XMLView id="detailView"
viewName="sap.ui.demo.fiori2.view.Detail"/>
        </midColumnPages>
    </FlexibleColumnLayout>
</mvc:View>
```

We add the detail page in `FlexibleColumnLayout's midColumnPages` aggregation in the `App.view.xml` file. This way the detail page will be displayed in the middle column.

webapp/view/Master.view.xml [MODIFY]

```
        ...
        <!-- DynamicPage Content -->
                    ...
                    ...
                    <items>
                        <ColumnListItem type="Navigation"
press=".onListItemPress">
                            <cells>
                                <ObjectIdentifier title="{products>Name}"
text="{products>ProductId}"/>
                                <ObjectNumber
                                    number="{
                                        parts:[
                                            {path:'products>Price'},
                                            {path:'products>CurrencyCode'}
                                        ],
                                        type: 'sap.ui.model.type.Currency',
                                        formatOptions: {showMeasure: false}
                                    }"
                                    unit="{products>CurrencyCode}"/>
                            </cells>
                        </ColumnListItem>
                    </items>
                    ...
```

We add a `press` handler to each `ColumnListItem` in the `Master.view.xml`.

webapp/controller/Master.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    'sap/ui/model/Sorter',
```

```
    'sap/m/MessageBox',
    'sap/f/library'
], function (JSONModel, Controller, Filter, FilterOperator, Sorter, MessageBox,
fioriLibrary) {
    "use strict";
        ...
        ...
        onSort: function () {
            this._bDescendingSort = !this._bDescendingSort;
            var oBinding = this.oProductsTable.getBinding("items"),
                oSorter = new Sorter("Name", this._bDescendingSort);
            oBinding.sort(oSorter);
        },

        onListItemPress: function () {
            var oFCL = this.oView.getParent().getParent();

            oFCL.setLayout(fioriLibrary.LayoutType.TwoColumnsMidExpanded);
        }
    });
});
```

In the `Master.controller.js`, we attach a `onListItemPress` function to the `press` handler, which changes the `layout` to `TwoColumnsBeginExpanded`. This means that there are going to be two columns, where the first one is larger than the second. For more information on the available layout types, see Types of Layout [page 2290].

Step 5: Using Object Page Layout as a Detail Page

In this step, we add `sap.uxap.ObjectPageLayout` to the detail page to display more information about each product.

The `ObjectPageLayout` control provides a layout that allows apps to easily display information related to a business object.

As of version 1.52, the control can have the same dynamic header that is used in the `sap.f.DynamicPage`. This ensures the availability of all the latest SAP Fiori 2.0 features, such as, having breadcrumbs navigation, navigation actions, and expanding/collapsing the header by tapping/clicking the title area or by selecting the available arrow buttons. For more information, see Object Page Headers [page 2488].

Compared to `sap.f.DynamicPage`, the `sap.uxap.ObjectPageLayout` can provide a more structured page content using an optional anchor bar and block content wrapped in sections and subsections that structure the information. For more information, see Object Page Layout [page 2482].

Preview

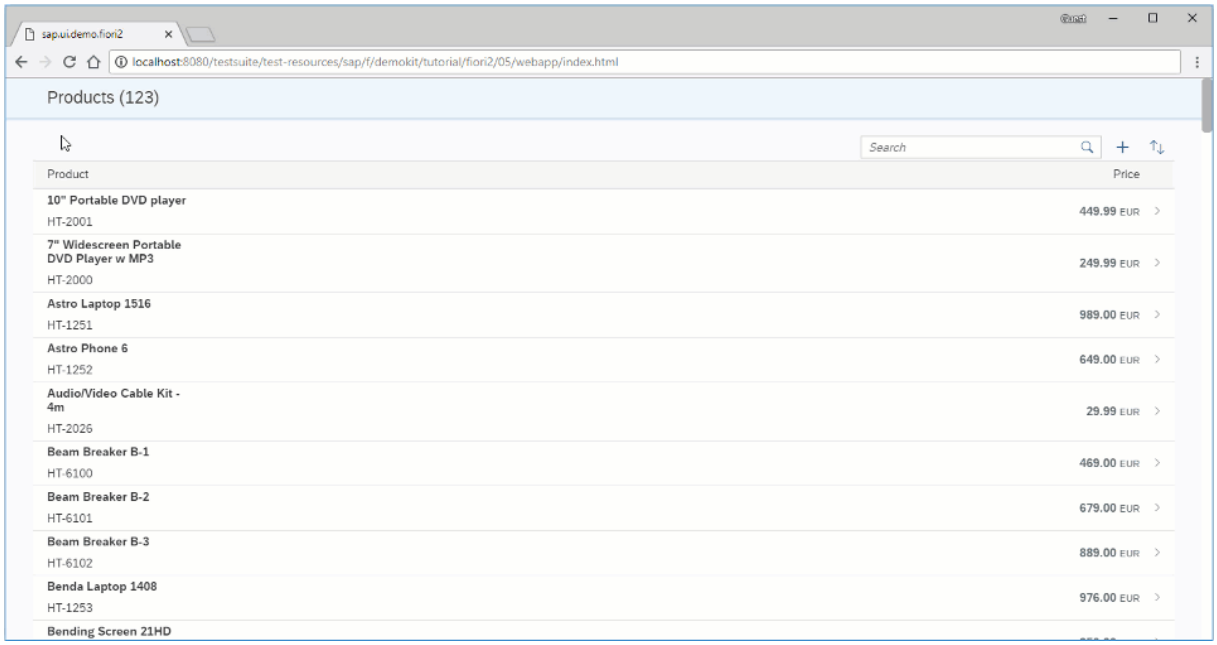


Figure 152: `ObjectPageLayout` with dynamic header for the detail page

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 5.

webapp/manifest.json [MODIFY]

```
{
    "_version": "1.12.0",
    "sap.app": {
        "id": "sap.ui.demo.fiori2",
        "type": "application",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui5": {
        "rootView": {
            "viewName": "sap.ui.demo.fiori2.view.App",
            "type": "XML",
            "async": true,
            "id": "fcl"
        },
        "dependencies": {
            "minUI5Version": "1.60.0",
            "libs": {
                "sap.ui.core": {},
                "sap.m": {},
                "sap.f": {},
                "sap.uxap": {}
```

```
            }
        },
        "config": {
            "fullWidth": true
        }
    }
}
```

First, we add the related libraries that we will need to the dependencies in the `manifest.json`.

webapp/view/Detail.view.xml [MODIFY]

```
<mvc:View
    xmlns="sap.uxap"
    xmlns:m="sap.m"
    xmlns:f="sap.f"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc">
    <ObjectPageLayout
        id="ObjectPageLayout"
        showTitleInHeaderContent="true"
        alwaysShowContentHeader="false"
        preserveHeaderStateOnScroll="false"
        headerContentPinnable="true"
        isChildPage="true"
        upperCaseAnchorBar="false">
    </ObjectPageLayout>
</mvc:View>
```

We add an instance of the `sap.uxap.ObjectPageLayout` control.

webapp/view/Detail.view.xml [MODIFY]

```
<mvc:View
    xmlns="sap.uxap"
    xmlns:m="sap.m"
    xmlns:f="sap.f"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc">
    <ObjectPageLayout
        id="ObjectPageLayout"
        showTitleInHeaderContent="true"
        alwaysShowContentHeader="false"
        preserveHeaderStateOnScroll="false"
        headerContentPinnable="true"
        isChildPage="true"
        upperCaseAnchorBar="false">
        <headerTitle>
            <ObjectPageDynamicHeaderTitle>
                <actions>
                    <m:ToggleButton
                        text="Edit"
                        type="Emphasized"/>
                    <m:Button
                        text="Delete"
                        type="Transparent"/>
                    <m:Button
```

```
                        text="Copy"
                        type="Transparent"/>
                    <m:Button
                        icon="sap-icon://action"
                        type="Transparent"/>
                </actions>
            </ObjectPageDynamicHeaderTitle>
        </headerTitle>
    </ObjectPageLayout>
</mvc:View>
```

We add the recommended dynamic header with an instance of the `ObjectPageDynamicHeaderTitle` in the `headerTitle` aggregation of the `ObjectPageLayout`. For more information, see Object Page Dynamic Header [page 2494] and Object Page Headers Comparison [page 2497].

webapp/view/Detail.view.xml [MODIFY]

```
        ...
        <headerTitle>
            <ObjectPageDynamicHeaderTitle>
                <actions>
                    <m:ToggleButton
                        text="Edit"
                        type="Emphasized"/>
                    <m:Button
                        text="Delete"
                        type="Transparent"/>
                    <m:Button
                        text="Copy"
                        type="Transparent"/>
                    <m:Button
                        icon="sap-icon://action"
                        type="Transparent"/>
                </actions>
            </ObjectPageDynamicHeaderTitle>
        </headerTitle>
        <headerContent>
            <m:FlexBox wrap="Wrap" fitContainer="true" alignItems="Stretch">
                <f:Avatar
                    displaySize="L"
                    displayShape="Square"
                    class="sapUiTinyMarginEnd">
                </f:Avatar>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Main Category"/>
                </m:VBox>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Subcategory"/>
                </m:VBox>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Price"/>
                </m:VBox>
            </m:FlexBox>
        </headerContent>
    </ObjectPageLayout>
</mvc:View>
```

We add content in the `headerContent` aggregation. We're using `sap.f.Avatar` as the recommended SAP Fiori 2.0 control for displaying images.

webapp/view/Detail.view.xml [MODIFY]

```
        ...
        <headerContent>
            <m:FlexBox wrap="Wrap" fitContainer="true" alignItems="Stretch">
                <f:Avatar
                    displaySize="L"
                    displayShape="Square"
                    class="sapUiTinyMarginEnd">
                </f:Avatar>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Main Category"/>
                </m:VBox>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Subcategory"/>
                </m:VBox>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Price"/>
                </m:VBox>
            </m:FlexBox>
        </headerContent>
        <sections>
            <ObjectPageSection title="General Information">
                <subSections>
                    <ObjectPageSubSection>
                        <blocks>
                            <form:SimpleForm
                                maxContainerCols="2"
                                editable="false"
                                layout="ResponsiveGridLayout"
                                labelSpanL="12"
                                labelSpanM="12"
                                emptySpanL="0"
                                emptySpanM="0"
                                columnsL="1"
                                columnsM="1">
                                <form:content>
                                    <m:Label text="Product ID"/>
                                    <m:Label text="Description"/>
                                    <m:Label text="Supplier"/>
                                </form:content>
                            </form:SimpleForm>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>

            <ObjectPageSection title="Suppliers">
                <subSections>
                    <ObjectPageSubSection>
                        <blocks>
                            <m:Table
                                id="suppliersTable"
                                items="{path : 'products>/ProductCollectionStats/
Filters/1/values'}">
                                <m:columns>
                                    <m:Column/>
                                </m:columns>
                                <m:items>
                                    <m:ColumnListItem type="Navigation">
                                        <m:cells>
                                            <m:ObjectIdentifier
text="{products>text}"/>
                                        </m:cells>
                                    </m:ColumnListItem>
                                </m:items>
```

```
                            </m:Table>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>
        </sections>
    </ObjectPageLayout>
</mvc:View>
```

Finally, we add page content in two separate sections with blocks. For more information, see Object Page Blocks [page 2504].

Step 6: Adding a Floating Footer

In this step, we add a floating footer to the detail page.

Preview

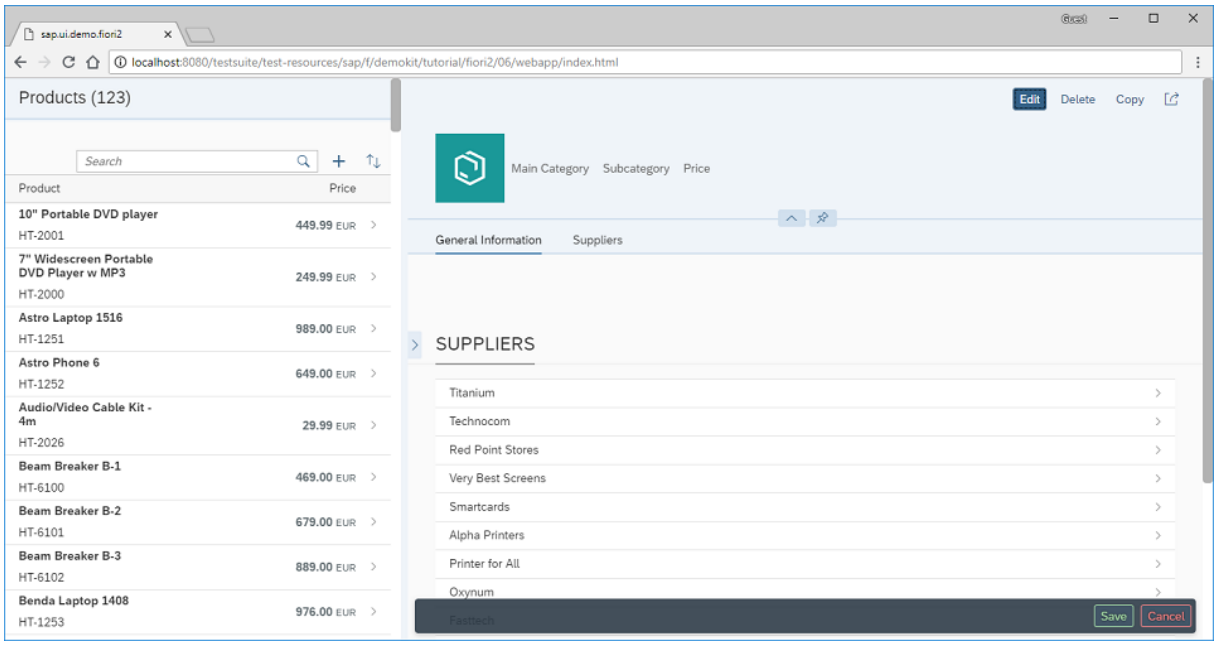


Figure 153: `ObjectPageLayout` with a floating footer

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 6.

webapp/view/App.view.xml [MODIFY]

```
<mvc:View
    displayBlock="true"
    height="100%"
    xmlns="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
    <FlexibleColumnLayout id="flexibleColumnLayout" stateChange="onStateChanged"
backgroundDesign="Solid">
        <beginColumnPages>
            <mvc:XMLView id="beginView"
viewName="sap.ui.demo.fiori2.view.Master"/>
        </beginColumnPages>
        <midColumnPages>
            <mvc:XMLView id="detailView"
viewName="sap.ui.demo.fiori2.view.Detail"/>
        </midColumnPages>
    </FlexibleColumnLayout>
</mvc:View>
```

First, we communicate changes to the layout with the use of the `stateChange` event.

webapp/view/Detail.view.xml [MODIFY]

```
        ...
        </sections>
        <footer>
            <m:OverflowToolbar>
                <m:ToolbarSpacer/>
                <m:Button type="Accept" text="Save"/>
                <m:Button type="Reject" text="Cancel"/>
            </m:OverflowToolbar>
        </footer>
    </ObjectPageLayout>
</mvc:View>
```

We add a footer inside the `sap.uxap.ObjectPageLayout`.

webapp/view/Detail.view.xml [MODIFY]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.Detail"
    xmlns="sap.uxap"
    xmlns:m="sap.m"
    xmlns:f="sap.f"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc">
    <ObjectPageLayout
        id="ObjectPageLayout"
        showTitleInHeaderContent="true"
        alwaysShowContentHeader="false"
        preserveHeaderStateOnScroll="false"
        headerContentPinnable="true"
        isChildPage="true"
```

```
        upperCaseAnchorBar="false">
        <headerTitle>
            <ObjectPageDynamicHeaderTitle>
                <actions>
                    <m:ToggleButton
                        text="Edit"
                        type="Emphasized"
                        press=".onEditToggleButtonPress"/>
                    <m:Button
                        text="Delete"
                        type="Transparent"/>
                    <m:Button
                        text="Copy"
                        type="Transparent"/>
                    <m:Button
                        icon="sap-icon://action"
                        type="Transparent"/>
                </actions>
            </ObjectPageDynamicHeaderTitle>
        </headerTitle>
        ...
```

We add a `press` event handler to the *Edit* button.

webapp/controller/Detail.controller.js [NEW]

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function (Controller) {
    "use strict";

    return Controller.extend("sap.ui.demo.fiori2.controller.Detail", {
        onEditToggleButtonPress: function() {
            var oObjectPage = this.getView().byId("ObjectPageLayout"),
                bCurrentShowFooterState = oObjectPage.getShowFooter();

            oObjectPage.setShowFooter(!bCurrentShowFooterState);
        }
    });
});
```

Finally, we create the controller and add a simple function to it to show and hide the footer.

Step 7: Routing

In this step, we utilize the `sap.f.routing.Router`.

Preview

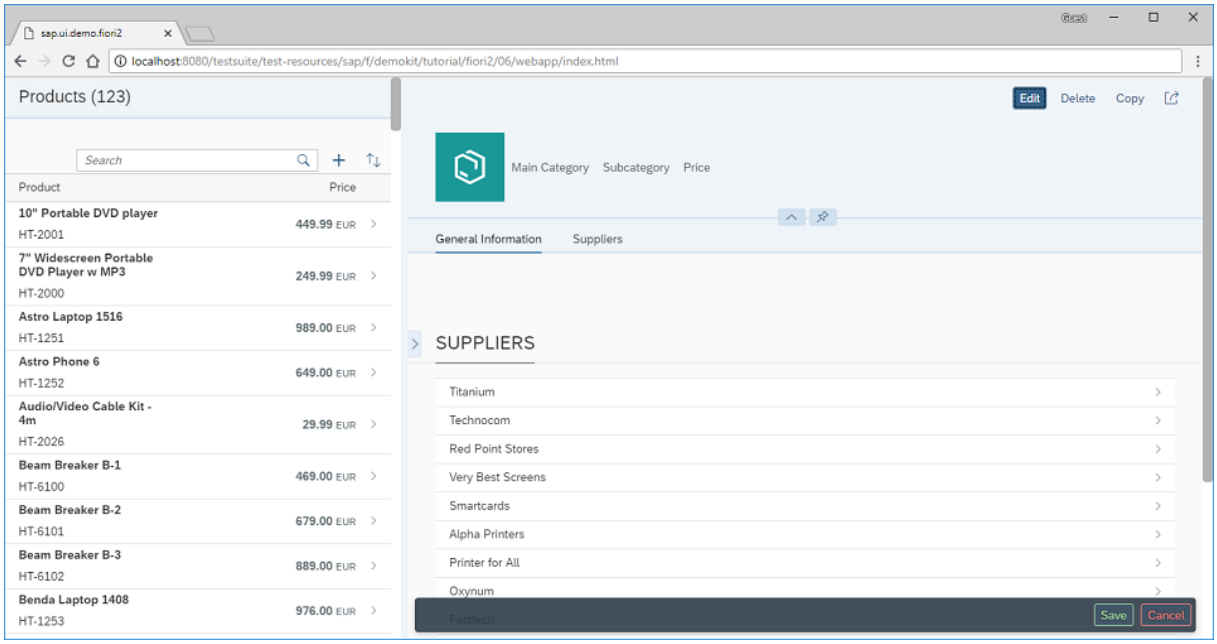


Figure 154: Changing layouts based on the **`sap.f.routing.Router`** (no visual changes to last step)

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 7.

webapp/views/App.view.xml [MODIFY]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.App"
    displayBlock="true"
    height="100%"
    xmlns="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
    <FlexibleColumnLayout
        id="flexibleColumnLayout"
        stateChange=".onStateChanged"
        backgroundDesign="Solid"
        layout="{/layout}"/>
</mvc:View>
```

We remove the hardcoded `beginColumnPages` and `endColumnPages` aggregations (since the router will add them automatically from now on), and we bind the `layout` property so that it can be changed easily from the controller.

webapp/controller/App.controller.js [NEW]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller"
], function (JSONModel, Controller) {
    "use strict";

    return Controller.extend("sap.ui.demo.fiori2.controller.App", {
        onInit: function () {
            this.oOwnerComponent = this.getOwnerComponent();
            this.oRouter = this.oOwnerComponent.getRouter();
            this.oRouter.attachRouteMatched(this.onRouteMatched, this);
        },

        onRouteMatched: function (oEvent) {
            var sRouteName = oEvent.getParameter("name"),
                oArguments = oEvent.getParameter("arguments");

            // Save the current route name
            this.currentRouteName = sRouteName;
            this.currentProduct = oArguments.product;
        },

        onStateChanged: function (oEvent) {
            var bIsNavigationArrow = oEvent.getParameter("isNavigationArrow"),
                sLayout = oEvent.getParameter("layout");

            // Replace the URL with the new layout if a navigation arrow was used
            if (bIsNavigationArrow) {
                this.oRouter.navTo(this.currentRouteName, {layout: sLayout,
product: this.currentProduct}, true);
            }
        },

        onExit: function () {
            this.oRouter.detachRouteMatched(this.onRouteMatched, this);
        }
    });
});
```

We access the router and bind to its `routeMatched` event. For more information, see Router [page 2296].

webapp/controller/Master.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    'sap/ui/model/Sorter',
    'sap/m/MessageBox',
```

```
    'sap/f/library'
], function (JSONModel, Controller, Filter, FilterOperator, Sorter, MessageBox,
fioriLibrary) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.Master", {
        onInit: function () {
            this.oView = this.getView();
            this._bDescendingSort = false;
            this.oProductsTable = this.oView.byId("productsTable");
            this.oRouter = this.getOwnerComponent().getRouter();
        },
        onSearch: function (oEvent) {
            var oTableSearchState = [],
                sQuery = oEvent.getParameter("query");
            if (sQuery && sQuery.length > 0) {
                oTableSearchState = [new Filter("Name", FilterOperator.Contains,
sQuery)];
            }
            this.oProductsTable.getBinding("items").filter(oTableSearchState,
"Application");
        },
        onAdd: function () {
            MessageBox.show("This functionality is not ready yet.", {
                icon: MessageBox.Icon.INFORMATION,
                title: "Aw, Snap!",
                actions: [MessageBox.Action.OK]
            });
        },
        onSort: function () {
            this._bDescendingSort = !this._bDescendingSort;
            var oBinding = this.oProductsTable.getBinding("items"),
                oSorter = new Sorter("Name", this._bDescendingSort);
            oBinding.sort(oSorter);
        },
        onListItemPress: function (oEvent) {
            var productPath =
oEvent.getSource().getBindingContext("products").getPath(),
                product = productPath.split("/").slice(-1).pop();
            this.oRouter.navTo("detail", {layout:
fioriLibrary.LayoutType.TwoColumnsMidExpanded, product: product});
        }
    });
});
```

We change the event handler for pressing an item from the master view to use the router instead of manually manipulating the `FlexibleColumnLayout` instance. When we call the router's `navTo` method, the router itself will change the `layout` property of the `FlexibleColumnLayout`.

webapp/controller/Detail.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function (Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.Detail", {
        onInit: function () {
            var oOwnerComponent = this.getOwnerComponent();

            this.oRouter = oOwnerComponent.getRouter();
            this.oModel = oOwnerComponent.getModel();
```

```
this.oRouter.getRoute("master").attachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detail").attachPatternMatched(this._onProductMatched,
this);
        },

        _onProductMatched: function (oEvent) {
            this._product = oEvent.getParameter("arguments").product ||
this._product || "0";
            this.getView().bindElement({
                path: "/ProductCollection/" + this._product,
                model: "products"
            });
        },
        onEditToggleButtonPress: function() {
            var oObjectPage = this.getView().byId("ObjectPageLayout"),
                bCurrentShowFooterState = oObjectPage.getShowFooter();
            oObjectPage.setShowFooter(!bCurrentShowFooterState);
        },

        onExit: function () {

this.oRouter.getRoute("master").detachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detail").detachPatternMatched(this._onProductMatched,
this);
        }
    });
});
```

We bind the table in the detail view to reflect the currently selected product from the master view.

webapp/Component.js [MODIFY]

```
sap.ui.define([
    'sap/ui/core/UIComponent',
    'sap/ui/model/json/JSONModel',
    'sap/f/library'
], function(UIComponent, JSONModel, fioriLibrary) {
    'use strict';
    return UIComponent.extend('sap.ui.demo.fiori2.Component', {
        metadata: {
            manifest: 'json'
        },
        init: function () {
            var oModel,
                oProductsModel,
                oRouter;
            UIComponent.prototype.init.apply(this, arguments);
            oModel = new JSONModel();
            this.setModel(oModel);
            // set products demo model on this sample
            oProductsModel = new JSONModel(sap.ui.require.toUrl('sap/ui/demo/
mock') + '/products.json');
            oProductsModel.setSizeLimit(1000);
            this.setModel(oProductsModel, 'products');
            oRouter = this.getRouter();
            oRouter.attachBeforeRouteMatched(this._onBeforeRouteMatched, this);
            oRouter.initialize();
        },
```

```
        _onBeforeRouteMatched: function(oEvent) {
            var oModel = this.getModel(),
                sLayout = oEvent.getParameters().arguments.layout;

            // If there is no layout parameter, set a default layout (normally
OneColumn)
            if (!sLayout) {
                sLayout = fioriLibrary.LayoutType.OneColumn;
            }

            oModel.setProperty("/layout", sLayout);
        }
    });
});
```

We initialize the router and bind to its `onBeforeRouteMatched` event, and we introduce a model, which will be used for the layout.

webapp/manifest.json [MODIFY]

```
{
    "_version": "1.12.0",
    "sap.app": {
        "id": "sap.ui.demo.fiori2",
        "type": "application",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui5": {
        "rootView": {
            "viewName": "sap.ui.demo.fiori2.view.App",
            "type": "XML",
            "async": true,
            "id": "fcl"
        },
        "dependencies": {
            "minUI5Version": "1.60.0",
            "libs": {
                "sap.ui.core": {},
                "sap.m": {},
                "sap.f": {},
                "sap.uxap": {}
            }
        },
        "config": {
            "fullWidth": true
        },
        "routing": {
            "config": {
                "routerClass": "sap.f.routing.Router",
                "viewType": "XML",
                "viewPath": "sap.ui.demo.fiori2.view",
                "controlId": "flexibleColumnLayout",
                "transition": "slide",
                "bypassed": {
                },
                "async": true
            },
            "routes": [
                {
                    "pattern": ":layout:",
```

```
                    "name": "master",
                    "target": [
                        "master",
                        "detail"
                    ]
                },
                {
                    "pattern": "detail/{product}/{layout}",
                    "name": "detail",
                    "target": [
                        "master",
                        "detail"
                    ]
                }
            ],
            "targets": {
                "master": {
                    "viewName": "Master",
                    "controlAggregation": "beginColumnPages"
                },
                "detail": {
                    "viewName": "Detail",
                    "controlAggregation": "midColumnPages"
                }
            }
        }
    }
}
```

Finally, we add the routing configuration in the `manifest.json`.

Step 8: Enhancing the Detail Page

With routing implemented, the model of the detail page is updated for each product. In this step, we enhance the detail page to show information specific for the selected product.

Preview

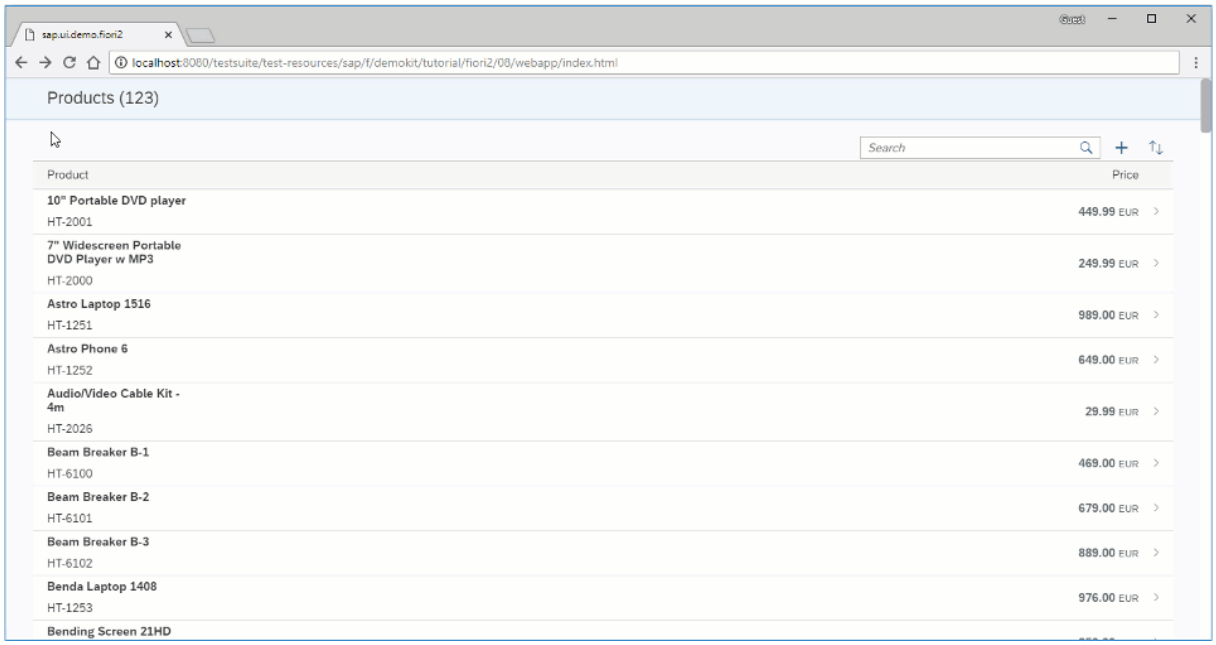


Figure 155: Enhanced detail page displaying information specific to the selected product

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 8.

webapp/view/Detail.view.xml [MODIFY]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.Detail"
    xmlns="sap.uxap"
    xmlns:m="sap.m"
    xmlns:f="sap.f"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc">
    <ObjectPageLayout
        id="ObjectPageLayout"
        showTitleInHeaderContent="true"
        alwaysShowContentHeader="false"
        preserveHeaderStateOnScroll="false"
        headerContentPinnable="true"
```

```
        isChildPage="true"
        upperCaseAnchorBar="false">
        <headerTitle>
            <ObjectPageDynamicHeaderTitle>
                <expandedHeading>
                    <m:Title text="{products>Name}" wrapping="true"
class="sapUiSmallMarginEnd"/>
                </expandedHeading>

                <snappedHeading>
                    <m:FlexBox wrap="Wrap" fitContainer="true"
alignItems="Center">
                        <m:FlexBox wrap="NoWrap" fitContainer="true"
alignItems="Center" class="sapUiTinyMarginEnd">
                            <f:Avatar
                                src="https://sapui5.hana.ondemand.com/
{products>ProductPicUrl}"
                                displaySize="S"
                                displayShape="Square"
                                class="sapUiTinyMarginEnd"/>
                            <m:Title text="{products>Name}" wrapping="true"/>
                        </m:FlexBox>
                    </m:FlexBox>
                </snappedHeading>
                <actions>
                ...
```

Using the `expandedHeading` and `snappedHeading` aggregations, we specify different content to be displayed in the title area depending on whether the header is expanded or collapsed.

webapp/view/Detail.view.xml [MODIFY]

```
        ...
        <headerContent>
            <m:FlexBox wrap="Wrap" fitContainer="true" alignItems="Stretch">
                <f:Avatar
                    src="https://sapui5.hana.ondemand.com/
{products>ProductPicUrl}"
                    displaySize="L"
                    displayShape="Square"
                    class="sapUiTinyMarginEnd">
                </f:Avatar>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Main Category"/>
                    <m:Text text="{products>MainCategory}"/>
                </m:VBox>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Subcategory"/>
                    <m:Text text="{products>Category}"/>
                </m:VBox>
                <m:VBox justifyContent="Center" class="sapUiSmallMarginEnd">
                    <m:Label text="Price"/>
                    <m:ObjectNumber number="{products>CurrencyCode}
{products>Price}" emphasized="false"/>
                </m:VBox>
            </m:FlexBox>
        </headerContent>
        ...
```

We adjust the `headerContent` so that the `sap.f.Avatar` displays the specific image of the selected product and the header displays the product's Main Category, Category and Price information, which is provided in the `products.json` we're using.

webapp/view/Detail.view.xml [MODIFY]

```
...
<sections>
    <ObjectPageSection title="General Information">
        <subSections>
            <ObjectPageSubSection>
                <blocks>
                    <form:SimpleForm
                        maxContainerCols="2"
                        editable="false"
                        layout="ResponsiveGridLayout"
                        labelSpanL="12"
                        labelSpanM="12"
                        emptySpanL="0"
                        emptySpanM="0"
                        columnsL="1"
                        columnsM="1">
                        <form:content>
                            <m:Label text="Product ID"/>
                            <m:Text text="{products>ProductId}"/>
                            <m:Label text="Description"/>
                            <m:Text text="{products>Description}"/>
                            <m:Label text="Supplier"/>
                            <m:Text text="{products>SupplierName}"/>
                        </form:content>
                    </form:SimpleForm>
                </blocks>
            </ObjectPageSubSection>
        </subSections>
    </ObjectPageSection>
    ...
```

We adjust the *General Information* section to display Product ID, Description and Supplier of the selected product.

Step 9: Adding a Detail-Detail Page

In this step, we create a detail-detail page using `sap.f.DynamicPage`, which is opened by choosing a supplier from the detail page.

Preview

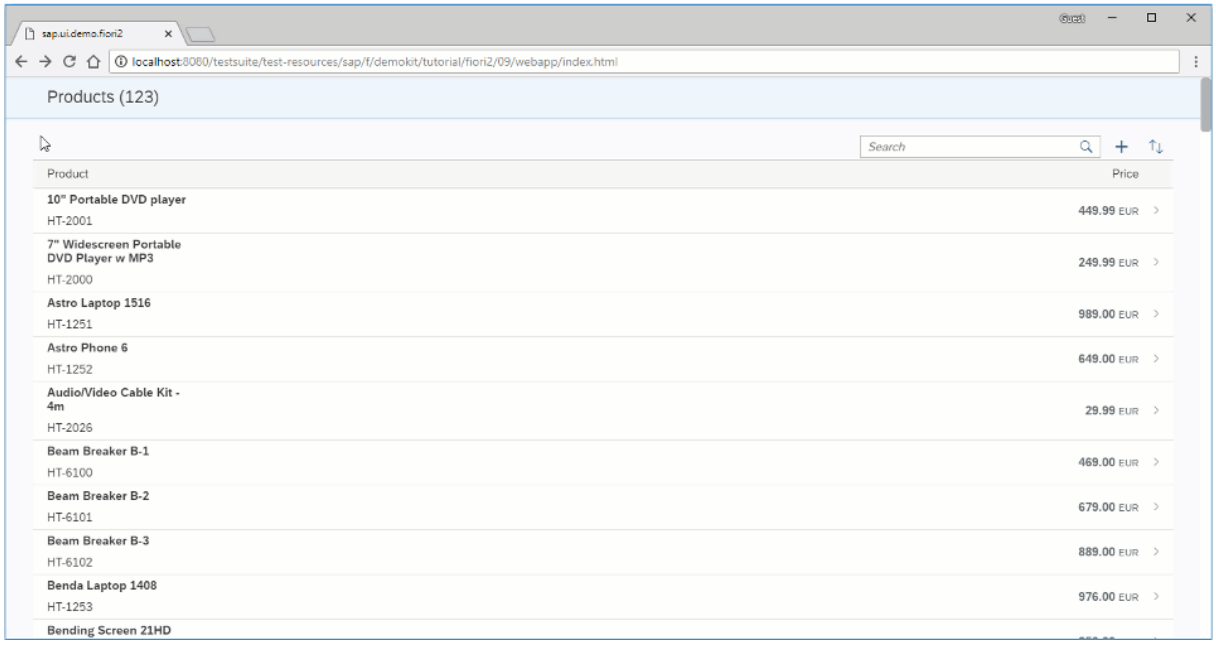


Figure 156: Detail-detail page displaying the name of the selected supplier

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 9.

webapp/view/DetailDetail.view.xml [NEW]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.DetailDetail"
    xmlns="sap.f"
    xmlns:m="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <DynamicPage toggleHeaderOnTitleClick="false">
        <title>
            <DynamicPageTitle>
                <heading>
                    <m:FlexBox wrap="Wrap" fitContainer="true"
alignItems="Center">
                        <m:Title text="{products>text}" wrapping="true"
class="sapUiTinyMarginEnd"/>
```

```
                    </m:FlexBox>
                </heading>
            </DynamicPageTitle>
        </title>
    </DynamicPage>
</mvc:View>
```

We create a detail-detail page view using `sap.f.DynamicPage` with only a title.

webapp/controller/DetailDetail.controller.js [NEW]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller"
], function (JSONModel, Controller) {
    "use strict";

    return Controller.extend("sap.ui.demo.fiori2.controller.DetailDetail", {
        onInit: function () {
            var oOwnerComponent = this.getOwnerComponent();

            this.oRouter = oOwnerComponent.getRouter();
            this.oModel = oOwnerComponent.getModel();


this.oRouter.getRoute("detailDetail").attachPatternMatched(this._onPatternMatch,
this);
        },

        _onPatternMatch: function (oEvent) {
            this._supplier = oEvent.getParameter("arguments").supplier ||
this._supplier || "0";
            this._product = oEvent.getParameter("arguments").product ||
this._product || "0";

            this.getView().bindElement({
                path: "/ProductCollectionStats/Filters/1/values/" +
this._supplier,
                model: "products"
            });
        },

        onExit: function () {

this.oRouter.getRoute("detailDetail").detachPatternMatched(this._onPatternMatch,
this);
        }
    });
});
```

We create the detail-detail page controller.

webapp/manifest.json [MODIFY]

```
            ...
            "routes": [
                {
```

```
                    "pattern": ":layout:",
                    "name": "master",
                    "target": [
                        "master",
                        "detail"
                    ]
                },
                {
                    "pattern": "detail/{product}/{layout}",
                    "name": "detail",
                    "target": [
                        "master",
                        "detail"
                    ]
                },
                {
                    "pattern": "detail/{product}/detailDetail/{supplier}/
{layout}",
                    "name": "detailDetail",
                    "target": [
                        "master",
                        "detail",
                        "detailDetail"
                    ]
                }
            ],
            "targets": {
                "master": {
                    "viewName": "Master",
                    "controlAggregation": "beginColumnPages"
                },
                "detail": {
                    "viewName": "Detail",
                    "controlAggregation": "midColumnPages"
                },
                "detailDetail": {
                    "viewName": "DetailDetail",
                    "controlAggregation": "endColumnPages"
                }
            }
        }
    }
}
```

We add the detail-detail page to our existing routes in the `manifest.json`.

webapp/view/Detail.view.xml [MODIFY]

```
            ...
            <ObjectPageSection title="Suppliers">
                <subSections>
                    <ObjectPageSubSection>
                        <blocks>
                            <m:Table
                                id="suppliersTable"
                                items="{path : 'products>/ProductCollectionStats/
Filters/1/values'}">
                                <m:columns>
                                    <m:Column/>
                                </m:columns>
                                <m:items>
                                    <m:ColumnListItem type="Navigation"
press=".onSupplierPress">
```

```
                                        <m:cells>
                                            <m:ObjectIdentifier
text="{products>text}"/>
                                        </m:cells>
                                    </m:ColumnListItem>
                                </m:items>
                            </m:Table>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>
        </sections>
        ...
```

We add a `press` event handler for each item in the *SUPPLIERS* table in the detail page.

webapp/controller/Detail.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    'sap/f/library'
], function (Controller, fioriLibrary) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.Detail", {
        onInit: function () {
            var oOwnerComponent = this.getOwnerComponent();
            this.oRouter = oOwnerComponent.getRouter();
            this.oModel = oOwnerComponent.getModel();

this.oRouter.getRoute("master").attachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detail").attachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detailDetail").attachPatternMatched(this._onProductMatched
, this);
        },
        onSupplierPress: function (oEvent) {
            var supplierPath =
oEvent.getSource().getBindingContext("products").getPath(),
                supplier = supplierPath.split("/").slice(-1).pop();

            this.oRouter.navTo("detailDetail", {layout:
fioriLibrary.LayoutType.ThreeColumnsMidExpanded, supplier: supplier, product:
this._product});
        },
        _onProductMatched: function (oEvent) {
        ...
```

We add an `onSupplierPress` function in the detail page controller in order to pass the data for the selected supplier and navigate to the detail-detail page.

webapp/controller/App.controller.js [MODIFY]

```
        ...
```

```
        onRouteMatched: function (oEvent) {
            var sRouteName = oEvent.getParameter("name"),
                oArguments = oEvent.getParameter("arguments");
            // Save the current route name
            this.currentRouteName = sRouteName;
            this.currentProduct = oArguments.product;
            this.currentSupplier = oArguments.supplier;
        },
        onStateChanged: function (oEvent) {
            var bIsNavigationArrow = oEvent.getParameter("isNavigationArrow"),
                sLayout = oEvent.getParameter("layout");
            // Replace the URL with the new layout if a navigation arrow was used
            if (bIsNavigationArrow) {
                this.oRouter.navTo(this.currentRouteName, {layout: sLayout,
product: this.currentProduct, supplier: this.currentSupplier}, true);
            }
        },
        onExit: function () {
            this.oRouter.detachRouteMatched(this.onRouteMatched, this);
        }
    });
});
```

Finally, we pass data for the supplier in the detail-detail page.

Step 10: Adding More Pages

In this step, we create an additional page that is displayed in a separate fullscreen column.

Preview

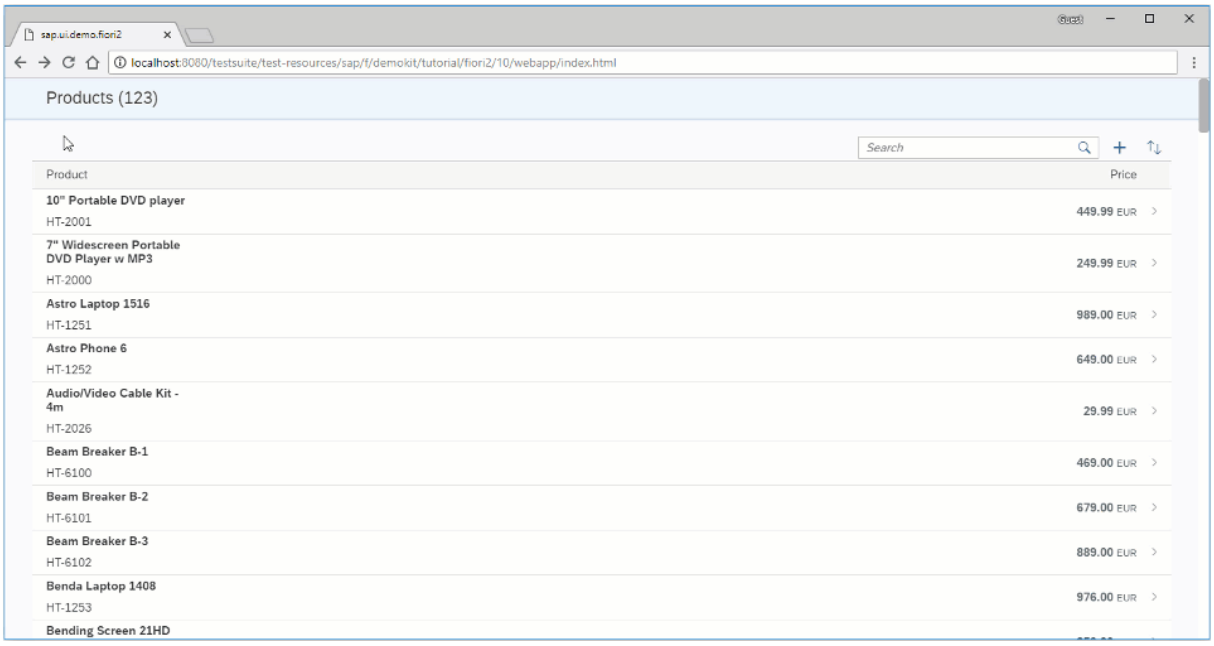


Figure 157: Additional page displayed in a separate fullscreen column

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 10.

webapp/view/AboutPage.view.xml [NEW]

```
<mvc:View
    xmlns="sap.m"
    xmlns:f="sap.f"
    xmlns:mvc="sap.ui.core.mvc">
    <f:DynamicPage toggleHeaderOnTitleClick="false">
        <!-- DynamicPage Title -->
        <f:title>
            <f:DynamicPageTitle>
                <f:heading>
                    <Title text="About supplier"/>
                </f:heading>
            </f:DynamicPageTitle>
        </f:title>
    </f:DynamicPage>
</mvc:View>
```

We create a simple additional page view.

webapp/manifest.json [MODIFY]

```
            ...
            "routes": [
                {
                    "pattern": "page2",
                    "name": "page2",
                    "target": "page2",
                    "layout": "EndColumnFullScreen"
                },
                {
                    "pattern": ":layout:",
                    "name": "master",
                    "target": [
                        "master",
                        "detail"
                    ]
                },
                {
                    "pattern": "detail/{product}/{layout}",
                    "name": "detail",
                    "target": [
                        "master",
                        "detail"
                    ]
                },
                {
                    "pattern": "detail/{product}/detailDetail/{supplier}/
{layout}",
                    "name": "detailDetail",
                    "target": [
                        "master",
```

```
                        "detail",
                        "detailDetail"
                    ]
                }
            ],
            "targets": {
                "master": {
                    "viewName": "Master",
                    "controlAggregation": "beginColumnPages"
                },
                "detail": {
                    "viewName": "Detail",
                    "controlAggregation": "midColumnPages"
                },
                "detailDetail": {
                    "viewName": "DetailDetail",
                    "controlAggregation": "endColumnPages"
                },
                "page2": {
                    "viewName": "AboutPage",
                    "controlAggregation": "endColumnPages"
                }
            }
        }
    }
}
```

Similar to the previous step, we add the additional page view to our existing routes in the `manifest.json`.

webapp/view/DetailDetail.view.xml [MODIFY]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.DetailDetail"
    xmlns="sap.f"
    xmlns:m="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <DynamicPage toggleHeaderOnTitleClick="false">
        <title>
            <DynamicPageTitle>
                <heading>
                    <m:FlexBox wrap="Wrap" fitContainer="true"
alignItems="Center">
                        <m:Title text="{products>text}" wrapping="true"
class="sapUiTinyMarginEnd"/>
                    </m:FlexBox>
                </heading>
            </DynamicPageTitle>
        </title>
        <content>
            <m:Link text="Navigate to next page…" press=".handleAboutPress"/>
        </content>
    </DynamicPage>
</mvc:View>
```

We add a link in the detail-detail page with a `press` event handler.

webapp/controller/DetailDetail.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller",
    'sap/f/library'
], function (JSONModel, Controller, fioriLibrary) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.DetailDetail", {
        onInit: function () {
            var oOwnerComponent = this.getOwnerComponent();
            this.oRouter = oOwnerComponent.getRouter();
            this.oModel = oOwnerComponent.getModel();

this.oRouter.getRoute("detailDetail").attachPatternMatched(this._onPatternMatch,
this);
        },
        handleAboutPress: function () {
            this.oRouter.navTo("page2", {layout:
fioriLibrary.LayoutType.EndColumnFullScreen});
        },
        _onPatternMatch: function (oEvent) {
        ...
```

Finally, we add a `handleAboutPress` function in the detail-detail page controller to navigate to the additional page without passing any data this time.

Step 11: Using the Flexible Column Layout Semantic Helper

In this step, we use the `sap.f.FlexibleColumnLayoutSemanticHelper` class to implement the recommended UX patterns for layout changes in the app.

`FlexibleColumnLayout` gives you the freedom to implement any app logic that involves changing the layout (showing/hiding columns) as a result of the user's actions. However, there are certain UX patterns that are considered as optimal and are recommended for SAP Fiori 2.0 apps. The `FlexibleColumnLayoutSemanticHelper` class helps you implement them by giving you tips about what layout to display when.

> **i** Note
>
> Using this class is NOT mandatory in order to build an app with the `FlexibleColumnLayout`, but makes it easier to achieve the optimal UX recommended in the SAP Fiori 2.0 design guidelines.

For more information, see Flexible Column Layout Semantic Helper [page 2294].

Preview

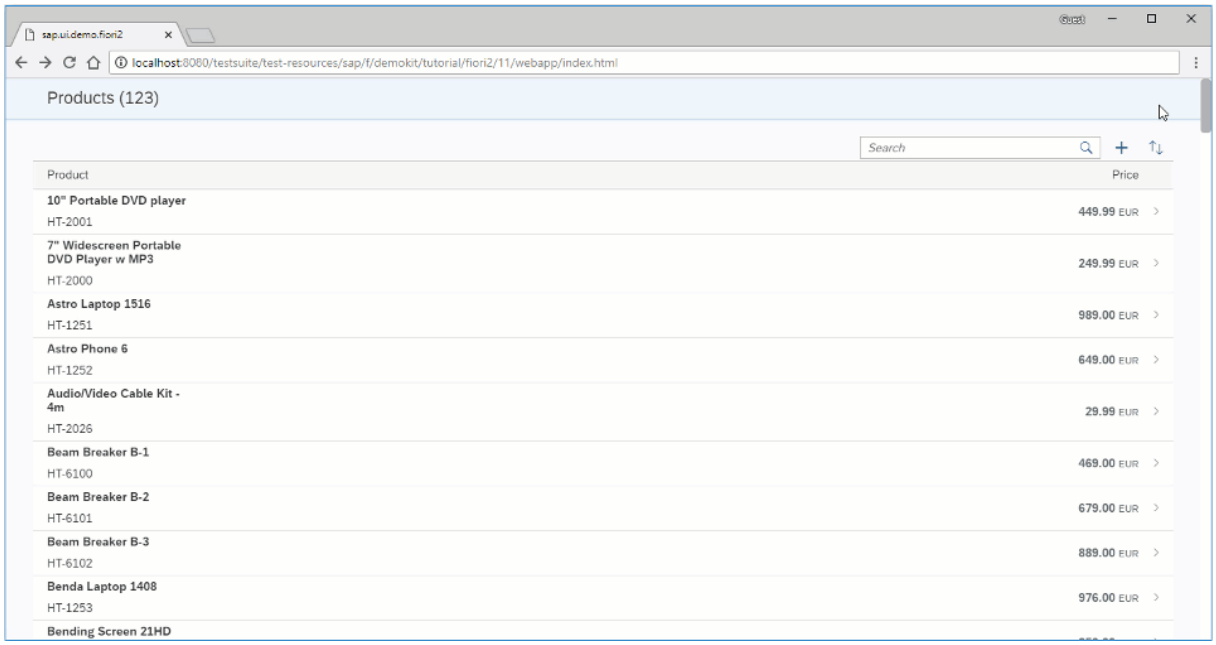


Figure 158: Master-detail-detail pattern using `sap.f.FlexibleColumnLayoutSemanticHelper`

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 11.

webapp/Component.js [MODIFY]

```
sap.ui.define([
    'sap/ui/core/UIComponent',
    'sap/ui/model/json/JSONModel',
    'sap/f/FlexibleColumnLayoutSemanticHelper',
    'sap/f/library'
], function(UIComponent, JSONModel, FlexibleColumnLayoutSemanticHelper,
fioriLibrary) {
    'use strict';
    return UIComponent.extend('sap.ui.demo.fiori2.Component', {
        metadata: {
            manifest: 'json'
        },
        init: function () {
            var oModel,
                oProductsModel,
                oRouter;
            UIComponent.prototype.init.apply(this, arguments);
            oModel = new JSONModel();
            this.setModel(oModel);
            // set products demo model on this sample
            oProductsModel = new JSONModel(sap.ui.require.toUrl('sap/ui/demo/
mock') + '/products.json');
            oProductsModel.setSizeLimit(1000);
```

```
            this.setModel(oProductsModel, 'products');
            oRouter = this.getRouter();
            oRouter.attachBeforeRouteMatched(this._onBeforeRouteMatched, this);
            oRouter.initialize();
        },
        getHelper: function () {
            return this._getFcl().then(function(oFCL) {
                var oSettings = {
                    defaultTwoColumnLayoutType:
fioriLibrary.LayoutType.TwoColumnsMidExpanded,
                    defaultThreeColumnLayoutType:
fioriLibrary.LayoutType.ThreeColumnsMidExpanded
                };
                return (FlexibleColumnLayoutSemanticHelper.getInstanceFor(oFCL,
oSettings));
            });
        },
        _onBeforeRouteMatched: function(oEvent) {
            var oModel = this.getModel(),
                sLayout = oEvent.getParameters().arguments.layout,
                oNextUIState;
            // If there is no layout parameter, query for the default level 0
layout (normally OneColumn)
            if (!sLayout) {
                this.getHelper().then(function(oHelper) {
                    oNextUIState = oHelper.getNextUIState(0);
                    oModel.setProperty("/layout", oNextUIState.layout);
                });
                return;
            }
            oModel.setProperty("/layout", sLayout);
        },

        _getFcl: function () {
            return new Promise(function(resolve, reject) {
                var oFCL = this.getRootControl().byId('flexibleColumnLayout');
                if (!oFCL) {
                    this.getRootControl().attachAfterInit(function(oEvent) {
                        resolve(oEvent.getSource().byId('flexibleColumnLayout'));
                    }, this);
                    return;
                }
                resolve(oFCL);

            }.bind(this));
        }
    });
});
```

First, we add a `getHelper` function in the `Component.js` file in order to pass the default `sap.f.FlexibleColumnLayout` parameters.

webapp/view/Detail.view.xml [MODIFY]

```
                ...
                <snappedHeading>
                    <m:FlexBox wrap="Wrap" fitContainer="true"
alignItems="Center">
                        <m:FlexBox wrap="NoWrap" fitContainer="true"
alignItems="Center" class="sapUiTinyMarginEnd">
                            <f:Avatar
                                src="https://sapui5.hana.ondemand.com/
{products>ProductPicUrl}"
```

```
                                displaySize="S"
                                displayShape="Square"
                                class="sapUiTinyMarginEnd"/>
                            <m:Title text="{products>Name}" wrapping="true"/>
                        </m:FlexBox>
                    </m:FlexBox>
                </snappedHeading>
                <navigationActions>
                    <m:OverflowToolbarButton
                        type="Transparent"
                        icon="sap-icon://full-screen"
                        press=".handleFullScreen"
                        tooltip="Enter Full Screen Mode"
                        visible="{= ${/actionButtonsInfo/midColumn/fullScreen} !
== null }"/>
                    <m:OverflowToolbarButton
                        type="Transparent"
                        icon="sap-icon://exit-full-screen"
                        press=".handleExitFullScreen"
                        tooltip="Exit Full Screen Mode"
                        visible="{= ${/actionButtonsInfo/midColumn/
exitFullScreen} !== null }"/>
                    <m:OverflowToolbarButton
                        type="Transparent"
                        icon="sap-icon://decline"
                        press=".handleClose"
                        tooltip="Close column"
                        visible="{= ${/actionButtonsInfo/midColumn/closeColumn} !
== null }"/>
                </navigationActions>
                <actions>
                    <m:ToggleButton
                        text="Edit"
                        type="Emphasized"
                        press=".onEditToggleButtonPress"/>
                    <m:Button
                        text="Delete"
                        type="Transparent"/>
                    <m:Button
                        text="Copy"
                        type="Transparent"/>
                    <m:Button
                        icon="sap-icon://action"
                        type="Transparent"/>
                </actions>
            </ObjectPageDynamicHeaderTitle>
        </headerTitle>
        ...
```

We add navigation actions for entering and exiting fullscreen and closing the column for the detail page.

webapp/controller/Detail.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function (Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.Detail", {
        onInit: function () {
            this.oOwnerComponent = this.getOwnerComponent();
            this.oRouter = this.oOwnerComponent.getRouter();
            this.oModel = this.oOwnerComponent.getModel();
```

```
this.oRouter.getRoute("master").attachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detail").attachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detailDetail").attachPatternMatched(this._onProductMatched
, this);
        },
        onSupplierPress: function (oEvent) {
            var supplierPath =
oEvent.getSource().getBindingContext("products").getPath(),
                supplier = supplierPath.split("/").slice(-1).pop(),
                oNextUIState;

            this.oOwnerComponent.getHelper().then(function (oHelper) {
                oNextUIState = oHelper.getNextUIState(2);
                this.oRouter.navTo("detailDetail", {
                    layout: oNextUIState.layout,
                    supplier: supplier,
                    product: this._product
                });
            }.bind(this));
        },
        _onProductMatched: function (oEvent) {
            this._product = oEvent.getParameter("arguments").product ||
this._product || "0";
            this.getView().bindElement({
                path: "/ProductCollection/" + this._product,
                model: "products"
            });
        },
        onEditToggleButtonPress: function() {
            var oObjectPage = this.getView().byId("ObjectPageLayout"),
                bCurrentShowFooterState = oObjectPage.getShowFooter();
            oObjectPage.setShowFooter(!bCurrentShowFooterState);
        },
        handleFullScreen: function () {
            var sNextLayout = this.oModel.getProperty("/actionButtonsInfo/
midColumn/fullScreen");
            this.oRouter.navTo("detail", {layout: sNextLayout, product:
this._product});
        },

        handleExitFullScreen: function () {
            var sNextLayout = this.oModel.getProperty("/actionButtonsInfo/
midColumn/exitFullScreen");
            this.oRouter.navTo("detail", {layout: sNextLayout, product:
this._product});
        },

        handleClose: function () {
            var sNextLayout = this.oModel.getProperty("/actionButtonsInfo/
midColumn/closeColumn");
            this.oRouter.navTo("master", {layout: sNextLayout});
        },
        onExit: function () {

this.oRouter.getRoute("master").detachPatternMatched(this._onProductMatched,
this);

this.oRouter.getRoute("detail").detachPatternMatched(this._onProductMatched,
this);
        }
    });
});
```

We create the handlers needed for the navigation actions.

webapp/view/DetailDetail.view.xml [MODIFY]

```
<mvc:View
    controllerName="sap.ui.demo.fiori2.controller.DetailDetail"
    xmlns="sap.f"
    xmlns:m="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <DynamicPage toggleHeaderOnTitleClick="false">
        <title>
            <DynamicPageTitle>
                <heading>
                    <m:FlexBox wrap="Wrap" fitContainer="true"
alignItems="Center">
                        <m:Title text="{products>text}" wrapping="true"
class="sapUiTinyMarginEnd"/>
                    </m:FlexBox>
                </heading>
                <navigationActions>
                    <m:OverflowToolbarButton
                        type="Transparent"
                        icon="sap-icon://full-screen"
                        press=".handleFullScreen"
                        tooltip="Enter Full Screen Mode"
                        visible="{= ${/actionButtonsInfo/endColumn/fullScreen} !
== null }"/>
                    <m:OverflowToolbarButton
                        type="Transparent"
                        icon="sap-icon://exit-full-screen"
                        press=".handleExitFullScreen"
                        tooltip="Exit Full Screen Mode"
                        visible="{= ${/actionButtonsInfo/endColumn/
exitFullScreen} !== null }"/>
                    <m:OverflowToolbarButton
                        type="Transparent"
                        icon="sap-icon://decline"
                        press=".handleClose"
                        tooltip="Close column"
                        visible="{= ${/actionButtonsInfo/endColumn/closeColumn} !
== null }"/>
                </navigationActions>
            </DynamicPageTitle>
        </title>
        <content>
            <m:Link text="Navigate to next page…" press=".handleAboutPress"/>
        </content>
    </DynamicPage>
</mvc:View>
```

Again, we add navigation actions for entering and exiting fullscreen and closing the column for the detail-detail page.

webapp/controller/DetailDetail.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
```

```
    "sap/ui/core/mvc/Controller"
], function (JSONModel, Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.DetailDetail", {
        onInit: function () {
            this.oOwnerComponent = this.getOwnerComponent();
            this.oRouter = this.oOwnerComponent.getRouter();
            this.oModel = this.oOwnerComponent.getModel();

this.oRouter.getRoute("detailDetail").attachPatternMatched(this._onPatternMatch,
this);
        },
        handleAboutPress: function () {
            var oNextUIState;
            this.oOwnerComponent.getHelper().then(function (oHelper) {
                oNextUIState = oHelper.getNextUIState(3);
                this.oRouter.navTo("page2", {layout: oNextUIState.layout});
            }.bind(this));
        },
        _onPatternMatch: function (oEvent) {
            this._supplier = oEvent.getParameter("arguments").supplier ||
this._supplier || "0";
            this._product = oEvent.getParameter("arguments").product ||
this._product || "0";
            this.getView().bindElement({
                path: "/ProductCollectionStats/Filters/1/values/" +
this._supplier,
                model: "products"
            });
        },
        handleFullScreen: function () {
            var sNextLayout = this.oModel.getProperty("/actionButtonsInfo/
endColumn/fullScreen");
            this.oRouter.navTo("detailDetail", {layout: sNextLayout, product:
this._product, supplier: this._supplier});
        },

        handleExitFullScreen: function () {
            var sNextLayout = this.oModel.getProperty("/actionButtonsInfo/
endColumn/exitFullScreen");
            this.oRouter.navTo("detailDetail", {layout: sNextLayout, product:
this._product, supplier: this._supplier});
        },

        handleClose: function () {
            var sNextLayout = this.oModel.getProperty("/actionButtonsInfo/
endColumn/closeColumn");
            this.oRouter.navTo("detail", {layout: sNextLayout, product:
this._product});
        },
        onExit: function () {

this.oRouter.getRoute("detailDetail").detachPatternMatched(this._onPatternMatch,
this);
        }
    });
});
```

And respectively, we create the handlers needed for the navigation actions in the detail-detail controller.

webapp/controller/Master.controller.js [MODIFY]

```
sap.ui.define([
```

```
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/Filter",
    "sap/ui/model/FilterOperator",
    'sap/ui/model/Sorter',
    'sap/m/MessageBox'
], function (JSONModel, Controller, Filter, FilterOperator, Sorter, MessageBox) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.Master", {
        onInit: function () {
            this.oView = this.getView();
            this._bDescendingSort = false;
            this.oProductsTable = this.oView.byId("productsTable");
            this.oRouter = this.getOwnerComponent().getRouter();
        },
        onSearch: function (oEvent) {
            var oTableSearchState = [],
                sQuery = oEvent.getParameter("query");
            if (sQuery && sQuery.length > 0) {
                oTableSearchState = [new Filter("Name", FilterOperator.Contains,
sQuery)];
            }
            this.oProductsTable.getBinding("items").filter(oTableSearchState,
"Application");
        },
        onAdd: function () {
            MessageBox.information("This functionality is not ready yet.",
{title: "Aw, Snap!"});
        },
        onSort: function () {
            this._bDescendingSort = !this._bDescendingSort;
            var oBinding = this.oProductsTable.getBinding("items"),
                oSorter = new Sorter("Name", this._bDescendingSort);
            oBinding.sort(oSorter);
        },
        onListItemPress: function (oEvent) {
            var productPath =
oEvent.getSource().getBindingContext("products").getPath(),
                product = productPath.split("/").slice(-1).pop(),
                oNextUIState;
            this.getOwnerComponent().getHelper().then(function (oHelper) {
                oNextUIState = oHelper.getNextUIState(1);
                this.oRouter.navTo("detail", {
                    layout: oNextUIState.layout,
                    product: product
                });
            }.bind(this));
        }
    });
});
```

We get the next layout from the semantic helper rather than hard coding them ourselves.

webapp/controller/App.controller.js [MODIFY]

```
sap.ui.define([
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/mvc/Controller"
], function (JSONModel, Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.fiori2.controller.App", {
        onInit: function () {
            this.oOwnerComponent = this.getOwnerComponent();
```

```
            this.oRouter = this.oOwnerComponent.getRouter();
            this.oRouter.attachRouteMatched(this.onRouteMatched, this);
        },
        onRouteMatched: function (oEvent) {
            var sRouteName = oEvent.getParameter("name"),
                oArguments = oEvent.getParameter("arguments");
            this._updateUIElements();
            // Save the current route name
            this.currentRouteName = sRouteName;
            this.currentProduct = oArguments.product;
            this.currentSupplier = oArguments.supplier;
        },
        onStateChanged: function (oEvent) {
            var bIsNavigationArrow = oEvent.getParameter("isNavigationArrow"),
                sLayout = oEvent.getParameter("layout");
            this._updateUIElements();
            // Replace the URL with the new layout if a navigation arrow was used
            if (bIsNavigationArrow) {
                this.oRouter.navTo(this.currentRouteName, {layout: sLayout,
product: this.currentProduct, supplier: this.currentSupplier}, true);
            }
        },
        // Update the close/fullscreen buttons visibility
        _updateUIElements: function () {
            var oModel = this.oOwnerComponent.getModel(),
                oUIState;
            this.oOwnerComponent.getHelper().then(function(oHelper) {
                oUIState = oHelper.getCurrentUIState();
                oModel.setData(oUIState);
            });
        },
        onExit: function () {
            this.oRouter.detachRouteMatched(this.onRouteMatched, this);
            this.oRouter.detachBeforeRouteMatched(this.onBeforeRouteMatched,
this);
        }
    });
});
```

Finally, we create a function in the `App.controller.js` to update the visibility of the master, detail, and detail-detail pages.

Step 12: Starting with Two Columns

In this step, we set up the app to start with an initial layout of two columns.

By default, the `FlexibleColumnLayout` starts off with one column. If your use case requires it, you can set the initial layout to start with two columns. The user can still navigate to a single-column layout by closing the detail page from the navigation actions or a three-column layout by selecting an item from the detail page.

Preview

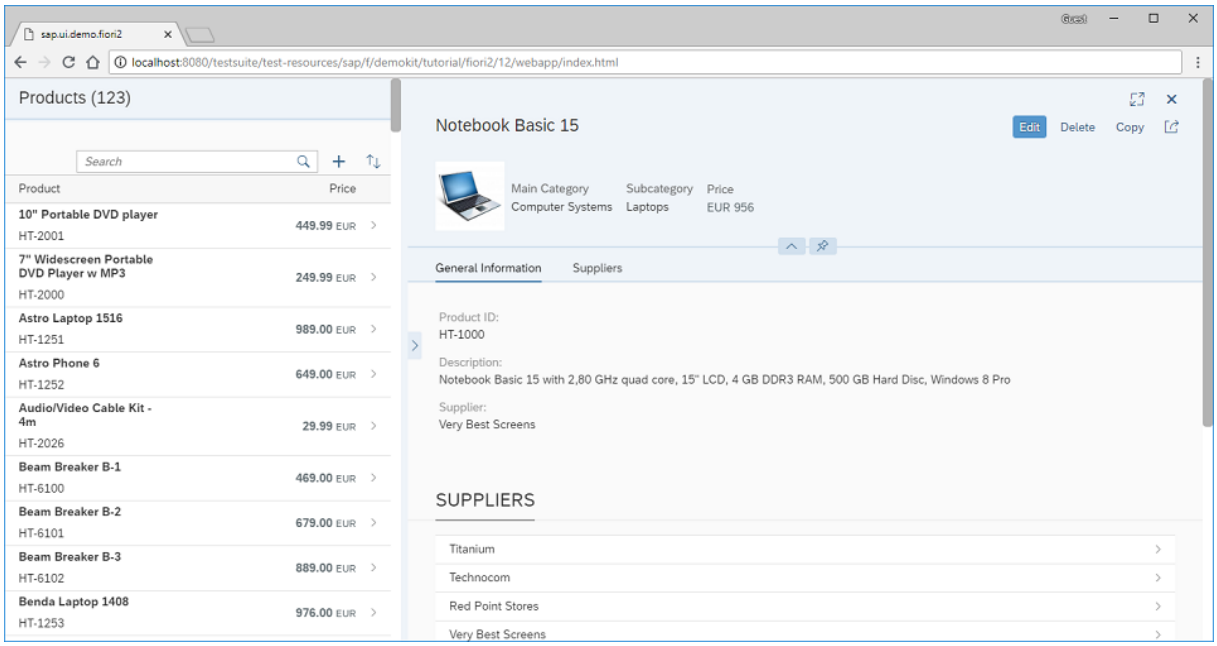


Figure 159: Initial layout with two columns

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 12.

webapp/Component.js [MODIFY]

```
        ...
        getHelper: function () {
            return this._getFcl().then(function(oFCL) {
                var oSettings = {
                    defaultTwoColumnLayoutType:
fioriLibrary.LayoutType.TwoColumnsMidExpanded,
                    defaultThreeColumnLayoutType:
fioriLibrary.LayoutType.ThreeColumnsMidExpanded,
                    initialColumnsCount: 2
                };
                return (FlexibleColumnLayoutSemanticHelper.getInstanceFor(oFCL,
oSettings));
            });
        },
        ...
```

We set the `initialColumnsCount` parameter of the `getHelper` method to `2`.

Step 13: Setting the Master-Detail Pattern

In this step, we set up the app to follow the master-detail pattern.

Apps using the master-detail pattern operate with a layout divided into two separate areas - a master list area and a details area. The master list area displays the items available to the user and the details area displays the details for an item that is selected in the master list. If your use case requires it, you can set the `FlexibleColumnLayout` to use a maximum of two columns. For more information, see the SAP Fiori Design Guidelines.

Preview

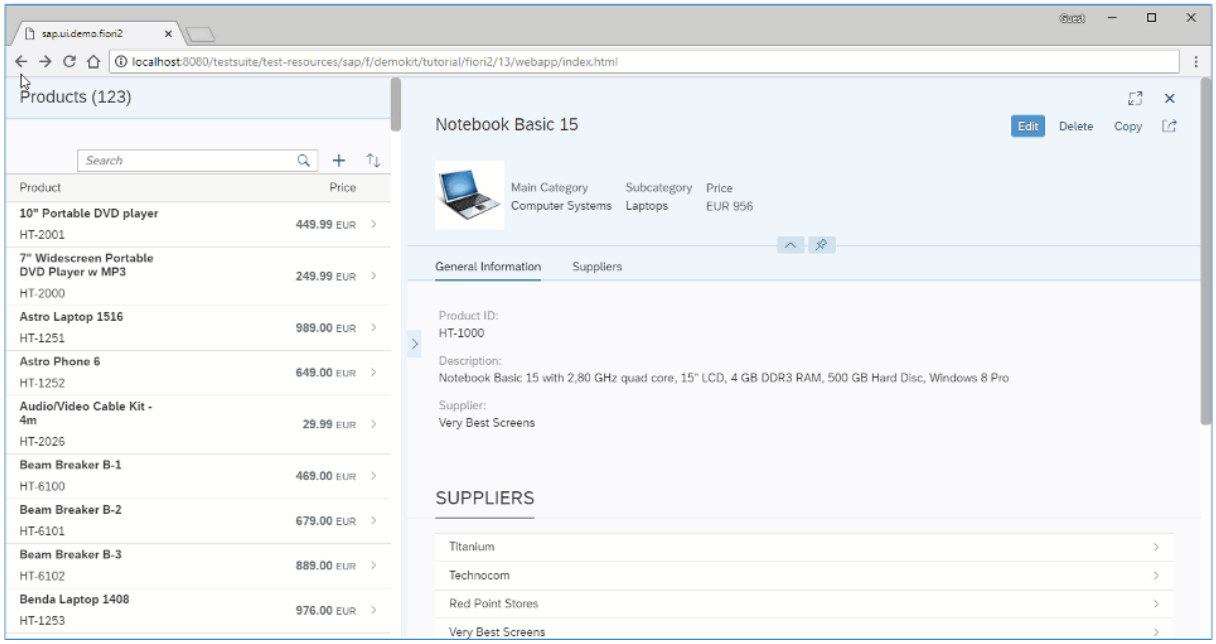


Figure 160: Master-detail pattern with `FlexibleColumnLayout`

Coding

You can view and download all files at SAP Fiori 2.0 App - Step 13.

webapp/Component.js [MODIFY]

```
        ...
        getHelper: function () {
            return this._getFcl().then(function(oFCL) {
                var oSettings = {
                    defaultTwoColumnLayoutType:
fioriLibrary.LayoutType.TwoColumnsMidExpanded,
```

```
                    defaultThreeColumnLayoutType:
fioriLibrary.LayoutType.ThreeColumnsMidExpanded,
                    initialColumnsCount: 2,
                    maxColumnsCount: 2
                };
                return (FlexibleColumnLayoutSemanticHelper.getInstanceFor(oFCL,
oSettings));
            });
        },
        ...
```

We set `maxColumnsCount` parameter of the `getHelper` method to `2`.

Rule Builder Control

In this tutorial you will learn how to embed a rule builder control to manage business rules in your application.

A business rule is a logic that defines some aspect of business and always resolves to either true or false. This logic can be maintained by non-technical users via a simplified rule language and user interface. Thus, business rules allow your application's customers to add their own logic without needing technical customization or coding.

The rule builder control enables business users to create and edit business rules in applications based on SAPUI5. The control defines SAP standard UX for creating business rules in an SAP Fiori application and should be the default UI component for SAP Fiori UI developers to add business rule capabilities.

The `sap.rules.ui` library provides controls to manage business rules. The two main components are:

- `RuleBuilder` – Provides visualization of the rule in the form of a decision table and text rule to simplify the creation and editing of the rule's business logic.
- Expression Language Services – Provides expression language services to support the readability and correctness of business rules.

The visualization provided by the `RuleBuilder` component contains text parts and the expression language services support the end user in creating and editing those text parts.

→ Tip

You do not have to do the tutorial steps sequentially; you can start the tutorial at any step you want. Just download the code, copy it to your workspace and make sure that the application runs by calling the `index.html` file.

You can view and download all the files required for steps 1 and 2 in decision table section at Rule Builder - Guided Decision Table. This is applicable only for decision table rules modeled using rule expression language.

You can view and download all the files required for steps 1 and 2 in text rule section at Rule Builder - Text Rule.

For more information, check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Expression Languages

You can model rules using an expression language. There are two expression languages supported by the rule builder:

- **Expression Language 1.0 (Rule Expression Language)**: Expression language 1.0 enable users to define business decision logic in simple readable syntax. This is the default language supported by the rule builder. A typical rule condition in expression language 1.0 is as shown below:
  `customer_name of the customer is equal to 'John`
- **Expression Language 2.0 (DMN SFEEL)**: Expression language 2.0 or DMN SFEEL is a subset of the Friendly Enough Expression Language (FEEL), provides a standard syntax for rule conditions, and reduces ambiguities while modeling a rule. A typical rule condition in expression language 2.0 is as shown below:
  `customer.customer name MATCHES 'John'`
  Key features of expression language 2.0 include:
  - **Autosuggest list** is a suggestion dropdown menu that lets you select the required element of the rule expression. You do not have to type the rule expression in the corresponding fields.
  - You can also type the rule expression in the field and select the corresponding vocabulary elements from the autosuggest list. Using free flow typing, you can also edit and delete the tokens in a rule expressions.

    > **Example**
    >
    > In the following rule expression:
    >
    > DO1.Equipment = 'Laptop' *AND* DO2.Date = 'Dec 3, 2020'
    >
    > - If the cursor is placed between *DO* and *1*, then all the data objects starting with DO are listed in the autosuggest list. Similarly, according to the cursor position, the corresponding data object or attribute name is listed.
    > - To change a date or timestamp value, edit the value in the *Fixed Value* field of the autosuggest list.
    > - If the cursor is placed before the attribute name *Equipment*, then the attribute name is deleted.
    > - If the cursor is placed in between or before the data object name, *DO1*, then both the data object and attribute name, *DO1.Equipment*, is deleted.
  - Expression language 2.0 supports the use of **vocabulary rules**. The rules which can be used in a rule expression are called vocabulary rules. The result returned by the vocabulary rule can be consumed in a rule expression.

The following are the objects for each expression language:

Expression Language	Expression Language Object
Expression language 1.0	ExpressionLanguage
Expression language 2.0	AstExpressionLanguage

> **Note**
>
> You can migrate your projects from expression language 1.0 to expression language 2.0 using the business rules API. For more information, see SAP Cloud Platform Business Rules. You should manually change

the expression language object to `AstExpressionLanguage` to load the migrated content in expression language 2.0.

Prerequisites

You should already have some general knowledge about SAPUI5 application development and the SAP rules framework.

For more information, see the documentation for the SAP HANA Rules Framework on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_HANA_RULES_FRAMEWORK.

This tutorial uses a mock server to provide the required data. Before proceeding with this tutorial, ensure that you are familiar with the concepts introduced in the following tutorials:

- Walkthrough [page 69], specifically Step 27: Mock Server Configuration [page 139]
- Mock Server [page 432]

> **i Note**
>
> Alternatively, you can use a properly configured backend system with the following implemented OData services:
>
> - Rules OData service – provides rule resources to create and edit all the elements required for rules.
> - Vocabulary OData service – provides vocabulary data resources.
>
> For more information, see the OData API reference on the SAP Help Portal at https://help.sap.com/doc/12167f8dbcc54650b47644c140fa0d0b/DEV/en-US/Rules_odata_api_index.html.

We will use a simple HTML page that will serve as a single-page application. There we will define the content of this page, which will include the meta tags, a script tag to load the SAPUI5 `sap.rules.ui` libraries, and the `RuleBuilder` control with data from the mock server.

File Structure

For this tutorial, you will be creating a folder with the following files:

- **<FolderName>** (folder)
  - `Component.js`
  - `index.html`
  - `Page.controller.js`
  - `Page.view.xml`
  - localService (folder)
    - rule (folder)
      - mockdata (folder)
      - `metadata.xml`
      - `responses.json`

- vocabulary (folder)
  - mockdata (folder)
  - `metadata.xml`

Decision Table

Create a business logic by defining the conditions in decision table, which is associated with the expression language.

Features

Refresh Data Object

The refresh data object feature reads the attributes of the data object and automatically fetches the predefined result attributes.

Cut/Copy/Paste

- The *Copy* option is used to duplicate the specific row and it can be inserted in any other row of the decision table using *Paste* option.
- The *Cut* option is used to remove the specific row and that can also be inserted in any other row of the decision table using paste option.

Hit Policy

For a decision table rule, you can specify how the rule engine should fetch the result. The two types of hit policies are first match and all match:

- *First Match*: The rule engine fetches the first occurrence that matches the condition and the corresponding result is returned.
- *All Match*: The rule engine fetches all the occurrences that matches the condition and returns them as result.

Access Modes

The access modes provided to the value in the decision table settings should be either `Editable` or `Hidden`.

- The hidden access sets that default value to all the rows corresponding to the attribute in decision table, where the result column gets hidden. The default value is `mandatory`.
- The editable access sets that default value to the new row, which is created after the settings are applied. The default value is `optional`.

Basic and Advanced Mode

- The *Basic* mode is explained as follows
  - It provides the easy way of creating a rule in decision table, which is applicable only for conditions column.

- If the operator is not set in the settings of the decision table, then the operator list will be provided as a dropdown in the popover of the decision table row. Once the operator is selected, you will get the input field option to set the value for the column.
- Different options are provided to set the values based on the selected data type such as, input field for **string** and **number**, date picker for **date**, dropdown for **boolean**, time picker for **time**.
- If the expression is given in the input field for the data type string and number, then the expression can be validated by clicking on the popover itself. If the data is wrong, then the expression will not be taken and the popover gets closed.
- If the data type is string, the value can be entered without quotes.
- **Exist in** operator is not supported.
- Auto suggestion is limited to advance mode which is not available in the basic mode.

- The *Advanced* mode enables you to create a rule with the auto suggestion and value help feature, which is applicable only for result column.

> **i** Note
>
> Basic and advanced modes are applicable only for decision table rules that are modeled using rule expression language.

Step 1: Creating an Initial Rule Control

In this step, we embed a rule control into an application view.

The `RuleBuilder` component is a container of rules controls, which can bundle different visualization means. Currently the only visualization available is the decision table. The `RuleBuilder` defines common rules UI consumption patterns and APIs to be followed by UI developers.

In this example, we create a decision table, which will use the guided input mode.

> **i** Note
>
> Guided input mode is predefined in the settings for each condition column in the data for this tutorial.

Coding

You can view and download all files at Rule Builder - Guided Decision Table.

Page.view.xml

```
<mvc:View
        xmlns:mvc="sap.ui.core.mvc"
        displayBlock="true"
        xmlns="sap.m"
```

```
        controllerName="sap.rules.ui.sample.GuidedDecisionTable.Page"
        viewName="sap.rules.ui.sample.GuidedDecisionTable.Page.view"
        xmlns:rules="sap.rules.ui">
    <Button id="editButton" press="handleEditButton" text="Edit"/>
    <rules:RuleBuilder id="ruleBuilder" types="DecisionTable" editable="false"/>
</mvc:View>
```

We add to the view a `RuleBuilder` control of type `DecisionTable`.

The above code will place and render the `RuleBuilder` control in the view without data. Initially, the control is set to Display mode, which means that the decision table cannot be edited. The *Edit* button allows users to change the mode for editing.

Now we need to connect the `ExpressionLanguage` object association to the `RuleBuilder`, which we do in the next step, and then the control loads its data via the OData model.

Step 2: Associating the Expression Language

Expression languages provide the required services for rule authoring, rule visualization, and rule content validation. The services provided by the expression language object include expression validations, expression parsing, auto-complete suggestions, retrieval of expression metadata and tokens, and performing runtime services such as fetching data objects, outputs, and so on.

The vocabulary OData model and the binding context path for the specific vocabulary are mandatory input for the expression language.

The expression language objects are an association of the `RuleBuilder` object, and it can be associated with multiple `RuleBuilder` objects.

Preview

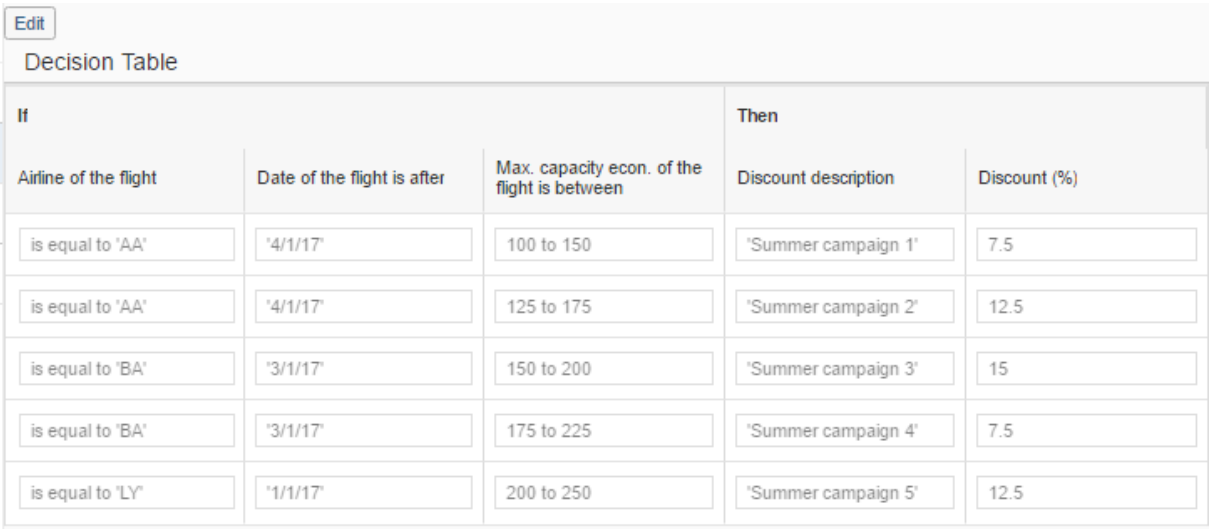


Figure 161: Decision Table with Guided Input

Coding

You can view and download all files at Rule Builder - Guided Decision Table. This is applicable only for rule expression language.

Before you begin, customize the `Page.controller.js` as per your requirements.

- Set the expression language object:
  For rule expression language:

  ```
  oExpressionLanguage = new sap.rules.ui.services.ExpressionLanguage();
  oRuleBuilder.setExpressionLanguage(oExpressionLanguage);
  ```

  For DMN SFEEL:

  ```
  oAstExpressionLanguage = new sap.rules.ui.services.AstExpressionLanguage();
  oRuleBuilder.setAstExpressionLanguage(oAstExpressionLanguage);
  ```

- Ensure that you have set the data before setting the vocabulary model for the expression language as shown:

  ```
  oExpressionLanguage.setData(data);
  oExpressionLanguage.setModel(that.oVocabularyModel);
  ```

Page.controller.js

```
sap.ui.define([
    'jquery.sap.global',
    'sap/ui/core/mvc/Controller',
    'sap/ui/model/odata/v2/ODataModel',
    'sap/rules/ui/services/ExpressionLanguage',   //For DMN SFEEL language, use
'AstExpressionLanguage'.
    'sap/ui/core/util/MockServer',
    'sap/m/MessageToast'
], function (jQuery, Controller, ODataModel, ExpressionLanguage, MockServer,
MessageToast) {    //For DMN SFEEL language, use 'AstExpressionLanguage' instead
of 'ExpressionLanguage'.
    "use strict";
    return Controller.extend("sap.rules.ui.sample.GuidedDecisionTable.Page", {
        onInit: function () {
            sap.ui.getCore().applyTheme("sap_belize");
            // apply compact density for desktop, the cozy design otherwise
            this.getView().addStyleClass(sap.ui.Device.system.desktop ?
"sapUiSizeCompact" : "sapUiSizeCozy");
            var mPath =
jQuery.sap.getModulePath("sap.rules.ui.sample.GuidedDecisionTable", "/");

            // Initialize Expression Language services
            this.oVocabularyMockServer = new MockServer({rootUri: "/sap/opu/
odata/SAP/vocabulary_srv/"});
            this.oVocabularyMockServer.simulate(
                mPath + "localService/vocabulary/metadata.xml",
                {'sMockdataBaseUrl': mPath + "localService/vocabulary/mockdata/"}
            );
            this.oVocabularyMockServer.start();
            this.oVocabularyModel = new ODataModel("/sap/opu/odata/SAP/
vocabulary_srv/");
            this.oExpressionLanguage = new ExpressionLanguage();                //
For DMN SFEEL, use 'new AstExpressionLanguage();'.
            this.oExpressionLanguage.setModel(this.oVocabularyModel);
            this.oExpressionLanguage.setBindingContextPath("/
Vocabularies('FA163E38C6481EE785F409DCAD583D43')");
```

```
            // Initialize the Rule Builder
            this.oRuleMockServer = new MockServer({rootUri: "/sap/opu/odata/SAP/
RULE_SRV/"});
            this.oRuleMockServer.simulate(
                mPath + "localService/rule/metadata.xml",
                {'sMockdataBaseUrl': mPath + "localService/rule/mockdata/"}
            );
            var aRequests = this.loadRequests(mPath);
            this.oRuleMockServer.setRequests(aRequests);
            this.oRuleMockServer.start();
            this.oRuleModel = new ODataModel({
                serviceUrl: "/sap/opu/odata/SAP/RULE_SRV/",
                defaultBindingMode: sap.ui.model.BindingMode.TwoWay
            });
            var oRuleBuilder = this.byId("ruleBuilder");
            oRuleBuilder.setModel(this.oRuleModel);
            oRuleBuilder.setExpressionLanguage(this.oExpressionLanguage);
            oRuleBuilder.setBindingContextPath("/
Rules(Id='FA163E38C6481EE785F409DCAD583D43',Version='000001')");
        },
        handleEditButton: function () {
            var oEditButton = this.byId("editButton");
            var oRuleBuilder = this.byId("ruleBuilder");
            var bEdit = (oEditButton.getText() === "Edit");
            oRuleBuilder.setEditable(bEdit);
            oEditButton.setText(bEdit ? "Display" : "Edit");
        },
        onAfterRendering: function () {
            // Line actions are not supported in this demo
            var oRuleBuilder = this.byId("ruleBuilder");
            var oDecisionTable = oRuleBuilder.getAggregation("_rule");
            var oToolbar = oDecisionTable.getAggregation("_toolbar");
            var arrContent = oToolbar.getContent();
            for (var i = 0; i < arrContent.length; i++) {
                if (arrContent[i].getMetadata().getName() === "sap.m.Button") {

arrContent[i].detachPress(arrContent[i].mEventRegistry.press[0].fFunction,
arrContent[i].mEventRegistry.press[0].oListner);
                    arrContent[i].attachPress(function (oEvent) {
                            var msg = 'Line action pressed';
                            MessageToast.show(msg);
                        }
                    );
                } else if (arrContent[i].getMetadata().getName() ===
"sap.m.MenuButton") {
                    var oMenu = arrContent[i].getMenu();

oMenu.detachItemSelected(oMenu.mEventRegistry.itemSelected[1].fFunction,
oMenu.mEventRegistry.itemSelected[1].oListner);
                    oMenu.attachItemSelected(function (oEvent) {
                            var msg = 'Line action pressed';
                            MessageToast.show(msg);
                        }
                    );
                }
            }
        },
        loadRequests: function (mPath) {
            // The mock server does not support 1 to 1 navigation.
            // Hence we provide the responses directly by adding custom requests
to the MockServer
            var oRresponses = jQuery.sap.sjax({
                type: "GET",
                url: mPath + "localService/rule/responses.json",
                dataType: "json"
                }
            ).data;
```

```
            var aRequests = this.oRuleMockServer.getRequests();
            var sMethod = "GET";
            var sPath = /Rules\
(Id='FA163E38C6481EE785F409DCAD583D43',Version='000001'\)\/DecisionTable\/
DecisionTableRows\/\$count/;
            var fnResponse1 = function (xhr) {
                xhr.respond(200, {
                    "Content-Type": "text/plain;charset=utf-8"
                }, "5");
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse1});

            sPath = /Rules\
(Id='FA163E38C6481EE785F409DCAD583D43',Version='000001'\)\/DecisionTable\/
DecisionTableRows\?\$skip=0&\$top=\d+&\$orderby=Sequence%20asc&\$expand=Cells/;
            var response_1 = this.response_1;
            var fnResponse2 = function (xhr) {
                xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_1);
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse2});
            sPath = /Rules\
(Id='FA163E38C6481EE785F409DCAD583D43',Version='000001'\)\/DecisionTable\/
DecisionTableColumns\/\$count/;
            var fnResponse3 = function (xhr) {
                xhr.respond(200, {
                    "Content-Type": "text/plain;charset=utf-8"
                }, "5");
            }
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse3});
            sPath = /Rules\
(Id='FA163E38C6481EE785F409DCAD583D43',Version='000001'\)\/DecisionTable\/
DecisionTableColumns\?\$skip=0&\$top=\d+&\$expand=Condition%2cResult/;
            var response_2 = this.response_2;
            var fnResponse4 = function (xhr) {
                xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_2);
            }
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse4});
            return aRequests;
        }
    });
});
```

This code adds an expression language object to the view controller, and connects it to the `RuleBuilder` as an association. For the expression language service, this code sets the model and then does all the necessary data binding internally (unlike other SAPUI5 controls where the developer defines the data binding). The data for the expression language is loaded via the vocabulary OData service.

The following are the code modifications that you can make to include additional functionalities:

- For reading specific vocabulary content like data objects, attributes, value help, rules or vocabulary rules, use the following code:

```
this.oVocabularyModel.read(sVocabularyPath, {
                urlParameters: {
                    "$expand": "DataObjects/Associations,DataObjects/
Attributes,ValueSources,Rules"
                },
```

- To set the vocabulary context shown in autosuggestion, use the following code:

```
oExpressionLanguage.setBindingContextPath("/Vocabularies(\'" + <vocabulary
context> + "\')");
```

> **i Note**
>
> Vocabulary context is `ProjectId` by default and all the data objects of the given project are listed in autosuggestion. The possible contexts include `RuleserviceId`, `RuleId` and `ProjectId`.

The code also sets a binding context path on `RuleBuilder` to the specific rule you are currently working on.

Now, when running the application, the decision table will be rendered filled with rule data from the mock server.

> **i Note**
>
> If you are using a back-end system with the relevant implemented OData services, use the following code:

```
sap.ui.define([
    'sap/ui/core/mvc/Controller',
    'sap/ui/model/odata/v2/ODataModel',
    'sap/rules/ui/services/ExpressionLanguage'         //For DMN SFEEL
language, use 'sap/rules/ui/services/AstExpressionLanguage'.
], function ( Controller, ODataModel, ExpressionLanguage) {   //For DMN SFEEL
language, use 'AstExpressionLanguage' instead of 'ExpressionLanguage'.
    "use strict";
    return Controller.extend("mySample.RuleBuilder.Page", {
        onInit: function () {
            // apply compact density for desktop, the cozy design otherwise
            this.getView().addStyleClass(sap.ui.Device.system.desktop ?
"sapUiSizeCompact" : "sapUiSizeCozy");

            // Initialize Expression Language services
            this.oVocabularyModel = new ODataModel("/vocabulary_srv/");
            this.oExpressionLanguage = new ExpressionLanguage();       //For
DMN SFEEL, use 'new AstExpressionLanguage()'.
            this.oExpressionLanguage.setModel(this.oVocabularyModel);
            this.oExpressionLanguage.setBindingContextPath("/
Vocabularies('<your rule ID>')");
            // Initialize the Rule Builder
            this.oRuleModel = new ODataModel({
                serviceUrl: "/sap/opu/odata/SAP/RULE_SRV/",
                defaultBindingMode: sap.ui.model.BindingMode.TwoWay
            });
            var oRuleBuilder = this.byId("ruleBuilder");
            oRuleBuilder.setModel(this.oRuleModel);
            oRuleBuilder.setExpressionLanguage(this.oExpressionLanguage);
            oRuleBuilder.setBindingContextPath("/Rules(Id='<your rule
ID>',Version='000001')");
        },
        handleEditButton: function () {
            var oEditButton = this.byId("editButton");
            var oRuleBuilder = this.byId("ruleBuilder");
            var bEdit = (oEditButton.getText() === "Edit");
            oRuleBuilder.setEditable(bEdit);
            oEditButton.setText(bEdit ? "Display" : "Edit");
        }
    });
});
```

Step 3: Changing the Decision Table Configuration

Decision table has a set of configurations that influence different aspects of the its functionality and the actions that business users can perform. The decision table configuration object has default values, which you can change. Part of the configuration of the decision table can be exposed to the end user via the decision table *Settings* dialog box.

Preview

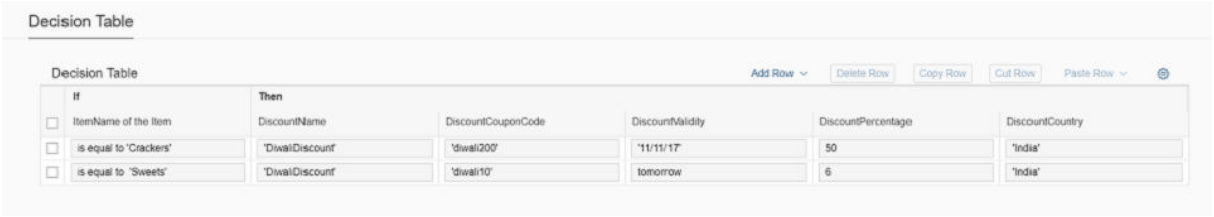


Figure 162: Decision Table with Settings Button

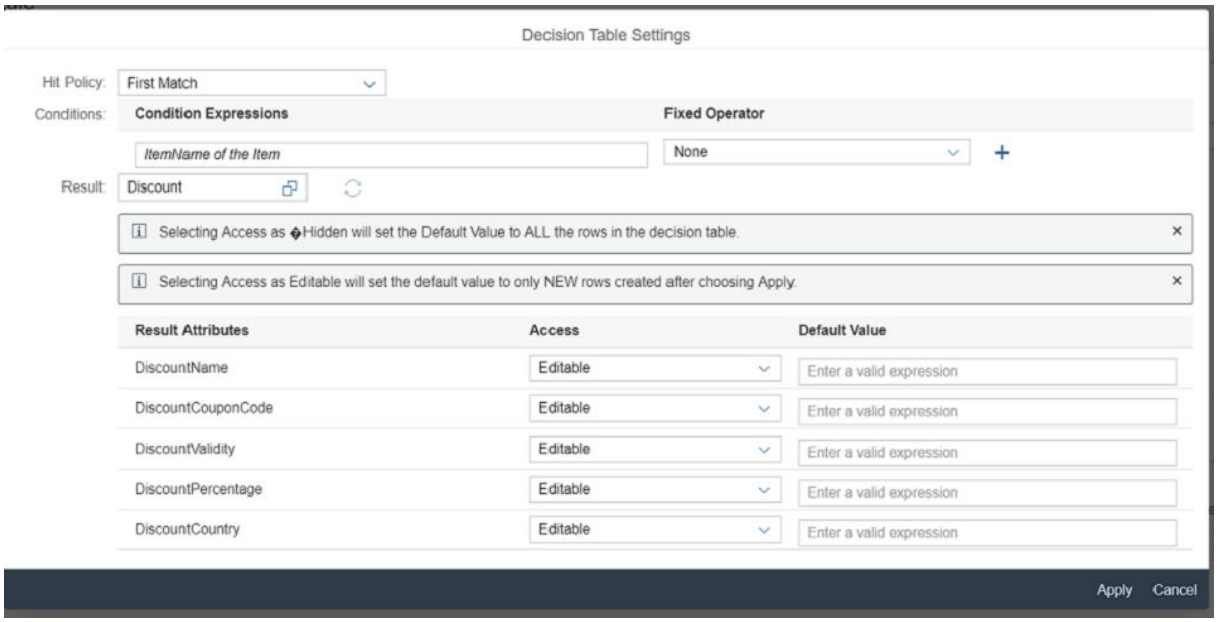


Figure 163: Decision Table Settings Dialog Box

Coding

You can view and download all files at Rule Builder - Guided Decision Table.

> **i Note**
>
> The new code described in this step is not included in the download sample files.

Page.view.xml

```
<mvc:View
        xmlns:mvc="sap.ui.core.mvc"
        displayBlock="true"
        xmlns="sap.m"
        controllerName="sap.rules.ui.sample.GuidedDecisionTable.Page"
        viewName="sap.rules.ui.sample.GuidedDecisionTable.Page.view"
        xmlns:rules="sap.rules.ui">
    <Button id="editButton" press="handleEditButton" text="Edit"/>
    <rules:RuleBuilder id="ruleBuilder" types="DecisionTable" editable="true">
        <rules:decisionTableConfiguration>
        <rules:DecisionTableConfiguration enableSettings="true"/>
        </rules:decisionTableConfiguration>
    </rules:RuleBuilder>
</mvc:View>
```

This code changes the decision table to display and enable the *Settings* icon ( ), which opens the *Settings* dialog box.

> **i** Note
>
> The Cell Format property of the decision table is deprecated from SAPUI5 version 1.52.8. The following is the new property is introduced.
>
> ```
> decisionTableFormat: {
>                     type: "sap.rules.ui.DecisionTableFormat",
>                     defaultValue: sap.rules.ui.DecisionTableFormat.CellFormat
>                 }
> ```
>
> The `enum` for type `sap.rules.ui.DecisionTableFormat` is
>
> ```
> sap.rules.ui.DecisionTableFormat = {
>         CellFormat: "CELLFORMAT",
>         RuleFormat: "RULEFORMAT"
>             };
> ```

The user has to set the the `enum` type to `RuleFormat` for the DecisionTable to be rendered based on rule format.

The value for the rule format is set at the time of rule creation to either basic or advanced mode depending on which the entire rule will be rendered based on basic mode or advanced mode.

Text Rule

Create a business logic by defining conditions (`if`, `else if`) in the form of text that has the result parts (`then`, `else`), which is associated with the rule expression language.

Features

Refresh Data Object

The refresh data object feature reads the attributes of the data object and automatically fetches the predefined result attributes.

Access Modes

The access modes provided to the value in the text rule settings should be either `Editable` or `Hidden`.

- The hidden access sets that default value to the attribute of data object in text rule, where it gets hidden from the text rule. The default value is `mandatory`.
- The editable access sets that default value to the new entries of data object which is created after the settings are applied and new `else` block is created. The default value is `optional`. When the result data object changes, by default all the attributes will have the access mode as `Editable` with no default value.

Operations

You can model a text rule without a result data object too. This type of text rule lets you perform operations on existing data objects or its attributes and is supported only in expression language 2.0. You can also add multiple result operations in a text rule.

> **i** Note
>
> Select *No Default Result* as the text rule result to perform the operations.

To view the list of operations that can be performed on any data object or attribute of your project, press `CTRL` + `SPACE` in the *Then* field of your text rule.

List of Operations:

Operation	Syntax	Input	Description	Example
Update	*UPDATE*(<Target>, <Source>)	`Target Entity`	The target data object or attribute that should be updated as per the value of the `Source Entity` or `Expression`.	*UPDATE*(`Employee_Table.Employee Name`, `Employee.Employee Name`)

Operation	Syntax	Input	Description	Example
		`Source Entity` or `Source Expression`.	The value of the `Target Entity` is updated to the value of the `Source Entity` or the value returned by the `Source Expression`. **i Note** You can also use vocabulary rules as the source expression by selecting it from the autosuggest list.	The data object, `Employee_Table.Employee Name` is updated as per the value of `Employee.Employee Name`.
Append	*APPEND*(<Target>, <Source>)	`Target Entity`	The target data object of type *Table* to which the `Source` has to be appended.	*APPEND*(`FlightTable, Flight`) The data object `Flight` is appended to the data object `FlightTable`.
		`Source Entity` or `Source Expression`	A data object that should be appended to the `Target` data object. Source data object should be of type *Structure* or *Table* or a rule that returns a data object of type *Structure* or *Table*.	

Value Help

While authoring a rule in advanced mode, the auto-complete suggestion list provides the corresponding list of values as value help items along with other auto-complete suggestions.

There are two types of value help:

- Value List : The values are created and maintained in the business rules by adding code and description.
- Service URL Mapping : The values are maintained outside the business rules and you have to configure the managed system to consume the values from external system. For more information on configuring managed system, see Configuring Managed System in SAP Cloud Platform Business Rules.

i Note

- The value list link is provided with the suggestions, only if the attribute is of type value help.
- The value help dialog can also be consumed by pressing **f4** while authoring a rule.
- To change the value help while authoring the rule, you can also click the value help attribute which opens the value help dialog.

- In the following example, the ID of an equipment is an attribute of type value help. While authoring the rule, the value help is provided in the suggestion which has the value list with descriptions as shown in the table. The value can be selected from the list and consumed in the rule.

```
If
        ID of an Equipment is equal to 'E001'
        Then
        Order
```

Table 10: Value List

Code	Description
E001	Smartphone
E002	Laptop
E003	Tablet
E004	Smartphone

- You can search for the value and description in the search bar.
- The advanced filter option is used to filter a value and description based on conditions which can be included or excluded. It can be applied in both value and description field.
  For Example:
  If you want to filter the description `Smartphone` from the list which has `Smartphone`, `Laptop`, and `Tablet`, then the condition can be applied in both the value and description field as the following.
  Include: Description is equal to `Smartphone`.
  Exclude: Value is equal to `E004`.
  So that the filtered result is `E001`, `Smartphone`.
- The value column can be sorted in both ascending and descending order.

> **i** Note
>
> The search and filter options are case sensitive.

Date Control

While authoring a rule in advanced mode, the auto-complete suggestion list of the Rule Builder UI control provides the date link where the value can be picked from the calendar. This date link is provided in the suggestion list, only if the attribute is of type date.

In the following example, the DOB of an employee is an attribute of type date. While authoring the rule, the date link is provided in the suggestion which opens a calendar from where the value can be selected and consumed in the rule.

```
    If
    DOB of an Employee is equal to '01/01/18'
    Then
    Discount
```

Step 1: Creating a Rule Control

In this step, we embed rule builder of type `TextRule` into an application view.

Coding

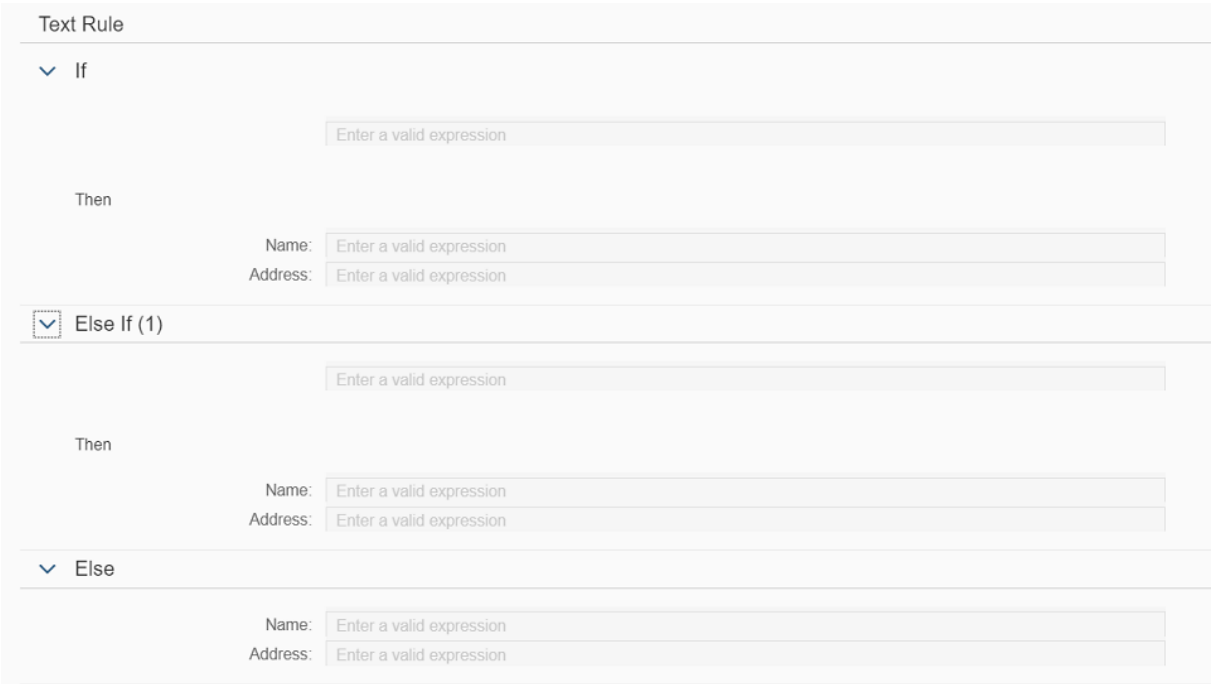
You can view and download all files at Rule Builder - Text Rule.

Page.view.xml

```
<mvc:View
        xmlns:mvc="sap.ui.core.mvc"
        displayBlock="true"
        xmlns="sap.m"
        controllerName="sap.rules.ui.sample.TextRule.Page"
        viewName="sap.rules.ui.sample.TextRule.Page.view"
        xmlns:rules="sap.rules.ui">
    <Button id="editButton" press="handleEditButton" text="Edit"/>
    <rules:RuleBuilder id="ruleBuilder" types="TextRule" editable="false"/>
</mvc:View>
```

Step 2: Associating the Expression Language

Preview



Coding

You can view and download all files at Rule Builder - Text Rule.

Page.controller.js

This code adds an expression language object to the view controller, and connects it to the `RuleBuilder`. For the expression language object, this code sets the model and then does all the necessary data binding internally (unlike other SAPUI5 controls where the developer defines the data binding). The data for expression language object is loaded via the vocabulary OData service.

Before you begin, customize the `Page.controller.js` as per your requirements.

- Set the expression language object:
  For rule expression language:

  ```
  oExpressionLanguage = new sap.rules.ui.services.ExpressionLanguage();
  oRuleBuilder.setExpressionLanguage(oExpressionLanguage);
  ```

  For DMN SFEEL:

  ```
  oAstExpressionLanguage = new sap.rules.ui.services.AstExpressionLanguage();
  oRuleBuilder.setAstExpressionLanguage(oAstExpressionLanguage);
  ```

- Ensure that you have set the data before setting the vocabulary model for the expression language as shown:

  ```
  oExpressionLanguage.setData(data);
  oExpressionLanguage.setModel(that.oVocabularyModel);
  ```

> **i Note**
>
> Text rule will not load if the batch mode is disabled. Set `setUseBatch` to `true` in the OData model used to enable batch mode.

The code also sets a binding context path on `RuleBuilder` to the specific rule you are currently working on.

```
 sap.ui.define([
    'jquery.sap.global',
    'sap/ui/core/mvc/Controller',
    'sap/ui/model/odata/v2/ODataModel',
    'sap/rules/ui/services/ExpressionLanguage',     //For DMN SFEEL, use 'sap/
rules/ui/services/AstExpressionLanguage'.
    'sap/ui/core/util/MockServer',
    'sap/m/MessageToast'
], function (jQuery, Controller, ODataModel, ExpressionLanguage, MockServer,
MessageToast) {   //For DMN SFEEL, use 'AstExpressionLanguage' instead of
'ExpressionLanguage'.
    "use strict";
    return Controller.extend("sap.rules.ui.sample.TextRule.Page", {
        /**
         * This sample uses the sap.ui.core.uti.MockServer. The RuleBuilder
control is meant to be used
         * with the Vocabulary OData service and the Rules OData service.
         * Hence, when using th eproper OData services the mockServer code
should be removed.
```

```
         */
        onInit: function () {
            sap.ui.getCore().applyTheme("sap_belize");
            // apply compact density for desktop, the cozy design otherwise
            this.getView().addStyleClass(sap.ui.Device.system.desktop ?
"sapUiSizeCompact" : "sapUiSizeCozy");
            var mPath = jQuery.sap.getModulePath("sap.rules.ui.sample.TextRule",
"/");
            // Initialiaze Expression Language services
            this.oVocabularyMockServer = new MockServer({rootUri: "/rule-service/
vocabulary_srv/"});
            this.oVocabularyMockServer.simulate(
                mPath + "localService/vocabulary/mockdata/metadata.xml",
                {'sMockdataBaseUrl': mPath + "localService/vocabulary/mockdata/"}
            );
            this.oVocabularyMockServer.start();
            this.oVocabularyModel = new ODataModel("/rule-service/
vocabulary_srv/");
            this.oExpressionLanguage = new ExpressionLanguage();         //For
DMN SFEEL, use 'new AstExpressionLanguage();'.
            this.oExpressionLanguage.setModel(this.oVocabularyModel);
            this.oExpressionLanguage.setBindingContextPath("/
Vocabularies('<project-id>')");
            // Initialiaze the Rule Builder

            this.oRuleMockServer = new MockServer({rootUri: "/rule-service/
rule_srv/"});
            this.oRuleMockServer.simulate(
                mPath + "localService/rule/mockdata/metadata.xml",
                {'sMockdataBaseUrl': mPath + "localService/rule/mockdata/"}
            );
            var aRequests = this.loadRequests(mPath);
            this.oRuleMockServer.setRequests(aRequests);
            this.oRuleMockServer.start();
            this.oRuleModel = new ODataModel({
                serviceUrl: "/rule-service/rule_srv/",
                defaultBindingMode: sap.ui.model.BindingMode.TwoWay
            });
            var oRuleBuilder = this.byId("ruleBuilder");
            oRuleBuilder.setModel(this.oRuleModel);
            oRuleBuilder.setExpressionLanguage(this.oExpressionLanguage);
            oRuleBuilder.setBindingContextPath("/Projects(Id='<project-
id>',Version='<project-version>')/Rules(Id='<rule-id>',Version='<rule-
version>')");
        },
        handleEditButton: function () {
            var oEditButton = this.byId("editButton");
            var oRuleBuilder = this.byId("ruleBuilder");
            var bEdit = (oEditButton.getText() === "Edit");
            oRuleBuilder.setEditable(bEdit);
            oEditButton.setText(bEdit ? "Display" : "Edit");
        },
        onAfterRendering: function () {
            /**
             * Line actions are not supported in this demo as they require a
functioning Rules oData service
             * This function overwites the line actions event handlers.
             * Please do not use this code when using proper OData services.
             */
            var oRuleBuilder = this.byId("ruleBuilder");
            var oDecisionTable = oRuleBuilder.getAggregation("_rule");
        },
        loadRequests: function (mPath) {
            // The mock server does not support 1 to 1 navigation.
            // Hence we provide the responses directly by adding custom requests
to the MockServer
            var oRresponses = jQuery.sap.sjax({
                type: "GET",
```

```
                url: mPath + "localService/rule/mockdata/responses.json",
                dataType: "json"
            }).data;
            var aRequests = this.oRuleMockServer.getRequests();
            var sMethod = "GET";
            var sPath = /Projects\(Id='<project-id>',Version='<project-
version>'\)\/Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/
TextRuleConditions\/\$count/;
            var fnResponse1 = function (xhr) {
                xhr.respond(200, {
                    "Content-Type": "text/plain;charset=utf-8"
                }, "3");
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse1});

            sPath = /Projects\(Id='<project-id>',Version='<project-version>'\)\/
Rules\(Id='<rule-id>',Version='<rule-version>'\)\?\$expand=TextRule/;
            var fnResponse2 = function (xhr) {
                xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_7);
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse2});

            var sPath = /Projects\(Id='<project-id>',Version='<project-
version>'\)\/Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/
TextRuleResults\/\$count/;
            var fnResponse3 = function (xhr) {
                xhr.respond(200, {
                    "Content-Type": "text/plain;charset=utf-8"
                }, "2");
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse3});

            sPath = /Projects\(Id='<project-id>',Version='<project-version>'\)\/
Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/TextRuleResults/;
            var fnResponse4 = function (xhr) {
                xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_3);
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse4});

            /Projects\(Id='<project-id>',Version='<project-version>'\)\/Rules\
(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/TextRuleConditions\?\
$expand=TextRuleResultExpressions/;
            var fnResponse5 = function (xhr) {
                xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_2);
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse5});

            sPath = /Projects\(Id='<project-id>',Version='<project-version>'\)\/
Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/TextRuleConditions\
(RuleId='<rule-id>',RuleVersion='<rule-version>',Id='1'\)\/
TextRuleResultExpressions\$count/;
            var fnResponse6 = function (xhr) {
                xhr.respond(200, {
                    "Content-Type": "text/plain;charset=utf-8"
                }, "2");
            };
```

```
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse6});

            sPath = /Projects\(Id='<project-id>',Version='<project-version>'\)\/
Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/TextRuleConditions\
(RuleId='<rule-id>',RuleVersion='<rule-version>',Id='1'\)\/
TextRuleResultExpressions\?\$skip=0&\$top=100/;
            var fnResponse7 = function (xhr) {
                 xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_8);
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse7});

            sPath = /Projects\(Id='<project-id>',Version='<project-version>'\)\/
Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/TextRuleConditions\
(RuleId='<rule-id>',RuleVersion='<rule-version>',Id='2'\)\/
TextRuleResultExpressions\$count/;
            var fnResponse8 = function (xhr) {
                xhr.respond(200, {
                    "Content-Type": "text/plain;charset=utf-8"
                }, "2");
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse8});

            sPath = /Projects\(Id='<project-id>',Version='<project-version>'\)\/
Rules\(Id='<rule-id>',Version='<rule-version>'\)\/TextRule\/TextRuleConditions\
(RuleId='<rule-id>',RuleVersion='<rule-version>',Id='2'\)\/
TextRuleResultExpressions\?\$skip=0&\$top=100/;
            var fnResponse9 = function (xhr) {
                 xhr.respondJSON(200, {
                    "Content-Type": "application/json;charset=utf-8"
                }, oRresponses.response_9);
            };
            aRequests.push({method: sMethod, path: sPath, response:
fnResponse9});

            return aRequests;
        }
    });
});
```

The following are the code modifications that you can make to include additional functionalities:

- For reading specific vocabulary content like data objects, attributes, value help, rules or vocabulary rules, use the following code:

```
this.oVocabularyModel.read(sVocabularyPath, {
                urlParameters: {
                    "$expand": "DataObjects/Associations,DataObjects/
Attributes,ValueSources,Rules"
                },
```

- To set the vocabulary context shown in autosuggestion, use the following code:

```
oExpressionLanguage.setBindingContextPath("/Vocabularies(\'" + <vocabulary
context> + "\')");
```

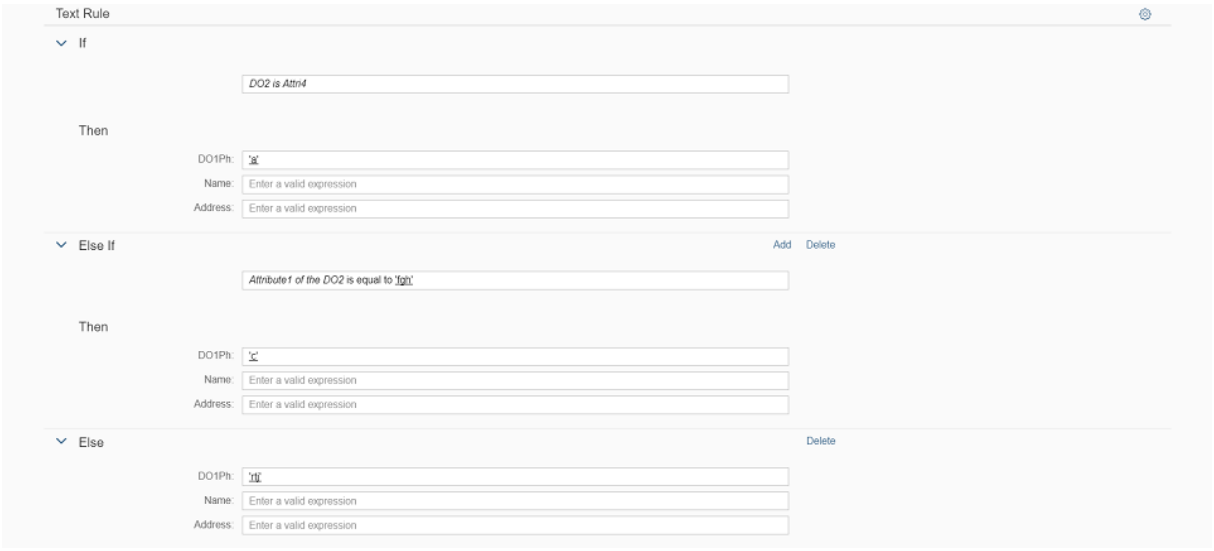
> **i Note**
>
> Vocabulary context is `ProjectId` by default and all the data objects of the given project are listed in autosuggestion. The possible contexts include `RuleserviceId`, `RuleId` and `ProjectId`.

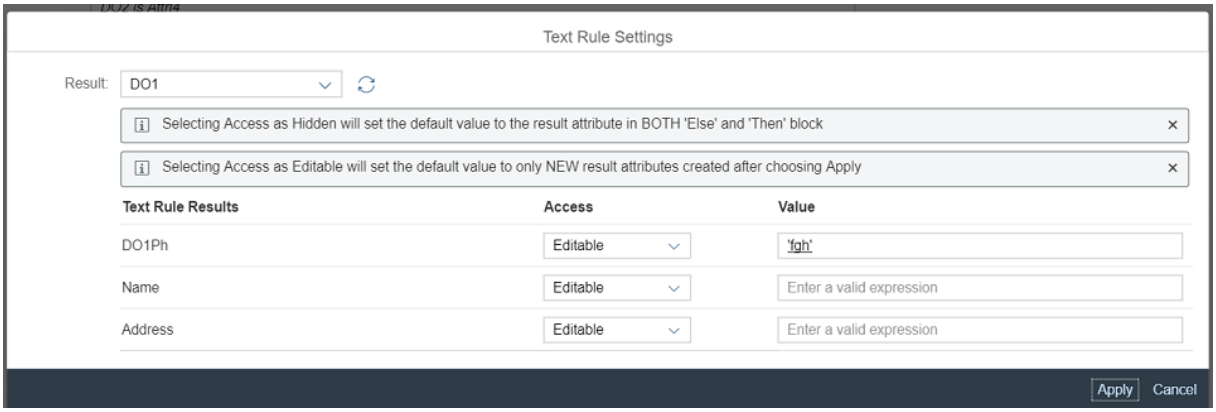
Step 3: Changing the Text Rule Configuration

Text rule has a set of configurations that influence different aspects of the its functionality and the actions that business users can perform. Part of the configuration of the text rule can be exposed to the end user via the text rule *Settings* dialog box.

Text rule has a condition part (`if, elseif`) and the result parts (`then, else`). In this type of rule authoring, the user can create a text rule which has different semantic fonts, colors for vocabulary (data), grammar (reserved words), inline error highlighting and provides the autocomplete feature to support user input for the rules (condition & results).

Preview





Coding

You can view and download all files at Rule Builder - Text Rule.

Page.view.xml

```
<mvc:View
        xmlns:mvc="sap.ui.core.mvc"
        displayBlock="true"
        xmlns="sap.m"
        controllerName="sap.rules.ui.sample.TextRule.Page"
        viewName="sap.rules.ui.sample.TextRule.Page.view"
        xmlns:rules="sap.rules.ui">
    <Button id="editButton" press="handleEditButton" text="Edit"/>
    <rules:RuleBuilder id="ruleBuilder" types="TextRule" editable="true">
        <rules:textRuleConfiguration>
        <rules:textRuleConfiguration enableSettings="true" enableElseIf=
"false"/>
        </rules:textRuleConfiguration>
    </rules:RuleBuilder>
</mvc:View>
```

The `enableSettings` option is `false` by default and can be set to `true` which enables the *Settings* icon, and opens the *Settings* dialog box. The `enableElse` and `enableElseIf` options are `true` by default and can be set to `false` which disables the `Else` and `ElseIf` parts in the condition.

Multiple `ElseIf` statements can also be configured in the condition.

Summary

We have now embedded a `TextRule` control in our application. We created a control that uses text input and bound it to data.

We also added a *Settings* button. You can click the *Settings* button to change settings for the text rule.

Smart Controls

In this tutorial you learn how to work with smart controls.

Smart controls are a specific category of SAPUI5 controls that have some special features in addition to the standard SAPUI5 features and thus make it easier to use the control in certain scenarios.

A primary example of such a feature is OData support: Typically, a smart control interprets OData metadata. In some cases, a smart control even persists an adapted version of the user interface that the user has defined for later usage.

In this tutorial, you see examples that should make the term “smart” even more tangible.

> → Tip
>
> You don't have to do all tutorial steps sequentially, you can also jump directly to any step you want. Just download the code from the previous step, copy it to your workspace and make sure that the application runs by calling the `webapp/index.html` file.

You can view and download the files for all steps in the *Samples* in the Demo Kit at Smart Controls. Depending on your development environment you might have to adjust resource paths and configuration entries.

For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Prerequisites

Preparation steps for the *Smart Controls* tutorial

You have already gone through the following tutorials:

- Walkthrough [page 69]
- Data Binding [page 219]

In addition, you need some background knowledge about OData and annotations that you can find here: http://www.sap.com/protocols/SAPData.

> **i Note**
>
> The smart controls require a default OData model, and named models are not supported.

You need a web server to host the files that are created in the tutorial steps, and you need the relevant SAPUI5 libraries, of course.

Please note that for each step there is a separate folder with its own copy of the files used.

File Structure

For each step we will create the following files:

- webapp (folder)
  - test (folder)
    - service (folder)
      - `metadata.xml`
      - `<Entity>.json` (could be more than one file)
      - server.js
  - `Component.js`
  - `index.html`
  - `manifest.json`
  - `<ControlName>.controller.js`
  - `<ControlName>.view.xml`
- `Component.js`

In some steps, additional files are needed. They will be explained in those steps. Since many of these files are almost identical from step to step, we show their content here but only briefly point to interesting points in these files if necessary. Please refer to the *Walkthrough* tutorial for further details on the general setup and the content of the outer `Component.js` file (the top-level file on the same level as the webapp folder) in which we define that the `index.html` will be shown in an iFrame.

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta charset="utf-8">
<title>SmartControls</title>
<script id="sap-ui-bootstrap"
    src="../../../../../../../../../resources/sap-ui-core.js"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-libs="sap.m, sap.ui.comp"
    data-sap-ui-bindingSyntax="complex"
    data-sap-ui-compatVersion="edge"
    data-sap-ui-preload="async"
    data-sap-ui-resourceroots='{
        "sap.ui.demo.smartControls": "./"
    }'>
</script>
<script>
    sap.ui.getCore().attachInit(function() {
        sap.ui.require([
            "sap/ui/demo/smartControls/test/service/server"
        ], function(server) {
            server.init();
            new sap.ui.core.ComponentContainer({
                name: "sap.ui.demo.smartControls",
                height: "100%"
            }).placeAt("content");
        });
    });
</script>
</head>
<body class="sapUiBody" id="content">
</body>
</html>
```

In this index file, you will recognize that we reference the library `sap.ui.comp` since this is the **main library** for the smart controls. This `index.html` file references the `Component.js` (through the name: "sap.ui.demo.smartControls") which always looks like this:

Component.js

```
sap.ui.define([
    "sap/ui/core/UIComponent"
], function(UIComponent) {
    "use strict";
    return UIComponent.extend("sap.ui.demo.smartControls.Component", {
        metadata: {
```

```
            manifest: "json"
        }
    });
});
```

In the `Component.js` file we use the `manifest.json` file.

manifest.json

```
{
  "_version": "1.8.0",
  "sap.app": {
    "id": "sap.ui.demo.smartControls",
    "type": "application",
    "title": "SAPUI5 Smart Controls",
    "description": "A simple app that explains the most important concepts of
smart controls in SAPUI5",
    "applicationVersion": {
        "version": "1.0.0"
    },
    "dataSources": {
      "mainService": {
        "uri": "/here/goes/your/serviceUrl/",
        "type": "OData",
        "settings": {
          "odataVersion": "2.0"
          "localUri": "localService/metadata.xml"
        }
      }
    }
  },
  "sap.ui": {
    "technology": "UI5"
  },
  "sap.ui5": {
    "rootView": {
        "viewName": "sap.ui.demo.smartControls.SmartField",
        "type": "XML"
        "async": true
    },
    "dependencies": {
      "minUI5Version": "1.30",
      "libs": {
        "sap.m": {},
        "sap.ui.comp": {}
      }
    },
    "models": {
       "": {
        "dataSource": "mainService",
        "settings": {
            "defaultBindingMode": "TwoWay"
        }
      }
    }
  }
}
```

In the `manifest.json` file we define the `rootView` and also the model of the application. Please note that the `TwoWay` binding mode ensures that an input validation is done automatically based on the metadata.

The last file that we wish to list here is the `server.js`:

server.js

```
sap.ui.define([
    "sap/ui/core/util/MockServer"
], function (MockServer) {
    "use strict";
    return {
        init: function () {
            // create
            var oMockServer = new MockServer({
                rootUri: "/here/goes/your/serviceUrl/"
            });
            // configure
            MockServer.config({
                autoRespond: true,
                autoRespondAfter: 1000
            });
            // simulate
            var sPath =
jQuery.sap.getModulePath("sap.ui.demo.smartControls.test.service");
            oMockServer.simulate(sPath + "/metadata.xml", sPath);
            // start
            oMockServer.start();
        }
    };
});
```

In this file we define the `MockServer` handling the server requests.

> **i** Note
>
> You might notice the rather flat setup of the files that is different from the setup propagated in the *Walkthrough* tutorial where the `Model-View-Controller` paradigm is reflected in the folder structure. We have chosen the flat setup for this tutorial since our examples all have exactly **one** single file for the view and **one** single file for the controller. Additional folders would add a complexity that we would like to avoid.

Step 1: Smart Field

One important building block of smart controls is the `SmartField` control that, depending on the OData metadata defined, allows you to render other controls and, for example, define fields with certain attributes based on the metadata.

So we start our journey into the world of smart controls by looking at the `SmartField` control and, in particular, an example in which a price together with its currency is displayed. Later we will see more complex examples.

Preview

Figure 164: Smart Field

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 1 - Smart Field.

SmartField.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.smartControls.SmartField"
    xmlns="sap.m"
    xmlns:smartForm="sap.ui.comp.smartform"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:sap.ui.layout="sap.ui.layout"
    xmlns:smartField="sap.ui.comp.smartfield">
    <smartForm:SmartForm editable="true">
        <smartForm:layout>
            <smartForm:ColumnLayout
                emptyCellsLarge="4"
                labelCellsLarge="4"
                columnsM="1"
                columnsL="1"
                columnsXL="1"/>
        </smartForm:layout>
        <smartForm:Group>
            <smartForm:GroupElement>
                <smartField:SmartField value="{Price}" id="idPrice"/>
            </smartForm:GroupElement>
        </smartForm:Group>
    </smartForm:SmartForm>
</mvc:View>
```

This view basically specifies a form with an appropriate layout, the content of which consists of a `SmartLabel` control along with a `SmartField` control . The connection between `SmartLabel` and `SmartField` is essential since metadata for `SmartLabel` is controlled via the binding of `SmartField`. `SmartField` and `SmartLabel` are connected by `id` and `labelFor`, respectively, in this case `idPrice`.

SmartField.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.smartControls.SmartField", {
        onInit: function() {
            this.getView().bindElement("/Products('4711')");
```

```
        }
    });
});
```

In the `SmartField.controller.js` file you will see that we bind the view to `"/Products('4711')"`.

metadata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0"
           xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
           xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/
metadata"
           xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema Namespace="com.sap.wt01"
                sap:schema-version="1" xmlns="http://schemas.microsoft.com/ado/
2008/09/edm">
            <EntityType Name="Product">
                <Key>
                    <PropertyRef Name="ProductId"/>
                </Key>
                <Property Name="ProductId" Type="Edm.String"/>
                <Property Name="Price" Type="Edm.String"
                          sap:unit="CurrencyCode" MaxLength="3" sap:label="Price"
                          sap:updatable="true"/>
                <Property Name="CurrencyCode" Type="Edm.String"
                          MaxLength="3" sap:label="Currency"
sap:semantics="currency-code"
                          sap:updatable="true"/>
            </EntityType>
            <EntityContainer m:IsDefaultEntityContainer="true"
                             sap:supported-formats="atom json">
                <EntitySet Name="Products" EntityType="com.sap.wt01.Product"
                           sap:updatable="true"/>
            </EntityContainer>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

The metadata document corresponds to the `$metadata` document of your OData service. You will find the `Price` and the `CurrencyCode` metadata here, and, in particular, you will see that the `Price` property defines `CurrencyCode` as its unit. The relationship is automatically picked up by `SmartField`, and it decides on the particular rendering of the price along with with its currency based on additional metadata, such as `Type` and `MaxLength`. With `sap:updatable="true"` we define that the field is editable; `sap:updatable="false"` would indicate that the field is read-only. All Boolean-like properties default to true if not specified otherwise. To make this clearer, however, we sometimes still include them in the `metadata.xml` document, like `sap:updatable="true"` in this case. Later, we will see other examples of how the `SmartField` control shows a "smart" behavior as to which controls are rendered on the UI.

Products.json

```
[{
```

```
    "ProductId": "4711",
    "Price": 856.49,
    "CurrencyCode": "EUR"
}]
```

Finally, we include the `Products.json` file (as referred to in the `metadata.xml` as `EntitySet`) in our example, which contains the data shown on the UI.

Related Information

Smart Field [page 2410]

Step 2: Smart Field with Value Help

You can use the `SmartField` control in combination with the `ValueHelpDialog` control that allow you to carry out a complex search in order to identify the value you are looking for.

We would like to stress the importance of this feature with a dedicated example, even though you might argue that this is just another feature of the `SmartField` control. In the following example, we see a value help for a currency code. By providing a value help, the user can find the correct currency by firing a query with complex input parameters. Surely, you can imagine that there are more complex examples in which specifying the correct value is almost impossible without performing a query. For example, when providing a customer ID, you would want to find this based on the last name of the customer.

We would like to emphasize here that the response to the query input (whether it is the response to the query input for the main table, as we will see later, or whether it is related to the value help, as we will see here) heavily depends on the server handling the request. In our case, the `MockServer` is not a full-fledged implementation that handles all OData requests as the user might expect. Therefore, please bear in mind that the examples are intended for a tutorial and not a real application.

Preview

Figure 165: Smart Field with Value Help

There is a small icon right next to the currency code. After pressing this icon you see a dialog (the `ValueHelpDialog` control) on which a query can be executed.

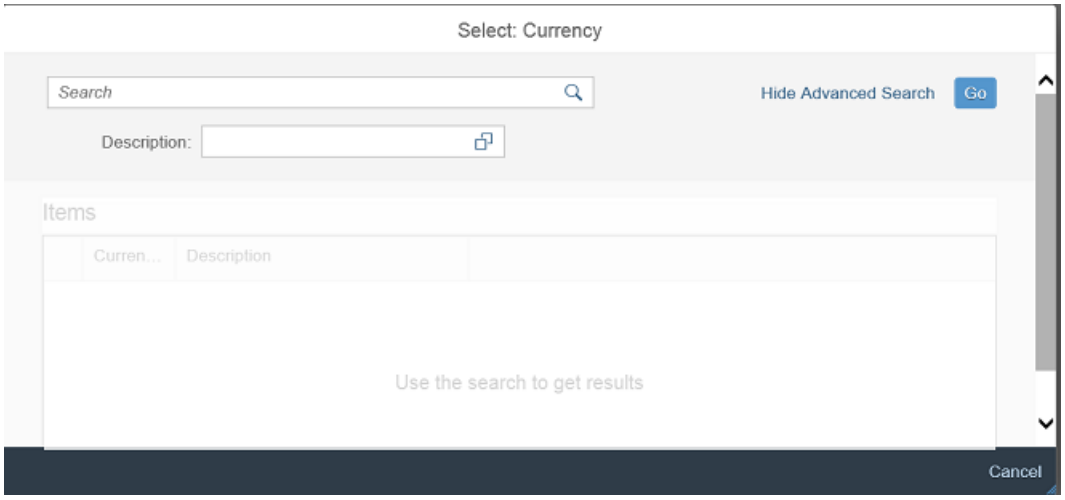


Figure 166: Value Help

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 2 - Smart Field with Value Help.

SmartFieldWithValueHelp.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.smartControls.SmartFieldWithValueHelp"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:smartForm="sap.ui.comp.smartform"
    xmlns:sap.ui.layout="sap.ui.layout"
    xmlns:smartField="sap.ui.comp.smartfield">
    <smartForm:SmartForm editable="true">
        <smartForm:layout>
            <smartForm:ColumnLayout
                emptyCellsLarge="4"
                labelCellsLarge="4"
                columnsM="1"
                columnsL="1"
                columnsXL="1"/>
        </smartForm:layout>
        <smartForm:Group>
            <smartForm:GroupElement>
                <smartField:SmartField value="{Price}" id="idPrice"/>
            </smartForm:GroupElement>
        </smartForm:Group>
    </smartForm:SmartForm>
</mvc:View>
```

For the view definition, we see that there is no difference to the previous example. This is an important fact since this exemplifies the underlying idea of what we mean by “smart”: Depending on the metadata, the control automatically adjusts its behavior.

SmartField.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";
    return
Controller.extend("sap.ui.demo.smartControls.SmartFieldWithValueHelp", {
        onInit: function() {
            this.getView().bindElement("/Products('4711')");
        }
    });
});
```

Again, this file is in essence identical with the controller in step 1.

metadata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema Namespace="com.sap.wt02"
            sap:schema-version="1" xmlns="http://schemas.microsoft.com/ado/
2008/09/edm">
            <EntityType Name="Product">
                <Key>
                    <PropertyRef Name="ProductId" />
                </Key>
                <Property Name="ProductId" Type="Edm.String" />
                <Property Name="Price" Type="Edm.String"
                    sap:unit="CurrencyCode" MaxLength="3" sap:label="Price"
                    sap:updatable="true" />
                <Property Name="CurrencyCode" Type="Edm.String"
                    MaxLength="3" sap:label="Currency" sap:semantics="currency-
code"
                    sap:updatable="true" />
            </EntityType>
            <EntityType Name="Currency">
                <Key>
                    <PropertyRef Name="CURR" />
                </Key>
                <Property Name="CURR" Type="Edm.String" MaxLength="4"
                    sap:display-format="UpperCase" sap:text="DESCR"
sap:label="Currency Code"
                    sap:filterable="false" />
                <Property Name="DESCR" Type="Edm.String" MaxLength="25"
                    sap:label="Description" />
            </EntityType>
            <EntityContainer m:IsDefaultEntityContainer="true"
                sap:supported-formats="atom json">
                <EntitySet Name="Products" EntityType="com.sap.wt02.Product" />
                <EntitySet Name="Currency" EntityType="com.sap.wt02.Currency" />
            </EntityContainer>
            <Annotations Target="com.sap.wt02.Product/CurrencyCode"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.Common.v1.ValueList">
                    <Record>
```

```
                        <PropertyValue Property="Label" String="Currency" />
                        <PropertyValue Property="CollectionPath"
String="Currency" />
                        <PropertyValue Property="SearchSupported" Bool="true" />
                        <PropertyValue Property="Parameters">
                            <Collection>
                                <Record
Type="com.sap.vocabularies.Common.v1.ValueListParameterOut">
                                    <PropertyValue Property="LocalDataProperty"
                                        PropertyPath="CurrencyCode" />
                                    <PropertyValue Property="ValueListProperty"
                                        String="CURR" />
                                </Record>
                                <Record

Type="com.sap.vocabularies.Common.v1.ValueListParameterDisplayOnly">
                                    <PropertyValue Property="ValueListProperty"
                                        String="DESCR" />
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
            </Annotations>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

As stated above, the metadata file is the place in which the difference to step 1 can be found - we have highlighted the changes. We will dig deeper into this file now.

First we inspect the added entity type:

```
<EntityType Name="Currency" sap:content-version="1">
<Key>
        <PropertyRef Name="CURR" />
    </Key>
    <Property Name="CURR" Type="Edm.String" MaxLength="4"
sap:display-format="UpperCase" sap:text="DESCR"
sap:label="Currency Code" sap:filterable="false"/>
    <Property Name="DESCR" Type="Edm.String" MaxLength="25"
sap:label="Description"/>
</EntityType>
```

We notice that we have set `sap:filterable="false"` for the `CURR` property. We do this, since we would otherwise also have a currency code search field in the dialog that we wish to avoid (default of `sap:filterable` is true).

Now let us look at the `ValueList` annotation:

```
<Annotations Target="com.sap.wt02.Product/CurrencyCode"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
<Annotation Term="com.sap.vocabularies.Common.v1.ValueList">
    <Record>
        <PropertyValue Property="Label" String="Currency"/>
        <PropertyValue Property="CollectionPath" String="Currency"/>
        <PropertyValue Property="SearchSupported" Bool="true"/>
        <PropertyValue Property="Parameters">
            <Collection>
<Record Type="com.sap.vocabularies.Common.v1.ValueListParameterOut">
                    <PropertyValue Property="LocalDataProperty"
PropertyPath="CurrencyCode" />
                    <PropertyValue Property="ValueListProperty" String="CURR" />
                </Record>
                <Record
```

```
Type="com.sap.vocabularies.Common.v1.ValueListParameterDisplayOnly">
                    <PropertyValue Property="ValueListProperty" String="DESCR" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
</Annotations>
```

With the metadata `Target="com.sap.wt02.Product/CurrencyCode"`, we define that the `CurrencyCode` of the `EntityType Product` will have a `ValueList` (or `ValueHelp`) associated to it. We set the property `SearchSupported` to true in order to get a general search field. This is the field in the dialog that has the *Search* shadow text.

```
<Record Type="com.sap.vocabularies.Common.v1.ValueListParameterOut">
                    <PropertyValue Property="LocalDataProperty"
PropertyPath="CurrencyCode" />
                    <PropertyValue Property="ValueListProperty" String="CURR" />
                </Record>
```

This specification defines that the value help will export the value of the `CURR` field to the `CurrencyCode` field using `ValueListParameterOut`. This export happens, for example, by selecting an entry in the list of currency values.

Lastly, as for the `ValueList` annotation, we specify with the following specification that the `DESCR` field is shown in the table (but only for display purposes in the sense that no interaction with the content of this field is possible):

```
<Record Type="com.sap.vocabularies.Common.v1.ValueListParameterDisplayOnly">
                    <PropertyValue Property="ValueListProperty" String="DESCR" />
</Record>
```

Products.json

```
[{
    "ProductId": "4711",
    "Price": 856.49,
    "CurrencyCode": "EUR"
}]
```

Since the product we show initially is the same as in step 1, there is no change to the `Products.json` file.

Currency.json

```
[{
    "CURR": "EUR",
    "DESCR": "European Euro"
},
{
    "CURR": "USD",
    "DESCR": "United States Dollar"
},
```

```
{
    "CURR": "GBP",
    "DESCR": "British Pound"
},
{
    "CURR": "DKK",
    "DESCR": "Danish Krone"
},
{
    "CURR": "INR",
    "DESCR": "Indian Rupee"
},
{
    "CURR": "NOK",
    "DESCR": "Norwegian Krone"
},
{
    "CURR": "SEK",
    "DESCR": "Swedish Krona"
},
{
    "CURR": "CHF",
    "DESCR": "Swiss Franc"
}]
```

In the newly added `Currency.json` file, we include the values needed for the currency entities.

Related Information

Smart Field [page 2410]

Step 3: Smart Field with Smart Link

We now show yet another but quite different feature of the `SmartField` control, `SmartField` used in combination with `SmartLink`, which allow you to embed a dialog with related cross-application links.

We will learn that a `SmartField` control in an XML view bound to an OData model with a `SemanticObject` annotation renders a special link that shows a dialog containing different cross-application links. These cross-application links are neither configured within the XML view nor directly specified in the OData metadata. The information about these links is extracted automatically when the view is running in the SAP Fiori launchpad or, more generally, the unified shell.

Preview

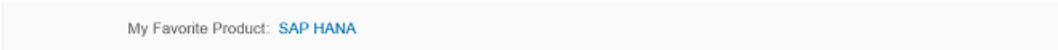


Figure 167: Smart Field with Smart Link

When you choose the link, a dialog opens:

Figure 168: Dialog with Navigation Targets

When you choose *Define Links*, a dialog opens where you can select the cross-application links you want to see.

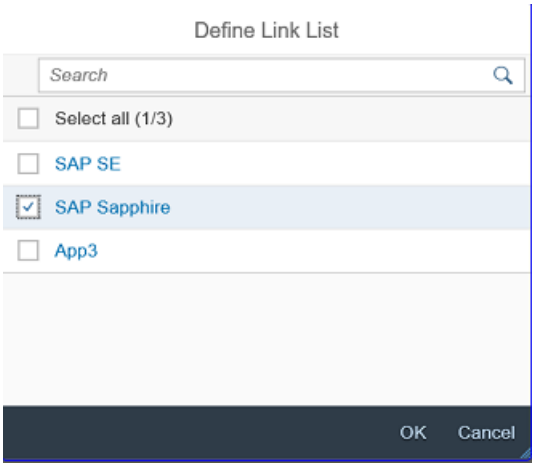


Figure 169: Link List

After your selection, the dialog looks like this:

Figure 170: Changed Links

You can choose *More* to go back to the link list.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 3 - Smart Field with Smart Link.

To have a working example, we include the `UShellCrossApplicationNavigationMock.js` class. This class basically mocks the required services, which would normally be available in the SAP Fiori launchpad. These services provide the cross-application navigation targets along with the URL parsing, thus making it possible to determine which link qualifies as a “fact sheet” target. As these services will be provided for in a real-world

scenario, we will not analyze the mock class in more detail and also not provide a code listing of the class. The class file can be found in the *Samples*.

SmartLink.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.smartControls.SmartLink"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:smartForm="sap.ui.comp.smartform"
    xmlns:sap.ui.layout="sap.ui.layout"
    xmlns:smartField="sap.ui.comp.smartfield">
    <smartForm:SmartForm editable="true">
        <smartForm:layout>
            <smartForm:ColumnLayout
                emptyCellsLarge="4"
                labelCellsLarge="4"
                columnsM="1"
                columnsL="1"
                columnsXL="1"/>
        </smartForm:layout>
        <smartForm:Group>
            <smartForm:GroupElement>
                <smartField:SmartField value="{Name}" id="idName"
editable="false"/>
            </smartForm:GroupElement>
        </smartForm:Group>
    </smartForm:SmartForm>
</mvc:View>
```

We recognize our setup of the previous two examples in the `view.xml`. We are referring to a different field, but apart from this, there is no substantial change. The `SmartField` control is rendered as a link in the display mode. We set `editable` to `false` to achieve the same effect in this example.

SmartLink.controller.js

```
sap.ui.define([
    'sap/ui/core/mvc/Controller', 'sap/ui/demo/smartControls/test/service/
UShellCrossApplicationNavigationMock'
], function(Controller, UShellCrossApplicationNavigationMock) {
    "use strict";
    return Controller.extend("sap.ui.demo.smartControls.SmartLink", {
        onInit: function() {
            this.getView().bindElement("/Products('4711')");
            UShellCrossApplicationNavigationMock.mockUShellServices({
                tutorial_03_Name: {
                    links: [
                        {
                            action: "sap_se",
                            intent: "http://www.sap.com",
                            text: "SAP SE"
                        }, {
                            action: "sap_sapphire",
                            intent: "http://www.sap.com/sapphire",
                            text: "SAP Sapphire"
                        }, {
```

```
                            action: "app3",
                            intent: "http://www.sap.com/hana",
                            text: "App3"
                        }
                    ]
                }
            });
        },
        onExit: function() {
            UShellCrossApplicationNavigationMock.unMockUShellServices();
        }
    });
});
```

We notice the instantiation of the mock class `UShellCrossApplicationNavigationMock` mentioned above and also the subsequent destroy.

metadata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema Namespace="com.sap.wt03"
            sap:schema-version="1" xmlns="http://schemas.microsoft.com/ado/
2008/09/edm">
            <EntityType Name="Product">
                <Key>
                    <PropertyRef Name="ProductId" />
                </Key>
                <Property Name="ProductId" Type="Edm.String" />
                <Property Name="Name" Type="Edm.String" sap:label="My Favorite
Product" />
            </EntityType>
            <EntityContainer m:IsDefaultEntityContainer="true"
                sap:supported-formats="json">
                <EntitySet Name="Products" EntityType="com.sap.wt03.Product" />
            </EntityContainer>
            <Annotations Target="com.sap.wt03.Product/Name"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"
                    String="tutorial_03_Name" />
            </Annotations>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

As you would expect, one piece of configuration is found in the metadata, namely the `SemanticObject` annotation. With this annotation we ensure that the `SmartField` embeds a special link control, the `SmartLink` control. Let's assume we are running in a unified shell that provides the services `CrossApplicationNavigation` and `URLParsing` (that we are mocking in our `UShellCrossApplicationNavigationMock` class). In this case, when the link is pressed, the `SmartLink` control triggers these service calls, analyzes the result, and renders the cross-application links accordingly. Since these services deliver configuration content of the unified shell, `SmartLink` is controlled by more than just OData metadata.

Products.json

```
[{
    "ProductId": "4711",
    "Name": "SAP HANA"
}]
```

We list the content of this for reasons of completeness. We note that this JSON file only contains the data shown for the link, nothing related to the dialog.

Related Information

Smart Field [page 2410]
Smart Link [page 2423]

Step 4: Smart Form

The `SmartForm` control is used to obtain a form-like layout for several controls.

`SmartForm` internally uses the `sap.ui.layout.form.Form` control. When using the `SmartForm` control in combination with the `SmartField` controls, the `view.xml` file remains very compact since required information about labels and headers is automatically extracted from the OData metadata. In addition, you can specify in `SmartForm` that it is toggle-editable in which case you get the option to switch between read-only and edit mode. In this case, the powerful features of the `SmartField` control really come to life, such as the value help and the smart links.

Preview

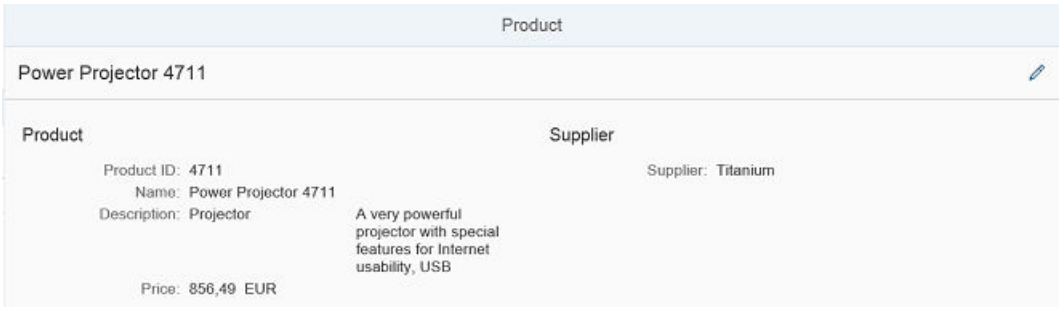


Figure 171: Smart Form with Several Smart Fields (Initial Read-Only Mode)

When you press the pencil icon, the dialog for `SmartForm` becomes editable:

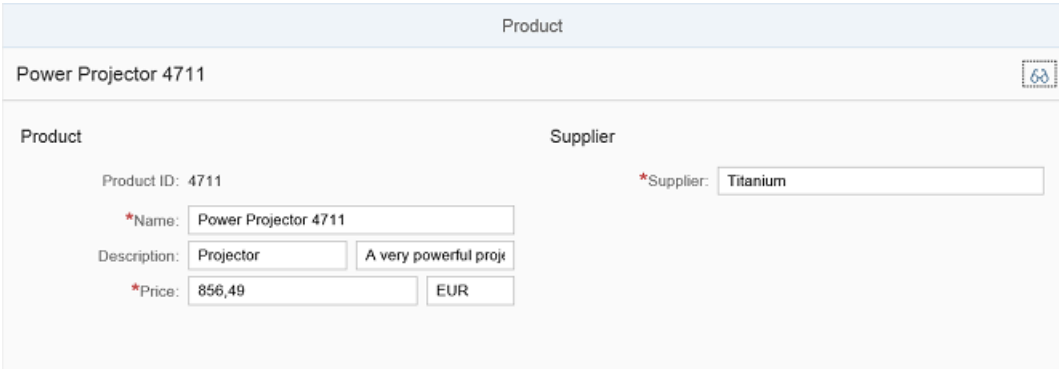


Figure 172: Smart Form with Several Smart Fields (Edit Mode)

When pressing the eyeglasses icon, you return to the display view of `SmartForm`.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 4 - Smart Form.

SmartForm.view.xml

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    controllerName="sap.ui.demo.smartControls.SmartForm"
    xmlns:smartForm="sap.ui.comp.smartform"
    xmlns:smartField="sap.ui.comp.smartfield">
    <smartForm:SmartForm
        id="smartForm"
        editTogglable="true"
        title="{Name}"
        flexEnabled="false">
        <smartForm:Group label="Product">
            <smartForm:GroupElement>
                <smartField:SmartField value="{ProductId}" />
            </smartForm:GroupElement>
            <smartForm:GroupElement>
                <smartField:SmartField value="{Name}" />
            </smartForm:GroupElement>
            <smartForm:GroupElement elementForLabel="1">
                <smartField:SmartField value="{CategoryName}" />
                <smartField:SmartField value="{Description}" />
            </smartForm:GroupElement>
            <smartForm:GroupElement>
                <smartField:SmartField value="{Price}" />
            </smartForm:GroupElement>
        </smartForm:Group>
        <smartForm:Group label="Supplier">
            <smartForm:GroupElement>
                <smartField:SmartField value="{SupplierName}" />
            </smartForm:GroupElement>
        </smartForm:Group>
```

```
    </smartForm:SmartForm>
</mvc:View>
```

We see that we have several new elements here. `Group` instructs the `SmartForm` to add a container for the child elements. In this case, we have two top-level containers of elements, one for `Product` and one for `Supplier`. With the `GroupElement` added as a wrapper control for `SmartFields`, we instruct the `SmartForm` to inspect the OData metadata and automatically add the labels found there. Within such `GroupElements`, we can even define a compound field having exactly one label in front. We do this in the example above in order to combine `CategoryName` with `Description`. With `elementForLabel="1"` we define that the label `Description` for `SmartField` (found in the OData metadata) is used for both fields. `flexEnabled="false"` is set to deactivate SAPUI5 flexibility, since these features are not part of this tutorial.

SmartForm.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.smartControls.SmartForm", {
        onInit: function() {
            this.getView().byId("smartFormPage").bindElement("/
Products('4711')");
        }
    });
});
```

The controller follows the pattern that we already know.

metadata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema Namespace="com.sap.wt04"
            sap:schema-version="1" xmlns="http://schemas.microsoft.com/ado/
2008/09/edm">
            <EntityType Name="Product">
                <Key>
                    <PropertyRef Name="ProductId" />
                </Key>
                <Property Name="ProductId" Type="Edm.String" Nullable="false"
                    sap:updatable="false" MaxLength="20" sap:label="Product
ID" />
                <Property Name="Name" Type="Edm.String" Nullable="false"
                    MaxLength="30" sap:label="Name" />
                <Property Name="CategoryName" Type="Edm.String"
sap:label="Category Description"
                    sap:updatable="true" />
                <Property Name="Description" Type="Edm.String" MaxLength="256"
                    sap:label="Description" sap:updatable="true" />
                <Property Name="Price" Type="Edm.String" Nullable="false"
```

```
                    sap:unit="CurrencyCode" MaxLength="3" sap:label="Price"
                    sap:updatable="true" />
                <Property Name="CurrencyCode" Type="Edm.String" Nullable="true"
                    MaxLength="3" sap:label="Currency" sap:semantics="currency-
code"
                    sap:updatable="true" />
                <Property Name="SupplierName" Type="Edm.String" Nullable="false"
                    sap:label="Supplier" sap:updatable="true" />
            </EntityType>
            <EntityContainer m:IsDefaultEntityContainer="true"
                sap:supported-formats="atom json">
                <EntitySet Name="Products" EntityType="com.sap.wt04.Product" />
            </EntityContainer>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

With `Nullable="false"` we define that the field is mandatory and therefore cannot be null. The label for the mandatory field is then marked with * on the UI. Other than that, there are no substantial differences in the metadata file. We only notice that the `sap:label` attributes defined here appear in the final form as explained before.

Products.json

```
[{
    "ProductId": "4711",
    "Name": "Power Projector 4711",
    "CategoryName": "Projector",
    "SupplierName": "Titanium",
    "Description": "A very powerful projector with special features for Internet
usability, USB",
    "Price": 856.49,
    "CurrencyCode": "EUR"
}]
```

We see that a few attributes have been changed and added to the JSON file, reflecting the fact that, in this step, more data is shown.

Related Information

Smart Form [page 2420]

Step 5: Smart Filter Bar and Smart Table

In this step, we will look at the `SmartTable` control along with the `SmartFilterBar` control that allow you to filter table entries.

In the context of `SmartTable`, there are several central features that can be activated:

- Table personalization

- View management with the `VariantManagement` control
- Export to Microsoft Excel

In this step, we will look at `SmartTable` without table personalization or view management. These will be treated as separate steps. In addition, we will first focus on `SmartTable` with `sap.m.Table` as the underlying table type. This table type is best suited for a responsive behavior and is even designed to be used on a smart phone or a tablet.

Preview

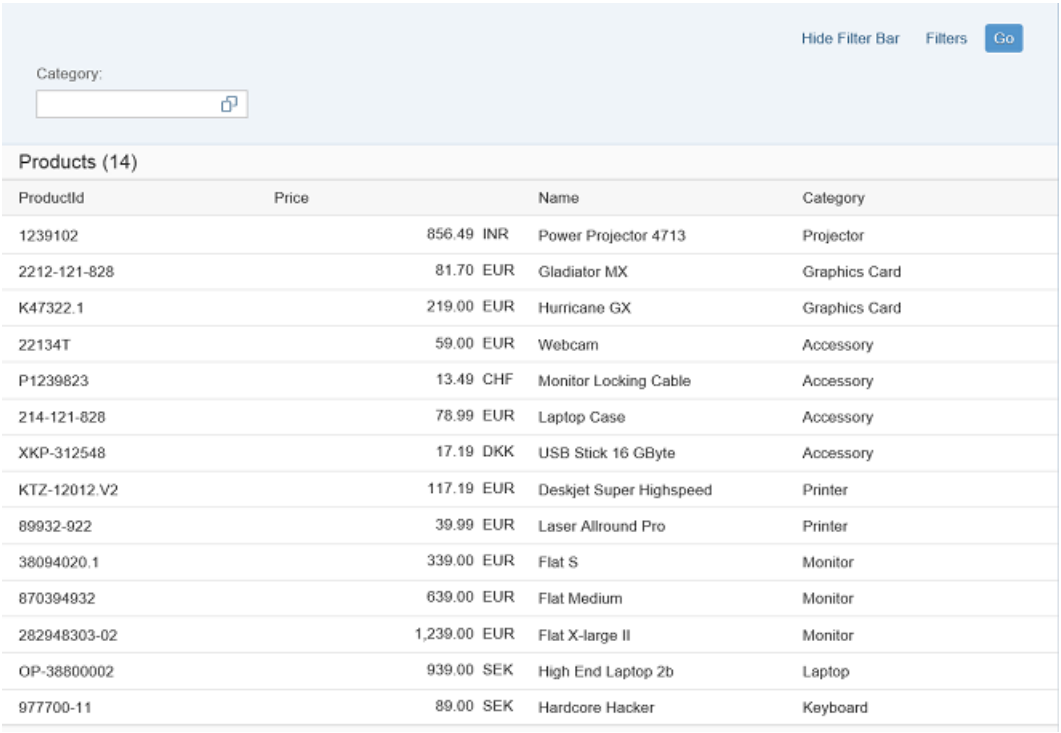


Figure 173: Initial Look of the Smart Filter Bar and the Smart Table, and the Results of Firing the Query

When choosing the *Filters* link, you see a popup:

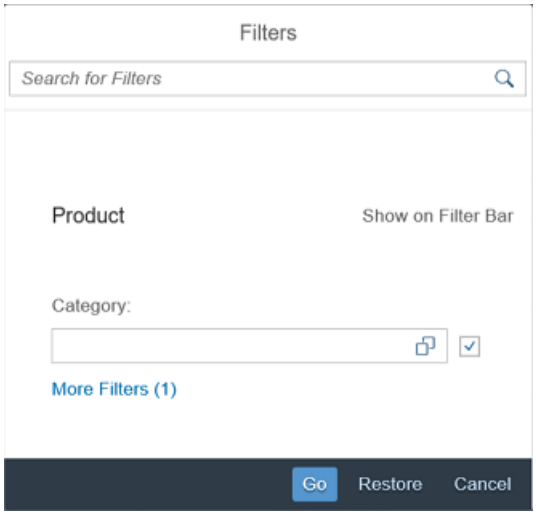


Figure 174: Dialog for Changing the Filter Values and Defining the Fields Displayed in the Filter Bar

Here you can press *More Filters*, which takes you to another dialog box.

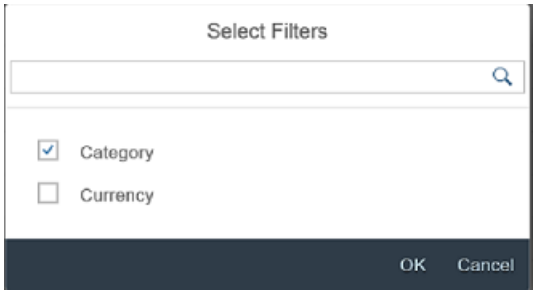


Figure 175: Dialog for Defining Additional Fields

When selecting the currency as an additional filter field for the table query and pressing *OK*, you can now select *Currency* as an additional field in the filter bar of the first dialog:

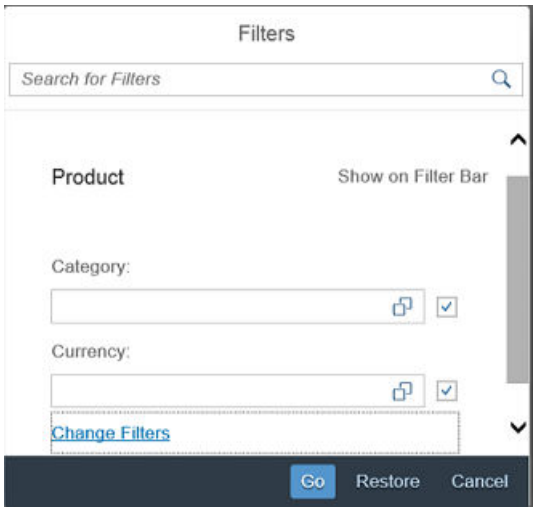


Figure 176: Currency Added to the Filter Bar

After selecting this additional field, we return to the table with the filter bar again and can use the value help for the newly added *Currency* field to restrict the results to those with *Currency = "EUR"*. Firing the query with this restriction, we see fewer results:

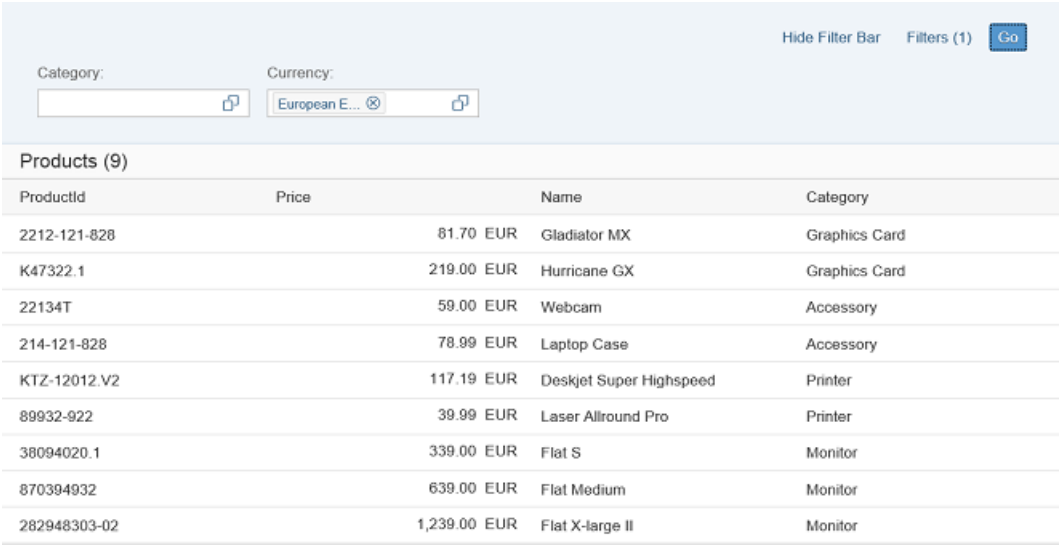


Figure 177: Query "EUR" Applied

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 5 - Smart Filter Bar and Smart Table.

SmartTable.view.xml

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    controllerName="sap.ui.demo.smartControls.SmartTable"
    xmlns:smartFilterBar="sap.ui.comp.smartfilterbar"
    xmlns:smartTable="sap.ui.comp.smarttable">
    <smartFilterBar:SmartFilterBar
        id="smartFilterBar"
        entitySet="Products">
        <smartFilterBar:controlConfiguration>
            <smartFilterBar:ControlConfiguration
                key="Category" visibleInAdvancedArea="true"
                preventInitialDataFetchInValueHelpDialog="false">
            </smartFilterBar:ControlConfiguration>
        </smartFilterBar:controlConfiguration>
    </smartFilterBar:SmartFilterBar>
    <smartTable:SmartTable
        id="smartTable_ResponsiveTable"
        smartFilterId="smartFilterBar"
        tableType="ResponsiveTable"
        editable="false"
        entitySet="Products"
        useVariantManagement="false"
        useTablePersonalisation="false"
        header="Products"
```

```
        showRowCount="true"
        useExportToExcel="false"
        enableAutoBinding="true">
    </smartTable:SmartTable>
</mvc:View>
```

We see that two new controls have been added to the `view.xml`. In the `SmartFilterBar` control we refer to an `entityType` which we will see later in `metadata.xml`. With the `ControlConfiguration` element that is added to the `controlConfiguration` aggregation of `SmartFilterBar`, we include the `Category` field in what we call the *Advanced* area of the filter bar. This is the area that can be hidden (or shown) using the toolbar *Hide Filter Bar*. We set the `preventInitialDataFetchInValueHelpDialog` property to `false` for an automatic execution of the query and thus showing of the results as soon as you open the value help. For `SmartTable` we define a few properties, some of which deserve special attention:

- `smartFilterId="smartFilterBar"`
  Ensures that the `SmartTable` can consume the `FilterBar` and the filter values defined there
- `tableType="ResponsiveTable"`
  Defines the underlying table as responsive (technically, it is an `sap.m.Table`)
- `useVariantManagement="false"`
  We will treat the case `true` in a later step.
- `useTablePersonalisation="false"`
  We will treat the case `true` in a later step.
- `header="Products"`
  Specifies the title for the table to be shown
- `showRowCount="true"`
  Specifies that the number of products appears after the title. In order for this count to work, the `SmartTable` needs to do the binding internally, which will be the case if either `smartFilterId` is specified or `enableAutoBinding` is set to `true` (see below).
- `useExportToExcel="false"`
  Offers an export to Microsoft Excel. In our case, we must set this value to `false` since the mock server does not support the proper format needed for such an export. The server must return a metadata document with `sap:supported-formats="xlsx"` to support this.
- `enableAutoBinding="true"`
  Defines whether the query is fired automatically initially, so `false` would mean the user must press *Go* to see any results in the table.

SmartTable.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";
    return Controller.extend("sap.ui.demo.smartControls.SmartTable");
});
```

We notice that because the `enableAutoBinding` property has aready been set in the `view.xml`, we don't have to do any binding in the controller.

metadata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema Namespace="com.sap.wt05"
            sap:schema-version="1" xmlns="http://schemas.microsoft.com/ado/
2008/09/edm">
            <EntityType Name="Product">
                <Key>
                    <PropertyRef Name="ProductId" />
                </Key>
                <Property Name="ProductId" Type="Edm.String"
                    sap:filterable="false" />
                <Property Name="Name" Type="Edm.String" MaxLength="30"
                    sap:label="Name" sap:filterable="false" />
                <Property Name="Category" Type="Edm.String" sap:label="Category"
                    sap:filterable="true" />
                <Property Name="Price" Type="Edm.String" sap:unit="CurrencyCode"
                    MaxLength="3" sap:label="Price" sap:filterable="false" />
                <Property Name="CurrencyCode" Type="Edm.String" MaxLength="3"
                    sap:label="Currency" sap:semantics="currency-code"
sap:filterable="true" />
            </EntityType>
            <EntityType Name="Currency">
                <Key>
                    <PropertyRef Name="CURR" />
                </Key>
                <Property Name="CURR" Type="Edm.String" MaxLength="4"
                    sap:display-format="UpperCase" sap:text="DESCR"
sap:label="Currency Code"
                    sap:filterable="false" />
                <Property Name="DESCR" Type="Edm.String" MaxLength="25"
                    sap:label="Description" />
            </EntityType>
            <EntityType Name="Category">
                <Key>
                    <PropertyRef Name="CAT" />
                </Key>
                <Property Name="CAT" Type="Edm.String" MaxLength="4"
                    sap:display-format="UpperCase" sap:text="DESCR"
sap:label="Category"
                    sap:filterable="false" />
                <Property Name="DESCR" Type="Edm.String" MaxLength="25"
                    sap:label="Description" />
            </EntityType>
            <EntityContainer m:IsDefaultEntityContainer="true"
                sap:supported-formats="atom json">
                <EntitySet Name="Products" EntityType="com.sap.wt05.Product" />
                <EntitySet Name="Currency" EntityType="com.sap.wt05.Currency" />
                <EntitySet Name="Category" EntityType="com.sap.wt05.Category" />
            </EntityContainer>
            <Annotations Target="com.sap.wt05.Product/CurrencyCode"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.Common.v1.ValueList">
                    <Record>
                        <PropertyValue Property="Label" String="Currency" />
                        <PropertyValue Property="CollectionPath"
String="Currency" />
                        <PropertyValue Property="SearchSupported" Bool="true" />
                        <PropertyValue Property="Parameters">
                            <Collection>
                                <Record
Type="com.sap.vocabularies.Common.v1.ValueListParameterOut">
```

```
                                    <PropertyValue Property="LocalDataProperty"
                                        PropertyPath="CurrencyCode" />
                                    <PropertyValue Property="ValueListProperty"
                                        String="CURR" />
                                </Record>
                                <Record
Type="com.sap.vocabularies.Common.v1.ValueListParameterDisplayOnly">
                                    <PropertyValue Property="ValueListProperty"
                                        String="DESCR" />
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
            </Annotations>
            <Annotations Target="com.sap.wt05.Product/Category"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.Common.v1.ValueList">
                    <Record>
                        <PropertyValue Property="Label" String="Category" />
                        <PropertyValue Property="CollectionPath"
String="Category" />
                        <PropertyValue Property="SearchSupported" Bool="true" />
                        <PropertyValue Property="Parameters">
                            <Collection>
                                <Record
Type="com.sap.vocabularies.Common.v1.ValueListParameterOut">
                                    <PropertyValue Property="LocalDataProperty"
                                        PropertyPath="Category" />
                                    <PropertyValue Property="ValueListProperty"
                                        String="CAT" />
                                </Record>
                                <Record
Type="com.sap.vocabularies.Common.v1.ValueListParameterDisplayOnly">
                                    <PropertyValue Property="ValueListProperty"
                                        String="DESCR" />
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
            </Annotations>
            <Annotations Target="com.sap.wt05.Product"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.UI.v1.LineItem">
                    <Collection>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Value" Path="ProductId" />
                        </Record>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Value" Path="Price" />
                        </Record>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Value" Path="Name" />
                        </Record>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Value" Path="Category" />
                        </Record>
                    </Collection>
                </Annotation>
            </Annotations>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

The `LineItem` annotation used here defines the columns that are created in the table. Only records defined in this annotation are created as table columns. Apart from this annotation, we have seen the remaining part before: We have two `ValueList` annotations that trigger a value help to be created for the associated fields, in our case the `CurrencyCode` and the `Category`. For `EntityType Name="Product"`, we have defined two fields as filterable. These are the two fields on which you can filter (and that are then also available as filter fields for table personalization as we will see in the next step). For `EntityType Name="Currency"`, we have only defined the description to be filterable to ensure that we only see the description field as search field and not the `CurrencyCode` field. The same applies to `EntityType Name="Category"`.

Category.json

```
[{
    "CAT": "PRO",
    "DESCR": "Projector"
},
{
    "CAT": "GCD",
    "DESCR": "Graphics Card"
},
{
    "CAT": "ACC",
    "DESCR": "Accessory"
},
{
    "CAT": "PRI",
    "DESCR": "Printer"
},
{
    "CAT": "MON",
    "DESCR": "Monitor"
},
{
    "CAT": "LAP",
    "DESCR": "Laptop"
},
{
    "CAT": "KBD",
    "DESCR": "Keyboard"
}]
```

This JSON file defines the possible value of the `Category`.

Currency.json

```
[{
    "CURR": "EUR",
    "DESCR": "European Euro"
},
{
    "CURR": "USD",
    "DESCR": "United States Dollar"
},
{
    "CURR": "GBP",
```

```
    "DESCR": "British Pound"
},
{
    "CURR": "DKK",
    "DESCR": "Danish Krone"
},
{
    "CURR": "INR",
    "DESCR": "Indian Rupee"
},
{
    "CURR": "NOK",
    "DESCR": "Norwegian Krone"
},
{
    "CURR": "SEK",
    "DESCR": "Swedish Krona"
},
{
    "CURR": "CHF",
    "DESCR": "Swiss Franc"
}]
```

This JSON file defines the possible value of the `CurrencyCode`, the same values we saw in step 2.

Products.json

```
[
  {
    "ProductId": "1239102",
    "Name": "Power Projector 4713",
    "Category": "Projector",
    "SupplierName": "Titanium",
    "Description": "A very powerful projector with special features for Internet
usability, USB",
    "WeightMeasure": 1467,
    "WeightUnit": "g",
    "Price": 856.49,
    "CurrencyCode": "INR",
    "Status": "Available",
    "Quantity": 3,
    "UoM": "PC",
    "Width": 51,
    "Depth": 42,
    "Height": 18,
    "DimUnit": "cm"
  },
.
.
.
]
```

We have only listed a part of the `Products.json` entries since the actual values are not so relevant. Please note that for your convenience we have included additional properties in this file to make it easier to experiment with changes to the `metadata.xml`, and possibly of the `view.xml`, to allow for a more hands-on experience.

Related Information

Smart Filter Bar [page 2413]
Smart Table [page 2444]

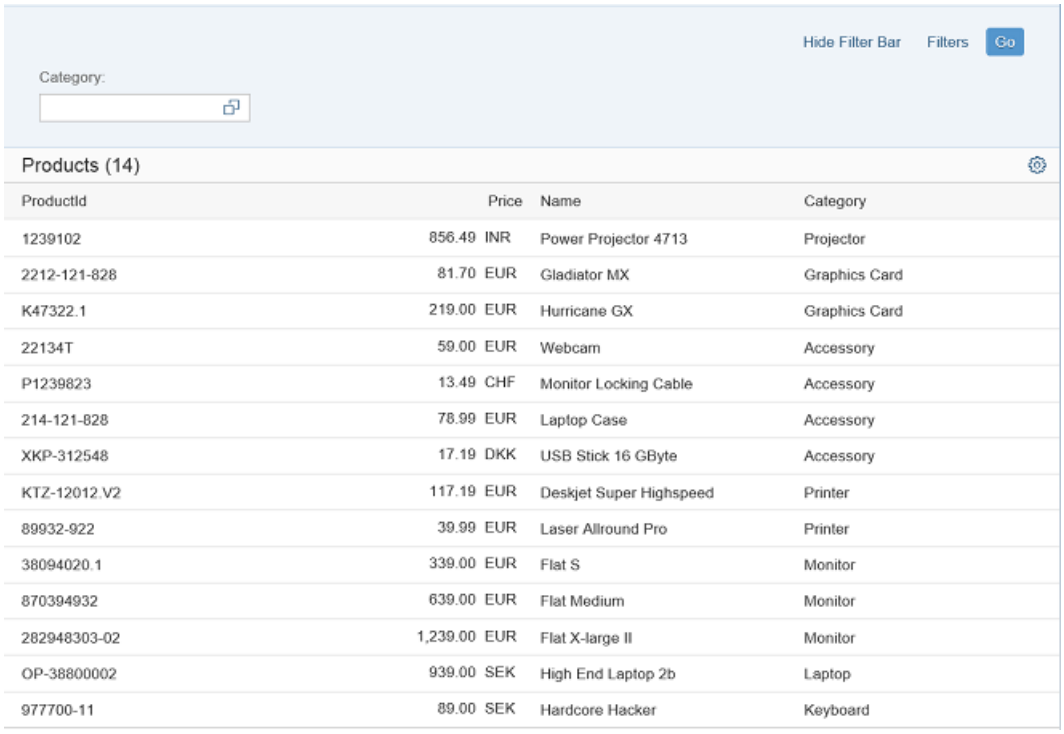
Step 6: Table Personalization

Table personalization offers you a dedicated dialog to specify which columns in the table are visible and in which order, how the data is sorted, whether grouping of the data is active, and whether table entries are filtered.

The settings in the dialog can also be persisted as you will see in the next step. In this step, we will use the same setup as in step 5 except that we enable table personalization in the `view.xml`.

Preview

Starting from the main UI, we can enter the different personalization possibilities by pressing the *Settings* icon in the upper right-hand corner of the toolbar:



ProductId	Price	Name	Category
1239102	856.49 INR	Power Projector 4713	Projector
2212-121-828	81.70 EUR	Gladiator MX	Graphics Card
K47322.1	219.00 EUR	Hurricane GX	Graphics Card
22134T	59.00 EUR	Webcam	Accessory
P1239823	13.49 CHF	Monitor Locking Cable	Accessory
214-121-828	78.99 EUR	Laptop Case	Accessory
XKP-312548	17.19 DKK	USB Stick 16 GByte	Accessory
KTZ-12012.V2	117.19 EUR	Deskjet Super Highspeed	Printer
89932-922	39.99 EUR	Laser Allround Pro	Printer
38094020.1	339.00 EUR	Flat S	Monitor
870394932	639.00 EUR	Flat Medium	Monitor
282948303-02	1,239.00 EUR	Flat X-large II	Monitor
OP-38800002	939.00 SEK	High End Laptop 2b	Laptop
977700-11	89.00 SEK	Hardcore Hacker	Keyboard

Figure 178: Table Personalization

After doing this, you see a popup with four different tabs:

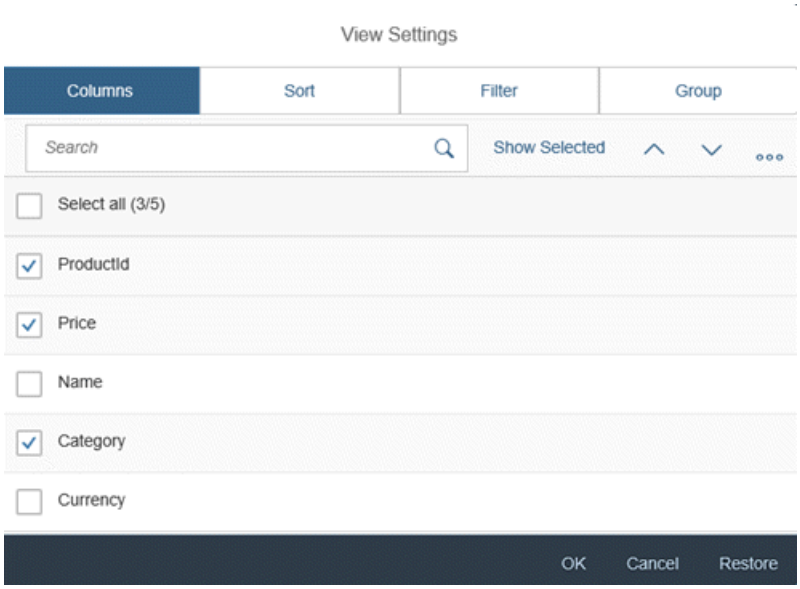


Figure 179: Settings Dialog: Column Visibility and Order

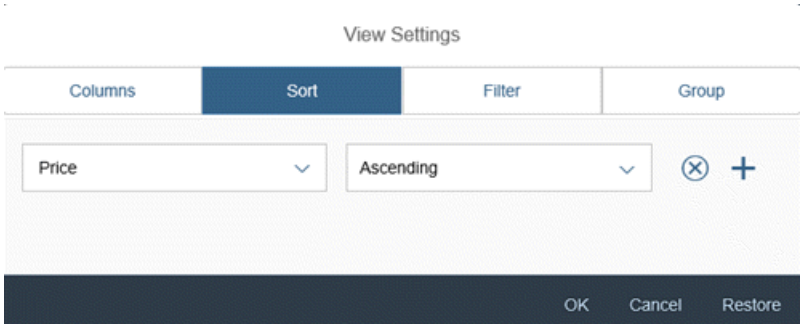


Figure 180: Settings Dialog: Sorting

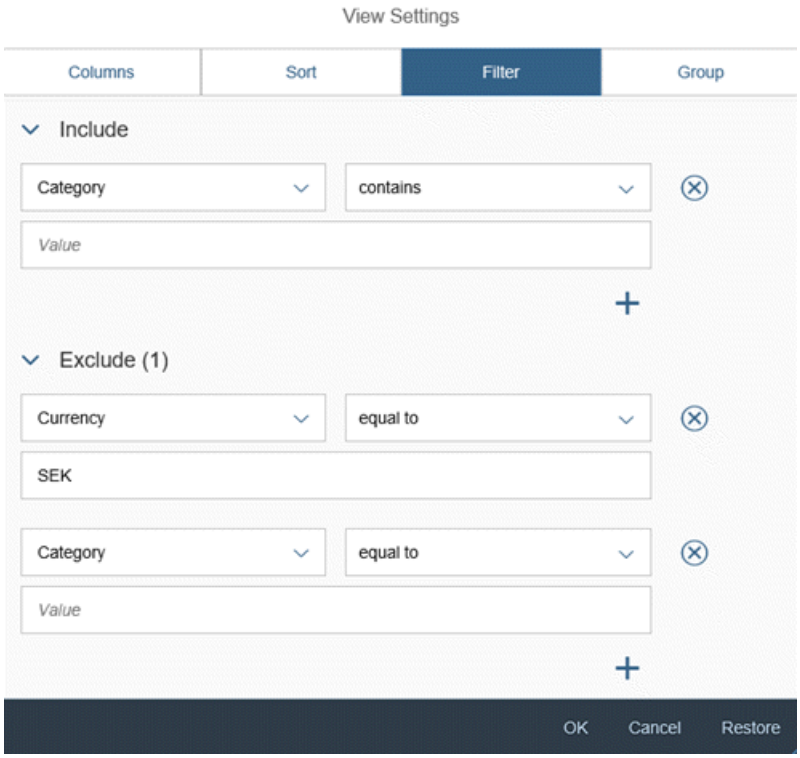


Figure 181: Settings Dialog: Filtering

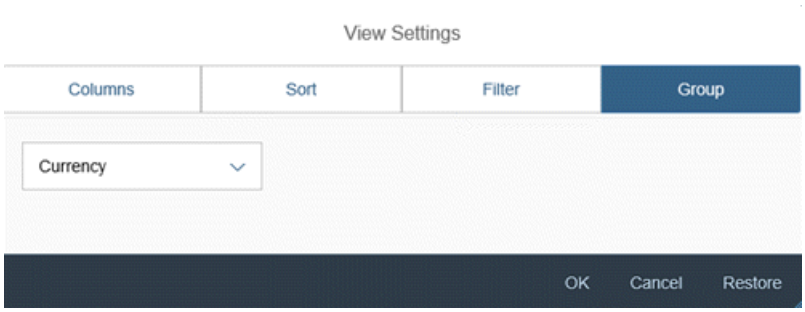


Figure 182: Settings Dialog: Grouping

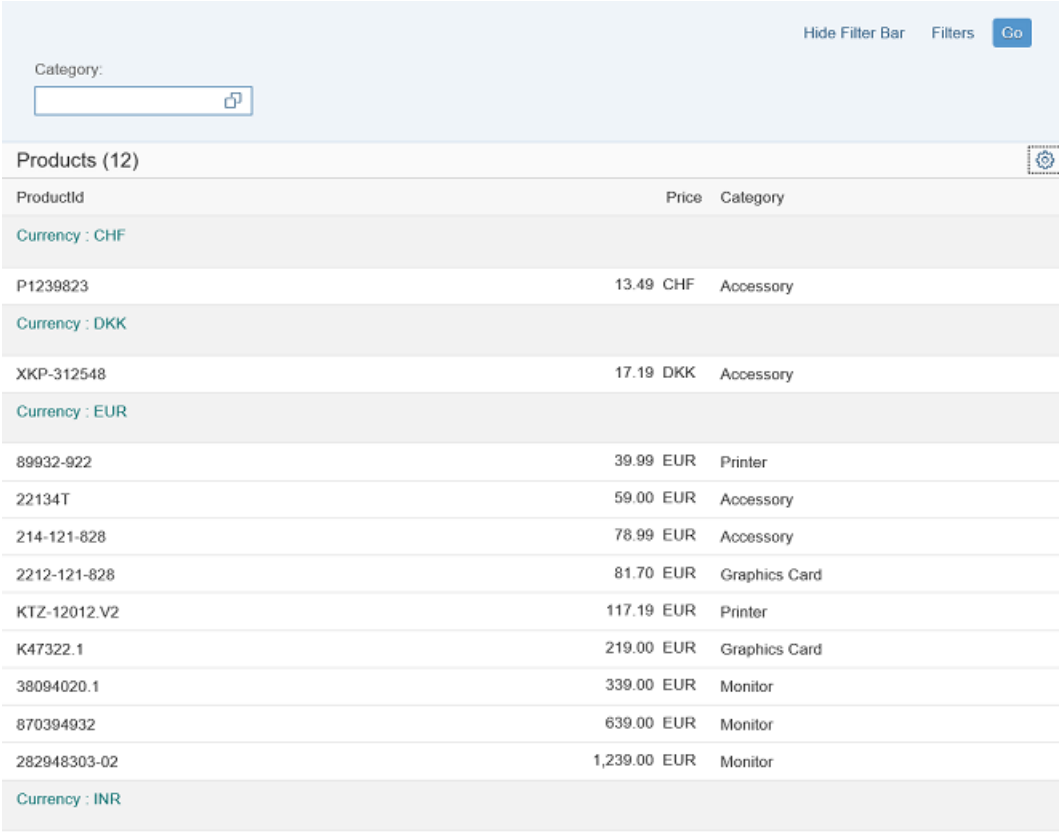


Figure 183: Resulting View with Personalization Applied

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 6 - Table Personalization.

As mentioned above, we use the same content as in the previous step, so we refrain from any code listings except for the `view.xml`. We should mention, however, that fields that are not specifically marked as `sap:sortable="false"`, `sap:filterable="false"` or `sap:groupable="false"` in the metadata will all be sortable, filterable, or groupable, respectively, and thus be available in the tabs of the personalization dialog.

Personalization.view.xml

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    controllerName="sap.ui.demo.smartControls.Personalization"
    xmlns:smartFilterBar="sap.ui.comp.smartfilterbar"
    xmlns:smartTable="sap.ui.comp.smarttable">
    <smartFilterBar:SmartFilterBar
        id="smartFilterBar"
        entitySet="Products">
        <smartFilterBar:controlConfiguration>
            <smartFilterBar:ControlConfiguration
                key="Category" visibleInAdvancedArea="true"
```

```
                preventInitialDataFetchInValueHelpDialog="false">
            </smartFilterBar:ControlConfiguration>
        </smartFilterBar:controlConfiguration>
    </smartFilterBar:SmartFilterBar>
    <smartTable:SmartTable
        id="smartTable_ResponsiveTable"
        smartFilterId="smartFilterBar"
        tableType="ResponsiveTable"
        editable="false"
        entitySet="Products"
        useVariantManagement="false"
        useTablePersonalisation="true"
        header="Products"
        showRowCount="true"
        useExportToExcel="false"
        enableAutoBinding="true">
    </smartTable:SmartTable>
</mvc:View>
```

To enable table personalization, we set `useTablePersonalisation` to `true`. As will become clear in the next section, you would typically use the table personalization together with view management since you can then also persist any changes done to the table.

Related Information

Personalization Dialog [page 2358]

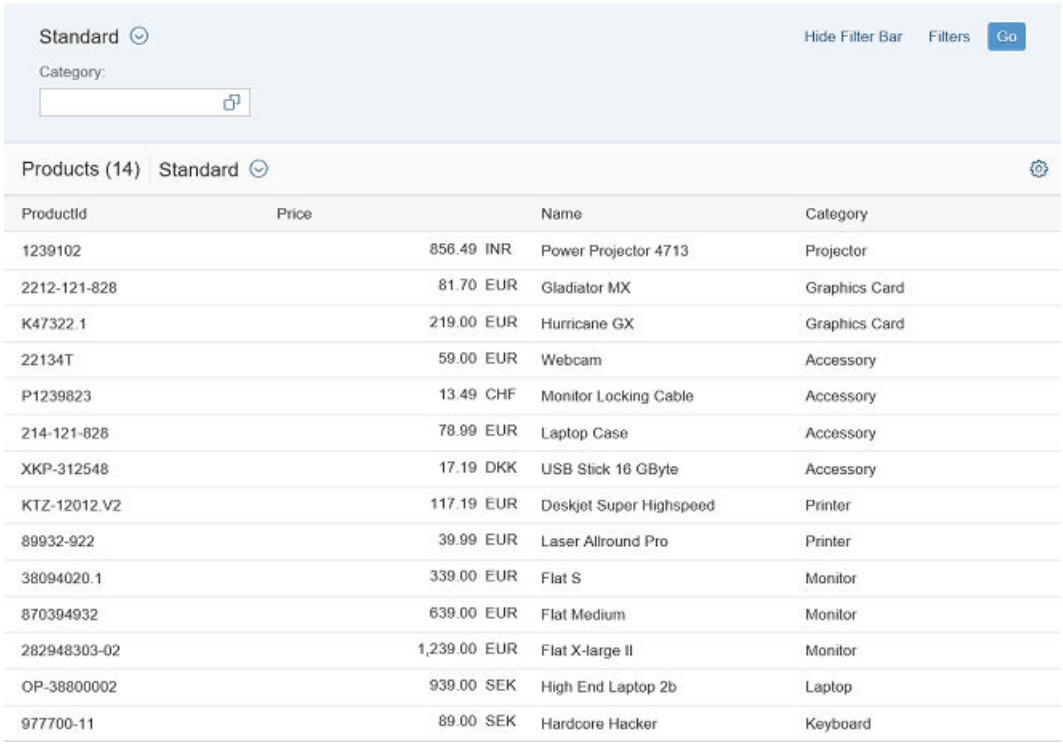
Step 7: View Management

The `VariantManagement` control allows you to handle views and makes it possible for the user to persist changes carried out on the UI and then later retrieve these changes.

For the smart controls, changes that are persisted include definitions of the filter used to query the results for the table and all changes done to the table with table personalization, visibility of columns and so on. Since we wish to provide stand-alone examples that can run on a local Web server, we do not connect to a real server on which the changes can be persisted so they can be retrieved later. Consequently, in our examples, changes are only kept in the current user session.

Preview

Initially, the UI looks as in the previous steps:



ProductId	Price	Name	Category
1239102	856.49 INR	Power Projector 4713	Projector
2212-121-828	81.70 EUR	Gladiator MX	Graphics Card
K47322.1	219.00 EUR	Hurricane GX	Graphics Card
22134T	59.00 EUR	Webcam	Accessory
P1239823	13.49 CHF	Monitor Locking Cable	Accessory
214-121-828	78.99 EUR	Laptop Case	Accessory
XKP-312548	17.19 DKK	USB Stick 16 GByte	Accessory
KTZ-12012.V2	117.19 EUR	Deskjet Super Highspeed	Printer
89932-922	39.99 EUR	Laser Allround Pro	Printer
38094020.1	339.00 EUR	Flat S	Monitor
870394932	639.00 EUR	Flat Medium	Monitor
282948303-02	1,239.00 EUR	Flat X-large II	Monitor
OP-38800002	939.00 SEK	High End Laptop 2b	Laptop
977700-11	89.00 SEK	Hardcore Hacker	Keyboard

Figure 184: Initial UI Without Personalization

We press *Filters* as shown in step 5 and then *Save* in the related dialog.

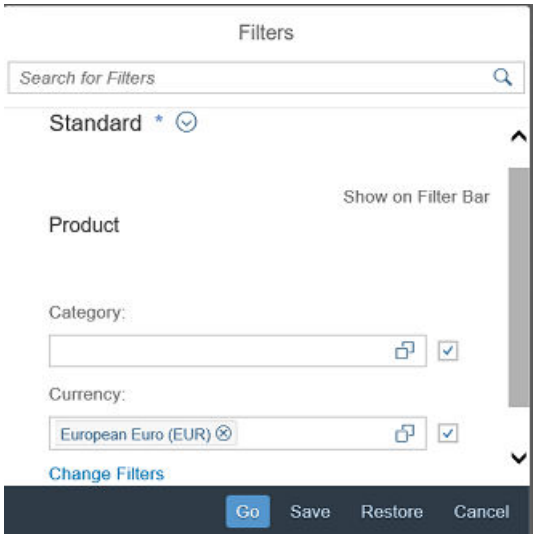


Figure 185: Specifying the Filter

After that, the following dialog is shown:

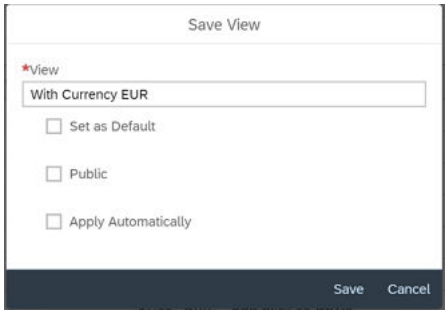
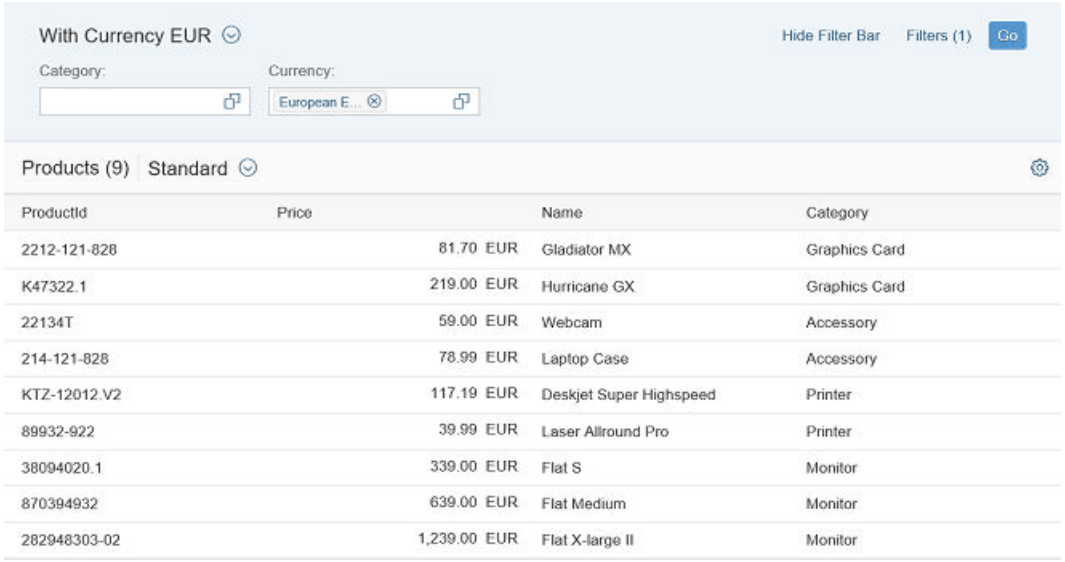


Figure 186: Defining the View Name

In this dialog, we specify the name under which this view is persisted, in our case `With Currency EUR`. With *Set as Default*, we can also specify whether this view is always used initially when navigating to this particular UI (since in our example we only persist within one browser session, this setting has no effect). With *Apply Automatically* we define that the query is fired automatically. The idea behind *Public* is that some popular but perhaps fairly complicated query settings that are used by several users can be automatically provided to all users. This *Public* option only has an effect when running on a real server and not on the mock server as in our example. If you choose *Public*, additional information regarding this function is required.

We verify these settings now and return to our main UI:



ProductId	Price	Name	Category
2212-121-828	81.70 EUR	Gladiator MX	Graphics Card
K47322.1	219.00 EUR	Hurricane GX	Graphics Card
22134T	59.00 EUR	Webcam	Accessory
214-121-828	78.99 EUR	Laptop Case	Accessory
KTZ-12012.V2	117.19 EUR	Deskjet Super Highspeed	Printer
89932-922	39.99 EUR	Laser Allround Pro	Printer
38094020.1	339.00 EUR	Flat S	Monitor
870394932	639.00 EUR	Flat Medium	Monitor
282948303-02	1,239.00 EUR	Flat X-large II	Monitor

Figure 187: UI with Active View

We notice that the *Currency* field is available in the filter bar and that we have already specified *EUR* as the currency.

Finally we now customize the table so that we do not see the `ProductId` column anymore. To do that, we press the *Settings* icon in the upper right-hand corner of the table and deselect `ProductId` in the dialog:

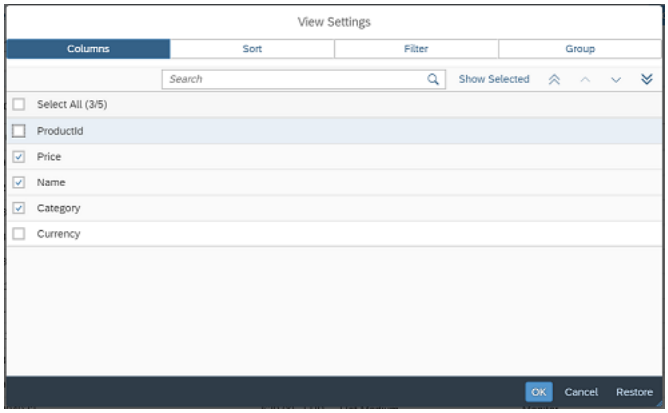
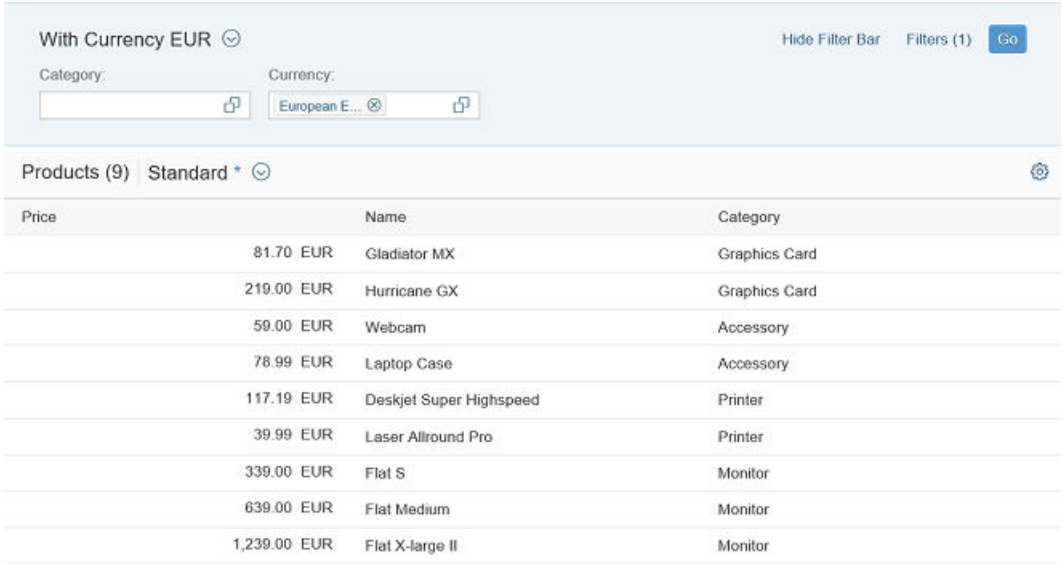


Figure 188: Remove `ProductId`

Returning to the main UI, we see that only the three columns required are now shown:



Price	Name	Category
81.70 EUR	Gladiator MX	Graphics Card
219.00 EUR	Hurricane GX	Graphics Card
59.00 EUR	Webcam	Accessory
78.99 EUR	Laptop Case	Accessory
117.19 EUR	Deskjet Super Highspeed	Printer
39.99 EUR	Laser Allround Pro	Printer
339.00 EUR	Flat S	Monitor
639.00 EUR	Flat Medium	Monitor
1,239.00 EUR	Flat X-large II	Monitor

Figure 189: Personalized Table

Figure 190:

The * right next to the *Standard* view indicates that a change has been made. We save this change by choosing the down-arrow symbol and *Save As* in the dialog, which takes us to a second dialog:

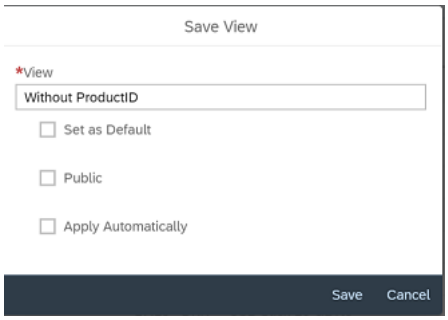
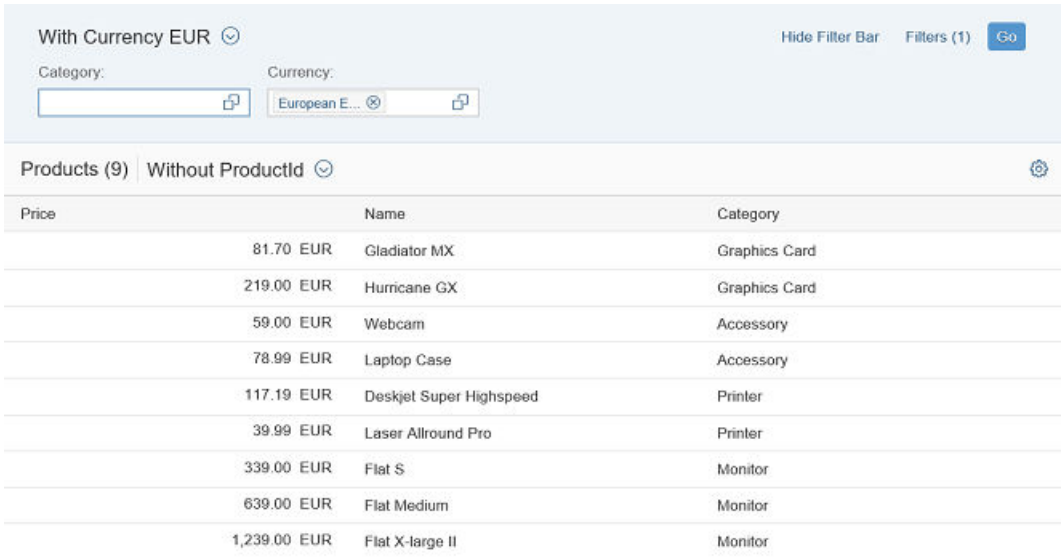


Figure 191: Specifying the View Name

Also, here we have the possibility to set this as default and to share the view. The latter is similar to the view for the `SmartFilterBar` control. After confirming the specified name, you return directly to the main UI.



Price	Name	Category
81.70 EUR	Gladiator MX	Graphics Card
219.00 EUR	Hurricane GX	Graphics Card
59.00 EUR	Webcam	Accessory
78.99 EUR	Laptop Case	Accessory
117.19 EUR	Deskjet Super Highspeed	Printer
39.99 EUR	Laser Allround Pro	Printer
339.00 EUR	Flat S	Monitor
639.00 EUR	Flat Medium	Monitor
1,239.00 EUR	Flat X-large II	Monitor

Figure 192: View for the Filter Bar and View for the Table

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 7 - Variant Management.

Since the coding is essentially identical with step 5 and step 6, we do not make any listing apart from the `view.xml` in which one property value has been changed and `persistencyKeys` have been specified.

VariantManagement.view.xml

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    controllerName="sap.ui.demo.smartControls.VariantManagement"
```

```
    xmlns:smartFilterBar="sap.ui.comp.smartfilterbar"
    xmlns:smartTable="sap.ui.comp.smarttable">
    <smartFilterBar:SmartFilterBar
        id="smartFilterBar"
        entitySet="Products"
        persistencyKey="SmartFilterPKey">
        <smartFilterBar:controlConfiguration>
            <smartFilterBar:ControlConfiguration
                key="Category"
                visibleInAdvancedArea="true"
                preventInitialDataFetchInValueHelpDialog="false">
            </smartFilterBar:ControlConfiguration>
        </smartFilterBar:controlConfiguration>
    </smartFilterBar:SmartFilterBar>
    <smartTable:SmartTable
        id="smartTable_ResponsiveTable"
        smartFilterId="smartFilterBar"
        tableType="ResponsiveTable"
        editable="false"
        entitySet="Products"
        useVariantManagement="true"
        useTablePersonalisation="true"
        header="Products"
        showRowCount="true"
        useExportToExcel="false"
        enableAutoBinding="true"
        persistencyKey="SmartTablePKey">
    </smartTable:SmartTable>
</mvc:View>
```

Component.js

```
sap.ui.define([
    "sap/ui/core/UIComponent",
    "sap/ui/fl/FakeLrepConnectorLocalStorage"
], function (UIComponent, FakeLrepConnectorLocalStorage) {
    "use strict";
    return UIComponent.extend("sap.ui.demo.smartControls.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {

FakeLrepConnectorLocalStorage.enableFakeConnector(sap.ui.require.toUrl("sap/ui/
demo/smartControls/lrep/component-test-changes.json"));
            UIComponent.prototype.init.apply(this, arguments);
        },
        destroy: function () {
            FakeLrepConnectorLocalStorage.disableFakeConnector();
            UIComponent.prototype.destroy.apply(this, arguments);
        }
    });
});
```

We add `FakeLrepConnectorLocalStorage` to our `Component.js` to enable the local storage, which is used by the `VariantManagement` control to save the client-side settings.

Related Information

Smart Variant Management [page 2457]

Step 8: Page Variant Management

In this step, we will look at the page variant of the `VariantManagement` control, an enhanced function of the `SmartVariantManagement` control that can handle multiple smart controls.

Basically, the paged version of the `VariantManagement` control is the same as the (non-paged) `VariantManagement` control. The difference is that the page variant is initialized with the `persistencyKey` property and can handle the persistency of multiple smart controls. Each smart control that uses personalization via the page variant has to provide a persistency key. The persistency itself will be stored under the `persistencyKey` of the `VariantManagement` control, and the relevant content for each control will be distributed accordingly based on each individual persistency key.

`persistencyKey` - `VariantManagement` control

- `persistencyKey` - `SmartFilterBar` control-relevant content
- `persistencyKey` `SmartTable` control-relevant content

Therefore, the persistent content of the page variant is the aggregated content of each registered smart control, and the individual persistency keys of the various smart controls are used to distinguish and distribute the relevant content.

Preview

As a page variant, the view management is now no longer part of the `SmartFilterBar` and `SmartTable` controls, but displayed in a central location instead:

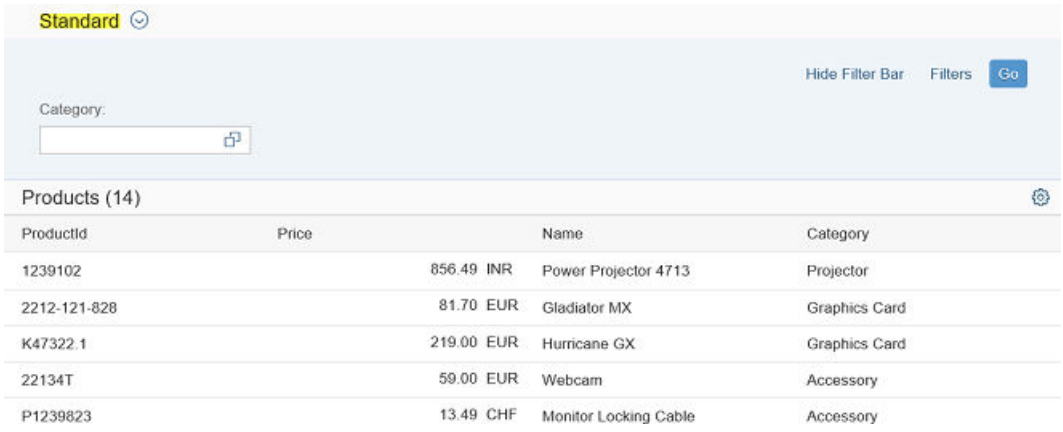


Figure 193: Central View Management

In addition, the `VariantManagement` control is no longer displayed in the *Filter* dialog of the `SmartFilterBar` control:

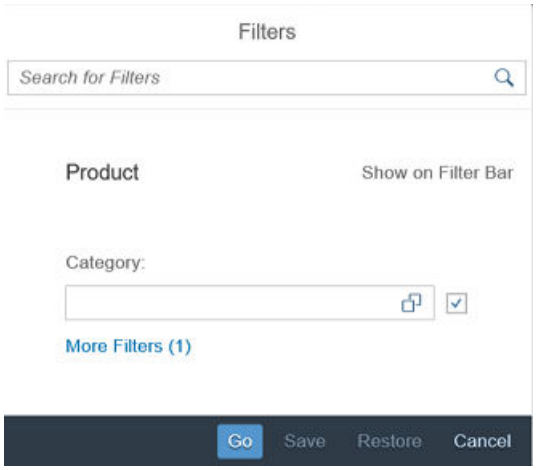


Figure 194: Filter Dialog Without View Management

Other than that, the page variant is just the same as the `VariantManagement` control.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 8 - Page Variant Management .

VariantManagement.view.xml

```
<mvc:View
          xmlns="sap.m"
       xmlns:mvc="sap.ui.core.mvc"
       controllerName="sap.ui.demo.smartControls.VariantManagement"
       xmlns:smartFilterBar="sap.ui.comp.smartfilterbar"
       xmlns:smartTable="sap.ui.comp.smarttable">
       <smartFilterBar:SmartFilterBar
             id="smartFilterBar"
             entitySet="Products"
             persistencyKey="SmartFilterPKey">
             <smartFilterBar:controlConfiguration>
                   <smartFilterBar:ControlConfiguration
                         key="Category"
                         visibleInAdvancedArea="true"
                         preventInitialDataFetchInValueHelpDialog="false">
                   </smartFilterBar:ControlConfiguration>
             </smartFilterBar:controlConfiguration>
       </smartFilterBar:SmartFilterBar>
       <smartTable:SmartTable
             id="smartTable_ResponsiveTable"
             smartFilterId="smartFilterBar"
             tableType="ResponsiveTable"
             editable="false"
             entitySet="Products"
```

```
            useVariantManagement="true"
            useTablePersonalisation="true"
            header="Products"
            showRowCount="true"
            useExportToExcel="false"
            enableAutoBinding="true"
            persistencyKey="SmartTablePKey">
        </smartTable:SmartTable>
</mvc:View>
```

The example shows the view management **without** a page variant.

VariantManagement.view.xml

```
<mvc:View
           controllerName="sap.ui.demo.smartControls.VariantManagement"
       xmlns="sap.m"
       xmlns:mvc="sap.ui.core.mvc"
       xmlns:html="http://www.w3.org/1999/xhtml"
       xmlns:smartVariantManagement="sap.ui.comp.smartvariants"
       xmlns:smartFilterBar="sap.ui.comp.smartfilterbar"
       xmlns:smartTable="sap.ui.comp.smarttable">
       <HBox
class="exPageVariantPadding">
       <smartVariantManagement:SmartVariantManagement
             id="pageVariantId"
             persistencyKey="PageVariantPKey" />
       </HBox>
       <smartFilterBar:SmartFilterBar
             id="smartFilterBar"
             entitySet="Products"
             smartVariant="pageVariantId"
             persistencyKey="SmartFilterPKey"
assignedFiltersChanged="onFiltersChanged">
             <smartFilterBar:controlConfiguration>
                   <smartFilterBar:ControlConfiguration
                         key="Category"
                         visibleInAdvancedArea="true"
                         preventInitialDataFetchInValueHelpDialog="false">
                   </smartFilterBar:ControlConfiguration>
             </smartFilterBar:controlConfiguration>
       </smartFilterBar:SmartFilterBar>
       <smartTable:SmartTable
             id="smartTable_ResponsiveTable"
             smartFilterId="smartFilterBar"
             smartVariant="pageVariantId"
             tableType="ResponsiveTable"
             editable="false"
             entitySet="Products"
             useVariantManagement="true"
             useTablePersonalisation="true"
             header="Products"
             showRowCount="true"
             useExportToExcel="false"
             enableAutoBinding="true"
             persistencyKey="SmartTablePKey">
       </smartTable:SmartTable>
</mvc:View>
```

The example shows the view management **with** a page variant.

Step 9: Smart Chart with Chart Personalization and View Management

In this step, we will look at the `SmartChart` control with the chart personalization and in combination with the `VariantManagement` control that allow you to use complex graphics along with other smart control features.

The `SmartChart` control can be used for visualizing data in a graphical manner. The `SmartChart` control creates a chart based on OData metadata and the configuration data that has been specified. The `entitySet` property must be specified to use the control. This property is used to fetch fields from OData metadata, from which the chart UI is generated.

There are several features that can be enabled in `SmartChart`, but for this tutorial, we will have a look at the following features:

- Chart personalization
- View management support
- Semantic object navigation using `SemanticObjectController`

Preview

This is what the smart chart looks like initially after firing the query:

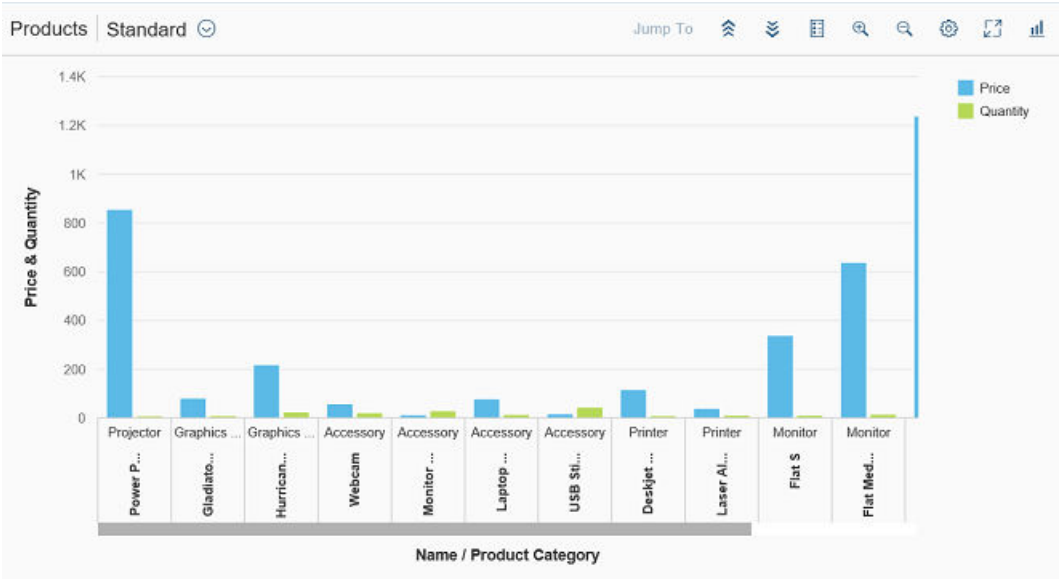


Figure 195: Initial Look of Smart Chart after Firing Query

The toolbar of the smart chart contains the header, and next to it is the view management dialog that has been explained in detail in one of the previous steps.

Next to the view management dialog is a button labeled *Jump To*. This button is disabled by default, but once a column has been selected inside the chart, this button gets enabled. When we click this button, a popup appears that contains details of the selected column. You can also navigate to related apps from here. This feature is enabled using `SemanticObjectController`. When we hover the mouse pointer over a column, a popup is displayed to provide additional information.

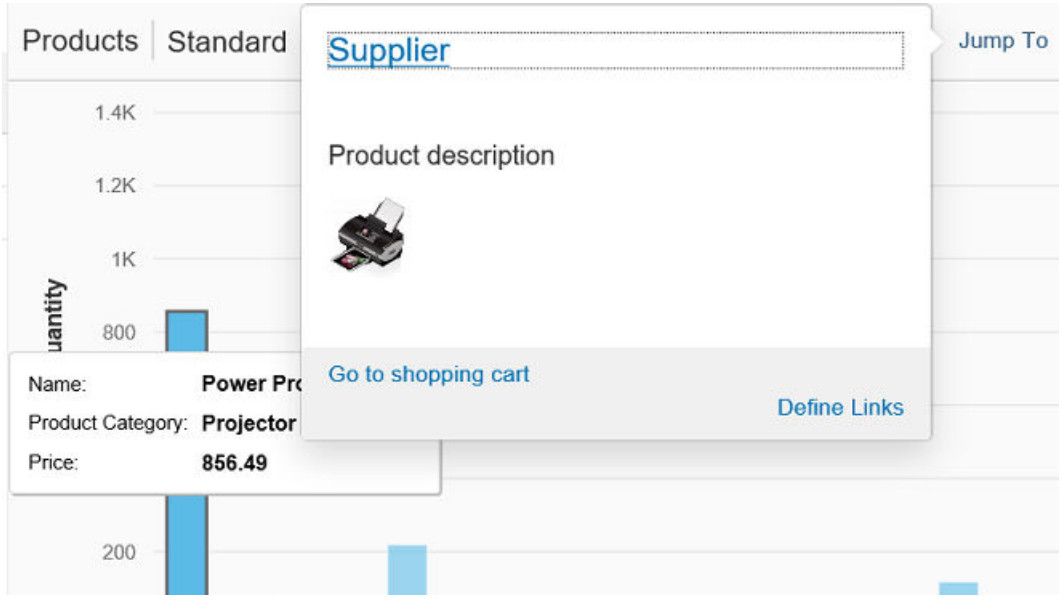


Figure 196: Semantic Navigation Feature

The button at the right-hand side of the toolbar can be used for selecting the chart type:

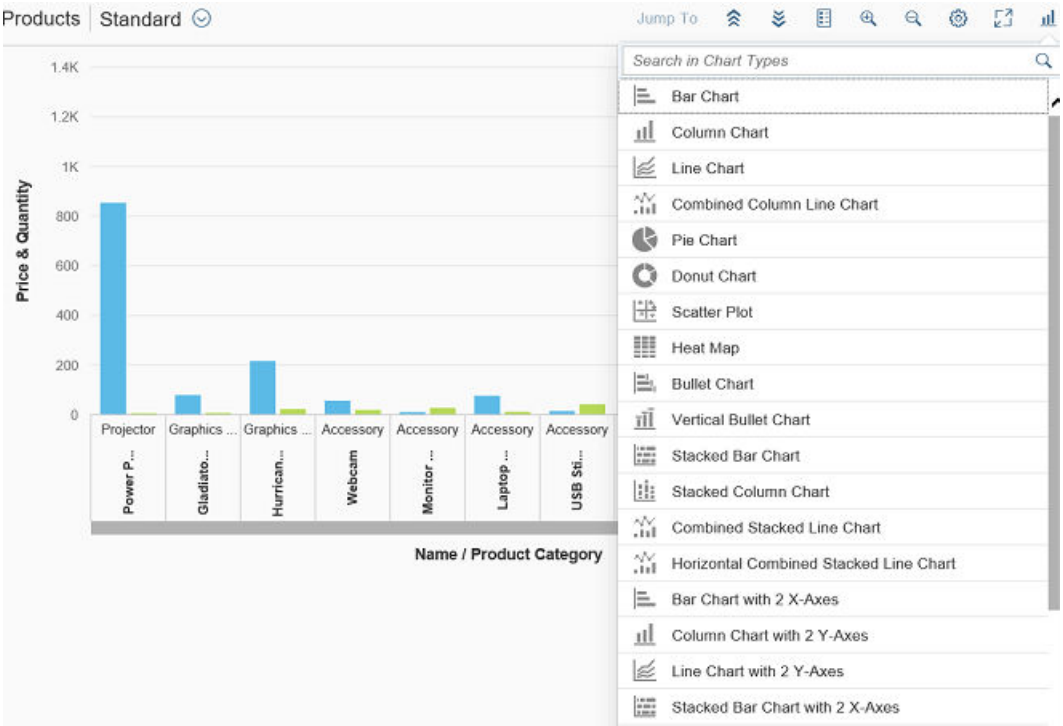


Figure 197: Selection of Chart Type

The two buttons next to the *Jump To* button can be used to drill up and drill down into the chart. Using this feature you can display even more detailed information in the chart:

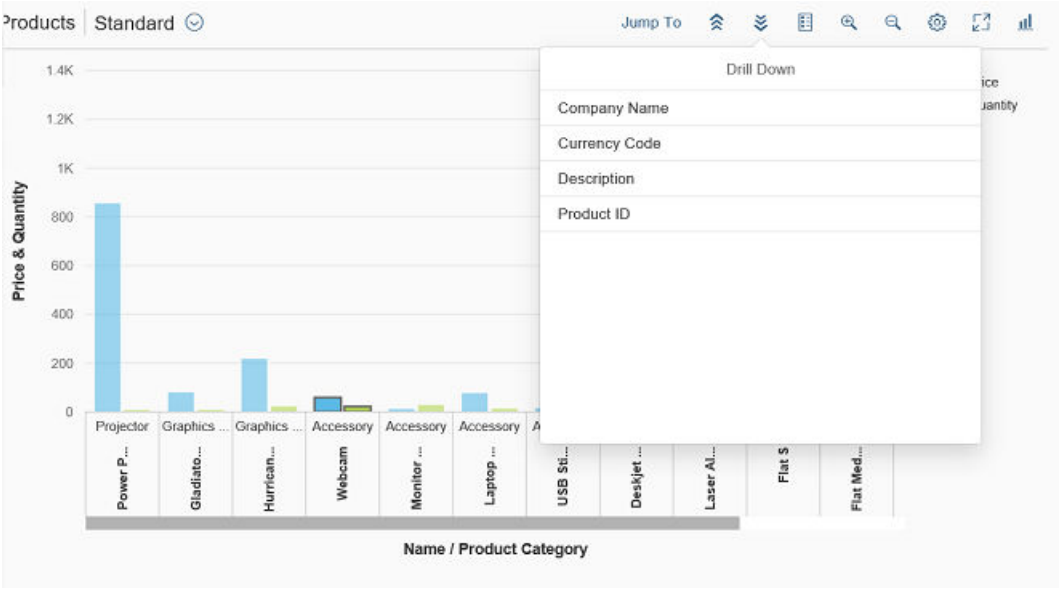


Figure 198: Drillup and Drilldown Features

If we set the `showDetailsButton` and `showDrillBreadcrumbs` properties to `true`, we can also use an alternative drilldown function: A button labeled *Drilldown* is shown. If a column is selected inside the chart, this button changes into a *Details* button. When you click this button, again, a popup appears that contains details of the selected column.

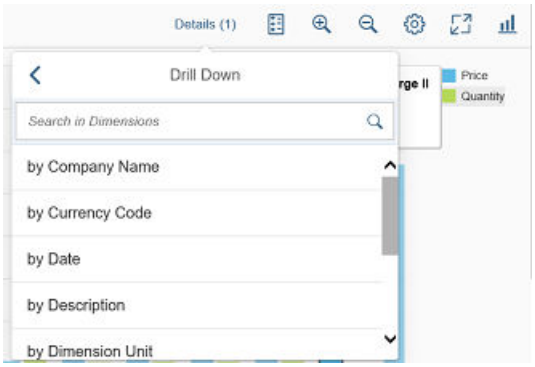


Figure 199: Drilldown by Dimensions

When you drill further down, you can see the breadcrumbs trail for the drilldown path on the left-hand side, which you can also use to drill up within the chart.

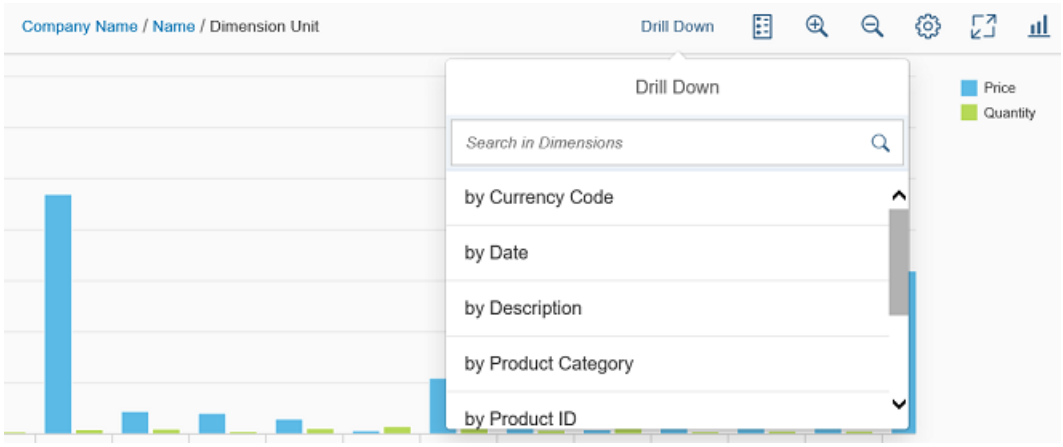


Figure 200: Further Drilldown

The next button can be used to toggle legend visibility:

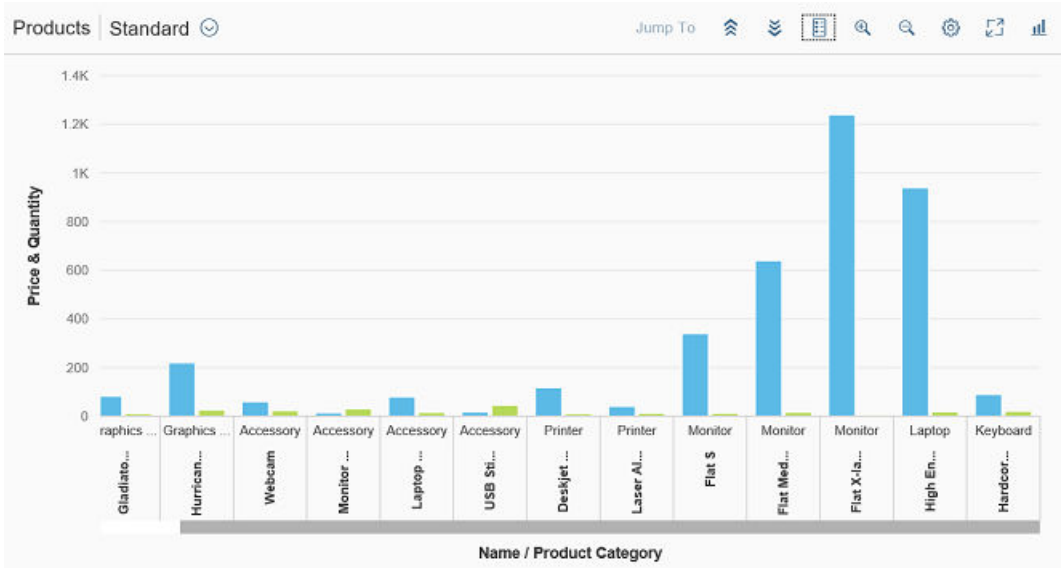


Figure 201: Toggle Legend Visibility

The two buttons next to this in the toolbar are used for zooming in or out. This will help you to get a clearer picture of a particular entity. The next button in the toolbar is the *Settings* icon that opens the personalization dialog for the chart:

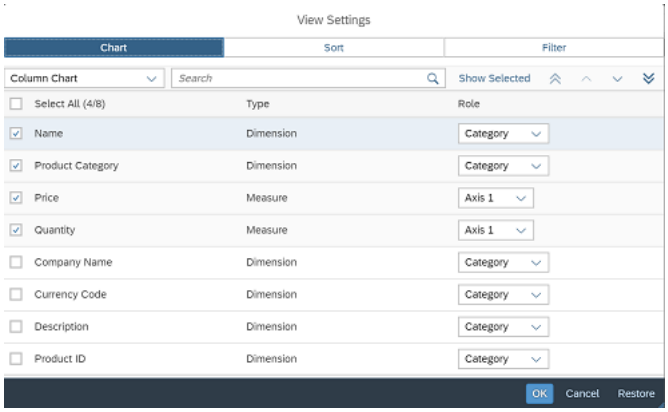


Figure 202: Dialog for Adding Measures and Dimensions

The button next to the settings can be used to display a chart in full screen mode. The chart will then take up 100 % width and height of the browser.

Coding

You can view and download all files in the *Samples* in the Demo Kit at Smart Controls - Step 9 - Smart Chart and at Samples.

SmartChart.view.xml

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc"
    xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:app="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"
    controllerName="sap.ui.demo.smartControls.SmartChart"
xmlns:sl="sap.ui.comp.navpopover"
    xmlns:smartChart="sap.ui.comp.smartchart">
    <smartChart:SmartChart enableAutoBinding="true"
        entitySet="Products" useVariantManagement="true"
        persistencyKey="SmartChart_Explored" useChartPersonalisation="true"
        header="Products">
        <smartChart:semanticObjectController>
            <sl:SemanticObjectController
                navigationTargetsObtained="onNavigationTargetsObtained"
navigate="onNavigate" />
        </smartChart:semanticObjectController>
    </smartChart:SmartChart>
</mvc:View>
```

We see that a new control has been added to the `view.xml`. In the `SmartChart` control, we refer to the entity type that we will see later in the `metadata.xml`. With `SemanticObjectController` that is added to the `semanticObjectController` aggregation of `SmartChart`, we can enable the display of linked data for a particular entity. We also set `enableAutoBinding="true"`, which enables automatic execution of the query

and thus shows the result as soon as the `SmartChart` control is loaded. We set `useVariantManagement="true"` and `persistencyKey="SmartChart_Explored"` to enable the view management. We also set `useChartPersonalisation="true"`, which enables the chart personalization.

SmartChart.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageBox"
], function(Controller, MessageBox) {
    "use strict";
    return Controller.extend("sap.ui.demo.smartControls.SmartChart", {
        onNavigationTargetsObtained: function(oEvent) {
            var oParameters = oEvent.getParameters();
            var oSemanticAttributes = oParameters.semanticAttributes;
            oParameters.show("Supplier", new sap.ui.comp.navpopover.LinkData({
                text: "Homepage",
                href: "http://www.sap.com",
                target: "_blank"
            }), [
                new sap.ui.comp.navpopover.LinkData({
                    text: "Go to shopping cart"
                })
            ], new sap.ui.layout.form.SimpleForm({
                maxContainerCols: 1,
                content: [
                    new sap.ui.core.Title({
                        text: "Product description"
                    }), new sap.m.Image({
                        src: "img/HT-1052.jpg", //
oSemanticAttributes.ProductPicUrl,
                        densityAware: false,
                        width: "50px",
                        height: "50px",
                        layoutData: new sap.m.FlexItemData({
                            growFactor: 1
                        })
                    }), new sap.m.Text({
                        text: oSemanticAttributes.Description
                    })
                ]
            }));
        },
        onNavigate: function(oEvent) {
            var oParameters = oEvent.getParameters();
            if (oParameters.text === "Homepage") {
                return;
            }
            MessageBox.show(oParameters.text + " has been pressed", {
                icon: sap.m.MessageBox.Icon.INFORMATION,
                title: "SmartChart demo",
                actions: [
                    sap.m.MessageBox.Action.OK
                ]
            });
        }
    });
});
```

The following two functions are defined:

- `onNavigationTargetObtained()`
- `onNavigate()`

`onNavigationTargetObtained()` is called when a column is selected and you click the *Jump To* button. With the click event the semantic parameters are obtained, and the function renders a navigation popover containing a simple form that presents detailed information of the entity selected as can be seen in figure 2.

`onNavigate()` is called when the links in the navigation popover are clicked. This is done to demonstrate how you add more functionality to the data in the navigation popover. In our example, we are just showing a message box and the navigation to one other link.

metadata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:Reference
        xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
        <edmx:Include Namespace="com.sap.vocabularies.Common.v1"
            Alias="Common" />
    </edmx:Reference>
    <edmx:Reference
        xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
        <edmx:Include Namespace="com.sap.vocabularies.UI.v1"
            Alias="UI" />
    </edmx:Reference>
    <edmx:DataServices m:DataServiceVersion="2.0">
        <Schema Namespace="com.sap.wt08"
            sap:schema-version="0" xmlns="http://schemas.microsoft.com/ado/
2008/09/edm">
            <EntityType Name="Product" sap:service-schema-version="1"
                sap:service-version="1" sap:semantics="aggregate"
                sap:content-version="1">
                <Key>
                    <PropertyRef Name="ProductId" />
                </Key>
                <Property Name="ProductId" Type="Edm.String" Nullable="false"
                    MaxLength="10" sap:aggregation-role="dimension"
sap:label="Product ID"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="Category" Type="Edm.String" Nullable="false"
                    MaxLength="40" sap:aggregation-role="dimension"
sap:label="Product Category"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="Name" Type="Edm.String" Nullable="false"
                    MaxLength="255" sap:aggregation-role="dimension"
sap:label="Name"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="Description" Type="Edm.String" Nullable="false"
                    MaxLength="255" sap:aggregation-role="dimension"
sap:label="Description"
```

```
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="SupplierName" Type="Edm.String" Nullable="false"
                    MaxLength="80" sap:aggregation-role="dimension"
sap:label="Company Name"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="Quantity" Type="Edm.Decimal" Nullable="false"
                    MaxLength="3" sap:aggregation-role="measure"
sap:label="Quantity"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="Price" Type="Edm.Decimal" Nullable="false"
                    Precision="23" Scale="4" sap:aggregation-role="measure"
sap:label="Price"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
                <Property Name="CurrencyCode" Type="Edm.String" Nullable="false"
                    MaxLength="5" sap:aggregation-role="dimension"
sap:label="Currency Code"
                    sap:creatable="false" sap:updatable="false"
sap:sortable="true"
                    sap:filterable="true" />
            </EntityType>
            <EntityContainer Name="com.sap.wt08"
                m:IsDefaultEntityContainer="true">
                <EntitySet Name="Products" EntityType="com.sap.wt08.Product"
                    sap:creatable="false" sap:updatable="false"
sap:deletable="false"
                    sap:pageable="false" sap:content-version="1" />
            </EntityContainer>
            <Annotations Target="com.sap.wt08.Product"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.UI.v1.LineItem">
                    <Collection>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Value" Path="Name" />
                            <Annotation
Term="com.sap.vocabularies.UI.v1.Importance"

EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/High" />
                        </Record>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Value" Path="Category" />
                            <Annotation
Term="com.sap.vocabularies.UI.v1.Importance"

EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/High" />
                        </Record>
                    </Collection>
                </Annotation>
            </Annotations>
            <Annotations Target="com.sap.wt08.Product"
                xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="com.sap.vocabularies.UI.v1.Chart">
                    <Record>
                        <PropertyValue Property="Title" String="Line Items" />
                        <PropertyValue Property="ChartType"
                            EnumMember="com.sap.vocabularies.UI.v1.ChartType/
Column" />
                        <PropertyValue Property="Dimensions">
                            <Collection>
                                <PropertyPath>Name</PropertyPath>
                                <PropertyPath>Category</PropertyPath>
```

```
                        </Collection>
                    </PropertyValue>
                    <PropertyValue Property="Measures">
                        <Collection>
                            <PropertyPath>Price</PropertyPath>
                            <PropertyPath>Quantity</PropertyPath>
                        </Collection>
                    </PropertyValue>
                </Record>
            </Annotation>
        </Annotations>
        <Annotations Target="com.sap.wt08.Product/Category"
            xmlns="http://docs.oasis-open.org/odata/ns/edm">
            <Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"
                String="SemanticObjectCategory" />
        </Annotations>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

The most important point to keep in mind is that analytical annotations along with chart annotations need to exist in the `metadata.xml`. The most prominent annotations are listed below:

- `sap:semantics="aggregate"`
  An analytical operation can be performed on the data. This has to be added to `EntityType` (in our case "Product")
- `sap:aggregation-role="dimension"`
  Defines that a property of `EntityType` is treated as a dimension.
- `sap:aggregation-role="measure"`
  Defines that a property of `EntityType` is treated as a measure.
- `Annotation Term="com.sap.vocabularies.UI.v1.Chart`
  Defines the UI annotation for rendering a chart.
- `EnumMember="com.sap.vocabularies.UI.v1.ChartType/Column"`
  Defines the default chart type.
- `Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"`
  Defines the `SemanticObject` annotation for `EntityType` (in our case the product category).

Products.json

```
[
  {
    "ProductId": "1239102",
    "Name": "Power Projector 4713",
    "Category": "Projector",
    "SupplierName": "Titanium",
    "Description": "A very powerful projector with special features for Internet
usability, USB",
    "WeightMeasure": 1467,
    "WeightUnit": "g",
    "Price": 856.49,
    "CurrencyCode": "EUR",
    "Status": "Available",
    "Quantity": 3,
    "UoM": "PC",
    "Width": 51,
    "Depth": 42,
    "Height": 18,
```

```
    "DimUnit": "cm"
  },
.
.
.
]
```

We have only listed a part of the `Products.json` entries since the actual values are not so relevant. Please note that for your convenience we have included additional properties in this file to make it easier to experiment with changes of the `metadata.xml`, and possibly of the `view.xml`, to allow for a more hands-on experience.

Summary

Summary of the *Smart Controls* tutorial

In this tutorial we have focused on conveying the big picture of the smart controls leaving quite a few features of the smart controls aside. For example, the `SmartTable` control can also be used in combination with the so-called analytical table, a table type that has additional features, such as displaying aggregation rows to display the sums. This information is controlled by the configuration in the `metadata.xml` document. In addition, the `SmartField` control as a basic building block can consume more annotations than we have been able to show here.

We would also like to mention that the OData services you have seen in this tutorial have version 2.0. With version 4.0, the behavior might change.

In the previous steps you have seen how smart controls can use different sources of configuration to make development easier. You may wonder, though, where exactly which configuration should be located. To answer this question, we need to think about from where the various configuration parts are coming. You should think of `metadata.xml` (that is the definition of entities, such as `EntityType` and annotations) as information associated with the service or the application logic, even though we have seen that annotations are more UI-specific than the more generic information found in the `EntityTypes` and the `EntityContainer`. Information in the `metadata.xml` is therefore predetermined for a certain amount of reuse: You can build several UIs (or parts of it) reusing this information. We have seen such a reuse several times in this document: The reuse of the OData service metadata document for `SmartFilterBar`, `SmartTable`, and the popups associated with the value help. In contrast to this, the configuration information that we specify in the `view.xml` is more specific to one concrete UI.

3D Viewer

In this tutorial, you will learn how to work with the controls in the Visual Interaction toolkit (`sap.ui.vk` library) to create applications with 3D viewing functionality.

> ⚠ Caution
>
> The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

The Visual Interaction toolkit provides controls for the visualization of 3D models in your application.

→ Tip

You do not have to do the tutorial steps sequentially. You can start the tutorial at any step you want, but you just need to ensure that you have downloaded all the relevant code required to work through the step.

You can view and download the files for all steps at 3D Viewer.

For more information check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]

Prerequisites

Prerequisite steps for the 3D Viewer tutorial.

⚠ Caution

The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

Before proceeding with the 3D Viewer tutorial, ensure that you are familiar with the concepts introduced in the following tutorials:

- Walkthrough [page 69]
- Data Binding [page 219]

File Structure

Please note that for each step in this tutorial, you will be creating a separate folder with its own copy of the following files:

- **<FolderName>** (folder)
  - controller (folder)
    - `App.controller.js`
  - i18n (folder)
    - `i18n.properties`
  - view (folder)
    - `App.view.xml`
  - `Component.js`
  - `index.html`
  - `manifest.json`

Step 1: 3D Viewer With Single File Loading

In this step, you will be creating a 3D Viewer application that allows a user to load a single 2D image or 3D model that is stored locally or remotely.

> ⚠ Caution
>
> The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

Preview

By the end of this step, you will have created a Viewer application that allows you to load a 2D or 3D resource. The Viewer application will look as follows:

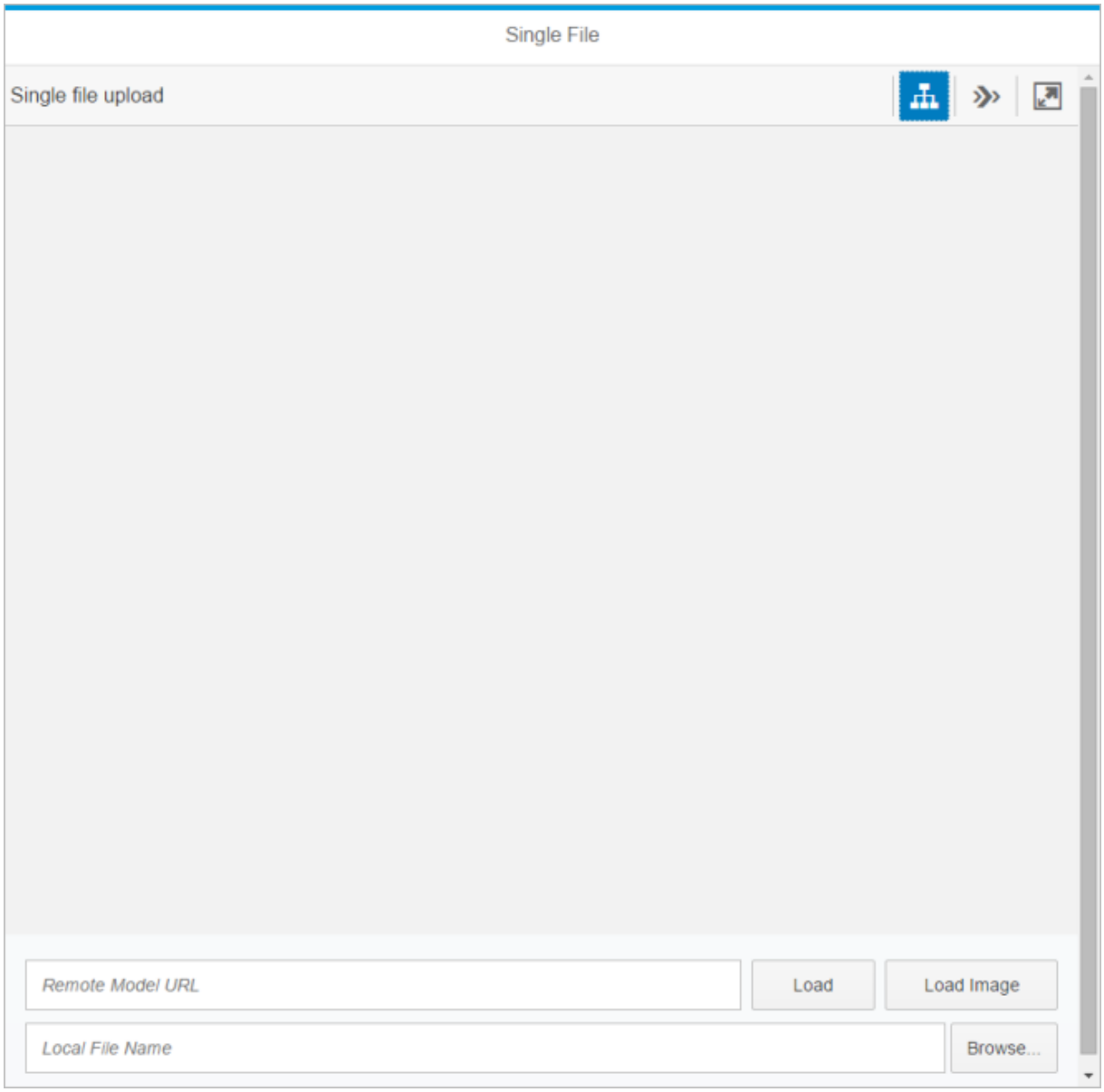


Figure 203: Viewer application with single file loading functionality

Coding

You can view and download all files at 3D Viewer - Step 1 - Single File.

index.html

In the first `script` element of this `index.html` file, we are referencing:

- the `sap.ui.vk` library, which contains the controls for adding 3D visualization functionality (the `sap.ui.vk` value in the `data-sap-ui-libs` property)
- the component file called `Component.js` (the `data-sap-ui-resourceroots` property)

We specify that resources related to `singleFile` are located in the same folder as this `index.html` file.

In the second `script` element, we create a function that will be called as soon as SAPUI5 is loaded and initialized. We add our application into this function.

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>
    <script id="sap-ui-bootstrap"
        src="../../../../../../../../../resources/sap-ui-core.js"
        data-sap-ui-libs="sap.ui.vk, sap.m"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-bindingSyntax="complex"
        data-sap-ui-resourceroots='{
            "singleFile": "./"
         }' >
    </script>
    <script>
    sap.ui.getCore().attachInit(function(){
        new sap.m.Shell({
            app: new sap.ui.core.ComponentContainer({
                name : "singleFile"
            })
        }).placeAt("content");
    });
    </script>
    </head>
    <body id="content" class="sapUiBody">
    </body>
</html>
```

Component.js

In the `Component.js` file, the only item for you to consider is to reference is the `manifest.json` file.

```
sap.ui.define([
    "sap/ui/core/UIComponent"
], function (UIComponent) {
    "use strict";
    return UIComponent.extend("singleFile.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
        }
    });
});
```

i18n.properties

The `i18n.properties` file contains all the user interface labels for the application, which includes the labels for he toolbar, the page title, the buttons, the text input fields, and the error message that will be displayed when no URL is specified.

```
# App Descriptor
appTitle=App title
appDescription=This is a description coming from the i18n as specified in
manifest.json
# Viewer Descriptor
viewerToolbarTitle=Single file load
# Page Descriptor
pageTitle=Single File
```

```
# Form Descriptor
formRemoteURL=Remote Model URL
buttonLoadModel=Load
buttonLoadImage=Load Image
formLocalFileName=Local File Name
# Message Toast
missingUrl=Please specify a URL
```

manifest.json

The `manifest.json` file contains information about the files that we need to use in our application.

In this file, we reference the `i18n.properties` file to specify what language the user interface of our application will have. We also specify what the root view of the application is; i.e. the page that is loaded first when the application is first started. This is set in the `rootView` property.

```
{
  "_version": "1.8.0",
  "sap.app": {
    "id": "singleFile",
    "type": "application",
    "i18n": "i18n/i18n.properties",
    "title": "{{appTitle}}",
    "description": "{{appDescription}}",
    "applicationVersion": {
      "version": "1.0.0"
    },
  },
  "sap.ui": {
    "technology": "UI5",
    "deviceTypes": {
      "desktop": true,
      "tablet": true,
      "phone": true
    }
  },
  "sap.ui5": {
    "rootView": "singleFile.view.App",
    "dependencies": {
      "minUI5Version": "1.30",
      "libs": {
        "sap.m": {}
      }
    },
    "models": {
      "i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "settings": {
          "bundleName": "singleFile.i18n.i18n"
        }
      }
    }
  }
}
```

App.view.xml

The `App.view.xml` file specifies how the page in the application will be laid out. We have one form container containing two form elements (`formElement`). The first `formElement` element contains the fields for loading 2D or 3D resources located remotely. We have specifed one input text field, and two buttons: one button for loading 2D images, and the other button for loading 3D models.

In the second `formElement`, we are using the `FileUploader` control to generate an input text field and a button to load 2D or 3D resources stored locally. We have specified the following file types as valid file types for loading using this `formElement`.

The labels for each of the fields are specified in the text attributes.

```
<mvc:View
   controllerName="singleFile.controller.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc"
   xmlns:vk="sap.ui.vk"
   xmlns:l="sap.ui.layout"
   xmlns:f="sap.ui.layout.form"
   xmlns:u="sap.ui.unified"
   displayBlock="true">
    <App id="singleFile">
        <Page
            title="{i18n>pageTitle}">
            <vk:Viewer
                id="viewer"
                toolbarTitle="{i18n>viewerToolbarTitle}"
                width="100%"
                height="85%"
                       />
                <f:Form
                layout="ResponsiveGridLayout"
                editable="true">
                    <f:layout>
                        <f:ResponsiveGridLayout/>
                    </f:layout>
                    <f:formContainers>
                        <f:FormContainer>
                            <f:formElements>
                                <f:FormElement>
                                    <f:fields>
                                        <Input
                                           value="{source>/remoteUrl}"
                                           valueLiveUpdate="true"
                                           placeholder="{i18n>formRemoteURL}">
                                            <layoutData>
                                                <l:GridData hCells="auto" />
                                            </layoutData>
                                        </Input>
                                        <Button
                                            text="{i18n>buttonLoadModel}"
                                            press="onPressLoadRemoteModel">
                                            <layoutData>
                                                <l:GridData hCells="2" />
                                            </layoutData>
                                        </Button>
                                        <Button
                                            text="{i18n>buttonLoadImage}"
                                            press="onPressLoadRemoteImage">
                                            <layoutData>
                                                <l:GridData hCells="3" />
                                            </layoutData>
                                        </Button>
                                    </f:fields>
                                </f:FormElement>
                                <f:FormElement>
                                    <f:fields>
                                        <u:FileUploader
                                           fileType="vds,png,jpg,jpeg,gif"
                                           placeholder="{i18n>formLocalFileName}"
                                           width="100%"
                                           change="onChangeFileUploader">
                                        </u:FileUploader>
```

```
                                    </f:fields>
                                </f:FormElement>
                            </f:formElements>
                        </f:FormContainer>
                    </f:formContainers>
                </f:Form>
            </Page>
        </App>
    </mvc:View>
```

App.controller.js

This file contains the logic for loading files into the Viewer application.

- The `handleEmptyUrl` function specifies what should occur if a user clicks on any of the buttons for loading, without having specified a URL to a resource first.
- The `loadModelIntoViewer` function specifies how the resource will be loaded into the Viewer application for viewing.
- The following event functions specify how the form elements should behave during certain events. The following list outlines what each of the functions do:
  - `onInit` - declares an empty structure when the controller is initialized. The empty structure is set as the model for the URLs.
  - `onPressLoadRemoteModel` - the logic for the button that loads 3D resources stored remotely.
  - `onPressLoadRemoteImage` - the logic for the button that loads 2D resources stored remotely.
  - `onChangeFileUploader` - the logic for the fields that load 2D or 3D resources stored locally.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/vk/ContentResource",
    "sap/m/MessageToast"
], function(Controller, JSONModel, ContentResource, MessageToast) {
    "use strict";
    var handleEmptyUrl = function(view) {
        var oBundle = view.getModel("i18n").getResourceBundle();
        var msg = oBundle.getText("missingUrl");
        MessageToast.show(msg);
    };
    var loadModelIntoViewer = function(viewer, remoteUrl, sourceType, localFile)
{
        //what is currently loaded in the view is destroyed
        viewer.destroyContentResources();
        var source = remoteUrl || localFile;
        if (source) {

            //content of viewer is replaced with new data
            var contentResource = new ContentResource({
                source: remoteUrl,
                sourceType: sourceType,
                sourceId: "abc"
            });
            //content: chosen path. content added to the view
            viewer.addContentResource(contentResource);
        }
    };
    return Controller.extend("singleFile.controller.App", {
        onInit: function() {
            var sourceData = {
                localFile: undefined,
                remoteUrl: undefined
            };
            var model = new JSONModel();
```

```
            model.setData(sourceData);
            this.getView().setModel(model, "source");
        },
        onPressLoadRemoteModel: function(event) {
            var view = this.getView();
            var sourceData = view.getModel("source").oData;
            var viewer = view.byId("viewer");
            if (sourceData.remoteUrl) {
                loadModelIntoViewer(viewer, sourceData.remoteUrl, "vds");
            } else {
                handleEmptyUrl(view);
            }
        },
        onPressLoadRemoteImage: function(event) {
            var view = this.getView();
            var sourceData = view.getModel("source").oData;
            var viewer = view.byId("viewer");
            if (sourceData.remoteUrl) {
                loadModelIntoViewer(viewer, sourceData.remoteUrl, "jpg");
            } else {
                handleEmptyUrl(view);
            }
        },
        onChangeFileUploader: function(event) {
            var view = this.getView();
            var viewer = view.byId("viewer");
            var localFile = event.getParameter("files")[0];
            //if user selects a local file
            if (localFile) {
                var fileName = localFile.name;
                var index = fileName.lastIndexOf(".");
                if (index >= 0 && index < fileName.length - 1) {
                    var sourceType = fileName.substr(index + 1);
                    loadModelIntoViewer(viewer, null, sourceType, localFile);
                }
            }
        }
    });
});
```

Testing the Application

To test that the application works, we will load a local 3D resource into the Viewer application.

Download the `boxTestModel.vds` file from the *Samples*. Click the *Browse...* button, navigate to the folder where the 3D resource is located, and load the model. Your screen should look like the following screenshot:

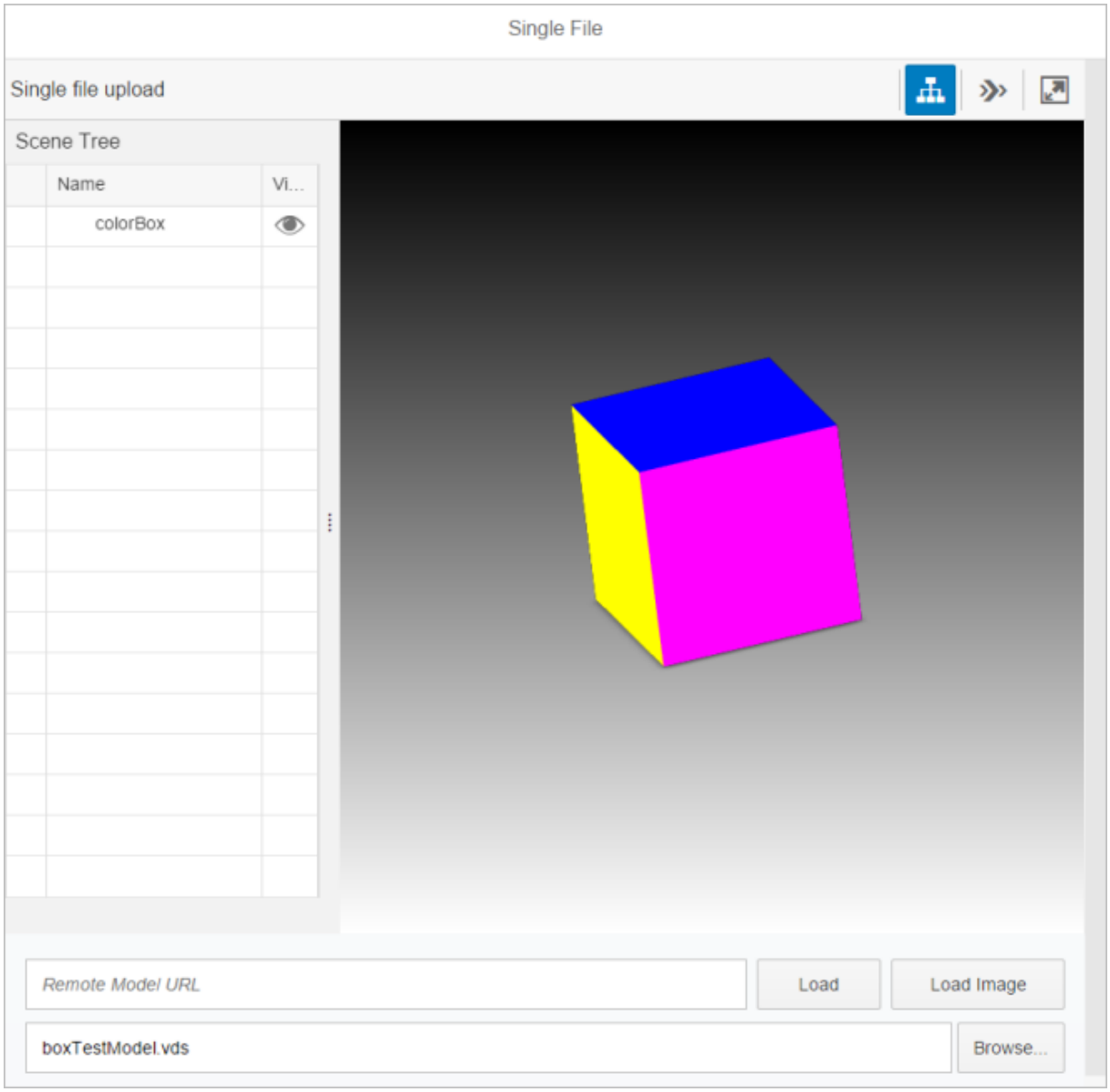


Figure 204: Viewer application loaded with a single VDS file

API Reference

- sap.ui.vk.Viewer
- sap.ui.vk.ContentResource

Step 2: 3D Viewer With Multiple File Loading

In this step, you will be creating a Viewer application that allows a user to load multiple 3D resources stored locally.

> ⚠ Caution
>
> The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

The content in this tutorial step references Step 1: 3D Viewer With Single File Loading [page 609] for comparative purposes.

Preview

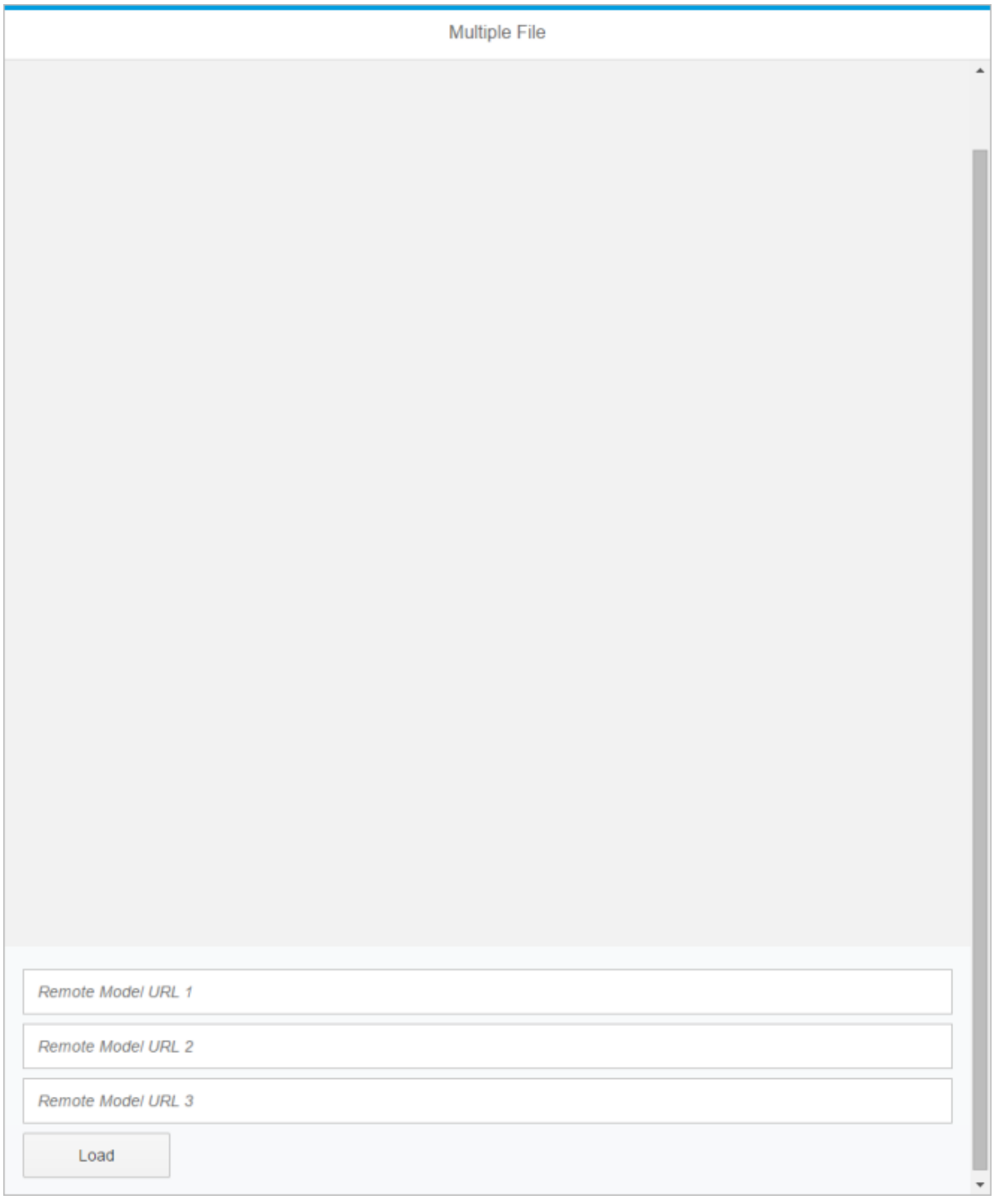


Figure 205: Viewer application with multiple file loading capability

Coding

You can view and download all files at 3D Viewer - Step 2 - Multiple File Loading .

index.html

Update the `index.html` file to reference the `multipleFiles` namespace, which will be the namespace we'll use for the sample application in this step.

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>
        <script id="sap-ui-bootstrap"
            src="../../../../../../../../../resources/sap-ui-core.js"
            data-sap-ui-libs="sap.ui.vk, sap.m"
            data-sap-ui-theme="sap_belize"
            data-sap-ui-bindingSyntax="complex"
            data-sap-ui-resourceroots='{
                "multipleFiles": "./"
            }'>
        </script>
        <script>
        sap.ui.getCore().attachInit(function(){
            new sap.m.Shell({
                app: new sap.ui.core.ComponentContainer({
                    name : "multipleFiles"
                })
            }).placeAt("content");
        });
        </script>
    </head>
    <body id="content" class="sapUiBody">
    </body>
</html>
```

Component.js

Update the `Component.js` file to reference the namespace specified for this application.

```
sap.ui.define([
        "sap/ui/core/UIComponent"
], function (UIComponent) {
        "use strict";
        return UIComponent.extend("multipleFiles.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
        }
    });
});
```

i18n.properties

In the `i18n.properties` file, we have labels for the toolbar, the page title, the three input fields, the *Load* button, and the error message that is displayed when the user attempts to load a model without specifying one to load.

```
# App Descriptor
appTitle=App title
appDescription=This is a description coming from the i18n as specified in
manifest.json

# Viewer Descriptor
```

```
viewerToolbarTitle=Upload multiple files

# Page Descriptor
pageTitle=Multiple File

# Form Descriptor
formRemoteURL1=Remote Model URL 1
formRemoteURL2=Remote Model URL 2
formRemoteURL3=Remote Model URL 3
buttonLoadModel=Load

# Message Toast
missingUrl=Please specify at least one URL
```

manifest.json

Update the `manifest.json` file so that it references the correct files.

```
{
    "_version": "1.8.0",
    "sap.app": {
        "id": "multipleFiles",
        "type": "application",
        "i18n": "i18n/i18n.properties",
        "title": "{{appTitle}}",
        "description": "{{appDescription}}",
        "applicationVersion": {
            "version": "1.0.0"
        },
    },
    "sap.ui": {
        "technology": "UI5",
        "deviceTypes": {
            "desktop": true,
            "tablet": true,
            "phone": true
        }
    },
    "sap.ui5": {
        "rootView": "multipleFiles.view.App",
        "dependencies": {
            "minUI5Version": "1.30",
            "libs": {
                "sap.m": {}
            }
        },
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "settings": {
                    "bundleName": "multipleFiles.i18n.i18n"
                }
            }
        }
    }
}
```

App.view.xml

This file specifies how the page in the application will be laid out. We only have one `formElement` in the form container, which contains the fields for loading 3D resources that are stored locally. In the element, we have specified three input text fields and one button for loading. The labels to use for each of the fields are specified in the text attributes.

```
<mvc:View
```

```
    controllerName="multipleFiles.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:vk="sap.ui.vk"
    xmlns:l="sap.ui.layout"
    xmlns:f="sap.ui.layout.form"
    xmlns:u="sap.ui.unified"
    displayBlock="true">
     <App id="multipleFiles">
         <Page
             title="{i18n>pageTitle}">
             <vk:Viewer
                 id="viewer"
                 toolbarTitle="{i18n>viewerToolbarTitle}"
                 width="100%"
                 height="85%"
                         />
             <f:Form editable="true">
                 <f:layout>
                     <f:ResponsiveGridLayout/>
                 </f:layout>
                 <f:formContainers>
                     <f:FormContainer>
                         <f:formElements>
                             <f:FormElement>
                                 <f:fields>
                                     <Input
                                            value="{source>/url1}"
                                            valueLiveUpdate="true"
                                            placeholder="{i18n>formRemoteURL1}">
                                         <layoutData>
                                             <l:GridData span="L12 M12 S12" />
                                         </layoutData>
                                     </Input>
                                     <Input
                                            value="{source>/url2}"
                                            valueLiveUpdate="true"
                                            placeholder="{i18n>formRemoteURL2}">
                                         <layoutData>
                                             <l:GridData span="L12 M12 S12" />
                                         </layoutData>
                                     </Input>
                                     <Input
                                            value="{source>/url3}"
                                            valueLiveUpdate="true"
                                            placeholder="{i18n>formRemoteURL3}">
                                         <layoutData>
                                             <l:GridData span="L12 M12 S12" />
                                         </layoutData>b
                                     </Input>
                                     <Button
                                             text="{i18n>buttonLoadModel}"
                                             press="onPressLoadRemoteModels">
                                         <layoutData>
                                             <l:GridData span="L2 M2 S2" />
                                         </layoutData>
                                     </Button>
                                 </f:fields>
                             </f:FormElement>
                         </f:formElements>
                     </f:FormContainer>
                 </f:formContainers>
             </f:Form>
         </Page>
     </App>
</mvc:View>
```

App.controller.js

Since we now have a different layout for the Viewer application compared to the sample Viewer application in Step 1: 3D Viewer With Single File Loading [page 609], we will need to change the logic for the application to accommodate for multiple file loading.

Add the `checkIfAllInputsEmpty` function to check whether the user has entered text into any of the input fields in the application. The `checkIfAllInputsEmpty` function returns the value `true` if the user hasn't entered any input at all, and the existing `handleEmptyUrl` function is called to display a message on the screen.

Update the `onInit` function so that we are specifying an empty data structure with three properties (`url1`, `url2`, and `url3`).

Replace the `loadModelIntoViewer` and `onPressLoadRemoteModel` functions with the following functions:

- `loadModelsIntoViewer` - loads the models into Viewer
- `onPressLoadRemoteModels` - handles the click event on the *Load* button

Remove the following functions:

- `onPressLoadRemoteImage` (since we are only loading 3D resources)
- `onChangeFileUploader` (since we are not using the FileUploader control for this application)

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/vk/ContentResource",
    "sap/m/MessageToast"
], function (Controller, JSONModel, ContentResource, MessageToast) {
    "use strict";

    //throws a Message Toast alert on the screen
    //when the user tries to load a model but there's no url specified
    var handleEmptyUrl = function (view) {
        var oBundle = view.getModel("i18n").getResourceBundle();
        var msg = oBundle.getText("missingUrl");
        MessageToast.show(msg);
    };

    //checks if all URL input fields are empty or not
    var checkIfAllInputsEmpty = function (urls) {
        var allEmpty = true;
        for (var i = 0; i < urls.length; i++) {
            if (urls[i]) {
                allEmpty = false;
                break;
            }
        }
        return allEmpty;
    }

    //loads the models from the URLs into the viewer
    var loadModelsIntoViewer = function (viewer, urls, sourceType) {
        //clears all the models currently loaded in the viewer
        viewer.destroyContentResources();

        //iterates through all URLs
        //and loads all models into the viewer
        for (var i = 0; i < urls.length; i++) {
            if (urls[i]) {
                var contentResource = new ContentResource({
                    source: urls[i],
                    sourceType: sourceType,
```

```
                    sourceId: "abc"
                   name: urls[i].split("/")[2]
                });
                //add current model to the viewer
                viewer.addContentResource(contentResource);
            }
        }
    };
    return Controller.extend("multipleFiles.controller.App", {
        //when the controller is initialized,
        //we declare an empty structure and
        //we set this as model for the URLs
        onInit: function () {
            var sourceData = {
                url1: "",
                url2: "",
                url3: ""
            };
            var model = new JSONModel();
            model.setData(sourceData);
            this.getView().setModel(model, "source");
        },

        //onPressLoadRemoteModels handles the click event on the LOAD button
        onPressLoadRemoteModels: function (event) {
            var view = this.getView();
            //set the source model to a variable
            var sourceData = view.getModel("source").getData;

            //get the current viewer control
            var viewer = view.byId("viewer");

            //create the list of URLs from the input fields
            var urls = [sourceData.url1, sourceData.url2, sourceData.url3];

            //if all URL inputs are empty show an alert on the screen
            //if at least one URL is specified, then take the URL list
            //and load all existing ones into the viewer
            if (checkIfAllInputsEmpty(urls)) {
                handleEmptyUrl(view);
            } else {
                loadModelsIntoViewer(viewer, urls, "vds");
            }
        }
    });
});
```

Testing the Application

To test that the application works, we will load three 3D resources into the Viewer application.

Download the following VDS files from the *Samples* in the Demo Kit:

- `boxTestModel.vds`
- `coneTestModel.vds`
- `cylinderTestModel.vds`

Type in the file path of the VDS files into each of the input text fields, and click on the *Load* button to load the 3D models. Your screen should look like the following screenshot:

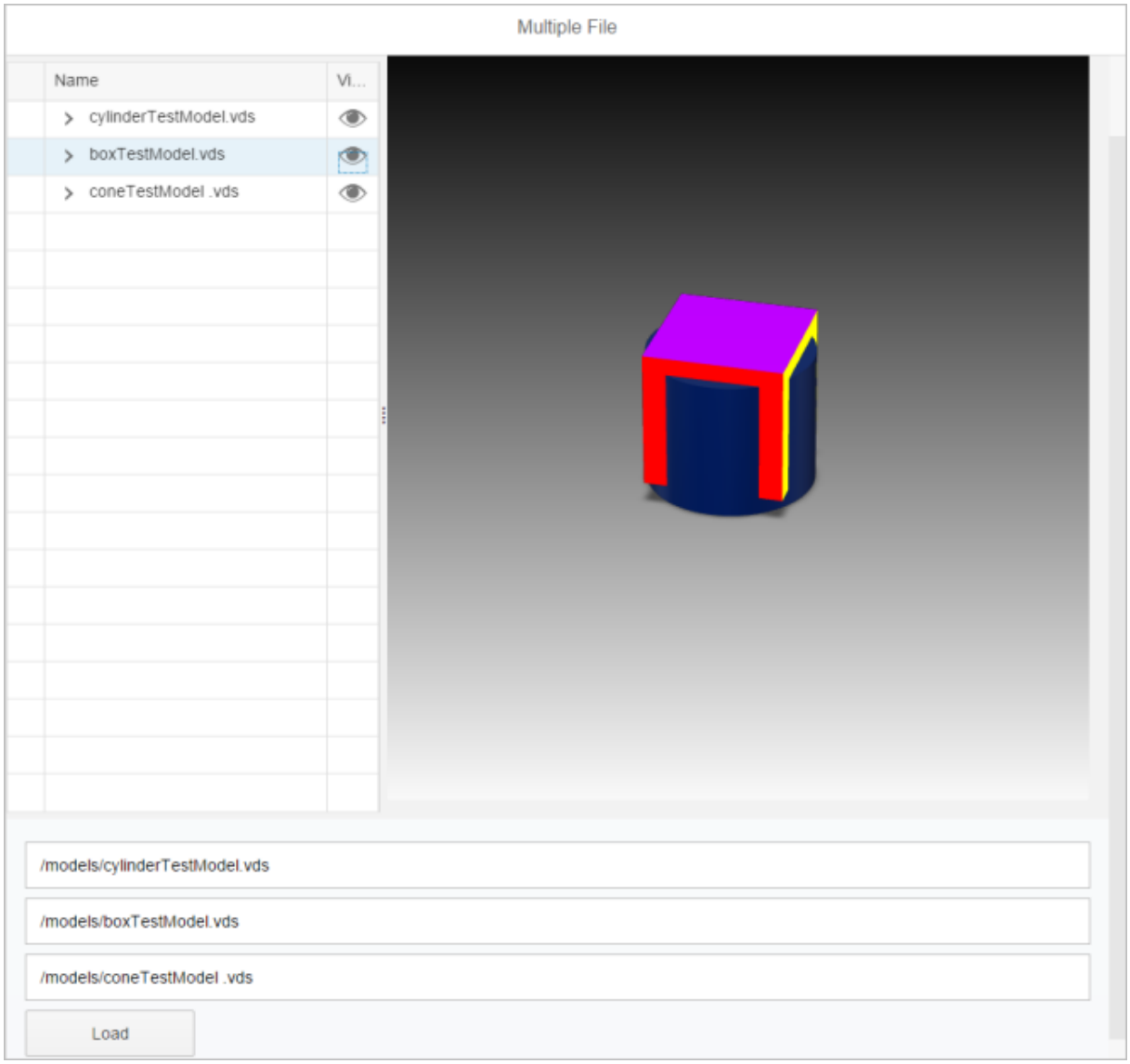


Figure 206: Viewer application loaded with three VDS files

API Reference

- sap.ui.vk.GraphicsCore

Step 3: 3D Viewer Using the Viewport Control

In this step, you will be creating a 3D Viewer application using the `sap.ui.vk.Viewport` control.

> ⚠ Caution
>
> The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

In previous steps, we utilized the `sap.ui.vk.Viewer` composite control to create a Viewer application capable of loading a 2D or 3D resource. Now, we will create a Viewer application with a pre-loaded resource without using the composite `sap.ui.vk.Viewer` control. Instead, we will use the following controls and library in `sap.ui.vk`, which are what you'll need at a minimum to display a 3D model in your application.

- `Viewport` control
- `ContentResource` control
- `GraphicsCore` library

We will build on this sample application in later steps of the 3D Viewer tutorial by introducing the other non-composite `sap.ui.vk` controls to create more complex Viewer applications.

Preview

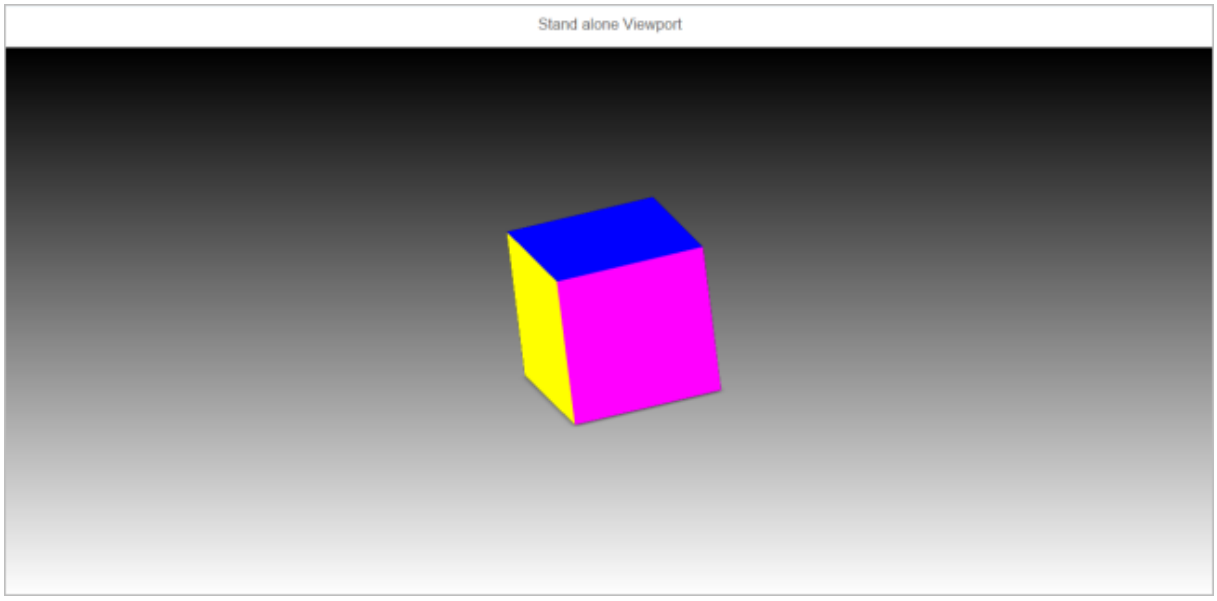


Figure 207: Viewer application that consists solely of a Viewport

Coding

You can view and download all files at 3D Viewer - Step 3 - Standalone Viewport.

index.html

Update the `index.html` file to reference the `standaloneViewport` namespace, which will be the namespace we'll use for the sample application in this step.

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>
        <script id="sap-ui-bootstrap"
            src="../../../../../../../../../resources/sap-ui-core.js"
            data-sap-ui-libs="sap.ui.vk"
```

```
            data-sap-ui-theme="sap_belize"
            data-sap-ui-bindingSyntax="complex"
            data-sap-ui-resourceroots='{
                "standaloneViewport": "./"
            }'>
        </script>
        <script>
        sap.ui.getCore().attachInit(function(){
            new sap.m.Shell({
                app: new sap.ui.core.ComponentContainer({
                    name : "standaloneViewport"
                })
            }).placeAt("content");
        });
        </script>
    </head>
    <body id="content" class="sapUiBody">
    </body>
</html>
```

Component.js

Update the `Component.js` file to reference the namespace specified for this application.

```
sap.ui.define([
        "sap/ui/core/UIComponent"
], function (UIComponent) {
        "use strict";
        return UIComponent.extend("standaloneViewport.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
        }
    });
});
```

i18n.properties

Because we are not creating any fields that a user can interact with, we only have one line of code which specifies what the label for the page title is.

```
# Page Descriptor
pageTitle=Standalone Viewport
```

manifest.json

Update the `manifest.json` file so that it references the correct files.

```
{
    "_version": "1.8.0",
    "sap.app": {
        "id": "standaloneViewport",
        "type": "application",
        "i18n": "i18n/i18n.properties",
        "title": "{{appTitle}}",
        "description": "{{appDescription}}",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui": {
```

```
        "technology": "UI5",
        "deviceTypes": {
            "desktop": true,
            "tablet": true,
            "phone": true
        }
    },
    "sap.ui5": {
        "rootView": "standaloneViewport.view.App",
        "dependencies": {
            "minUI5Version": "1.30",
            "libs": {
                "sap.m": {}
            }
        },
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "settings": {
                    "bundleName": "standaloneViewport.i18n.i18n"
                }
            }
        }
    }
}
```

App.view.xml

Because the Viewport is the only item that we need to display on the application screen, we only need to have the `<vk:Viewport>` element added to this file. In the element's attributes, we specify the Viewport's width and height on the screen, and also give it an arbitrary identifier value.

```
<mvc:View
        controllerName="standaloneViewport.controller.App"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc"
        xmlns:vk="sap.ui.vk"
        xmlns:l="sap.ui.layout"
        xmlns:f="sap.ui.layout.form"
        xmlns:u="sap.ui.unified"
        displayBlock="true">
    <App id="standaloneViewport">
        <Page
            title="{i18n>pageTitle}">
            <vk:Viewport
                    id="viewport"
                    width="100%"
                    height="50%"/>
        </Page>
    </App>
</mvc:View>
```

App.controller.js

The logic in this `App.controller.js` file can be summarized as follows:

1. Create a Content Resource that stores a pre-specified model
2. Initiate a scene in our application in the following order:
    1. Create a Graphics Core instance
    2. Create a Viewport that is bound to the Graphics Core instance
    3. Load the Content Resource to the Graphics Core for rendering on the Viewport

```
sap.ui.define([
```

```
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/vk/ContentResource",
], function (Controller, JSONModel, ContentResource) {
    "use strict";

    var contentResource = new sap.ui.vk.ContentResource({
        source: "/models/boxTestModel.vds",
        sourceType: "vds",
        id: "abc123"
    });

    return Controller.extend("standaloneViewport.controller.App",{
        onInit: function() {

            var mainScene;

            jQuery.sap.require("sap.ui.vk.GraphicsCore");
            var graphicsCore = new sap.ui.vk.GraphicsCore({},{
                antialias: true,
                alpha: true,
                premultipliedAlpha: false
            });

            var view = this.getView();
            var viewport = view.byId("viewport");

            viewport.setGraphicsCore(graphicsCore);
            graphicsCore.loadContentResourcesAsync([contentResource],
function(sourcesFailedToLoad){
                if (sourcesFailedToLoad){
                    jQuery.sap.log.error("Some of content resources cannot be
loaded.");
                } else {
                    var scene = graphicsCore.buildSceneTree([contentResource]);
                    if (scene){
                        mainScene = scene;
                        viewport.setScene(mainScene);
                    } else {
                        jQuery.sap.log.error("Failed to load viewport");
                    }
                }
            });
        }
    });
});
```

We'll now break the code down to look at each part in more detail.

Create a New Content Resource

Create a `contentResource` object that specifies the resource to load. In this case, we're pre-loading the `boxTestModel.vds` model into the application. This occurs before the scene in our Viewer application is initiated.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/vk/ContentResource",
], function (Controller, JSONModel, ContentResource) {
    "use strict";
    var contentResource = new sap.ui.vk.ContentResource({
        source: "/models/boxTestModel.vds",
        sourceType: "vds",
        id: "abc123"
```

Create a New Graphics Core Instance

Create a new Graphics Core instance. We're only specifying one input parameter, which is for the WebGL context attributes. We then get the current viewport, and attach it to the Graphics Core instance so the model we've loaded can be rendered.

```
return Controller.extend("standaloneViewport.controller.App",{
    onInit: function() {

    var mainScene;
        jQuery.sap.require("sap.ui.vk.GraphicsCore");
        var graphicsCore = new sap.ui.vk.GraphicsCore({},{
            antialias: true,
            alpha: true,
            premultipliedAlpha: false
        });
        var view = this.getView();
        var viewport = view.byId("viewport");

        viewport.setGraphicsCore(graphicsCore);
```

Load the Content Resource for Rendering

Now that we've associated the viewport with the Graphics Core, we can load our model to be displayed on the Viewport. In the following code block, we have specified some checks to make sure that the model loads correctly. If no resource is loaded, or if there is an error loading the resource, we throw the following error on the screen "Some of content resources cannot be loaded". Otherwise, we build a scene with the loaded resource, and then display this scene on the viewport. If the scene itself does not load into the Viewport, we throw an error saying the scene could not be built "Failed to build the scene."

```
graphicsCore.loadContentResourcesAsync([contentResource],
function(sourcesFailedToLoad){
    if (sourcesFailedToLoad){
        jQuery.sap.log.error("Some of content resources cannot be loaded.");
    } else {
        var scene = graphicsCore.buildSceneTree([contentResource]);
            if (scene){
                mainScene = scene;
                viewport.setScene(mainScene);
            } else {
                jQuery.sap.log.error("Failed to load viewport");
            }
    }
});
}
});
```

API Reference

- sap.ui.vk.Viewport

Related Information

Viewport [page 2480]

Step 4: Adding a Scene Tree

In this step, you will be adding an `sap.ui.vk.SceneTree` control to your 3D Viewer application.

> ⚠ Caution
>
> The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

The following `sap.ui.vk` controls are introduced in this step:

- `SceneTree` control
- `ViewStateManager` control

We will use the `SceneTree` control to create a tree structure that displays the hierarchy of the nodes for the loaded model. We will then use the `ViewStateManager` control to link the Scene Tree with the model loaded into the Viewport, so that we can visually associate a selection in the scene with its corresponding node as displayed in the Scene Tree.

The content in this step builds on the code from Step 3: 3D Viewer Using the Viewport Control [page 624], and code changes performed in this step of the tutorial are done in relation to the files in Step 3: 3D Viewer Using the Viewport Control [page 624].

Preview

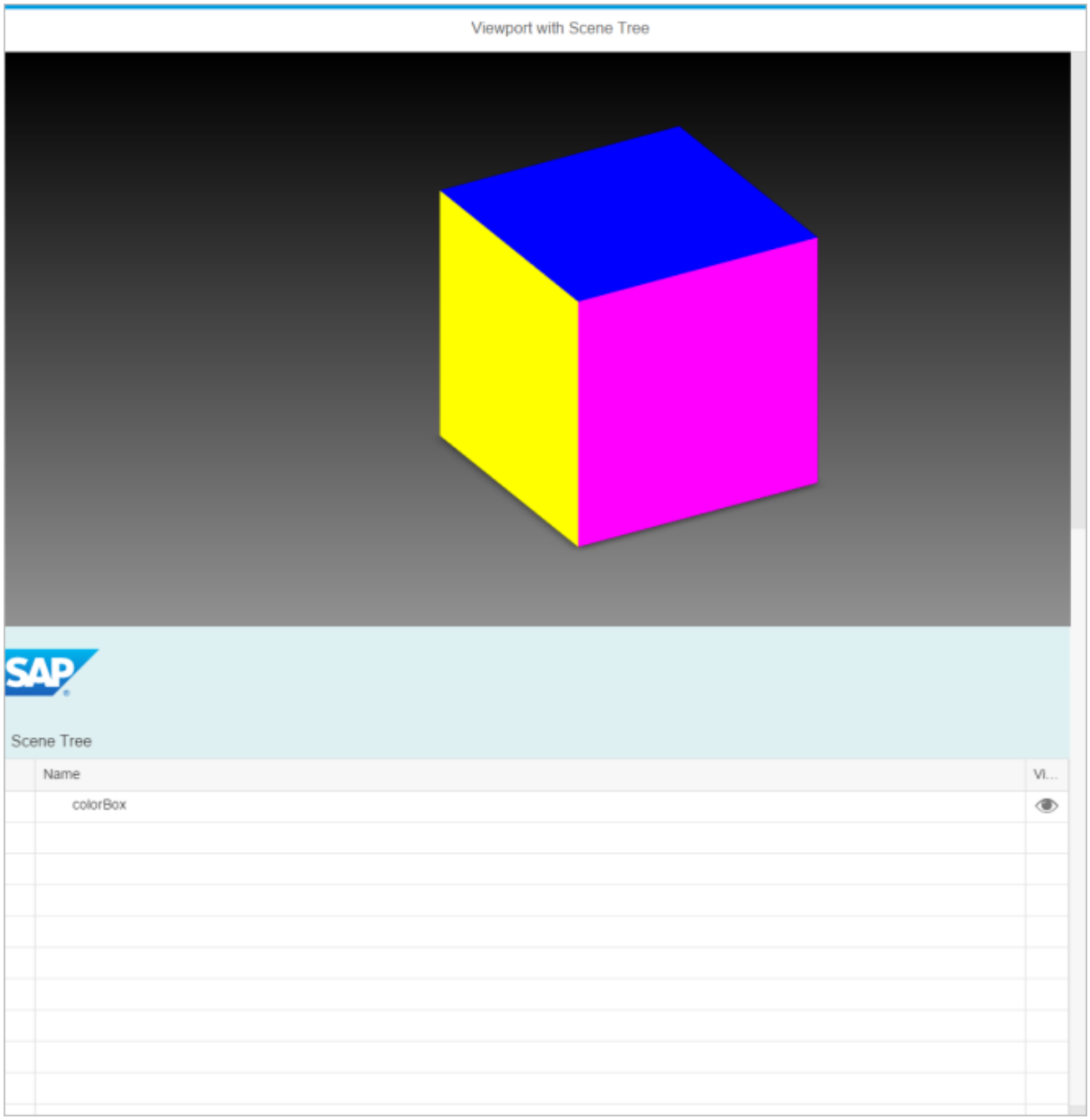


Figure 208: Viewer application with a Viewport and a Scene Tree

Coding

You can view and download all files at 3D Viewer - Step 4 - Viewport with Scene Tree.

index.html

Update the `index.html` file to reference the `viewportScenetree` namespace, which will be the namespace we'll use for the sample application in this step.

```
<!DOCTYPE HTML>
```

```
<html>
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>
    <script id="sap-ui-bootstrap"
        src="../../../../../../../../../resources/sap-ui-core.js"
        data-sap-ui-libs="sap.ui.vk, sap.m"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-bindingSyntax="complex"
        data-sap-ui-resourceroots='{
            "viewportScenetree": "./"
        }'>
    </script>
    <script>
    sap.ui.getCore().attachInit(function(){
        new sap.m.Shell({
            app: new sap.ui.core.ComponentContainer({
                name : "viewportScenetree"
            })
        }).placeAt("content");
    });
    </script>
    </head>
    <body id="content" class="sapUiBody">
    </body>
</html>
```

Component.js

Update the `Component.js` file to reference the namespace specified for this application.

```
sap.ui.define([
    "sap/ui/core/UIComponent"
], function (UIComponent) {
    "use strict";
    return UIComponent.extend("viewportScenetree.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
        }
    });
});
```

i18n.properties

Update the page title to say "Viewport with Scene Tree".

```
# Page Descriptor
pageTitle=Viewport with Scene Tree
```

manifest.json

Update the `manifest.json` file so that it references the correct files.

```
{
    "_version": "1.8.0",
    "sap.app": {
        "id": "viewportScenetree",
        "type": "application",
        "i18n": "i18n/i18n.properties",
        "title": "{{appTitle}}",
```

```
        "description": "{{appDescription}}",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui": {
        "technology": "UI5",
        "deviceTypes": {
            "desktop": true,
            "tablet": true,
            "phone": true
        }
    },
    "sap.ui5": {
        "rootView": "viewportScenetree.view.App",
        "dependencies": {
            "minUI5Version": "1.30",
            "libs": {
                "sap.m": {}
            }
        },
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "settings": {
                    "bundleName": "viewportScenetree.i18n.i18n"
                }
            }
        }
    }
}
```

App.view.xml

To display the Scene Tree in your application, add the `<vk:SceneTree>` element. In the element's attribute's, specify the Scene Tree control's width and height on the screen, and give it an arbitrary identifier value.

```
<mvc:View
        controllerName="viewportScenetree.controller.App"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc"
        xmlns:vk="sap.ui.vk"
        xmlns:l="sap.ui.layout"
        xmlns:f="sap.ui.layout.form"
        xmlns:u="sap.ui.unified"
        displayBlock="true">
    <App id="viewportScenetree">
        <Page
                title="{i18n>pageTitle}">
            <vk:Viewport
                    id="viewport"
                    width="100%"
                    height="25%"/>
            <vk:SceneTree
                    id="scenetree"
                    width="100%"
                    height="25%"/>
        </Page>
    </App>
</mvc:View>
```

App.controller.js

For the most part, the code in the `App.controller.js` file will be the same as the content of the `App.controller.js` file for the application in Step 3: 3D Viewer Using the Viewport Control [page 624]. The

highlighted sections in the following code block indicate the additions and changes made in the code to incorporate a Scene Tree in the 3D Viewer application.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/vk/ContentResource",
], function (Controller, JSONModel, ContentResource) {
    "use strict";
    var contentResource = new sap.ui.vk.ContentResource({
        source: "/models/boxTestModel.vds",
        sourceType: "vds",
        id: "abc123"
    });
    return Controller.extend("viewportScenetree.controller.App",{
        onInit: function() {
            var mainScene;
            jQuery.sap.require("sap.ui.vk.GraphicsCore");
            var graphicsCore = new sap.ui.vk.GraphicsCore({},{
                antialias: true,
                alpha: true,
                premultipliedAlpha: false
            });
            var view = this.getView();
            var viewport = view.byId("viewport");
            var sceneTree = view.byId("scenetree");
            viewport.setGraphicsCore(graphicsCore);
            graphicsCore.loadContentResourcesAsync([contentResource],
function(sourcesFailedToLoad){
                if (sourcesFailedToLoad){
                    jQuery.sap.log.error("Some of content resources cannot be
loaded.");
                } else {
                    var scene = graphicsCore.buildSceneTree([contentResource]);
                    if (scene){
                        mainScene = scene;
                        viewport.setScene(mainScene);
                        var viewStateManager =
graphicsCore.createViewStateManager(mainScene.getDefaultNodeHierarchy());
                        viewport.setViewStateManager(viewStateManager);
                        sceneTree.setScene(mainScene, viewStateManager);
                    } else {
                        jQuery.sap.log.error("Failed to build the scene tree.");
                    }
                }
            });
        }
    });
});
```

Let us look at the changes in more detail.

The following code line adds a new Scene Tree object.

```
var sceneTree = view.byId("scenetree");
```

We create a new object called `viewStateManager` that gets the node hierarchy of the resource that's loaded into the scene. Then, we associate the created `viewStateManager` object with our viewport. We also associate the nodes in the `viewStateManager` object, as well as the resource loaded into the main scene with the Scene Tree.

```
var viewStateManager =
graphicsCore.createViewStateManager(mainScene.getDefaultNodeHierarchy());
viewport.setViewStateManager(viewStateManager);
```

```
sceneTree.setScene(mainScene, viewStateManager);
```

Finally, we changed the message for the condition that determines whether the viewport and the scene tree loaded successfully or not.

```
jQuery.sap.log.error("Failed to build the scene tree.");
```

API Reference

- sap.ui.vk.SceneTree

Related Information

Scene Tree [page 2470]

Step 5: Adding Step Navigation

In this step, you will be adding the `sap.ui.vk.StepNavigation` control to a 3D Viewer application.

> ⚠ Caution
>
> The controls in the `sap.ui.vk` library are currently flagged as experimental. For more information, see Compatibility Rules [page 17].

Sometimes, you may encounter a 3D model that has a sequence of animations associated with it. The `StepNavigation` control allows you to display the steps in the animation sequence, navigate to the individual steps in the animation sequence, and play the animation in a single step or in all of the steps.

The content in this step builds on the code from Step 4: Adding a Scene Tree [page 630], and code changes performed in this step of the tutorial are done in relation to the files in Step 4: Adding a Scene Tree [page 630].

Preview

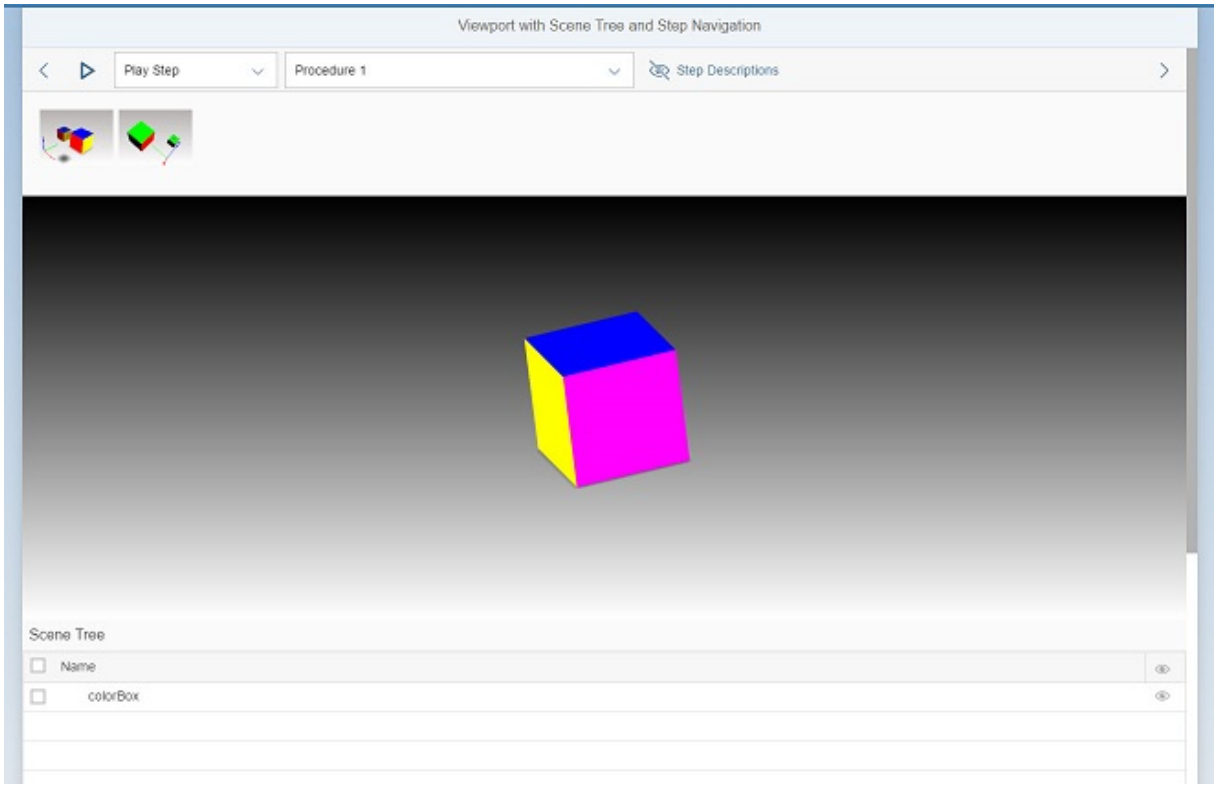


Figure 209: Viewer application with a Step Navigation, Viewport, and Scene Tree

Coding

You can view and download all files at 3D Viewer - Step 5 - Viewport with Scene Tree and Step Navigation.

index.html

Update the `index.html` file to reference the `viewportScenetreeStepnav` namespace, which will be the namespace we'll use for the sample application in this step.

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>
    <script id="sap-ui-bootstrap"
        src="../../../../../../../../../resources/sap-ui-core.js"
        data-sap-ui-libs="sap.ui.vk, sap.m, sap.ui.core"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-bindingSyntax="complex"
        data-sap-ui-resourceroots='{
            "viewportScenetreeStepnav": "./"
        }'>
    </script>
    <script>
    sap.ui.getCore().attachInit(function(){
        new sap.m.Shell({
            app: new sap.ui.core.ComponentContainer({
                name : "viewportScenetreeStepnav"
```

```
            })
        }).placeAt("content");
    });
    </script>
    </head>
    <body id="content" class="sapUiBody">
    </body>
</html>
```

Component.js

Update the `Component.js` file to reference the namespace specified for this application.

```
sap.ui.define([
    "sap/ui/core/UIComponent"
], function (UIComponent) {
    "use strict";
    return UIComponent.extend("viewportScenetreeStepnav.Component", {
        metadata: {
            manifest: "json"
        },
        init: function () {
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
        }
    });
});
```

i18n.properties

Because we are not creating any fields that a user can interact with, we only have one line of code which specifies what the label for the page title is.

```
# Page Descriptor
pageTitle=Viewport with Scene Tree and Step Navigation
```

manifest.json

Update the `manifest.json` file so that it references the correct files.

```
{
    "_version": "1.8.0",
    "sap.app": {
        "id": "viewportScenetreeStepnav",
        "type": "application",
        "i18n": "i18n/i18n.properties",
        "title": "{{appTitle}}",
        "description": "{{appDescription}}",
        "applicationVersion": {
            "version": "1.0.0"
        }
    },
    "sap.ui": {
        "technology": "UI5",
        "deviceTypes": {
            "desktop": true,
            "tablet": true,
            "phone": true
        }
    },
    "sap.ui5": {
        "rootView": "viewportScenetreeStepnav.view.App",
        "dependencies": {
            "minUI5Version": "1.30",
```

```
            "libs": {
                "sap.m": {}
            }
        },
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "settings": {
                    "bundleName": "viewportScenetreeStepnav.i18n.i18n"
                }
            }
        }
    }
}<vk:StepNavigation>
```

App.view.xml

In this file, we have added a `<vk:StepNavigation>` element to this file. We have specified the width and height of the `StepNavigation` control on the screen.

```
<mvc:View
        controllerName="viewportScenetreeStepnav.controller.App"
        xmlns="sap.m"
        xmlns:mvc="sap.ui.core.mvc"
        xmlns:vk="sap.ui.vk"
        xmlns:l="sap.ui.layout"
        xmlns:f="sap.ui.layout.form"
        xmlns:u="sap.ui.unified"
        displayBlock="true">
    <App id="viewportScenetreeStepnav">
        <Page
                title="{i18n>pageTitle}">
                <vk:StepNavigation
                id="StepNavigation"
                width="100%"
                    height="17.5%"/>
            <vk:Viewport
                    id="viewport"
                    width="100%"
                    height="50%"/>
            <vk:SceneTree
                    id="scenetree"
                    width="100%"
                    height="50%"/>
        </Page>
    </App>
</mvc:View>
```

App.controller.js

The highlighted sections in the following code block indicate the additions and changes made in the code to incorporate step navigation in the 3D Viewer application.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/vk/ContentResource",
], function (Controller, JSONModel, ContentResource) {
    "use strict";
    var contentResource = new sap.ui.vk.ContentResource({
        source: "/models/boxTestModel.vds",
        sourceType: "vds",
        id: "abc123"
    });
    return Controller.extend("viewportScenetreeStepnav.controller.App",{
```

```
        onInit: function() {
            var mainScene;
            jQuery.sap.require("sap.ui.vk.GraphicsCore");
            var graphicsCore = new sap.ui.vk.GraphicsCore({},{
                antialias: true,
                alpha: true,
                premultipliedAlpha: false
            });
            var view = this.getView();
            var viewport = view.byId("viewport");
            var sceneTree = view.byId("scenetree");
            var stepNavigation = view.byId ("StepNavigation");
            viewport.setGraphicsCore(graphicsCore);
            graphicsCore.loadContentResourcesAsync([contentResource],
function(sourcesFailedToLoad){
                if (sourcesFailedToLoad){
                    jQuery.sap.log.error("Some of content resources cannot be
loaded.");
                } else {
                    var scene = graphicsCore.buildSceneTree([contentResource]);
                    if (scene){
                        mainScene = scene;
                        viewport.setScene(mainScene);
                        var viewStateManager =
graphicsCore.createViewStateManager(mainScene.getDefaultNodeHierarchy());
                        viewport.setViewStateManager(viewStateManager);
                        sceneTree.setScene(mainScene, viewStateManager);
                        stepNavigation.setScene(mainScene);
                    } else {
                        jQuery.sap.log.error("Failed to build the viewport, the
scene tree, and the step navigation.");
                    }
                }
            });
        }
    });
});
```

Let us look at the changes in more detail.

The following line of code adds a new Step Navigation object.

```
var stepNavigation = view.byId ("StepNavigation");
```

We then associate the scene with the Step Navigation object, so that the animation sequences in the model are displayed in the `StepNavigation` control.

```
stepNavigation.setScene(mainScene);
```

Finally, we changed the message for the condition that determines whether the Viewport, the Scene Tree, and the Step Navigation controls loaded successfully or not:

```
jQuery.sap.log.error("Failed to build the Viewport, the Scene Tree, and the Step
Navigation.");
```

API Reference

- sap.ui.vk.StepNavigation

Related Information

Step Navigation [page 2474]

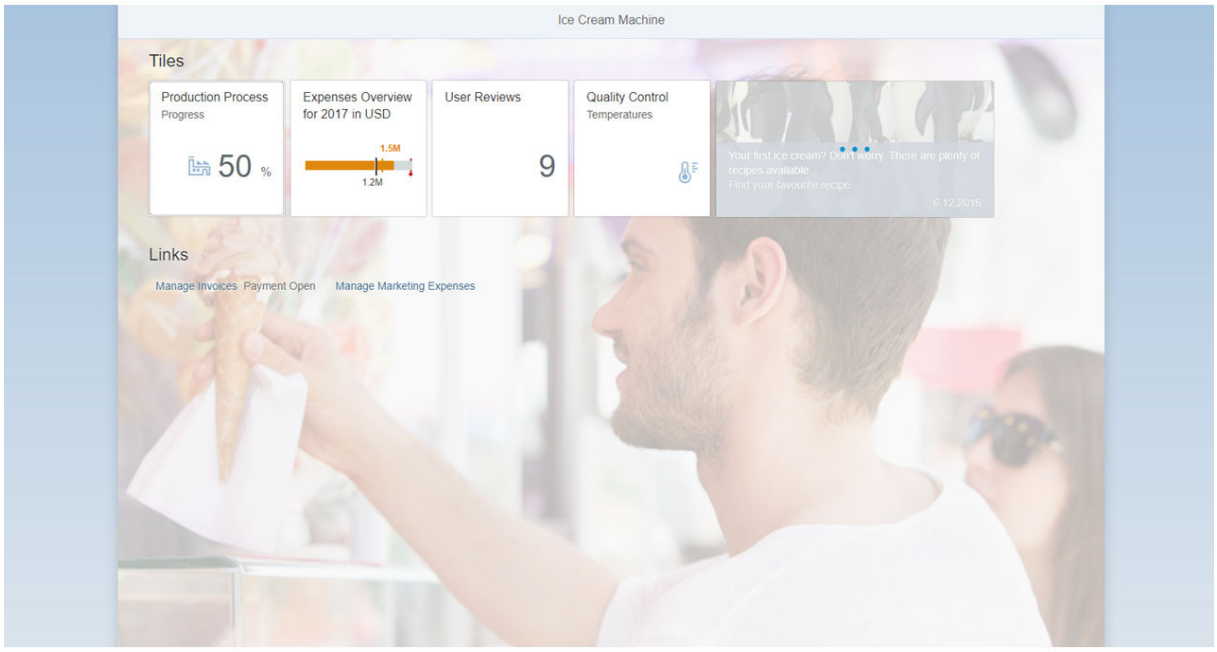
Ice Cream Machine

In this tutorial, we will show you how to use SAPUI5 controls like Generic Tiles, Micro Charts, and Process Flow.

Welcome to our **Ice Cream Machine** tutorial.

To get all of the important information about the production process at a glance, we will create a new start page. We will use the `GenericTile` control in different scenarios, the `NumericContent`, and several micro charts. The `ProcessFlow` will be used in a further view to display detailed data in the production process. We will create micro charts that show more specific information as in the `ProcessFlow`. In another view, the `ChartContainer` will be used to display the test results of the newly built ice cream machine. Finally, we will create an additional view with a `Timeline` control that will display our customer reviews.

Preview



Step 1: Initial Application

In the first step, we will explain how to get started with a development environment.

Choose your development environment

In this tutorial, you don't have to worry about creating any test data. Everything is already included in the initial app and you can reference the data in every step. You can view and download the samples for all steps in the Demo Kit at Ice Cream Machine.

> **i** Note
>
> Please keep in mind that we will not explain every single step of SAPUI5 application development, as we are focusing on selected controls. We will not cover topics like data binding, navigation, or the MVC pattern. For more information about these topics, see Essentials [page 691].

Creating an app

You can do this tutorial with SAP Web IDE (from SAPUI5 version 1.50) or with your own development environment. Just download the initial version of the final app from the sample here, extract the archive, and start coding.

For more information, check the following sections of the tutorials overview page (see Get Started: Setup, Tutorials, and Demo Apps [page 38]):

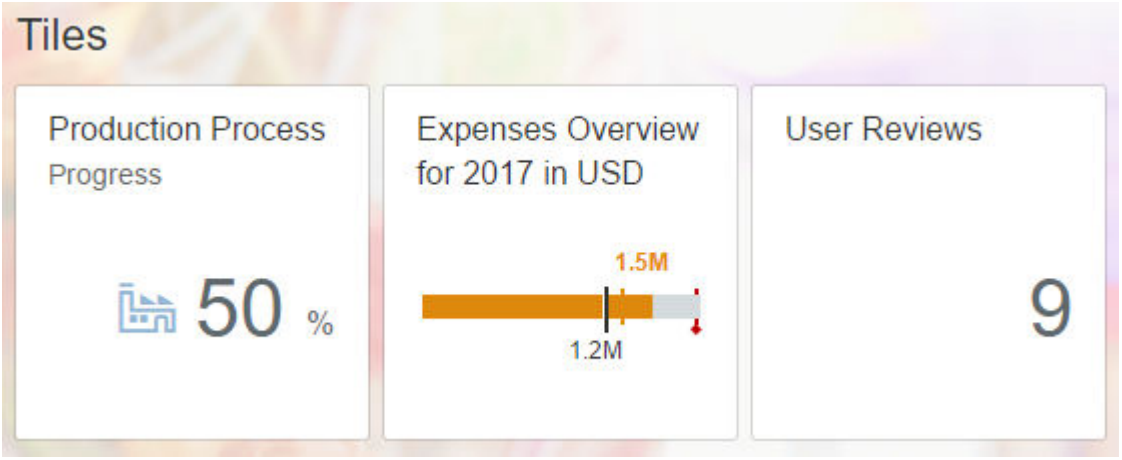
- Downloading Code for a Tutorial Step [page 40]
- Adapting Code to Your Development Environment [page 40]
- App Development Using SAP Web IDE [page 44]

Step 2: KPI Tile and Chart Tile on the Start Page

In this step, we will create our start page with the KPI Tile (`GenericTile` with `NumericContent`) and Chart Tile (`GenericTile` with `MicroChart`).

Preview

When you have completed this step, your start page will include the following tiles:



GenericTile with NumericContent

Let's start by creating the first `GenericTile` that shows the progress of the production process as percentage data (compared to all production steps). By selecting the **Production Process** tile, the user can navigate to another view to see the complete production process. We will implement this navigation in step 5.

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 2 - KPI Tile and Chart Tile in Start Page.

Startpage.view.xml

```
<mvc:View
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
02.controller.Startpage"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    xmlns:layout="sap.ui.layout">
    <Page title="{i18n>title}">
        <layout:VerticalLayout class="sapUiResponsiveMargin">
            <Title
                titleStyle="H2"
                text="{i18n>startpageTilesGroupTitle}"
                class="sapUiTinyMarginBegin" />
            <layout:HorizontalLayout allowWrapping="true">
                <GenericTile
                    class="sapUiTinyMarginBegin sapUiTinyMarginTop"
```

```
                    header="{i18n>startpagePFTileTitle}"
                    subheader="{i18n>startpagePFTileSubTitle}">
                    <tileContent>
                        <TileContent>
                            <content>
                                <NumericContent
                                    scale="%"
                                    value="{
                                        path: 'process>/Nodes',
                                        formatter: '.getProgress'
                                    }"
                                    icon="sap-icon://factory" />
                            </content>
                        </TileContent>
                    </tileContent>
                </GenericTile>
            </layout:HorizontalLayout>
        </layout:VerticalLayout>
    </Page>
</mvc:View>
```

First, we are going to add a `GenericTile` with the *header* and *subheader* properties in order to display the header and subheader . In the `TileContent` aggregation, the `TileContent` is added with a `NumericContent`. This lets you display a scale or unit with the *scale* property. The *value* property of the `NumericContent` displays the percentage value of the production progress by using a custom formatter. In our case, it is *.getProgress*. You can use any icon from the SAPUI5 icon poolas the *icon* property.

We will create another `GenericTile` with `NumericContent` in a similar way, which will navigate to another view. You can see customer reviews there. The number of reviews is displayed in this `GenericTile`. Navigation to the reviews view is implemented in step 5.

Startpage.view.xml

```
mvc:View
     controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
02.controller.Startpage"
     xmlns:mvc="sap.ui.core.mvc"
     xmlns="sap.m"
     xmlns:layout="sap.ui.layout">
     <Page title="{i18n>title}">
         <layout:VerticalLayout class="sapUiResponsiveMargin">
             <Title
                 titleStyle="H2"
                 text="{i18n>startpageTilesGroupTitle}"
                 class="sapUiTinyMarginBegin" />
             <layout:HorizontalLayout allowWrapping="true">
                ...
                <GenericTile
                     class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                     header="{i18n>startpageUserReviewsTileTitle}">
                     <tileContent>
                         <TileContent>
                             <content>
                                 <NumericContent
                                     value="{
                                         path: 'reviews>/UserReviews',
                                         formatter: '.getEntityCount'
                                     }"
                                         indicator="None"
                                         valueColor="Neutral" />
                             </content>
                         </TileContent>
                     </tileContent>
                 </GenericTile>
```

```
            </layout:HorizontalLayout>
        </layout:VerticalLayout>
    </Page>
</mvc:View>
```

In the `NumericContent`, we only want to display the number of user reviews. Since we don't want to show indicators, we set the *indicator* property to **None**. We are choosing **Neutral** as a value for the *valueColor* property to display a neutral semantic color.

Startpage.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
02.controller.Startpage", {
        ...
        getProgress: function(aNodes) {
                    if (!aNodes || aNodes.length === 0) {
                       return 0;
                    }
                    var iSum = 0;
                    for (var i = 0; i < aNodes.length; i++) {
                        iSum += aNodes[i].state === "Positive";
                    }
                    var fPercent = (iSum / aNodes.length) * 100;
                    return fPercent.toFixed(0);
                },
                getEntityCount: function(entities) {
                    return entities && entities.length || 0;
                }
    });
});
```

The *getProgress* function is the formatter function that belongs to the value binding of the `NumericContent` in the first `GenericTile`. It is used to return the ratio of positive process steps as compared to the complete number of process steps.

To get the number of user reviews in another `GenericTile`, we will add a new formatter called *.getEntityCount*. This formatter returns the number of entities found in the **reviews** JSON model.

GenericTile with BulletMicroChart

In the second tile on the start page, you can find an overview of the project expenses. We will use the `BulletMicroChart` control for this overview.

Startpage.view.xml

```
mvc:View
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
02.controller.Startpage"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    xmlns:layout="sap.ui.layout"
    xmlns:microchart="sap.suite.ui.microchart">
    <Page title="{i18n>title}">
        <layout:VerticalLayout class="sapUiResponsiveMargin">
```

```
            <Title
                titleStyle="H2"
                text="{i18n>startpageTilesGroupTitle}"
                class="sapUiTinyMarginBegin" />
            <layout:HorizontalLayout allowWrapping="true">
                                ...
               <GenericTile
                     class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                     header="{
                         parts: [
                             'i18n>startpageCCTileTitle',
                             'business>/year',
                             'business>/currency'
                         ],
                         formatter: '.formatMessage'
                     }">
                     <tileContent>
                         <TileContent>
                             <content>
                                 <microchart:BulletMicroChart
                                     size="Responsive"
                                     targetValue="{business>/plannedExpenses}"
                                     targetValueLabel="{
                                        path: 'business>/plannedExpenses',
                                        formatter: '.formatNumber'
                                    }"
                                     actualValueLabel="{
                                        path: 'business>/expenses',
                                        formatter: '.formatNumber'
                                    }"
                                     minValue="0">
                                     <microchart:actual>
                                         <microchart:BulletMicroChartData
                                             value="{business>/expenses}"
                                             color="Critical" />
                                     </microchart:actual>
                                     <microchart:thresholds>
                                         <microchart:BulletMicroChartData
                                             value="{business>/expensesCritical}"
                                             color="Critical" />
                                         <microchart:BulletMicroChartData
                                             value="{business>/budget}"
                                             color="Error" />
                                     </microchart:thresholds>
                                 </microchart:BulletMicroChart>
                             </content>
                         </TileContent>
                     </tileContent>
                 </GenericTile>
            </layout:HorizontalLayout>
        </VBox>
    </Page>
</mvc:View>
```

We are adding a new `GenericTile`. Here, we are setting only the *header* property of the tile because we don't need a subheader. We want to display only the Expenses overview. The `BulletMicroChart` is in the *TileContent*. Do not forget to add a namespace for the `sap.suite.ui.microchart` library in the **.view.xml*.

We want to show our business data in the `BulletMicroChart`. To do so, we are adding the `BulletMicroChartData` element to the **actual** aggregation of the chart control. This data element has two properties:

1. The *value* property that is bound to a property in the data model
2. The *color* property that is set to **Critical** to show when the expenses are about to exceed the limit. This property shows the amount of money already spent on this project.

Similarly, we can add **thresholds** to the chart to show when our expenses become **critical** but still manageable. For expenses that are too high for the company, the **Error** color applies. This time, the *value* properties are directly bound to the properties in the data model. No formatting is needed as the values are used only by the chart and are not displayed directly.

Finally, we will add a *targetValue* to our chart to display an indicator that shows the planned spending.

Startpage.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/format/NumberFormat",
    "sap/base/strings/formatMessage"
], function(Controller, NumberFormat) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
02.controller.Startpage", {
        ...
        formatNumber: function(value) {
            var oFloatFormatter = NumberFormat.getFloatInstance({
                style: "short",
                decimals: 1
            });
            return oFloatFormatter.format(value);
        }
    });
});
```

Since the production of our ice cream has already cost us a lot of money and the monetary values are very high, we need to provide custom data labels in order to have a neat chart design. The custom data labels can be added via the *actualValueLabel* and *targetValueLabel* properties that are set on the `BulletMicroChart`. In both instances, we use data binding with a custom formatter function from the *Startpage.controller.js* file to format the values using a standard SAPUI5 number formatter.

Step 3: Launch Tile and Slide Tile

In this step, we will create a new Launch Tile (`GenericTile` with `ImageContent`) and a `SlideTile`.

Preview

Initializing the models

To configure the next `GenericTiles`, you need to create models that contain news data. You can find the source code of the model data in */model/data/News.json*.

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 3 - Launch Tile and Slide Tile.

Startpage.controller.js

```
sap.ui.define([
       "sap/ui/core/mvc/Controller",
"sap/ui/model/json/JSONModel",
       "sap/ui/core/format/NumberFormat",
       "sap/base/strings/formatMessage"
], function(jQuery, Controller, JSONModel, NumberFormat) {
       "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
03.controller.Startpage", {
     onInit: function() {
                var sDataPath = sap.ui.require.toUrl("sap/suite/ui/commons/
demokit/tutorial/icecream/03/model/data") + "/News.json";
                var oModel = new JSONModel(sDataPath);
                this.getView().setModel(oModel, "news");
                }
                ...
    });
});
```

You instantiate the models in the *onInit* hook function. They are then available when needed. If you choose a different resource root in your *index.html* file, keep in mind that you have to adapt the module path to the **.json* files whenever you load the file into a controller.

GenericTile with ImageContent

With the *ImageContent* control, you can create the second tile that contains only one image. With that tile, you can also navigate to another view (we will explain this implementation later).

Startpage.view.xml

```
<mvc:View
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
03.controller.Startpage"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    xmlns:layout="sap.ui.layout"
    xmlns:microchart="sap.suite.ui.microchart">
    <Page title="{i18n>title}">
        <layout:VerticalLayout class="sapUiResponsiveMargin">
            <Title
                titleStyle="H2"
                text="{i18n>startpageTilesGroupTitle}"
                class="sapUiTinyMarginBegin" />
            <layout:HorizontalLayout allowWrapping="true">
                ...
            <GenericTile
                    class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                    header="{i18n>startpageTestResultsTileTitle}"
                    subheader="{i18n>startpageTestResultsTileSubTitle}">
```

```
                    <tileContent>
                        <TileContent>
                            <content>
                                <ImageContent src="sap-icon://temperature" />
                            </content>
                        </TileContent>
                    </tileContent>
                </GenericTile>

            </layout:HorizontalLayout>
        </layout:VerticalLayout>
    </Page>
</mvc:View>
```

Here, the `TileContent` has only one *ImageContent* with an icon set in the *src* property. If you like, you can choose another icon or image.

SlideTile with Two GenericTiles

With the `SlideTile`, we want to display news about the ice cream machine as well as the user reviews.

Startpage.view.xml

```
<mvc:View
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
03.controller.Startpage"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    xmlns:layout="sap.ui.layout"
    xmlns:microchart="sap.suite.ui.microchart">
    <Page title="{i18n>title}">
        <layout:VerticalLayout class="sapUiResponsiveMargin">
            <Title
                titleStyle="H2"
                text="{i18n>startpageTilesGroupTitle}"
                class="sapUiTinyMarginBegin" />
            <layout:HorizontalLayout allowWrapping="true">
                ..
         <SlideTile
                    class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                    tiles="{news>/News}">
                    <GenericTile
                        backgroundImage="{news>image}"
                        state="{news>state}"
                        frameType="TwoByOne">
                        <tileContent>
                            <TileContent
                                footer="{
                                    path: 'news>date',
                                    formatter: '.formatJSONDate'
                                }">
                                <content>
                                    <NewsContent
                                        contentText="{news>content}"
                                        subheader="{news>subheader}">
                                    </NewsContent>
                                </content>
                            </TileContent>
                        </tileContent>
                    </GenericTile>
                </SlideTile>
            </layout:HorizontalLayout>
```

```
        </layout:VerticalLayout>
    </Page>
</mvc:View>
```

First of all, the `SlideTile` has a wider frame type than the standard `GenericTile`. The default value for the *frameType* property of `GenericTile` is **OneByOne**. This means that it has the standard width and height. The standard frame type of the `SlideTile` is **TwoByOne** which means that the tile is twice as wide as the default one. The data of the **news** model is bound via the **tiles** aggregation and uses the provided `GenericTile` template. All properties in the `GenericTile` are bound to the **news** model. The image you've chosen for the *backgroundImage* property should already have the proper size. As you have all the images you need as part of this tutorial, you only need to refer to the respective image in the model.

With the `NewsContent`, we can put text on top of the background image in the correct layout. In the *NewsContent*, you set the title with the *contentText* property. You set the subtitle using the *subheader* property.

The *date* field in the *news* model is not in a user-friendly format. This is why we are adding a new formatter that converts the date value.

Startpage.controller.js

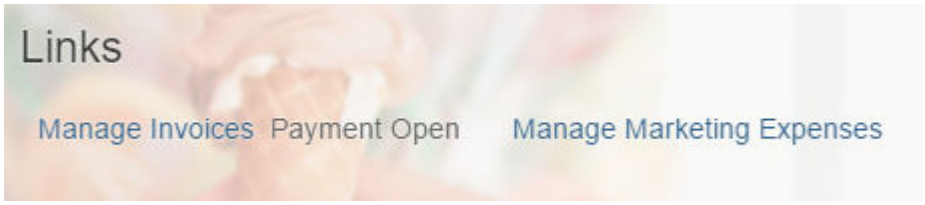
```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/format/NumberFormat",
    "sap/base/strings/formatMessage"
], function(Controller, JSONModel, NumberFormat) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
03.controller.Startpage", {
        ...
formatJSONDate: function(date) {
            var oDate = new Date(Date.parse(date));
            return oDate.toLocaleDateString();
        }

        ...
    });
});
```

In the controller, we need to add the implementation of the *formatJSONDate* formatter function. Based on the language settings of your browser or local machine, the JSON date string is converted to a string that corresponds to the time and date settings.

Step 4: Generic Tiles in Line Mode

In this step, we will create two `GenericTiles` in line mode.

Preview



Two GenericTiles in LineMode

There are tiles that you don't use as often as other tiles. To save space, you can reduce the tiles and only show the header and subheader. Use `LineMode` as the *mode* property of the `GenericTile`.

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 4 - Generic Tiles in Line Mode.

Startpage.view.xml

```
<mvc:View
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
04.controller.Startpage"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m"
    xmlns:layout="sap.ui.layout"
    xmlns:microchart="sap.suite.ui.microchart">
    <Page title="{i18n>title}">
        <layout:VerticalLayout class="sapUiResponsiveMargin">
            ...
            <Title
                titleStyle="H2"
                text="{i18n>startpageLinksGroupTitle}"
                class="sapUiTinyMarginBegin sapUiMediumMarginTop" />
            <layout:HorizontalLayout allowWrapping="true">
                <GenericTile
                    header="{i18n>startpageLineTile1Title}"
                    subheader="{i18n>startpageLineTile1SubTitle}"
                    mode="LineMode"
                    class="sapUiTinyMarginBegin" />
                <GenericTile
                    header="{i18n>startpageLineTile2Title}"
                    mode="LineMode"
                    class="sapUiTinyMarginBegin" />
            </layout:HorizontalLayout>
        </layout:VerticalLayout>
    </Page>
</mvc:View>
```

We need to create a new layout container for the tiles that we want to show. You do this to separate the two tile types and their alignment. These two tiles can be created in a similar way as the`GenericTiles`, except that the *mode* property must be set to `LineMode`.

Step 5: Navigating from the Start Page to Other Pages

In this step, we will show how to set up navigation using the standard routing pattern.

First, please create three empty views and the corresponding controllers:

- `ProcessFlow.view.xml`
- `ChartContainer.view.xml`
- `Reviews.view.xml`

Startpage.view.xml

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 5 - Navigating from the Start Page to Other Pages.

```
<mvc:View
    ...
    <Page title="{i18n>title}">
        <layout:VerticalLayout class="sapUiResponsiveMargin">
            ...
            <layout:HorizontalLayout allowWrapping="true">
                <GenericTile
                    class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                    header="{i18n>startpagePFTileTitle}"
                    subheader="{i18n>startpagePFTileSubTitle}"
                    press=".onNavToProcessFlow">
                    <tileContent>
                        ...
                    </tileContent>
                </GenericTile>
                    ...
                <GenericTile
                    class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                    header="{i18n>startpageUserReviewsTileTitle}"
                    press=".onNavToReviews">
                    <tileContent>
                        ...
                    </tileContent>
                </GenericTile>
                <GenericTile
                    class="sapUiTinyMarginBegin sapUiTinyMarginTop"
                    header="{i18n>startpageTestResultsTileTitle}"
                    subheader="{i18n>startpageTestResultsTileSubTitle}"
                    press=".onNavToChartContainer">
                    <tileContent>
                        ...
                    </tileContent>
                </GenericTile>
                ...
            </layout:HorizontalLayout>
            ...
        </layout:VerticalLayout>
    </Page>
</mvc:View>
```

We will add the *press* event to the `GenericTiles` with the *Production Process*, *User Reviews*, and *Quality Control* titles. This is to trigger navigation with a function. The press event function will be implemented in the controller file.

Startpage.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/core/format/NumberFormat",
    "sap/base/strings/formatMessage"
], function(Controller, JSONModel, NumberFormat) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
05.controller.Startpage", {
        ...
        onNavToProcessFlow: function() {
            this.getRouter().navTo("processFlow");
        },

        onNavToChartContainer: function() {
            this.getRouter().navTo("chartContainer");
        },
        onNavToReviews: function() {
            this.getRouter().navTo("reviews");
        },

        getRouter: function() {
            return this.getOwnerComponent().getRouter();
        }
});
```

`getRouter` returns the router instance of the component.

`onNavToProcessFlow` is called when the user clicks on the first tile. It triggers the navigation to the view with the `ProcessFlow`.

`onNavToChartContainer` is called when the user clicks on the fourth tile. It triggers the navigation to the view with the `ChartContainer`.

`onNavToReviews` is called when the user clicks the third tile. It triggers the navigation to the view with the `Timeline`.

Step 6: Chart Container

In this step, we will use the *ChartContainer* control to display information in a detailed view.

In the previous steps, we created a start page with embedded tiles. We prepared the routing configuration for navigation from the *Quality Control* tile to another view.

In the *Quality Control* view, we want to display the temperatures needed for different flavors and the ideal temperature for getting the best ice cream with a smooth consistency. We will use a chart to visualize the data and a table to get a detailed view with the help of the `ChartContainer` control.

Preview

Chart Content

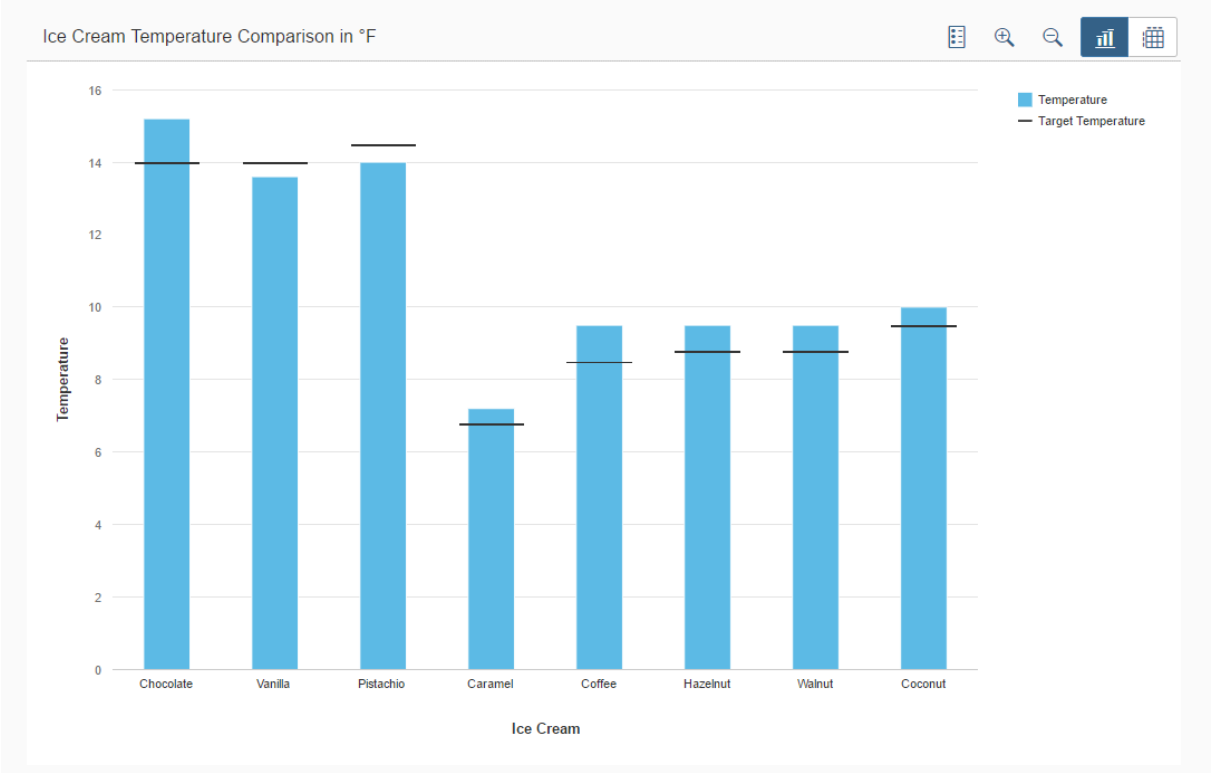


Table Content

Ice Cream Temperature Comparison in °F

Ice Cream ID	Temperature	Target Temperature
Chocolate	15.2	14
Vanilla	13.6	14
Pistachio	14	14.5
Caramel	7.2	6.8
Coffee	9.5	8.5
Hazelnut	9.5	8.8
Walnut	9.5	8.8
Coconut	10	9.5

ChartContainer

The `ChartContainer` control manages various data views in one container. The single controls (for example, *VizFrame* and *Tables*) are embedded in the `ChartContainerContent` controls that were added to the *content* aggregation of the `ChartContainer`.

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 6 - Chart Container.

ChartContainer.view.xml

```
mvc:View
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:suite="sap.suite.ui.commons"
    xmlns:core="sap.ui.core"
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
06.controller.ChartContainer">
    <Page
        title="{i18n>chartContainerPageTitle}"
        showNavButton="true"
        navButtonPress=".onNavButtonPressed"
        backgroundDesign="Solid">
        <content>
            <l:Grid defaultSpan="L12 M12 S12" class="sapUiResponsiveMargin"
vSpacing="0" hSpacing="0">
                <suite:ChartContainer
                        title="{
                            parts: [
                                'i18n>chartContainerBulletChartTitle',
                                'business>/temperatureUnit'
                            ],
                            formatter: '.formatMessage'
                        }">
                    <suite:ChartContainerContent
                        icon="sap-icon://vertical-bullet-chart"
                        title="{i18n>chartContainerBulletChartContentTitle}">
                        <suite:content>
                            <core:Fragment

fragmentName="sap.suite.ui.commons.demokit.tutorial.icecream.
06.fragment.VizChart"
                                    type="JS" />
                        </suite:content>
                    </suite:ChartContainerContent>
                </suite:ChartContainer>
            </l:Grid>
        </content>
    </Page>
</mvc:View>
```

In the toolbar, there is a corresponding button for each `ChartContainerContent` from which you can select the content. You can have an icon on the button if you set the *icon* property of the `ChartContainerContent` to the respective SAPUI5 icon URI. The *title* property of the `ChartContainerContent` determines the tooltip text for the button.

Since `VizFrame` is not part of this tutorial, we have provided the `VizChart.fragment.js` JavaScript file in the folder fragment that can be used without modifications.

ChartContainer.controller.js

```
sap.ui.define([
     "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
06.controller.ChartContainer", {
onNavButtonPressed: function() {
            var oRouter = this.getOwnerComponent().getRouter();
            oRouter.navTo("home");
        }
    });
});
```

In the `ChartContainer.view.xml`, we will add the backward navigation with the `showNavButton` and `navButtonPress` properties. The function that is referenced by the `navButtonPress` event retrieves the router of the component and navigates to the **home** route as specified in `manifest.json`.

Table with details

In the next step, we are going to display detailed information about the data shown in the chart. We will be using a table in which each value is displayed, so that no information is lost.

ChartContainer.view.xml

```
mvc:View
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:suite="sap.suite.ui.commons"
    xmlns:core="sap.ui.core"
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
06.controller.ChartContainer">
    <Page
        title="{i18n>chartContainerPageTitle}"
        showNavButton="true"
        navButtonPress=".onNavButtonPressed"
        backgroundDesign="Solid">
        <content>
            <l:Grid defaultSpan="L12 M12 S12" class="sapUiResponsiveMargin"
vSpacing="0" hSpacing="0">
                <suite:ChartContainer
                        title="{
                            parts: [
                                'i18n>chartContainerBulletChartTitle',
                                'business>/temperatureUnit'
                            ],
                            formatter: '.formatMessage'
                        }">
                    ...
                <suite:ChartContainerContent
                     icon="sap-icon://table-chart"
                     title="{i18n>chartContainerTableContentTitle}">
                     <suite:content>
                         <Table items="{/Temperatures}">
                             <columns>
                                 <Column>
                                     <Text
text="{i18n>chartContainerIceCreamId}" />
                                 </Column>
                                 <Column>
                                     <Text
```

```
text="{i18n>chartContainerTemperature}" />
                                    </Column>
                                    <Column>
                                        <Text
text="{i18n>chartContainerTargetTemperature}" />
                                    </Column>
                                </columns>
                                <items>
                                    <ColumnListItem>
                                        <cells>
                                            <Text text="{id}" />
                                            <Text text="{temperature}" />
                                            <Text text="{target}" />
                                        </cells>
                                    </ColumnListItem>
                                </items>
                            </Table>
                        </suite:content>
                    </suite:ChartContainerContent>
                </suite:ChartContainer>
            </l:Grid>
        </content>
    </Page>
</mvc:View>
```

We are going to create another `ChartContainerContent` with a **table**. This table contains three columns that were created with the **Column** element with a **text** that will be shown in the header. The values were created with the `ColumnListItem` that has a *text* control with a bound *text* property for each cell.

Step 7: Header Container and Radial Micro Chart

In this step, we use the `HeaderContainer` and `RadialMicroChart`.

In the last steps, we have implemented the user interaction handling to allow navigation from the **Production Process** tile to the `ProcessFlow` view. With the **Production Process** tile, we only have an aggregated view of the progress of the production process. Therefore, we want to provide a detailed overview of the current production status in this step. We will add the production flow, the states of the manufacturing steps, their dependencies, and critical steps if necessary.

First, we add the `ProcessFlow` control to this view, so that we see the current status of the production process. The production process consists of six steps represented by the following `ProcessFlowLaneHeaders`:

- Order
- Manufacturing
- Assembly
- Marketing
- Delivery
- Payment

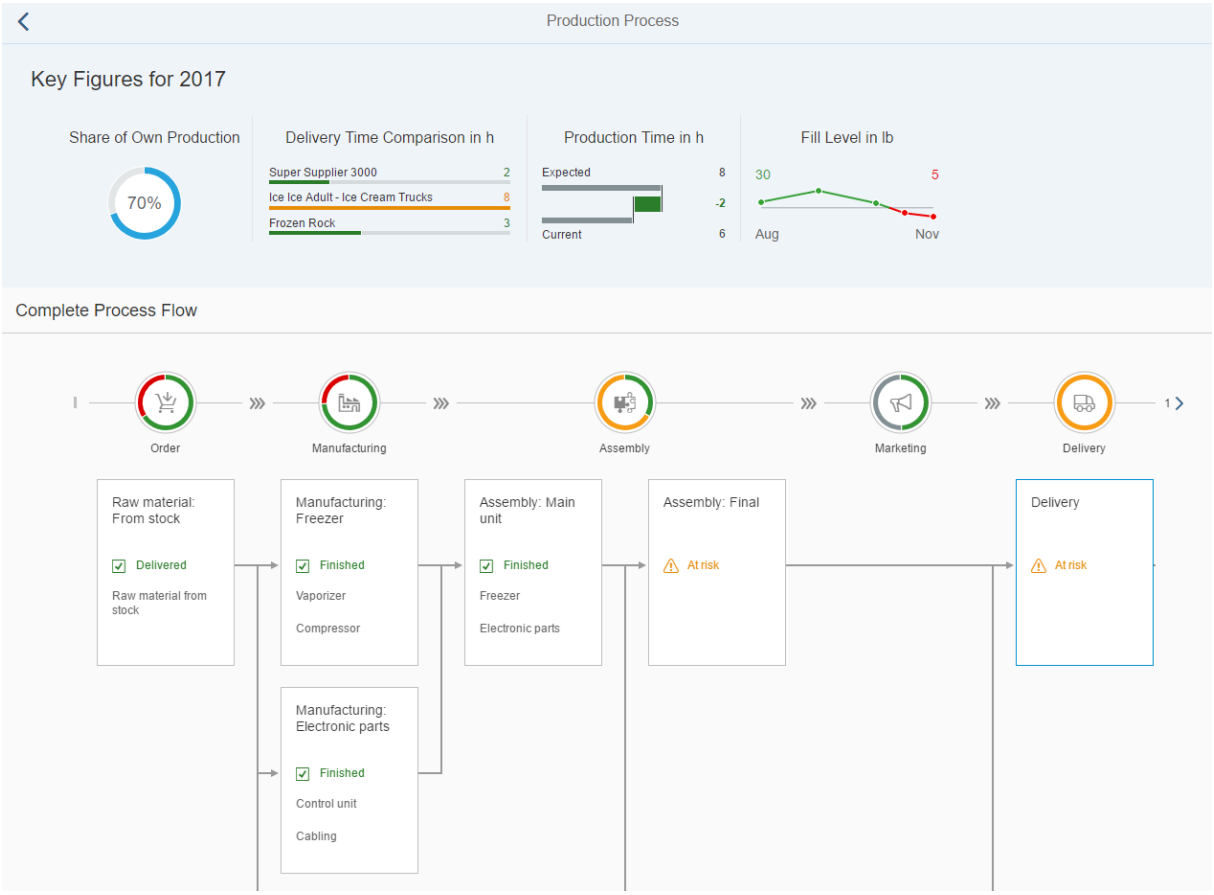
The `ProcessFlowLaneHeaders` use icons that stand for the meaning of the lanes. The lane headers aggregate the status of the nodes that are assigned to a particular lane. Each lane consists of `ProcessFlowNodes` that describe an activity in this production step. The structure of the nodes and their relationships for `ProcessFlow` are defined in the *ProcessFlowData.json* file.

The header area of the `ProcessFlow` should contain a `HeaderContainer` with four micro charts that provide the following information:

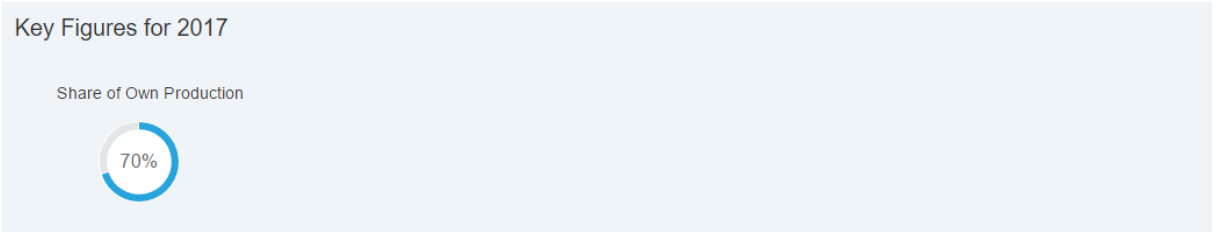
- Share of Own Production (your own production rate)
- Delivery Time Comparison in h (comparison of suppliers' delivery time)
- Production Time in h (comparison of expected and current production times)
- Fill Level in lb (displays the fill level changes over time)

Preview of the final ProcessFlow

This is the final view of this page:



Preview of this step



HeaderContainer with RadialMicroChart

In the header area, we can display additional values for the business logic of the production process. We can use charts to visualize the information. To distinguish this part from the ProcessFlow, we use the *ObjectHeader* in the header area. We want to add further charts in the following steps and therefore, we create an instance of the *HeaderContainer* that contains our charts.

The first chart contains the share of the parts being produced by us that should be included in the final product. This value is already available in the data model. You can visualize the percentage value by using a `RadialMicroChart`. We first create a `HeaderContainer` with a single `RadialMicroChart`.

ProcessFlow.view.xml

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 7 - Header Container and Radial Micro Chart.

```
<mvc:View
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.suite.ui.commons"
    xmlns:m="sap.m"
    xmlns:mc="sap.suite.ui.microchart"
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
07.controller.ProcessFlow">
    <m:Page
        title="{i18n>processFlowTitle}"
        showNavButton="true"
        navButtonPress=".onNavButtonPressed"
        backgroundDesign="Solid">
        <m:content>
            <m:ObjectHeader
                responsive="true"
                title="{
                    parts: [
                        'i18n>processFlowChartsTitle',
                        'business>/year'
                    ],
                    formatter: '.formatMessage'
                }">
                <m:headerContainer>
                    <m:HeaderContainer
                        scrollStep="200"
                        scrollTime="500"
                        showDividers="true"
                        class="sapUiSmallMargin">
                        <m:FlexBox
                            width="12rem"
                            height="10rem"
                            alignItems="Center"
                            justifyContent="Center"
                            direction="Column">
                            <m:Title
                                text="{i18n>processFlowChartsShareOwnProduction}"
                                class="sapUiSmallMargin"/>
                            <m:FlexBox width="6rem" height="6rem">
                                <mc:RadialMicroChart percentage="{business>/
shareOwnProduction}"/>
                            </m:FlexBox>
                        </m:FlexBox>
                    </m:HeaderContainer>
                </m:headerContainer>
            </m:ObjectHeader>
        </m:content>
    </m:Page>
```

```
</mvc:View>
```

The `HeaderContainer` has *scrollStep* and *scrollTime* properties that will be used to change the scroll step and the animation speed of the `HeaderContainer`. We keep the *showDividers* default value as we want to split each chart in the `HeaderContainer`.

The `FlexBox` is used as a container for the *Title* and the inner *FlexBox* for the `RadialMicroChart` and this leads to the correct sizing. To visualize the share of your own production, the value should be set in the *percentage* property of the `RadialMicroChart`. With the *Size* enumeration, you can use the chart in one of the fixed sizes. If not set, the default *Responsive* size is used and the width and height will adapt to the surrounding container. We set the *width* and *height* of the *FlexBox* instance.

ProcessFlow.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/base/strings/formatMessage"
], function (Controller, formatMessage) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
07.controller.ProcessFlow", {
        formatMessage: formatMessage,
        onNavButtonPressed: function () {
            this.getOwnerComponent().getRouter().navTo("home");
        }
    });
});
```

When the user clicks on the **Back** button, *onNavButtonPressed* is called and this triggers the navigation back to the start page view.

Step 8: Comparison Micro Chart

In this step, we will create the `ComparisonMicroChart` on the **Production Process** page.

Preview

ComparisonMicroChart

We would like to include information about our suppliers' delivery times in order to compare them. We are going to use the `ComparisonMicroChart` to visualize the comparison of several values.

ProcessFlow.view.xml

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 8 - Comparison Micro Chart.

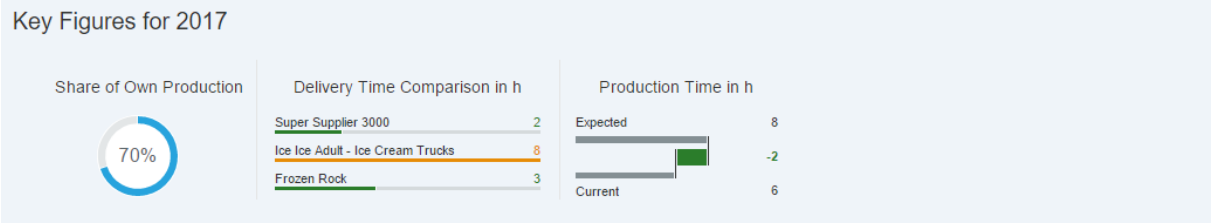
```
mvc:View
    ...
        <m:headerContainer>
            <m:HeaderContainer
                scrollStep="200"
                scrollTime="500"
                showDividers="true"
                class="sapUiSmallMargin">
                ...
                <m:FlexBox
                    width="16rem"
                    height="10rem"
                    alignItems="Center"
                    justifyContent="Center"
                    direction="Column">
                    <m:Title
                        text="{
                            parts: [
                                'i18n>processFlowChartsDeliveryTimeComparison',
                                'suppliers>/timeMeasure'
                            ],
                            formatter: '.formatMessage'
                        }"
                        class="sapUiSmallMargin" />
                    <m:FlexBox width="16rem" height="6rem" renderType="Bare">
                        <mc:ComparisonMicroChart
                            size="Responsive"
                            press=".press"
                            data="{suppliers>/Suppliers}">
                            <mc:data>
                                <mc:ComparisonMicroChartData
                                    title="{suppliers>id}"
                                    value="{suppliers>deliveryTime}"
                                    displayValue="{suppliers>deliveryTime}"
                                    color="{suppliers>deliveryTimeSemantics}" />
                            </mc:data>
                        </mc:ComparisonMicroChart>
                    </m:FlexBox>
                </m:FlexBox>
            </m:HeaderContainer>
        </m:headerContainer>
</mvc:View>
```

The structure that surrounds the `ComparisonMicroChart` is similar to the structure we used for the `RadialMicroChart`. The chart dimensions are inherited from *FlexBox* because the *isResponsible* property is being used. The internal structure is different because the `ComparisonMicroChart` contains an aggregation of the `ComparisonMicroChartData` items. Each item is responsible for a particular line in the chart.

Step 9: Delta Micro Chart

In this step, we will create the `DeltaMicroChart` on the **Production Process** page.

Preview



DeltaMicroChart

We are going to add a chart to visualize the difference between the time required for production compared with the estimated time. You use the `DeltaMicroChart` to compare two separate values.

ProcessFlow.view.xml

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 9 - Delta Micro Chart.

```
mvc:View
    ...
        <m:headerContainer>
            <m:HeaderContainer
                scrollStep="200"
                scrollTime="500"
                showDividers="true"
                class="sapUiSmallMargin">
                ...
                <m:FlexBox
                    width="12rem"
                    height="10rem"
                    alignItems="Center"
                    justifyContent="Center"
                    direction="Column">
                    <m:Title
                        text="{
                            parts: [
                                'i18n>processFlowChartsProductionTime',
                                'business>/timeMeasure'
                            ],
                            formatter: '.formatMessage'
                        }"
                        class="sapUiSmallMargin" />
                    <m:FlexBox width="12rem" height="6rem" renderType="Bare">
                        <mc:DeltaMicroChart
                            size="Responsive"
                            color="{business>/
productionTimeComparisonCriticality}"
                            value1="{business>/expectedProductionTime}"
```

```
                            value2="{business>/currentProductionTime}"
                            title1="{i18n>processFlowChartsExpected}"
                            title2="{i18n>processFlowChartsCurrent}"
                            displayValue1="{business>/expectedProductionTime}"
                            displayValue2="{business>/currentProductionTime}"
                            deltaDisplayValue="{
                                parts: [
                                    'business>/expectedProductionTime',
                                    'business>/currentProductionTime'
                                ],
                                formatter: '.getValuesDelta'
                        }" />
                    </m:FlexBox>
                </m:FlexBox>
            </m:HeaderContainer>
        </m:headerContainer>
    ...
</mvc:View>
```

The `DeltaMicroChart` is used in the responsive mode embedded in the *Flexbox* that sets the dimensions. We are simply using the respective values from the data model for binding the *displayValue1* and *displayValue2* properties. For the *deltaDisplayValue*, we need to perform an advanced calculation based on the values before we can set them as a property. This is why we will be using the *.getValuesDelta* formatter function.

ProcessFlow.controller.js

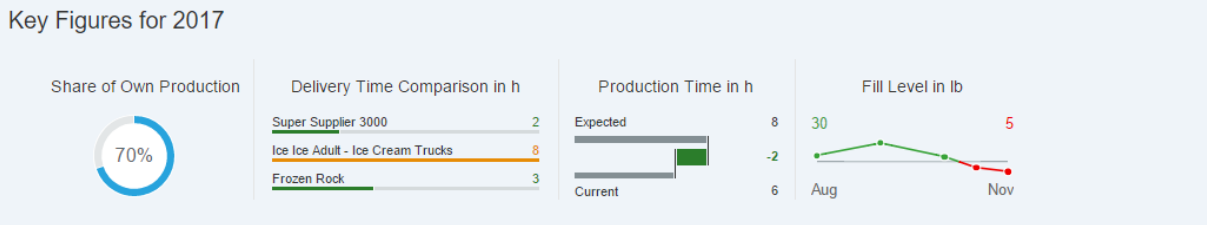
```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/base/strings/formatMessage"
], function(Controller) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
09.controller.ProcessFlow", {
        ...
        getValuesDelta: function(fFirstValue, fSecondValue) {
            return fSecondValue - fFirstValue;
        }
    });
});
```

This formatter takes the respective values and returns a string that will then be set as a property of the `DeltaMicroChart`.

Step 10: Line Micro Chart

In this step, we will create the `LineMicroChart` on the **Production Process** page.

Preview



LineMicroChart

With the `LineMicroChart`, we want to visualize how the fill level is going to change during the course of the production process and we want to determine the threshold values that fall below a specified level.

ProcessFlow.view.xml

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 10 - Line Micro Chart.

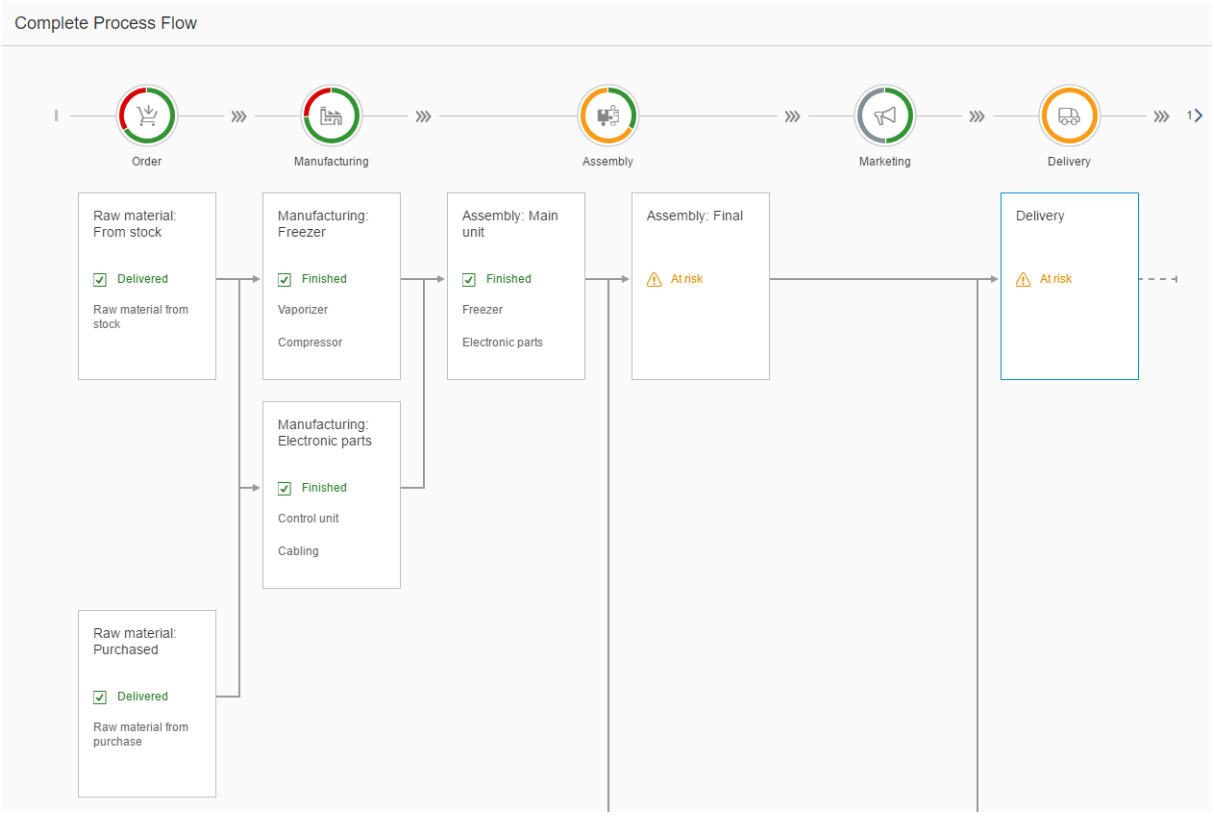
```
mvc:View
    ...
        <m:headerContainer>
            <m:HeaderContainer
                scrollStep="200"
                scrollTime="500"
                showDividers="true"
                class="sapUiSmallMargin">
                ...
                <m:FlexBox
                    width="12rem"
                    height="10rem"
                    alignItems="Center"
                    justifyContent="Center"
                    direction="Column">
                    <m:Title
                        text="{
                            parts: [
                                'i18n>processFlowChartsFillLevel',
                                'business>/fillLevel/measure'
                            ],
                            formatter: '.formatMessage'
                        }"
                        class="sapUiSmallMargin" />
                    <m:FlexBox width="12rem" height="6rem" renderType="Bare">
                        <mc:LineMicroChart
                            size="Responsive"
                            threshold="{business>/fillLevel/threshold}"
                            leftTopLabel="{business>/fillLevel/valueBegin}"
                            leftBottomLabel="{business>/fillLevel/timeBegin}"
                            rightTopLabel="{business>/fillLevel/valueEnd}"
                            rightBottomLabel="{business>/fillLevel/timeEnd}"
                            color="{business>/fillLevel/color}"
                            showPoints="true"
                            points="{
                                path: 'business>/fillLevel/timeSeries',
                                templateShareable: true
                            }">
                            <mc:LineMicroChartPoint
                                x="{business>time}"
                                y="{business>level}" />
                        </mc:LineMicroChart>
                    </m:FlexBox>
                </m:FlexBox>
            </m:HeaderContainer>
        </m:headerContainer>
    ...
</mvc:View>
```

You can find the information about the fill levels in the *points* aggregation of the `LineMicroChart` control. Property **x** of the `LineMicroChartPoint` contains the dimension of the chart and the time. Property **y** contains the fill level during the production process. The **threshold** property separates the records by their fill level into an upper and a lower layer. With the *color* property, we can apply a different color to each layer, depending on the specified threshold.

Step 11: Process Flow

In this step, we will create the `ProcessFlow` on the **Production Process** page.

Preview



ProcessFlow

The `ProcessFlow` in the center of the page will be used to visualize the current state of the production process. Each production step is represented by a lane. The colors of the state segments of `ProcessFlowHeader` are calculated internally according to a `ProcessFlowNode` of a lane. The following semantic colors are used:

- Grey for **Neutral**
- Green for **Good**
- Orange for **Critical**
- Red for **Error**.

The **Assembly** lane in the middle is a *merged* lane that consists of nodes that are bound together by a parent-child relation. As a result, the usual width will be doubled.

The `ProcessFlowNode` elements **Spare parts: Purchased** and **Delivery** are highlighted (refer to the *focused* property on the API). They are of high interest because their states are critical.

ProcessFlow.view.xml

You can view and download this step in the Demo Kit at Process Flow.

```
<mvc:View
    ...
        <m:ObjectHeader
            ...
            <m:headerContainer>
                <m:HeaderContainer
                    ...
                </m:HeaderContainer>
            </m:headerContainer>
        </m:ObjectHeader>
        <m:Panel headerText="{i18n>processFlowProcessTitle}">
            <ProcessFlow
                id="processflow1"
                scrollable="false"
                nodes="{process>/Nodes}"
                lanes="{process>/Lanes}"
                nodePress=".onNodePressed">
                <nodes>
                    <ProcessFlowNode
                        laneId="{process>lane}"
                        nodeId="{process>id}"
                        title="{process>title}"
                        titleAbbreviation="{process>titleAbbreviation}"
                        isTitleClickable="{process>isTitleClickable}"
                        children="{process>children}"
                        state="{process>state}"
                        stateText="{process>stateText}"
                        texts="{process>texts}"
                        highlighted="{process>highlighted}"
                        focused="{process>focused}" />
                </nodes>
                <lanes>
                    <ProcessFlowLaneHeader
                        laneId="{process>id}"
                        iconSrc="{process>icon}"
                        text="{process>label}"
                        position="{process>position}" />
                </lanes>
            </ProcessFlow>
        </m:Panel>
    ...
</mvc:View>
```

First, we are going to add a `ProcessFlow` control to our view. By setting the *scrollable* property to **false**, the `ProcessFlow` will use the default browser scrolling.

Next, we will bind the *nodes* and *lanes* aggregations. Templates will be added to the corresponding aggregations (nodes and lanes).

The template for `ProcessFlowNodes` is created using the following properties:

- The *laneId* that is a unique identifier for the lane the node belongs to.
- The *nodeId* that is the current node identifier.
- The **child** property that defines the parent-child relation to other nodes using their *nodeIds*.
- The **state** property that determines the property of the node and influences the aggregated state of the lane directly.

The next template for the lanes will be created using a `ProcessFlowLaneHeader` with a *laneId* that we already know from the `ProcessFlowNode`. The *iconSrc* property of the `ProcessFlowLaneHeader` accepts any icon. The text below the state of the `ProcessFlowLaneHeader` is set with the *text* property of the lane header. The order of the lanes is defined by the *position* property of the lane header.

ProcessFlow.controller.js

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/base/strings/formatMessage"
], function(Controller, MessageToast) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
11.controller.ProcessFlow", {
        ...
        onNodePressed: function(oEvent) {
             var sItemTitle = oEvent.getParameters().getTitle();

MessageToast.show(this.getResourceBundle().getText("processFlowNodeClickedMessage
", [sItemTitle]));
        },

        getResourceBundle: function() {
            return this.getOwnerComponent().getModel("i18n").getResourceBundle();
        }
    });
});
```

The *onNodePressed* event listener is set on the *onNode* event of the `ProcessFlow`. It is called when the user clicks on the *node*. The title text of the *node* will be displayed in the message toast.

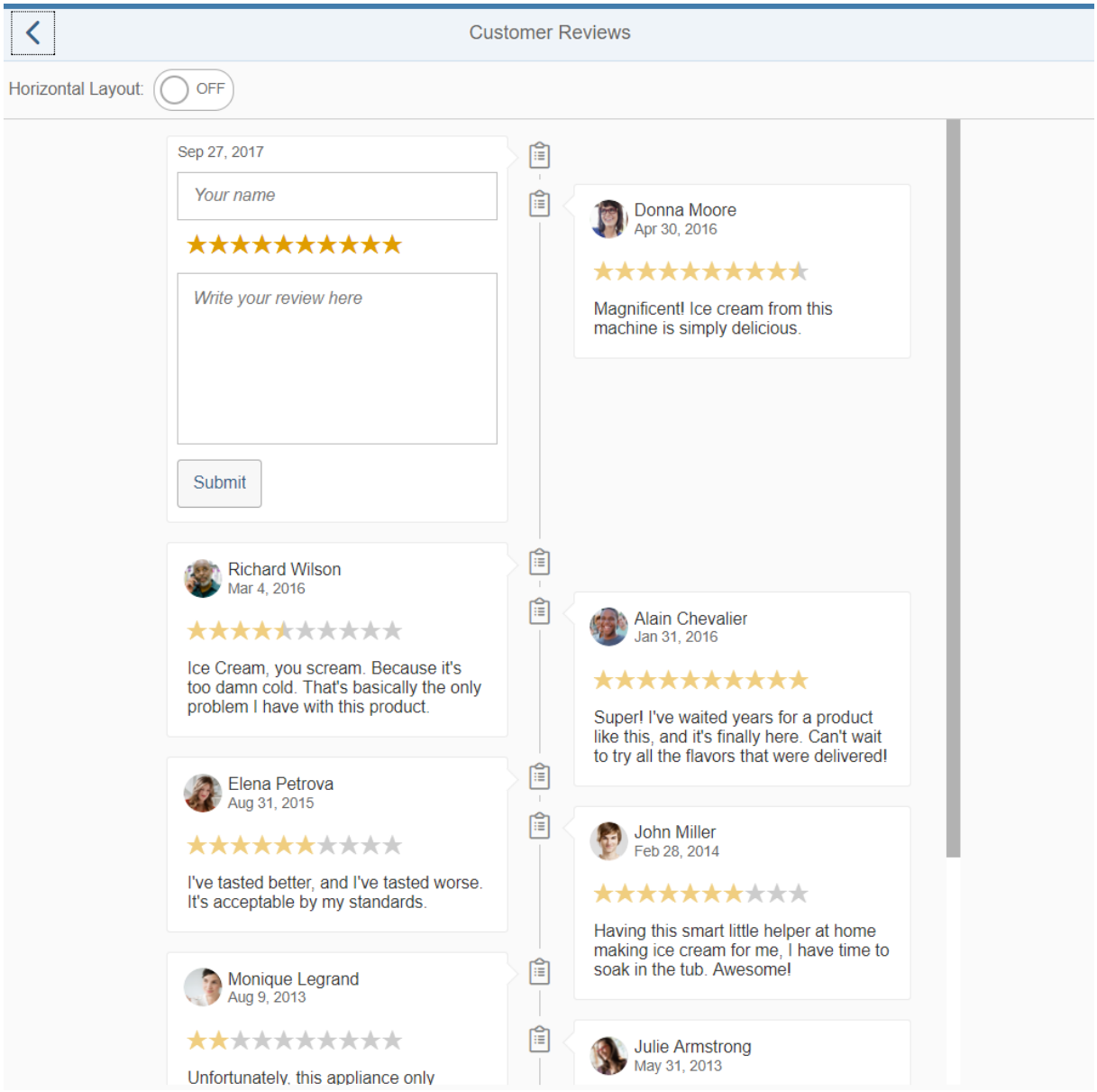
Step 12: Timeline

Use this step to learn how to set up a page where your customers can post their reviews of the ice cream machine.

In this step, we will use the `Timeline` control that allows the users to write and submit reviews. The reviews are arranged in chronological order along the timeline axis.

You can view and download this step in the Demo Kit at Ice Cream Machine - Step 12 - Timeline.

Preview



Setting Up a Reviews Page with a Timeline

To perform this step, you need one of the views you created in step 5, `Reviews.view.xml`, as well as the controller you defined for this view, `Reviews.controller.js`.

Reviews.view.xml

In the `Reviews.view.xml` view, add a `Timeline` element that will display a timeline with customer reviews.

You may also need to add a `page` and a `toolbar` if the view does not include them yet.

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:suite="sap.suite.ui.commons"
    controllerName="sap.suite.ui.commons.demokit.tutorial.icecream.
12.controller.Reviews">
    <Page
        title="{i18n>timelineTitle}"
        showNavButton="true"
        navButtonPress=".onNavButtonPressed"
        backgroundDesign="Solid">
        <Toolbar>
            <Label text="{i18n>layoutSwitchLabel}"/>
            <Switch change=".onHorizontalSwitchChange"/>
        </Toolbar>
        <suite:Timeline
            id="timeline"
            enableDoubleSided="true"
            growing="false"
            groupBy="dateTime"
            lazyLoading="true"
            content="{reviews>/UserReviews}"
            textHeight="automatic"
            showHeaderBar="false">
            <suite:TimelineItem
                dateTime="{
                    path: 'reviews>date',
                    formatter: '.formatDateTime'
                }"
                userPicture="{=!${reviews>template} ? ${reviews>userPic} : null}"
                title="{=!${reviews>template} ? ${reviews>user} : null}"
                text="{=!${reviews>template} ? ${reviews>quote} : null}"
                filterValue="{=!${reviews>template} ? ${reviews>rating} : null}">
                <suite:embeddedControl>
                    <VBox>
                        <Input
                            value="{reviews>user}"
                            visible="{reviews>template}"
                            placeholder="{i18n>newReviewUserNameHint}"/>
                        <RatingIndicator
                            enabled="{reviews>template}"
                            value="{reviews>rating}"
                            maxValue="10"
                            iconSize="1rem"
                            class="sapUiTinyMargin"/>
                        <Text
                            text="{reviews>quote}"
                            visible="{=!${reviews>template}}"
                            class="sapUiTinyMargin"/>
                        <TextArea
                            value="{reviews>quote}"
                            growing="false"
                            height="150px"
                            width="100%"
                            visible="{reviews>template}"
                            placeholder="{i18n>newReviewUserCommentHint}"
                            valueLiveUpdate="true"/>
                        <Button
                            visible="{reviews>template}"
                            text="{i18n>newReviewButtonText}"
                            press=".addReview"/>
                    </VBox>
                </suite:embeddedControl>
            </suite:TimelineItem>
        </suite:Timeline>
    </Page>
```

```
</mvc:View>
```

Reviews.controller.js

In the reviews controller, define a function for adding a review, `addReview`, along with the functions for adjusting date and time format, `formatDateTime`, dynamically updating the rating, `onRatingChange`, layout switching, `onHorizontalSwitchChange`, and navigating back to the start page, `onNavButtonPressed`.

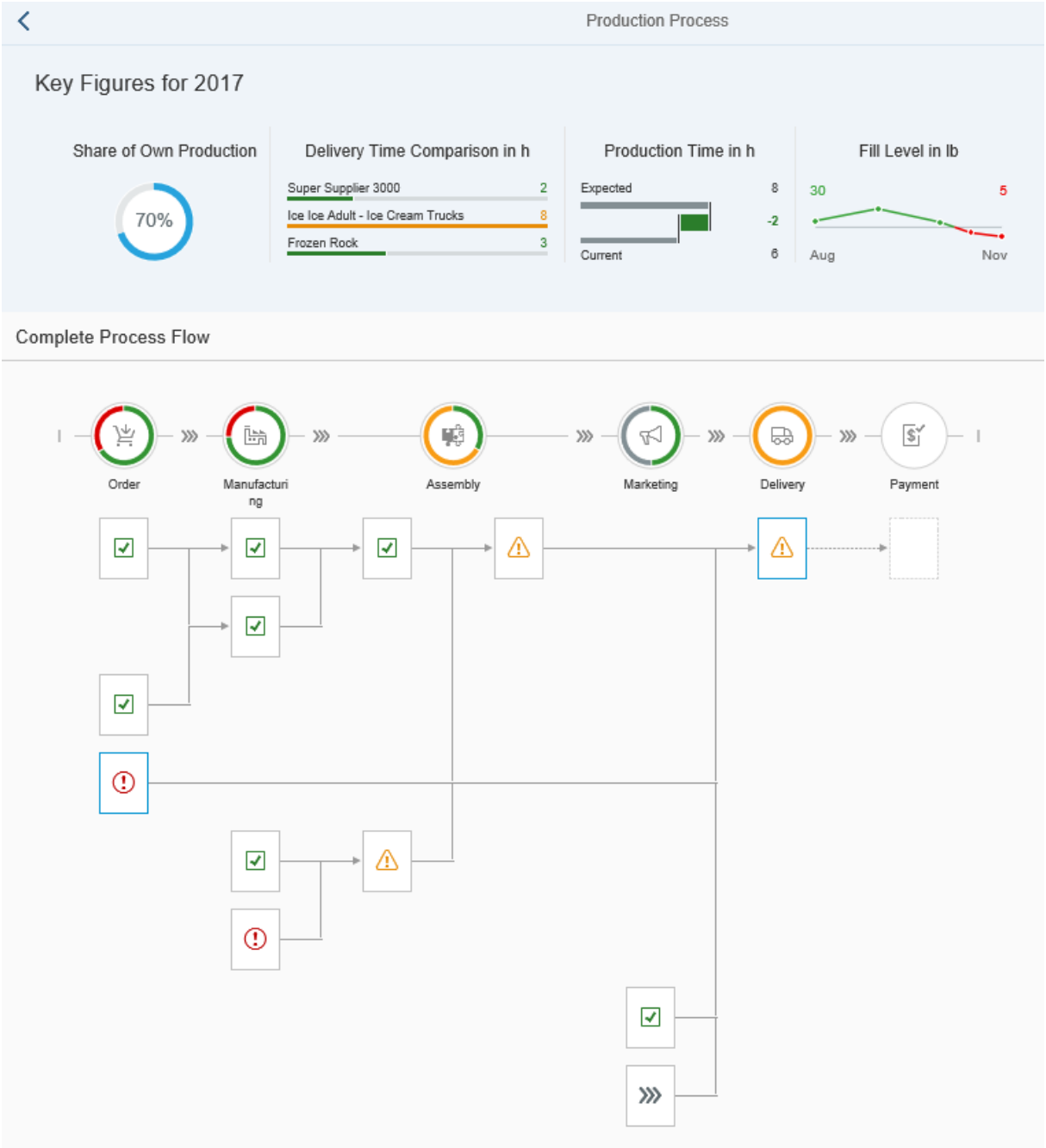
```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/format/DateFormat"
], function(Controller, DateFormat) {
    "use strict";
    return Controller.extend("sap.suite.ui.commons.demokit.tutorial.icecream.
12.controller.Reviews", {
        onInit: function() {
            this.oTimeline = this.byId("timeline");
        },
        addReview: function() {
            var oModel = this.getView().getModel("reviews"),
                oData = oModel.getData(),
                oTemplateEntry = oData.UserReviews[0];
            oTemplateEntry.template = false;
            // Add new template entry to the beginning
            oData.UserReviews.unshift({
                "user": "",
                "userPic": "",
                "rating": 10,
                "quote": "",
                "dateTime": "now",
                "template": true
            });
            oModel.setData(oData);
        },
        formatDateTime: function(dateTime) {
            var oDateInstance = DateFormat.getDateInstance();
            return oDateInstance.format(oDateInstance.parse(dateTime));
        },
        onNavButtonPressed: function() {
            var oRouter = this.getOwnerComponent().getRouter();
            oRouter.navTo("home");
        },
        onHorizontalSwitchChange: function(event) {
            if (event.getParameter("state")) {
                this.oTimeline.setAxisOrientation("Horizontal");
            } else {
                this.oTimeline.setAxisOrientation("Vertical");
            }
        }
    });
});
```

The initial reviews data is pulled from the model data in the */model/data/Reviews.json* file that is included in the download materials for this tutorial.

Step 13: Optimizing the Process Flow Layout

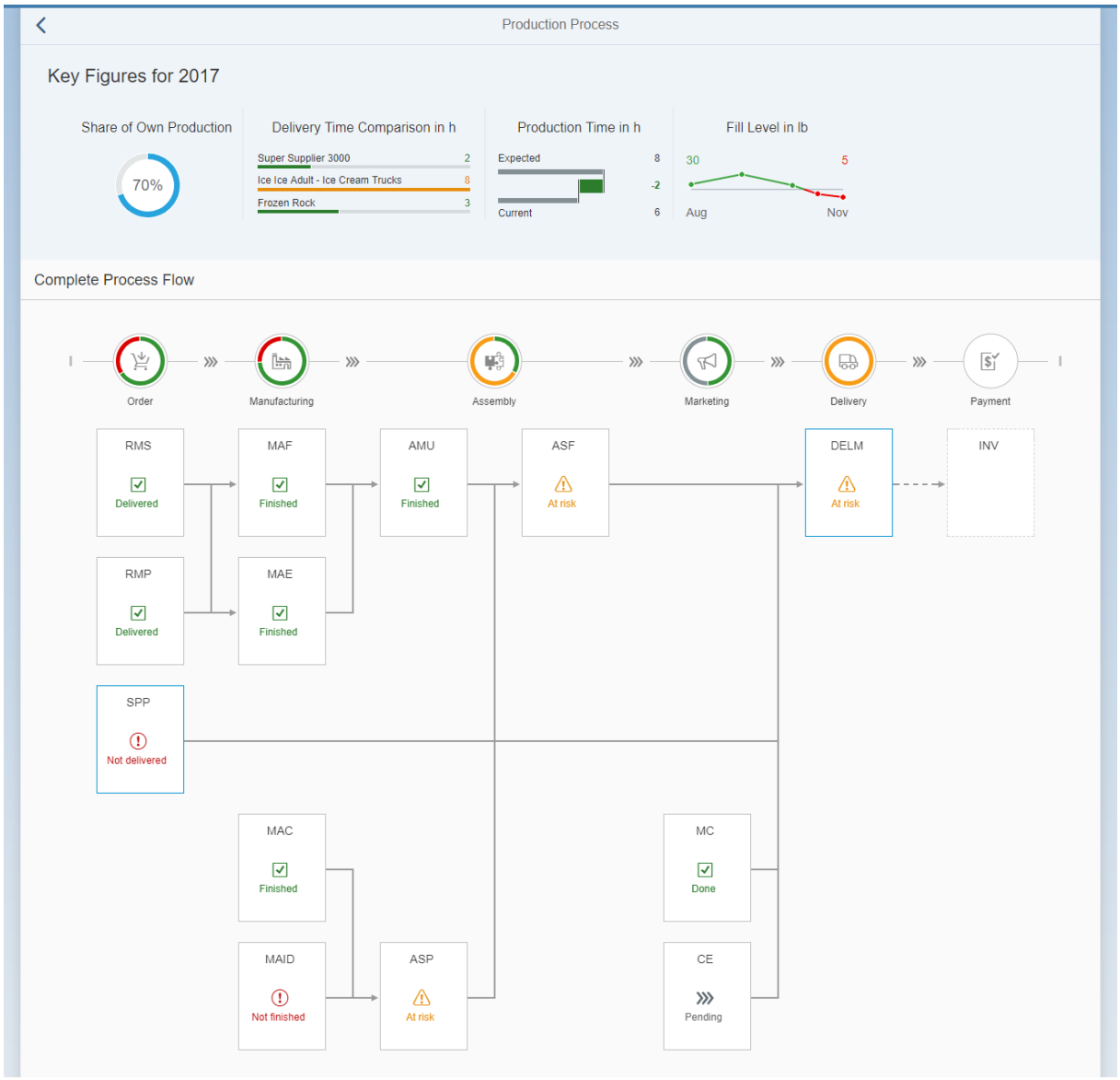
In the last step, we will optimize the `ProcessFlow` layout.

When you have completed the previous steps, your **Production Process** page will look like this:



You may have seen that some `ProcessFlow` nodes are not placed in a perfect way. For example, there is too much space for the *Marketing* node and available space is not used. To optimize the position of the nodes, the

`ProcessFlow` control provides the *optimizeLayout* API method. With this method, you can rearrange the nodes. Check it out and see the results:



Please keep in mind that this optimization has an influence on the app performance as rendering times will increase.

For more information, see the API Reference in the Demo Kit.

Demo Apps

With the Demo Kit, we deliver some demo apps that show you how you can use the various features and controls of SAPUI5.

You can open the apps directly at Demo Apps. You can also download the source files there to have a look at the code.

We have the following categories of demo apps:

- Showcase apps that show you how to use specific controls or features
- Apps that are created with our tutorials (see Get Started: Setup, Tutorials, and Demo Apps [page 38])
- Template apps (see App Templates: Kick Start Your App Development [page 1399])
- Other demo apps for Key User Adaptation, Application Path Framework (APF), and Charts

The following tables give an overview of what each demo app shows.

Features, Layouts, and Specific Topics

Demo App	Feature	Layouts	Specific Topic

Shopping Cart	XML View [page 787]
	Busy Indicators [page 2253]
	Device Adaptation [page 1433]
	Expression Binding [page 845]
	Input validation (see Validation Messages [page 1065])
		`sap.ui.layout.BlockLayout`
		`sap.ui.layout.VerticalLayout`
		`sap.ui.layout.form.SimpleForm`
			Filtering, custom filter
			Sorting
			Formatting
			Behavior-driven Development with Gherkin [page 1233]
			Mock Server [page 1222]
			Local storage

Browse Orders	XML View [page 787]
	Busy Indicators [page 2253]
	Device Adaptation [page 1433]
	Expression Binding [page 845]
		`sap.ui.layout.ResponsiveGridLayout`
		`sap.ui.layout.form.SimpleForm`
		`sap.f.FlexibleColumnLayout`
			Sorting
			Grouping
			Formatting
			Mock Server [page 1222]

Shop Administration Tool	XML View [page 787]
	Busy Indicators [page 2253]
	Device Adaptation [page 1433]
	Expression Binding [page 845]
	Custom Controls [page 2158]
		`sap.ui.layout.BlockLayout`
		`sap.ui.layout.ResponsiveGridLayout`
		`sap.uxap.ObjectPageLayout`
			Formatting

Employee Directory	XML View [page 787]	`sap.ui.layout.form.SimpleForm`	Mock Server [page 1222] Routing and Navigation [page 1072]
Hello World	JS View [page 803]		

Demo App	Feature	Layouts	Specific Topic
Bulletin Board	XML View [page 787] Busy Indicators [page 2253]	`sap.ui.layout.form.SimpleForm`	Sorting Formatting Mock Server [page 1222] Custom type
Manage Products	XML View [page 787] Busy Indicators [page 2253]		Sorting Formatting Mock Server [page 1222]
Worklist Template	XML View [page 787] Busy Indicators [page 2253]		Filtering Formatting Mock Server [page 1222] Sorting

Worklist (FLP) Template	XML View [page 787]
	JS View [page 803]
	Busy Indicators [page 2253]
		`sap.ushell.ui.shell.RightFloatingContainer`
		`sap.ushell.ui.shell.ShellAppTitle`
		`sap.ushell.ui.shell.ShellHeader`
		`sap.ushell.ui.shell.ShellHeadItem`
		`sap.ushell.ui.shell.ShellId`
		`sap.ushell.ui.shell.ShellLayout`
		`sap.ushell.ui.shell.SplitContainer`
			Filtering
			Formatting
			Mock Server [page 1222]
			Sorting
			Integration with SAP Fiori launchpad
			SAP Fiori launchpad sandbox

Master-Detail Template	XML Fragments [page 1005] XML View [page 787] Busy Indicators [page 2253] Device Adaptation [page 1433] Expression Binding [page 845]	`sap.f.FlexibleColumnLayout`	Formatting List selector Mock Server [page 1222] Sorting

Demo App	Feature	Layouts	Specific Topic

Master-Detail (FLP) Template	XML Fragments [page 1005]
	XML View [page 787]
	JS View [page 803]
	Busy Indicators [page 2253]
	Device Adaptation [page 1433]
	Expression Binding [page 845]
		`sap.ushell.ui.shell.ShellHeadItem`
		`sap.ushell.ui.shell.ShellAppTitle`
		`sap.ushell.ui.shell.ShellLayout`
			Formatting
			Mock Server [page 1222]
			Sorting
			Integration with SAP Fiori launchpad
			SAP Fiori launchpad sandbox

UI Adaptation at Runtime			SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]
UI Adaptation at Runtime for SAP Fiori Elements			SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]
APF Demo Application	XML View [page 787] JS View [page 803]	`sap.ui.layout.VerticalLayout`	Developing Apps with Analysis Path Framework (APF) [page 2040] Component Container
APF Configuration Modeler	XML View [page 787] JS View [page 803]	`sap.ushell.ui.shell.ShellHeader`	APF Configuration Modeler [page 2054] Component Container
Chart Demo App	XML View [page 787]		

Controls

Demo App	sap.m	sap.m.semantic	Other Libraries
Shopping Cart	Carousel		
	ColumnListItem		
	DatePicker		
	FormattedText		
	LightBox		
	List		
	MessagePage		
	MessagePopover		
	NavContainer		
	NotificationListItem		
	ObjectListItem		
	PullToRefresh		
	RangeSlider		
	SearchField		
	SegmentedButton		
	StandardListItem		
	Toolbar		
	Wizard		
Browse Orders	IconTabBar	DetailPage	
	List	GroupSelect	
	ObjectHeader	MasterPage	
	PullToRefresh	SendEmailAction	
	SearchField		
	SegmentedButton		
	SplitApp		
	Table		

Demo App	sap.m	sap.m.semantic	Other Libraries
Shop Administration Tool	App ColumnListItem List MessagePopover ResponsivePopover SearchField StandardListItem Table Toolbar		sap.tnt.NavigationListItem sap.tnt.ToolHeader sap.tnt.ToolPage Micro charts of sap.ui.comp.smartmicrochart
Employee Directory	App IconTabBar List Toolbar		
Hello World	App		
Bulletin Board	App ColumnListItem IconTabBar Toolbar	FullscreenPage SendEmailAction	
Manage Products	App Toolbar	FullscreenPage SendEmailAction	
Worklist Template	App ColumnListItem MessagePage SearchField Table Toolbar	SemanticPage SendEmailAction	

Demo App	sap.m	sap.m.semantic	Other Libraries
Worklist (FLP) Template	App ColumnListItem FormattedText HBox List MessagePage OverflowToolbar Page ScrollContainer SearchField Table Toolbar ToggleButton Vbox	FullscreenPage SendEmailAction	
Master-Detail Template	ColumnListItem OverflowToolbar List MessagePage ObjectHeader Page SearchField Table Toolbar ViewSettingsDialog	titleHeading SemanticPage FilterAction SendEmailAction	sap.f.FlexibleColumnLayout (2 columns)

Demo App	sap.m	sap.m.semantic	Other Libraries
Master-Detail (FLP) Template	App	DetailPage	
	Button	FilterAction	
	ColumnListItem	GroupSelect	
	FormattedText	MasterPage	
	IconTabBar	SendEmailAction	
	HBox	SortSelect	
	List		
	MessagePage		
	ObjectListItem		
	ObjectNumber		
	ObjectHeader		
	PullToRefresh		
	ScrollContainer		
	SearchField		
	SplitApp		
	Table		
	Toolbar		
	ToolSpacer		
	ViewSettingsDialog		

Demo App	sap.m	sap.m.semantic	Other Libraries
APF Demo Application	App Bar Button CheckBox FacetFilter FacetFilterItem FormattedText Label MessagePopover NavContainer OverflowToolbarButton Page ScrollContainer SearchField Title ToggleButton Toolbar ToolSpacer		

Demo App	sap.m	sap.m.semantic	Other Libraries
APF Configuration Modeler	App Bar Button Column ColumnListItem Input Label Page ScrollContainer Title Toolbar ToolSpacer Vbox		
Chart Demo App	List SplitApp Toolbar		sap.viz.ui5.4controls.Popover sap.viz.ui5.controls.VizFrame D3 charts (https://d3js.org)

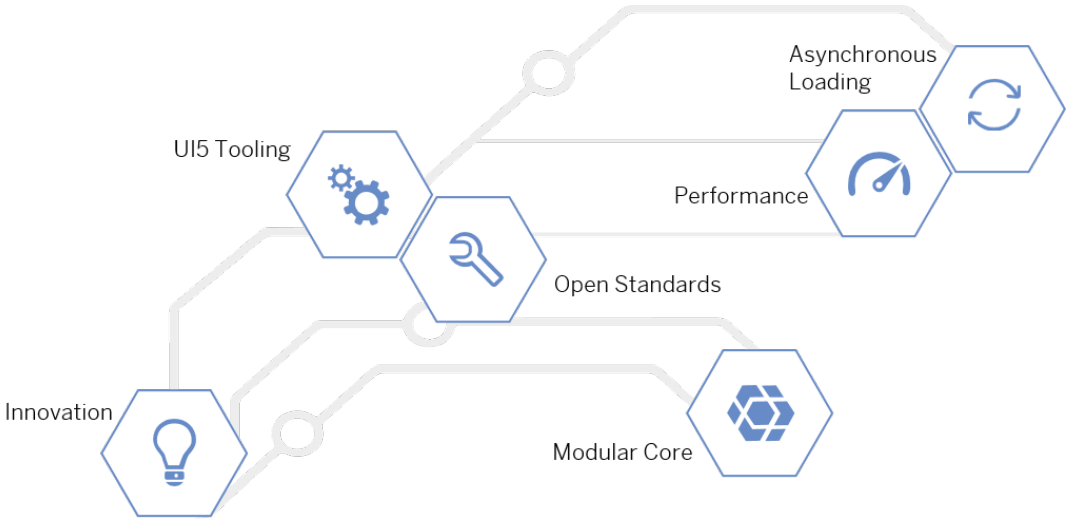
Best Practices for App Developers

In this section, we have compiled a set of best practice recommendations to help you develop high-quality SAPUI5 apps.

The best practices articles are built around the key concepts below. Where applicable, we show hands-on examples and refer you to our tutorials as well as to our the detailed documentation.

Key Concepts

Hover over the shapes to find out more about some key benefits of SAPUI5.



- Best Practices for App Developers [page 680]
- Best Practices for App Developers [page 680]
- Best Practices for App Developers [page 680]
- Best Practices for App Developers [page 680]
- Best Practices for App Developers [page 680]
- Best Practices for App Developers [page 680]

> → Tip
>
> All demo apps, templates, and tutorials in the SAPUI5 Demo Kit follow these recommendations.

Load Only What You Really Need

The amount of resources and data that your app loads will directly affect the performance of your app. You should declare all dependencies and remove unused libraries and classes from your code.

Keep Your Library Dependencies Up To Date

A library preload file, the library styles and text translations are loaded for every library you define in the application descriptor or the SAPUI5 bootstrap. Always define libraries in the manifest and remove all libraries that you do not intend to use in your code.

```
"sap.ui5": {
    "dependencies": {
        "minUI5Version": "1.60.0",
        "libs": {
            "sap.ui.core": {},
            "sap.m": {},
```

```
            "sap.ui.layout": {}
        }
    }
    ...
}
```

- Learn how: Walkthrough Tutorial Step 10: Descriptor for Applications [page 91]
- Find out more: Descriptor for Applications, Components, and Libraries [page 734]

Declare Local Dependencies

In the JavaScript files of your app, define all dependencies to SAPUI5 framework classes and app resources via `sap.ui.define`. If you have unused dependencies, you should remove them right away.

The UI5 Build and Development tooling can then create a "cleaned-up" version of your app that only contains the resources you really need. The so-called application preload will greatly speed up the initial load time of your app.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/m/MessageToast",
    "sap/base/Log"
], function (Controller, MessageToast, Log) {
    ...
```

- Learn how: Walkthrough Tutorial Step 10: Descriptor for Applications [page 91]
- Find out more:
  - Modules and Dependencies [page 1094]
  - https://help.sap.com/viewer/825270ffffe74d9f988a0f0066ad59f0/Cloud/en-US/dfb26ef028624cf486a8bbb0bfd459ff.html
  - https://github.com/SAP/ui5-tooling

Use Lazy Loading

Use controls like `sap.m.List` or UI patterns that support displaying data selectively or with pagination. Make sure that your backend service is designed to deliver small chunks of data as well.

```
<List
    growing="true"
    growingThreshold="20"
    ...>
```

- Learn how: Testing Tutorial Step 7: Changing the Table to a Growing Table [page 399]
- Find out more: Growing Feature for Table and List [page 2342]

Use the MVC Concept

MVC (Model-View-Controller) is a concept for structuring your software. It makes it easier to maintain and to extend your apps.

The MVC pattern divides your application into three individual parts that interact with each other: the model, the views, and the controllers. There are some best practices for each of these parts:

Model: Keep Everything Organized

It's simple: Use the right folder structure! If you arrange and structure your files and folders in a smart way, this makes coding much easier and also makes for sound performance when you load your application.

- Find out more: Folder Structure: Where to Put Your Files [page 1428]

View: Use XML Views

There are many view types, for example JavaScript, JSON, or HTML. However, we strongly recommend that you use XML views and fragments. XML clearly separates the view and the application logic, is easy to manipulate and can be parsed by tools like the layout editor in SAP Web IDE. That's why we also used XML views in all our tutorials, demo apps, and guides.

- Learn how: Walkthrough Tutorial Step 4: XML Views [page 76]
- Find out more: XML View [page 787]

Controller: Find the Best Location for Your Controllers

Every view you create should have its own controller with a corresponding file name. For example: If your view is called `App.view.xml`, then the matching controller should be named `App.controller.js`.

There is one special case: The so called `BaseController` is not directly related to a view. It is quite common that several controllers use the same functions. You can place these shared functions in the `BaseController` from which all other instantiated controllers will inherit. In other words: Every function you place in the `BaseController` is available for all your controllers. This makes your app code definitely easier to maintain, and you save some lines of code!

The controllers are written in JavaScript and contain all the app logic. They should be placed in the `controller` folder. However, not all JavaScript code belongs in the `controller` folder. For example, formatter logic. The main function of this type of JavaScript file is to format data. That's why you should place it in the `models` folder of your application.

- Learn how: Walkthrough Tutorial Step 5: Controllers [page 79]
- Find our more: Controller [page 807]

Keep Your Views Short and Simple

The view part of your app reflects what users can see and interact with. You should use a suitable set of UI controls that match your scenario and keep things simple.

Use `sap.m` as the Default Namespace

Most bread-and-butter controls are located in the `sap.m` namespace, which makes it the perfect default namespace. If you want to add other controls and layouts, you can define an additional namespace. For your own namespaces, you should keep the alias short and simple as well. You will typically use it in many places, and a short alias keeps your code tidy.

```
<mvc:View
    xmlns="sap.m"
    xmls:l="sap.ui.layout"
    xmlns:mvc="sap.ui.core.mvc">
    <App>
        <Page>
            <l:HorizontalLayout>
            ...
```

- Find out more: Namespaces in XML Views [page 788]

Remove Clutter From Your Views

It's easy to save a few bytes and make your code a lot cleaner:

- Don't define properties that are set to their default values.
- Remove unused namespace aliases.
- Omit the `content` or `items` tag for controls that define default aggregations.
- Use self-closing XML tags for controls that don't define any aggregations.

> → Tip
>
> Samples may contain more code that you actually need. When you copy code from a sample, it's best to remove all properties that won't be used in your views.

```
<SearchField change=".onSearch"/>
<List items="{/Products}" headerText="Search Results">
    <StandardListItem title="{Name}"/>
</List>
</Panel>
```

Clean Up Your Aggregation Templates

If you have bound aggregations, Avoid using complex or nested controls. Remember: The template below will be repeated for every entity in your data. If the template is more complex than necessary, this may lead to performance issues at runtime and slow down your app.

```
<List
    items={/Products}>
    <StandardListItem
        title="{Name}"
        description="{Text}"/>
```

- Learn how: Data Binding Tutorial Step 12: Aggregation Binding Using Templates [page 244]
- Find out more: Aggregation Handling in XML Views [page 788]

Think About View Modularization Early On

Things may get a little messy as your app is growing with your requirements. Therefore, name your views semantically. If a view is getting too "heavy", you should outsource parts of it to a separate view. With XML fragments and XML composites, you can flexibly reuse parts of yor UI elsewhere.

```
<App>
    <Page>
        <myXMLComposites:SearchPanel title="Find employees"/>
        <mvc:XMLView viewName="EmployeList"/>
    </Page>
</App>
```

- Learn how: Walkthrough Tutorial Step 15: Nested Views [page 104]
- Find out more:
  - XML Composite Controls [page 2229]
  - Reusing UI Parts: Fragments [page 1004]

Choose Clever UI Patterns

SAPUI5 offers a huge collection of feature-rich UI controls, often giving you multiple implementation choices.

Aim for the simplest possible pattern to implement your use case. For more information, see the Samples, and filter for "layout".

Use Stable IDs

If you keep the IDs of controls, elements, and components stable, you can be sure that other SAPUI5 features will be able to identify them correctly during processing.

Background

SAPUI5generates IDs for controls, elements, or components dynamically if you don't set them yourself. This sounds convenient, but might lead to problems when the corresponding elements are processed later on by other SAPUI5 features. So it's a good idea to use stable IDs instead of dynamic IDs.

How to Make IDs Stable

For this, you use the `id` property or attribute of the respective element. For a list of the elements for which you can set stable ID, see the related link below.

Here's an example of an XML view **without** stable IDs:

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page>
        <content>
            <Table>
            </Table>
        </content>
    </Page>
</mvc:View>
```

At runtime, the `Page` and the `Table` would get dynamically generated IDs like `__page0` and `__table0`. However, these generated IDs can change whenever the control structure of the app changes.

If you define stable IDs for the two controls in the example above, it could look like this:

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page id="page">
        <content>
            <Table id="table">
            </Table>
        </content>
    </Page>
</mvc:View>
```

The controls will now always be identified by these IDs.

In the case of views, the sequence of instantiation also plays a role: If there are two views with unstable IDs in the app, they get the generated IDs `__view0` and `__view1` depending on the order the views are opened. This makes it impossible to correctly identify them when they are processed by other features.

→ Tip

You should choose a semantic name for your IDs that makes it easier for you to identify them later.

For more information about naming restrictions, some testing options to check for unstable IDs, as well as the features that require stable IDs, see the related link below.

Related Information

Stable IDs: All You Need to Know [page 1442]

Make Your App CSP Compliant

CSP stands for Content Security Policy and is a security standard to prevent cross-site scripting or other code injection attacks.

It's strongly recommended that you make your SAPUI5 applications CSP compliant - after all, you want your apps to be secure. The main thing you have to do is to remove all scripts that directly execute code from your HTML pages.

Define Initial Components in a Declarative Way

Don't use directly executable code in your HTML files, because this makes them vulnerable. Instead, enable the `ComponentSupport` module in the bootstrapping script. Then, declare your desired component in the body via a `div` tag. This will instantiate the component when the `onInit` is executed.

```
...
<script id="sap-ui-bootstrap"
    src="resources/sap-ui-core.js"
    data-sap-ui-preload="async"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-oninit="module:sap/ui/core/ComponentSupport">
</script>
<body class="sapUiBody" id="content">
    <div data-sap-ui-component data-name="sap.ui.demo.walkthrough" data-
id="container" data-settings='{"id" : "walkthrough"}'></div>
</body>
```

- Learn how: Walkthrough Tutorial Step 9: Component Configuration [page 88]
- Find our more: Declarative API for Initial Components [page 731]

Separate Scripts From HTML Files in Your Test Folder

Because the HTML files in your test folder do not directly open your application, you can't use the new `ComponentSupport` feature here. To make them CSP compliant, you need to put the executable script code in

a separate file on the same level as the HTML file. You can then refer to this file in your HTML file inside a `script` tag in the head:

New script file:

```
window.suite = function() {
    "use strict";
    var oSuite = new parent.jsUnitTestSuite(),
        sContextPath = location.pathname.substring(0,
location.pathname.lastIndexOf("/") + 1);
    oSuite.addTestPage(sContextPath + "unit/unitTests.qunit.html");
    oSuite.addTestPage(sContextPath + "integration/opaTests.qunit.html");
    return oSuite;
};
```

HTML file:

```
<head>
    ...
    <script src="testsuite.qunit.js" data-sap-ui-testsuite></script>
</head>
</html>
```

- Learn how: Testing Tutorial Step 10: Test Suite and Automated Testing [page 411]
- Find out more: Content Security Policy [page 1481]

Use Asynchronous Loading

Asynchronous loading is the way to go: It makes your applications a lot faster and, through that, better to use.

As OpenUI5 is evolving, the loading processes in the background were significantly improved. To get the best out of these changes in the core and to speed up your app, we recommend that you switch on asynchronous loading. With asynchronous loading, files are retrieved in parallel. This is much quicker than with synchronous loading, where files are retrieved sequentially. There are a few possibilities to do that:

Use a Bootstrapping Tag in HTML Files

Add the bootstrapping tag `data-sap-ui-async="true"` to your `index.html` file. This loads the modules for all declared libraries asynchronously. If you have other HTML files in your app, you should do this there as well.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>My App</title>
    <script
        id="sap-ui-bootstrap"
        ...
        data-sap-ui-async="true">
    </script>
...
```

- Learn how: Walkthrough Tutorial Step 2: Bootstrap [page 72]

- Find out more: Bootstrapping: Loading and Initializing [page 692]

Add the `async` Property to the `manifest.json`

To also load all application-specific configuration settings asynchronously, set the `async` property in the metadata of the `manifest.json` file to `true`.

```
"sap.ui5": {
        "rootView": {
            "viewName": "sap.ui.demo.worklist.view.App",
            "type": "XML",
            "async": true,
            "id": "app"
        },
```

- Learn how: Walkthrough Tutorial Step 10: Descriptor for Applications [page 91]
- Find out more: Descriptor for Applications, Components, and Libraries [page 734]

Is Your Application Ready for Asynchronous Loading?

Find a collection of information that helps you to find out if your application is ready for asynchronous loading.

Applications benefit from the configuration of the SAPUI5 module loader to work asynchronously. However, if an application or library is not yet fully compatible with asynchronous loading, you may encounter issues. To avoid running into these issues, the following list provides information that helps you to find out whether your application needs to be adapted to enable asynchronous loading. The list is not exhaustive, but points to additional information and should give you a good start in getting your applications ready for asynchronous loading.

- For asynchronous loading, your application modules have to facilitate the concept for defining and handling of modules in SAPUI5 that is aligned with the asynchronous module definition (AMD) standard. For an overview, see Best Practices for Loading Modules [page 1100].
- Existing applications may still use synchronous variants of factory methods. To make your application ready for asynchronous loading, you need to use asynchronous variants of factory methods instead. For information how you can replace the synchronous variants with asynchronous variants, see Legacy Factories Replacement [page 1124].
- The global access to legacy APIs triggers a compatibility layer to load such modules synchronously and needs to be replaced to enable your application for asynchronous loading. For information how you replace the global access, see Legacy jQuery.sap Replacement [page 1109].
- The Support Assistant also helps you to identify issues in your application, especially issues related to synchronous or asynchronous loading. For information about the Support Assistant, see Support Assistant [page 1339].
- If your application relies on certain points in time, you may run into runtime issues when you switch from synchronous to asynchronous module loading and the points in time on which your application relies are different due to this change. Here are two examples of such issues:
  An event gets triggered before potential listeners had the chance to attach themselves. Whereas this may have worked for synchronous loading where the timing of module loading and initializing is different, this causes issues for asynchronous loading. To resolve this and to ensure that all modules are properly loaded

and exectued, all necessary module dependencies for the module that actually triggers the event need to be handled by `sap.ui.require` or `sap.ui.define`.
A controller listens on the `EventBus` for a certain event that is triggered by another controller. Depending on the loading time of the corresponding views, the `init` controller methods may be called at a different point in time. For information how to resolve this, see Best Practices for Loading Modules [page 1100].

- Issues with asynchronous loading can also occur, if your application uses XML views that are configured to be loaded asynchronously via the `manifest` property, for example for the root view `sap.ui5/rootView/async=true`. To detect such issues, we recommend to do extensive (automatic) testing to ensure the application continues to work as expected.

Known Incompatibilities

The `sap.viz` library uses another module loader in addition to the SAPUI5 module loader in some scenarios. In combination with the `async=true` configuration parameter, this currently leads to issues and may break your application.

Performance Checklist

Follow these steps to apply performance best practices to your application.

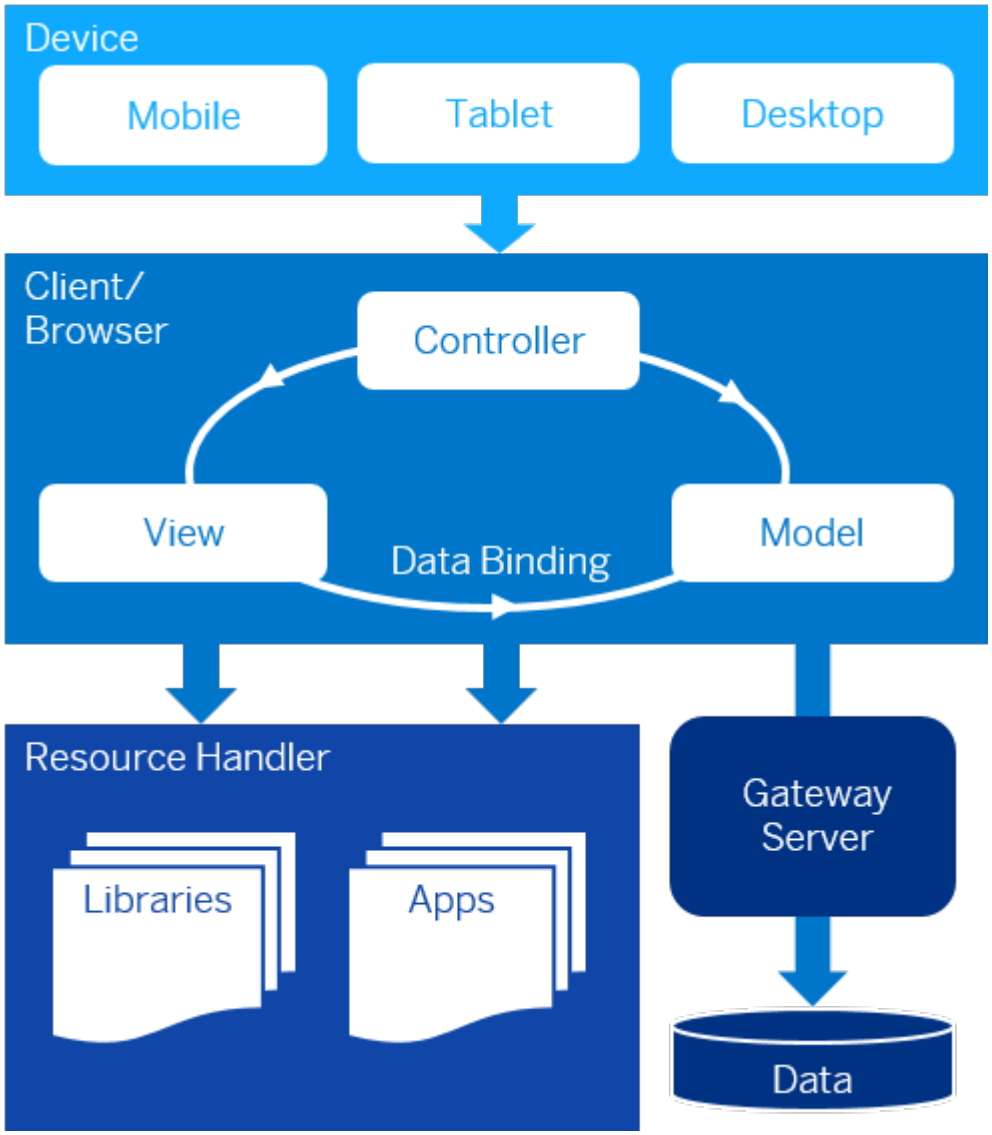
In addition to applying best practices, always stay up to date with the framework, for instance via the SAPUI5 Release Notes and the What's New in SAPUI5 [page 6].

1. Use the UI5 Support Assistant to Check for Known Issues [page 1339]
2. Enable Asynchronous Loading in the Bootstrap [page 1434]
3. Ensure the Root View and Routing are Configured to Load Targets Asynchronously [page 1435]
4. Make Use of Asynchronous Module Loading (AMD Style) [page 1436]
5. Use `manifest.json` instead of the Bootstrap to define Dependencies [page 1436]
6. Load SAPUI5 from the Content Delivery Network (CDN) [page 1437]
7. Ensure that all Resources are Available to Avoid 404 Errors [page 1438]
8. Use "manifest first" to load the Component [page 1438]
9. Ensure that Library Preloads are Enabled [page 1438]
10. Ensure that Application Resources are Loaded as Component Preload [page 1438]
11. Check Network Requests [page 1439]
12. Migrate `jquery.sap.*` Modules to their Modularised Variants [page 1439]
13. Migrate Synchronous Variants of UI5 Factories to Asynchronous Variants [page 1440]
14. Use the OData V2 Model Preload [page 1440]
15. Use OData V2 Metadata Caching [page 1440]
16. Check Lists and Tables [page 1440]
17. Further Optimize your Code [page 1441]

Essentials

This chapter and its sections describe the development concepts of SAPUI5, such as the Model View Controller, data binding, and components. Use this section as a reference.

SAPUI5 Architecture



SAPUI5 is a client UI technology based on JavaScript, CSS and HTML5.

Apps developed with SAPUI5 run in a browser on any device (mobile, tablet or desktop PC).

When users access an SAPUI5 app, a request is sent to the respective server to load the application into the browser. The view accesses the relevant libraries. Usually the model is also instantiated and business data is fetched from the database.

Depending on the environment in which SAPUI5 is used, the libraries or your applications can be stored, for example, on an SAP NetWeaver Application Server or an SAP Cloud Platform, and business data can be accessed, for example, using the OData model through a SAP Gateway.

Artifacts in the Framework

The top-level structural unit is called a **library**. Libraries are the master artifacts in the extensibility concept. They bundle a set of controls and related types and make them consumable by Web applications. There are predefined and standard libraries, like `sap.m`, with many commonly used controls. At the same time, it treats custom UI libraries as first-class citizens, making it easy for you to write and use your own controls alongside the predefined ones.

A UI **element** is the basic building block of our user interfaces; it is a reusable entity with properties, events, methods, and relations. The most important relations are aggregations to other UI elements, and in this way a tree structure of elements can be created.

From a developer's point of view, a **control** (e.g. `Button`, `Label`, `TextField`, or `Table`) is the most important artifact. It is an object which controls the appearance and user interaction of a rectangular screen region. It is a special kind of user interface element which can be used as the root of such a tree structure. In this way, it serves as an entry point, especially for rendering. Besides controls, there are also other **non-control elements**, which cannot be used as the root of such a tree structure, but only as a dependent part within it (e.g. `TableRow`, `TableCell`).

Data types are first-class entities in the meta model. This allows reuse of types across libraries and extensibility of the type system. The core library (technically, this is the `sap.ui.core` library) already defines a core set of types that can be used in other libraries.

Bootstrapping: Loading and Initializing

To use SAPUI5 features in your HTML page, you have to load and initialize the SAPUI5 library.

You can use the SAPUI5 bootstrap script in your page to initialize SAPUI5 runtime automatically as soon as the script is loaded and executed by the browser. For simple use cases as well as the default SAPUI5 installation, this is sufficient to build and run UIs. In addition to this, you can specify the set of SAPUI5 libraries and the theme used for your application in the configuration settings.

> **i** Note
>
> If you run your app standalone, the bootstrap is added to your HTML page. In an SAP Fiori launchpad environment, the launchpad executes the bootstrap and no additional HTML page is needed to display the app.

The following code snippet shows a typical bootstrap script tag:

```
<script id="sap-ui-bootstrap"
     type="text/javascript"
     src="resources/sap-ui-core.js"
     data-sap-ui-theme="sap_belize"
     data-sap-ui-libs="sap.m"
```

```
      data-sap-ui-compatVersion="edge">
</script>
```

The attributes `data-sap-ui-theme="sap_belize"` and `data-sap-ui-libs="sap.m"` already provide examples of how SAPUI5 runtime can be configured to the needs of an application.

Overview of Bootstrap Files

SAPUI5 provides several bootstrap files for different use cases. The following table gives an overview of the most important resources and the respective use cases. The resource names refer to the `resources/` folder in the SAPUI5 installation. The actual base URL depends on your platform and administrative setup.

Resource	Description
`sap-ui-core.js`	This is the standard bootstrap file, which we recommend to use for typical use cases. It already contains jQuery, `jquery-ui-position` and only the minimum required parts of the core library (`sap.ui.core`). Required files are loaded dynamically using XMLHttpRequest (XHR). For more information, see Standard Variant for Bootstrapping [page 694].
Content Delivery Network (CDN)	You can access the libraries externally from a CDN. For more information see Variant for Bootstrapping from Content Delivery Network [page 696].
`sap-ui-core-nojQuery.js`	You use this bootstrap file for applications with their own jQuery version. It also contains the minimum required parts of the core library, but **not** jQuery and `jquery-ui-position`. For more information, see noJQuery Variant for Bootstrapping [page 698].
`sap/ui/core/library-preload.js`	This file contains most of the modules that are contained in the `sap.ui.core` library, but the modules are parsed and executed only on demand, and not immediately. ⚠ Caution An application must not reference this file. If the configuration option is set to `preload`, SAPUI5 automatically loads the file. For more information, see Standard Variant for Bootstrapping [page 694].
`sap-ui-core-lean.js`	This bootstrap file is similar to the `sap-ui-core.js` file, but in this use case only the jQuery and one SAPUI5 file are

Resource	Description
	loaded immediately and the other files are loaded dynamically. **⚠ Caution** This use case is usually **not** used and may be removed in future.
`sap-ui-custom*.js`	File names that match this pattern are reserved for custom merged files used by the application. **i Note** The proposed naming scheme for these files needs to be adapted in future versions for the same encapsulation reasons as mentioned above.

Standard Variant for Bootstrapping

The standard variant for bootstrapping loads all JavaScript modules of a library in advance with one single request for performance reasons.

The library preload file `library-preload.js` contains all modules of a certain library. These modules will only be executed on demand, if the application requires them. Using preloads significantly reduces the number of roundtrips since the single modules are bundled in one file.

> **i Note**
>
> An application must **not** reference the `library-preload.js`. If preload files exist, SAPUI5 automatically loads them. The dependencies to libraries are defined as part of the manifest namespace `sap.ui5/dependencies/libs`. For further information, see Descriptor for Applications, Components, and Libraries [page 734].

Option 1 (Recommended)

By setting the `async=true` configuration parameter, the module loader loads the modules and preload files asynchronously. You can enable it in an existing application by specifying the `sap-ui-async` configuration parameter in the start URL, or by adding the `data-sap-ui-async` attribute to the bootstrap tag:

```
<script
    id="sap-ui-bootstrap"
    src="resources/sap-ui-core.js"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-async="true"
    data-sap-ui-onInit="module:my/app/main"
```

```
    data-sap-ui-resourceRoots='{"my.app": "./"}'
></script>
```

> **i Note**
>
> Before you use the `async` configuration parameter, make sure your app is ready for asynchronous loading, see Best Practices for Loading Modules [page 1100] and Is Your Application Ready for Asynchronous Loading? [page 689].

Option 2

Alternatively, you can influence the loading behavior of the preload files without affecting other single modules by setting the `preload` configuration parameter to one of the following values:

- `async` (recommended)
  If you set the `preload` configuration option to `async`, the runtime loads the modules for all declared libraries asynchronously. Thus, for any code that follows the SAPUI5 bootstrap tag, the framework cannot make sure that the classes are already available. Therefore, the application must delay the access to the SAPUI5 APIs by using the `Core.attachInitEvent` method. SAPUI5 supports the `async` mode only for libraries that are loaded by the SAPUI5 core. Libraries that are loaded dynamically by using the `sap.ui.getCore().loadLibrary()` API will be loaded synchronously by default for compatibility reasons. Only when a configuration object with a property of `async:true` is passed, the bundle is loaded asynchronously.
- `sync`
  If you set the `preload` configuration parameter to `sync`, the runtime loads the modules for all declared libraries synchronously. After processing the bootstrap tag, all preload files of all libraries are loaded and the libraries are initialized as usual. The `preload=sync` mode should be transparent for most applications.
- `auto`
  The `preload=auto` configuration parameter is the default value. This mode checks whether SAPUI5 runtime uses optimized sources. If optimized sources are used, it enables the `preload=sync` option to further optimize the runtime. For normal or debug sources, the preload is deactivated.

You can easily check this with an existing application by specifying the `sap-ui-preload=<mode>` parameter in the start URL or by adding the `data-sap-ui-preload` attribute to the bootstrap tag:

```
<script
    id="sap-ui-bootstrap"
    src="resources/sap-ui-core.js"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-preload="async"
    data-sap-ui-onInit="module:my/app/main"
    data-sap-ui-resourceRoots='{"my.app": "./"}'
></script>
```

> **i Note**
>
> Using the `async=true` or `preload=async` configuration parameters requires extensive testing and cooperation on application side to ensure a stable and fully working application. It is, therefore, not activated automatically, but only by configuration. If you encounter issues, or if you want to prepare your application in advance, see Is Your Application Ready for Asynchronous Loading? [page 689].

> **i Note**
>
> Preload sources are always optimized. However, using the `debug=true` configuration parameter always disables the loading of preload files.

> **i Note**
>
> You can combine the `async` or `preload` configuration parameters with other bootstrap variants such as `sap-ui-core-noJQuery`.

Related Information

noJQuery Variant for Bootstrapping [page 698]
Configuration of the SAPUI5 Runtime [page 699]
Is Your Application Ready for Asynchronous Loading? [page 689]

Variant for Bootstrapping from Content Delivery Network

SAPUI5 can either be loaded locally with a relative path from an SAP Web server or externally from a Content Delivery Network (CDN).

> **i Note**
>
> Loading SAPUI5 from a CDN improves your app performance: You can load from a server that (in most cases) is much closer to your location, and you can benefit from the caching mechanism and the language fallback logic.

Bootstrapping From SAPUI5 CDN

SAPUI5 application hosted on SAP Cloud Platform are allowed to leverage the SAPUI5 CDN to retrieve the SAPUI5 distribution layer artifacts.

Specific Version

Check the available versions with the respective maintenance status at https://ui5.sap.com/versionoverview.html.

You can refer to a specific version by using a versioned URL as in the following example:

```
<script id="sap-ui-bootstrap"
    type="text/javascript"
    src="https://sapui5.hana.ondemand.com/1.42.6/resources/sap-ui-core.js"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-libs="sap.m"></script>
```

The first segment of the URL after the host name is used to specify a concrete version.

Default Version

The default version of our libraries has the generic URL https://sapui5.hana.ondemand.com/resources/sap-ui-core.js (SAPUI5).

> ⚠ Caution
>
> The default version is constantly being upgraded and this might have an impact on the stability of your application. Use this version for testing purposes only.
>
> If you want to use the default version, you can use the following bootstrap scripts:

```
<script id="sap-ui-bootstrap"
    type="text/javascript"
    src="https://sapui5.hana.ondemand.com/resources/sap-ui-core.js"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-libs="sap.m"></script>
```

Cache Control

The cache control is different for dynamic and static resources. If you refer to the latest maintenance version (dynamic), you have a maximum cache age of one week, if you refer to a specific (static) version, you have a maximum cache age of 10 years. In both cases, cross-origin resource sharing (CORS) headers are set, so that you can consume resources from the central location without any proxy in between.

> ℹ Note
>
> The Cache Buster is only needed if you consume SAPUI5 without a concrete version in the URL. When you consume SAPUI5 with the concrete version in the URL, this is not needed, as the content served by that unique URLs will never change and can be cached forever.

Bootstrapping from Custom CDN

SAPUI5 applications hosted on platforms other than SAP Cloud Platform can leverage a custom CDN to retrieve the SAPUI5 distribution layer artifacts. For this, the SAPUI5 distribution layer artifacts must be deployed on an SAP web server.

To use your custom CDN with the SAPUI5 ABAP repository, you need to configure this CDN as an external location in the customizing. For more information, see the documentation for the Customizing activity *Configure SAPUI5 Bootstrapping* in Customizing under *SAP NetWeaver* > *UI Technologies* > *SAPUI5*.

Related Information

Multi-Version Availability of SAPUI5

Versioning of SAPUI5 [page 29]

noJQuery Variant for Bootstrapping

The noJQuery variant supports bootstrapping for an application that already integrates jQuery or uses a different jQuery version than SAPUI5.

In this variant, you include the `resources/sap-ui-core-noJQuery.js` file in your HTML page. Make sure that jQuery and `jquery-ui-position` have been loaded beforehand. The following code snippet shows an example:

```
    <!-- include some jQuery version -->
    <script src="my-jQuery-min.js" ></script>
    <!-- application does not have its own jquery-ui-position, so it might use
the one from SAPUI5 -->
    <script src="resources/sap/ui/thirdparty/jqueryui/jquery-ui-position.js" ></
script>
    <!-- now booting SAPUI5 -->
    <script
            id="sap-ui-bootstrap"
            src="resources/sap-ui-core-nojQuery.js"
            data-sap-ui-libs="sap.m"
            data-sap-ui-theme="sap_belize" >
    </script>
```

Initialization Process

The initialization process starts after SAPUI5 runtime is loaded.

The initialization of the SAPUI5 runtime comprises the following steps:

1. The jQuery plugins, which are mainly located in the `jQuery.sap` namespace, provide fundamental functionality of SAPUI5, such as the modularization concept, a logging framework, performance measurement, and so on.
2. The global object `sap` is defined.
3. The `sap.ui.core.Core` class is executed with all its dependencies.
4. The runtime configuration is determined from different sources.
5. All libraries and modules declared in the configuration as well as their dependencies are loaded.
6. For each loaded library, the CSS file of the configured theme is loaded.
7. When all libraries are loaded and the document is ready, the `initEvent` of the core is fired and all registered handlers are executed.

Initialization Readiness

The optimal point in time to execute or start an application is after the framework has been initialized. You can use the `attachInit` function to determine this point in time: The callback of the `attachInit` function is executed directly after the framework has been initialized.

```
sap.ui.getCore().attachInit(function(){
    // application can be started
});
```

As an alternative, you can also use a bootstrap module, see Standard Variant for Bootstrapping [page 694].

Loading of Additional Resources During Bootstrap

The SAPUI5 runtime loads and interprets additional resources for the control libraries during bootstrap.

The files are loaded in the following sequence:

1. Library bootstrap file `/<context-path>/resources/<library-name>/library.js`
   A JavaScript file that contains the JavaScript code for all enumeration types provided by the library as well as library-specific initialization code that is independent from the controls in the library. The file calls the `sap.ui.getCore().initLibrary` method with an object that describes the content of the library (list of contained controls, elements etc.). For libraries that have been developed with SAPUI5 application development tools or the SAPUI5 offline build tools, this file is generated automatically during the build
2. Library style sheet file `/<context-path>/resources/<library-name>/themes/<theme-name>/library.css`
   SAPUI5A standard CSS file that contains all styles relevant for this library. For application development tools, this file is generated automatically during the build.

Dynamic Loading of Libraries

SAPUI5 provides the `sap.ui.getCore().loadLibary()` method to load libraries at runtime in addition to the libraries declared in the runtime configuration.

After loading, you can use all controls from the library. For these additional libraries, the same restriction apply as for the declared libraries: Accessing the document object model (DOM) is only possible after the `document.ready` event of the HTML page. Also, rendering applies for these libraries in the same way as for the declared libraries.

Configuration of the SAPUI5 Runtime

SAPUI5 provides several options for the configuration of the SAPUI5 runtime, such as runtime default values and script tag attributes.

When the SAPUI5 bootstrap script is included in a page, the SAPUI5 runtime will automatically be initialized as soon as the script is loaded and executed by the browser. For simple use cases and for a default SAPUI5 installation, this should already be sufficient to build and run UIs. The only additional information that usually is specified, is the set of libraries and the theme that is used.

So a typical bootstrap script looks like this:

```
<script id="sap-ui-bootstrap"
            type="text/javascript"
            src="resources/sap-ui-core.js"
            data-sap-ui-theme="sap_belize"
            data-sap-ui-libs="sap.m"
            data-sap-ui-compatVersion="edge">
```

```
</script>
```

For more information see Bootstrapping: Loading and Initializing [page 692].

You can use the following ways to provide configuration information.

Default Values

The easiest way to specify a configuration value is **not to specify** it. The SAPUI5 runtime contains a default value for each configuration option. As long as you don't have to change the value, you don't specify it.

Individual Script Tag Attributes

For each configuration option, you can have one attribute in the bootstrap script tag.

The attributes have to provide the following information:

- Attribute name
  The attribute name is composed of the name of the configuration option and the `data-sap-ui-` prefix. The first part of the prefix (`data-`) is necessary to comply with the W3C recommendations for custom attributes in HTML. The second part (`-sap-ui-`) separates SAPUI5 attributes from custom attributes defined by any other framework.

  > **i Note**
  >
  > Attribute names in HTML are case-insensitive and this also applies to the configuration attribute names. However, SAPUI5 has defined some configuration options names in camel case, for example `originInfo`. SAPUI5 converts these names automatically to lower case when accessing the configuration.

- Value
  Element attributes in HTML have a `string` value by definition. For configuration options of type `string`, the attribute value is equivalent to the value of the option.

  > **i Note**
  >
  > If the value contains specific HTML characters, such as '<' or '>', or if the value contains the same quote character that is used to wrap the attribute value, the usual HTML escape mechanisms must be used: Use entities for the specific HTML characters, for example `<` instead of `<`, and switch the type of quotes from single to double or vice versa.

  For configuration options that are **not** of type `string`, the format of the allowed values has to be defined as follows:

Type	Notation/Values
`string`	String; escaped according to the HTML conventions

Type	Notation/Values
`boolean`	`true` and `x` are both accepted as true values (case-insensitive), all others are false. We recommend to use `false` for false values
`int`	Any integer value
`string array`	Comma-separated list of values; comma is not supported in the values (no escaping)
map from string to string	JavaScript object literal (preferably JSON syntax)

Single and Complex Configuration Attributes

The attribute `data-sap-ui-config` makes it possible to provide a single attribute with the configuration information for the SAPUI5 runtime.

You can use this attribute instead of attaching individual options with individual configuration attributes to the script tag. Its content is similar to the global configuration object, but without the enclosing parenthesis: It is a comma separated list of key-value pairs.

> **i** Note
>
> The usual HTML escape mechanisms must be used if the value contains specific HTML characters (<, >, &) or the quote character that is used to enclose the attribute value.

```
<script id="sap-ui-bootstrap"
    type="text/javascript"
    src="resources/sap-ui-core.js"
    data-sap-ui-config="theme:'sap_belize',
    libs:'sap.m'"
    >
</script>
```

Global Configuration Objects

The global configuration object is a property in the global `window` object with property name `sap-ui-config`. The property must be a simple object, where each property represents the configuration option of the corresponding name.

To avoid conflicts with typical JavaScript coding, the name of the `window` property is not a valid JavaScript identifier. The name structure is chosen to avoid conflicts with SAP objects. To define the object, quotes must be used. If a configuration option has a name that is not a valid JavaScript identifier or that is a reserved token in JavaScript, the property name in the configuration object must be quoted. Currently, such a configuration option does **not** exist.

As the configuration is evaluated during bootstrap, the configuration object must be created **before** SAPUI5 is bootstrapped. Otherwise, the contained configuration cannot be evaluated. As a consequence, using the global configuration object requires another script tag in front of the bootstrap script tag. It is up to the application

whether it uses an inline script tag or a separate JavaScript file, which is loaded via a script tag, for this purpose. If you use a dedicated file, it may require more work initially, but offers the following advantages:

- Several pages can share the file and, thus, use the same configuration.
- The Content Security Policy (CSP) mechanism as introduced, for example, by Firefox 4.0 and others requires the use of a file.

The following code snippet shows an example for an inline script tag:

```
<script type="text/javascript">
            window["sap-ui-config"] = {
            theme : "sap_belize",
            libs : "sap.m",
            };
</script>
<script id="sap-ui-bootstrap"
            src="resources/sap-ui-core.js"
        >
</script>
```

This option requires an additional script or script tag, but it offers the following advantages:

- Possibility to share configuration between pages
- Can be used in environments where the scrip tag cannot be influenced, for example, because it is created out of some configuration, like in some mashup frameworks
- Allows to provide configuration before the core boots

URL Parameters

Configuration parameters can be added to the URL of an app.

The URL parameter name is composed of the name of the configuration option and the `sap-ui-` prefix, for example like `index.html?sap-ui-debug=true`.

> **i Note**
>
> The W3C proposed that the `data-` prefix is not needed and not even allowed here as all URL parameters are kind of custom parameters.

The value of a URL parameter is of type `string` and the same type mapping as for HTML attributes applies. However, URLs require a different encoding than HTML; they use, for example % encoding instead of entity encoding.

For security reasons, only some configuration options can be set via URL parameters. An application can set the `ignoreUrlParameters` option to `true` to disable URL configuration parameters completely.

Runtime Configuration Object

The runtime configuration object enables you to modify a limited set of configuration options at runtime.

The configuration options above are evaluated during the SAPUI5 runtime boots. After that, all changes to these parameters are ignored. To read the final configuration result, you can use the `sap.ui.getCore().getConfiguration()` method.

The same object also provides set methods for a very limited set of configuration options that can be modified at runtime. The runtime and/or the controls can react on these configuration changes. The most prominent (and so far only) example for such a configuration option is the theme.

Order of Significance

1. Attributes of the DOM reference override the system defaults.
2. URL parameters override the DOM attributes; empty URL parameters reset the parameter to its system default.
3. If you call setters at runtime, any previous settings calculated during object creation are overwritten with the new value.

Configuration Options and URL Parameters

The complete list of configuration options available in SAPUI5 can be found in the *API Reference* under `sap.ui.core.Configuration`. The following table shows a subset of the available configuration options.

> **i** Note
>
> A subset of these configuration parameters can also be used as URL parameter ("URL: Yes"). The URL parameter name is composed of the name of the configuration option and the `sap-ui-` prefix, for example like `sap-ui-debug=true`.

Option	Type
`accessibility`	Type: `boolean` Default value: `true` URL: Yes Modifiable at runtime: No If set to `true`, the SAPUI5 controls are rendered for or running in accessibility mode.
`animationMode`	Type: `string` Default value: `full` URL: Yes Modifiable at runtime: Yes The following animation modes are available:

Option	Type
	• `full`: all animations are shown • `basic`: a reduced, more light-weight set of animations • `minimal`: no animations are shown, except animations of fundamental functionality • `none`: deactivates the animation completely This parameter replaces the deprecated Boolean `animation` parameter. For all controls that implement the `animation` parameter, the `animationMode` is set as follows: • If `animation` is set to `true`, this is interpreted as `animationMode full` • If `animation` is set to `false`, this is interpreted as `animationMode minimal`
`appCacheBuster`	Type: `true | string[]` Default value: [] URL: Yes Modifiable at runtime: Yes, with `AppCacheBuster` API (see Application Cache Buster: Enhanced Concept [page 1136]) If set to a non empty list of URLs, the `AppCacheBuster` will be activated and will load component version info files from the configured set of URLs (see Application Cache Buster [page 1134]).
`areas`	Type: `string[]` Default value: `null` URL: No This configuration parameter defines UI areas that shall be created in advance; use to create new UI areas and `sap.ui.getCore().getUIArea(id).destroy()` to delete existing UI areas at runtime.
`autoAriaBodyRole`	Type: `boolean` Default value: `true` URL: No Modifiable at runtime: No Determines whether the framework automatically adds the ARIA role `application` to the HTML body.

Option	Type
`bindingSyntax`	Type: `string` Default value: • As of SAPUI5 1.28, in combination with `data-sap-ui-compatVersion="edge"`: `complex` • `default` The meaning of the default value `'default'` depends on the compatibility version `sapCoreBindingSyntax`. If the compatibility version is at least 1.28, the 'complex' binding syntax is assumed, otherwise the 'simple' binding type. In other words: applications that configured a general compatibility version of 1.28 (or higher or 'edge'), will automatically run with the 'complex' binding syntax. URL: No Modifiable at runtime: No This configuration parameter defines whether the simple or the complex binding syntax is used. The parameter only affects bindings that are defined as strings, for example in the constructor of a control, or when specifying a binding in a declarative view, such as XML view or HTML view.
`calendarType`	Type: `gregorian | islamic | japanese | persian` (case-sentitive) Default value: If there is no value defined, the actual value is determined from the locale data for the configured locale. URL: Yes Modifiable at runtime: See API Reference: `sap.ui.core.Configuration.setCalendarType`. Defines the calendar type that is used for locale-dependent, date-related features (for example, formatting or parsing date and time).
`debug`	Type: `boolean` or `string` Default value: `false` URL: Yes Modifiable at runtime: No If set to `true`, the debug sources are loaded; if the bootstrap code is loaded from an optimized source, the

Option	Type
	bootstrap will be aborted and start anew from a debug source. You can also specify a comma-separated list as `string` that contains all modules that should be loaded as debug source. Example: `index.html?sap-ui-debug=sap/ui/model/odata/v2/` will load all debug sources for all modules of the OData V2 model. All others modules will be taken from the preload (if preload is active). You can use the following patterns: • A trailing slash (`/`) means that the complete package should be included (shortcut for `/**/*`) Example: `sap/ui/model/odata/v2/` loads everything from the `sap/ui/model/odata/v2/` package as debug source (also nested packages `sap/ui/model/odata/v2/**/*`). • `**/` matches any package or sequence of packages Example: `**/v2/` loads any package named `v2` as debug sources like `odata/v2`, `json/v2/` etc. • `*` matches any part of a simple name Example: `sap/ui/model/*` matches all files directly contained in the model package, but not in nested packages (for example, not `v2` or `v4`) **i Note** You can also select the debug sources in the technical information dialog. For more information, see Technical Information Dialog [page 1322].
`formatLocale`	Type: `string` Default value: `undefined` URL: Yes Modifiable at runtime: No This configuration parameter defines the locale used for formatting purposes; the default values for the locale are derived from the language.
`frameOptions`	Type: `string` Default value: `default` URL: No

Option	Type
	Modifiable at runtime: No Frame options mode; for more information, see Frame Options [page 1478]
`frameOptionsConfig`	Type: `object` Default value: undefined URL: No Modifiable at runtime: No Advanced frame options configuration; for more information, see Frame Options [page 1478]
`ignoreUrlParams`	Type: `boolean` Default value: `false` URL: No Modifiable at runtime: No Security-relevant parameter that allows applications to disable configuration modifications via URL parameters.
`inspect`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No If set to `true`, the `sap-ui-debug.js` module is included and provides some supportability features
`language`	Type: `string` Default value: `user language` URL: Yes Modifiable at runtime: Yes, with restrictions. This configuration parameter defines the language that shall be used for localized texts, formatting, and so on. For more information, see API Reference: `sap.ui.core.Configuration.setLanguage` and Identifying the Language Code / Locale [page 1269].
`libs`	Type: `string[]` Default value: `[ ]` URL: No

Option	Type
	Modifiable at runtime: Yes This configuration parameter defines a list of libraries that shall be loaded initially; use the `loadLibrary()` method to load further libraries. For more information, see: loadLibrary
`logLevel`	Type: `0 | 1 | 2 | 3 | 4 | 5 | 6 | NONE | FATAL | ERROR | WARNING | INFO | DEBUG | ALL` Default value: `ERROR` URL: Yes Modifiable at runtime: Yes This configuration parameter sets the log level to the given value; for minified (productive) sources, the default level is `ERROR`, for debug sources it is `DEBUG`. At runtime, you can modify the log level by using the `sap/base/Log.setLevel` method.
`manifestFirst`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: Yes, by using option with same name in the `sap.ui.component` API If set to `true`, the descriptor for a component is read and evaluated first, before loading the component code (`Component.js`).
`modules`	Type: `string[]` Default value: `[ ]` URL: No This configuration parameter defines a list of JavaScript modules that shall be loaded after the core has been initialized.
`noConflict`	Type: `boolean` Default value: `false` URL: No Modifiable at runtime: No

Option	Type
	If set to `true`, SAPUI5 forces jQuery into `noConflict` mode.
`noDuplicateIds`	Type: `boolean` Default value: `true` URL: Yes Modifiable at runtime: No If set to `true`, this configuration parameter enforces that the same IDs are **not** used for multiple controls; we highly recommend this check as duplicate IDs may cause unforeseeable issues and side effects.
`onInit`	Type: `code` or `string` Default value: `undefined` URL: No Modifiable at runtime: No This configuration setting defines code that has to be executed after the initialization. The use of this parameter with string value as (like `"myinitfunction();"`) code is no longer recommended as it requires `eval` and therefore might conflict with stronger content security policies. Either use it only in the form `window["sap-ui-config"].onInit = function() { ... }` or use the runtime API `sap.ui.getCore().attachInit()` instead. If you define a `string`, this can be a reference to a function or a name of a module. Functions are resolved from the global namespace (like `"myapp.initFunction"`). Modules are indicated by the prefix `module:` (like `"module:myapp/main/Module")`. The module will be loaded and executed after the initialization.
`originInfo`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No If set to `true`, additional information for text resources is provided that allows to determine the origin of a translated text on the UI

Option	Type
`preload`	Type: not specified, `auto`, `sync`, or `async` Default value: `auto` URL: No Modifiable at runtime: No This configuration parameter defines the loading behaviour of the so-called preload files. They contain all modules of a library. The contained modules are only loaded, but not executed until they are used by the application. The values are used as follows: • When set to `auto`, SAPUI5 runtime automatically uses `preload=sync` when the `async` bootstrap configuration parameter is set to false (`async=false`) or not set at all. The `preload` files are loaded asynchronously in case `async=true` is set. • When set to `sync`, the preload files for the declared libraries are loaded synchronously. • When set to `async`, the preload files are loaded asynchronously. However, we recommend to use the `async=true` configuration parameter in the bootstrap instead, because it switches more module/ related APIs to `async` including the loading behaviour of the preload files. • For any other value (for example blank), the preload feature is deactivated and modules are loaded on demand.
`async`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No This configuration setting enables the module loader to load both, modules and library-preload files asynchronously. Activating this feature requires intensive testing and cooperation on application side to ensure a stable and fully working application. In case you encounter issues, or if you want to prepare your application in advance, see Is Your Application Ready for Asynchronous Loading? [page 689].
`preloadLibCss`	Type: `string[]` Default value: `[]`

Option	Type
	URL: Yes Modifiable at runtime: No This configuration setting specifies a list of UI libraries using the same syntax as the `libs` property, for which the SAPUI5 core does not include the `library.css` stylesheet in the head of the page. If the list starts with an exclamation mark (!), no stylesheet is loaded at all for the specified libs. In this case, it is assumed that the application takes care of loading CSS, for example, a manually merged, single CSS file. Otherwise, the Core instructs the backend to create a merged CSS for the specified libs. In both cases, if the first libraries name is an asterisk (*), it will be expanded to the list of already configured libraries. **i Note** The `merge` feature is currently only available for Java and only for apps that include the additional backend component `resource-ext`. Without the merge, applications can include their own merged CSS file and suppress the loading of the standard `library.css`.
`resourceRoots`	Type: `object` Default value: undefined URL: No Modifiable at runtime: With `sap.ui.loader.config({paths: ...})`, a map can be used to define locations for resources, see `sap.ui.loader` in the API reference. To provide a URL location that is **not** overwritten by a component later on, `final` can be set to `true`, for example: `{'url' : '/that/is/the/prefix/', 'final' : true}` For more information, see jQuery.sap.registerModulePath
`rtl`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: Yes, with restrictions. For more information, see API Reference: `sap.ui.core.Configuration.setLanguage` and

Option	Type
	API Reference: `sap.ui.core.Configuration.setRTL`. If set to `true`, all controls are rendered in right-to-left (RTL) mode; not yet determined automatically.
`statistics`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No Activates end-to-end traces and measurement of response times For more information, see Interaction Tracking for Performance Measurement [page 1382]
`theme`	Type: `string` Default value: `base` URL: Yes Modifiable at runtime: Yes This configuration parameter defines the theme that shall be used for the current page; you can change the theme at runtime by calling `sap.ui.getCore().applyTheme()`. **Theme Root:** When the theme string contains an at-sign (`@`), anything before the `@` is assumed to denote the ID of the theme while anything after the `@` is assumed to represent the URL location of the theme. To defend against XSS attacks, only tthe server-relative part of the URL is used, any host or port prefix will be ignored.
`themeRoots`	Type: `object` Default value: undefined URL: No Modifiable at runtime: `sap.ui.getCore().setThemeRoot()` This configuration parameter defines the location of themes.
`trace`	Type: `boolean` Default value: `false`

Option	Type
	URL: No Modifiable at runtime: No If set to `true`, this configuration parameter activates an overlay div that contains a trace.
`trailingCurrencyCode`	Type: `boolean` Default value: `true` URL: Yes Modifiable at runtime: No By default the currency codes are shown after the amount. If set to `false`, the currency code will be shown as configured by the locale-specific patterns of the Common Locale Data Repository (CLDR).
`uidPrefix`	Type: `string` Default value: '--' URL: No Modifiable at runtime: No Prefix to be used for automatically generated control IDs; must be chosen carefully to avoid conflicts with IDs defined by the application or DOM IDs.
`versionedLibCss`	Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No If set to `true`, the version parameters are included in requests to the library theme resource (for example, the parameter `library.css?version=1.0.1&sap-ui-dist-version=1.0.2` is added. `version` contains the library version and `sap-ui-dist-version` the version of the SAPUI5 distribution . This applies to the following resources: • `library(-RTL).css` (or any other variation) • `library-parameters.json` URLs within the CSS or parameters are not modified.
`weinreId`	Type: `string`

Option	Type
	Default value: URL: Yes Modifiable at runtime: No
`weinreServer`	Type: `string` Default value: URL: No Modifiable at runtime: No URL to a WEINRE server to be used for debugging purposes; if set, SAPUI5 automatically includes the WEINRE target modules.
`whitelistService`	Type: `string` Default value: URL: No Modifiable at runtime: No URL to a whitelist service; see Whitelist Service [page 1478]

Experimental Options

The options listed in the table below are 'experimental'. They may be removed in future versions, or their definition or behavior may change in an incompatible way. Experimental options are identified by the name prefix `xx-`. Experimental configuration options are used for support scenarios where SAPUI5 development needs the freedom to evolve supportability features over time. Others are related to experimental features where the underlying feature still may change. When an experimental configuration option becomes mature, the `xx-` prefix is removed from the definition. For compatibility reasons, the old name with the `xx-` prefix will still be supported.

Option	Type
`xx-componentPreload`	**i Note** This is an experimental feature and may be modified or removed in future versions. Type: `sync | async |off` Default value: same as `preload` URL: Yes Modifiable at runtime: No

Option	Type
	Allows to suppress the preload of component resources (`Component-preload.js`). By default, the component resources are automatically preloaded when preloads are active in general (e.g. when running against the optimized SAPUI5 runtime and not running in debug mode). With this parameter, the preload can be switched off without affecting the library preload. `sync` or `async` has no meaning, both are accepted to be compatible with the library preload, but the code that creates a component decides whether this happens synchronously or asynchronously. .
`xx-debugModuleLoading`	**i Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No When set to `true`, the SAPUI5 module loading feature produces `DEBUG` output for every required, executed, or required but already loaded module. This can help to analyse issues with dependency order, and so on.
`xx-debugRendering`	**i Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` Default value: `false` URL: Yes Modifiable at runtime: No When set to `true`, some components of the SAPUI5 rendering system (`RenderManager`, `UIArea`) create a far more verbose debug output for rendering steps, for example: • Which controls have to be rendered? • Who invalidated the control? (stacktrace) • Was one rendering run sufficient, or have there been multiple runs?
`xx-libraryPreloadFiles`	**i Note** This is an experimental feature and may be modified or removed in future versions. Type: `string[]` Default value: `both` URL: Yes Modifiable at runtime: No

Option	Type
	Allows to enforce the use of a specific preload file type: • for all libraries: `?sap-ui-xx-libraryPreloadFiles=json` • for individual libraries (might be a comma separated list): `?sap-ui-xx-libraryPreloadFiles=sap.m:none,sap.ui.layout:json` • for a combination of both: `?sap-ui-xx-libraryPreloadFiles=both,sap.m:none,sap.ui.layout:js` Possible values for the file types are • `none` (no preload at all) • `json` (only try to load `library-preload.json`) • `js` (only try to load `library-preload.js`) • `both` (first try `js`, then `json`). Any other value will be ignored. The default is `both` for all libraries.
`xx-noless`	**i Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` Default value: `false` URL: Yes (only!) Modifiable at runtime: No Only useful at development time: when set to `true`, the browser-based compilation of LESS theming files is suppressed, only the CSS that is created at built-time will be loaded.
`xx-nosync`	**i Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` | `warn` Default value: `false` URL: Yes Modifiable at runtime: No When set to `warn`, any use of synchronous XHRs will be reported with a warning in the console. When set to `true`, such calls will cause an error.
`xx-showLoadErrors`	**i Note** This is an experimental feature and may be modified or removed in future versions. When executing a loaded module synchronously, some browsers do not provide a proper error location. By setting this configuration parameter to `true`, SAPUI5 can be advised to load a failed script a second time,

Option	Type
	but asynchronously with a script tag. This usually results in an easier to understand syntax error message and a code location.
`xx-supportedLanguages`	**Note** This is an experimental feature and may be modified or removed in future versions. Type: `string[]` Default value: `[]` URL: Yes Modifiable at runtime: No With this option the client can be instructed to limit its backend requests for translatable texts to the configured set of languages. An empty value or the value `*` allows any language, the value `default` limits the requests to the set of languages that are delivered with SAPUI5.
`xx-test-mobile`	**Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` Default value: `false` Modifiable at runtime: No This configuration parameter activates support for mobile device-specific events, such as touch events. This enables you to test standard SAPUI5 controls on mobile devices.
`xx-viewCache`	**Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` Default value: `true` URL: Yes Modifiable at runtime: No Allows to disable the view caching, for example, during development. (See XML View Cache [page 797].)
`xx-waitForTheme`	**Note** This is an experimental feature and may be modified or removed in future versions. Type: `boolean` Default value: `false` URL: Yes

Option	Type
	Modifiable at runtime: No If set to `true`, the first (initial) rendering of the application will be delayed until the theme has been loaded and applied (until `Core.isThemeApplied()`). Helps to avoid FOUC (flash of unstyled content).

Compatibility Version Information

Compatibility version flags allow applications to react to incompatible changes in SAPUI5.

> ⚠ Caution
>
> The concept of compatibility versions has been abandoned as of version 1.28. Therefore, there will be no new compatibility version flags in the future. If you start building a new application please set `data-sap-ui-compatVersion="edge"` on your SAPUI5 bootstrap tag.

As described in the compatibility rules, changes to SAPUI5 features are compatible, see Compatibility Rules [page 17]. In some cases, however, it may make sense to change the behavior of a feature, for example, to change the default values or to use an optimized implementation and these changes may lead to incompatibilities.

> ℹ Note
>
> We recommend to adapt to new feature versions as soon as possible.

The compatibility version configuration works as follows:

- A version flag is introduced if a feature change is incompatible.
- The version flag has to be defined in the SAPUI5 bootstrap tag either globally (`data-sap-ui-compatVersion` or individually for each feature (for example `data-sap-ui-compatVersion-xyz`). Example with `compatVersion "1.18"`

```
<script id="sap-ui-bootstrap"
        type="text/javascript"
        src="resources/sap-ui-core.js"
        data-sap-ui-theme="sap_belize"
        data-sap-ui-libs="sap.m"
        data-sap-ui-compatVersion="1.18"
        data-sap-ui-compatVersion-xyz="1.16"
        >
</script>
```

- If no version is defined, the default behavior of the feature applies.
- If an explicit version is specified, the behavior of the specified version is applied.
- If a version edge is specified, the newest behavior of the feature is applied.

- A fallback mechanism is implemented. The following table is an example of possible configuration options for feature "xyz":

data-sap-ui-compatVersion	data-sap-ui-compatVersion-xyz	Default feature xyz	Resulting compatibility version
--	--	1.14	1.14
1.16	--	1.14	1.16
--	1.16	1.14	1.16
1.18	1.16	1.14	1.16
edge	..	1.14	1.18

SAPUI5 supports the following compatibility version flags:

Flag	Description
`data-sap-ui-compatVersion-flexBoxPolyfill`	The `flexBoxPolyfill` for Internet Explorer 9 was deprecated in 1.16 due to functional deficiencies. When the compatibility version is 1.16 or higher, the polyfill is not active at all. Otherwise, the buggy implementation behaves as before, so that it still works in existing applications. Default value: 1.14
`data-sap-ui-compatVersion-sapMeTabContainer`	The `TabContainer` was deprecated in 1.15. When the compatibility version is 1.16 or higher, an error is logged to the console indicating that `sap.m.IconTabBar` should be used instead. Default value: 1.14
`data-sap-ui-compatVersion-sapMeProgessIndicator`	--
`data-sap-ui-compatVersion-sapMGrowingList`	--
`data-sap-ui-compatVersion-sapMListAsTable`	--
`data-sap-ui-compatVersion-sapMDialogWithPadding`	By default, the content area of `Dialog` had paddings. To make the padding consistent with other popups, the padding is removed for compatibility versions 1.16 or higher. If the padding is still needed inside the content area of `Dialog`, add the CSS style class `sapUiPopupWithPadding` to `Dialog` by calling the `addStyleClass` function. Default value: 1.14
`data-sap-ui-bindingSyntax`	This configuration parameter defines whether the simple or the complex binding syntax is used. The parameter only

Flag	Description
	affects bindings that are defined as strings, for example in the constructor of a control, or when specifying a binding in a declarative view, such as XML view or HTML view. For versions lower than 1.28, the default value is `default` which only has very limited features. As of version 1.28, the default is `complex`.

Related Information

Compatibility Rules [page 17]

Structuring: Components and Descriptor

SAPUI5 provides faceless components for services that deliver data from the back end system, and UI components that extend components and add rendering functionality. The descriptor provides a central, machine-readable and easy-to-access location for storing metadata associated with an application, an application component, or a library.

Components

Components are independent and reusable parts used in SAPUI5 applications.

An application can use components from different locations from where the application is running. Thus, components can be developed by different development teams and be used in different projects. Components also support the encapsulation of closely related parts of an application into a particular component. This makes the structure of an application and its code easier to understand and to maintain.

> **i** Note
>
> Constraints due to cross-origin issues also apply to components.

SAPUI5 provides the following two types of components:

- Faceless components (class: `sap.ui.core.Component`)
  Faceless components do **not** have a user interface and are used, for example, for a service that delivers data from a back-end system.
- UI components (class: `sap.ui.core.UIComponent`)
  UI components extend components and add rendering functionality to the component. They represent a screen area or element on the user interface, for example, a button or a shell, along with the respective settings and metadata. `sap.ui.core.UIComponent` extends `sap.ui.core.Component` and adds rendering functionality to the component.

The `sap.ui.core.Component` class is the base class for UI and faceless components and provides the metadata for both types of components. To extend the functionality, components can inherit from their base class or from another component.

Components are loaded and created via the component factory function `sap.ui.component`. You can either pass the name of the component or the URL of the descriptor file (`manifest.json`) to load it via the descriptor, see Manifest First Function [page 735]. We recommend loading the component using the descriptor (if available) - it improves performance during the initial load since the loading process can be parallelized and optimized.

After loading the descriptor, the component factory can load the dependencies (SAPUI5 libraries and other dependent components) in parallel next to the component preload, and also models can be preloaded.

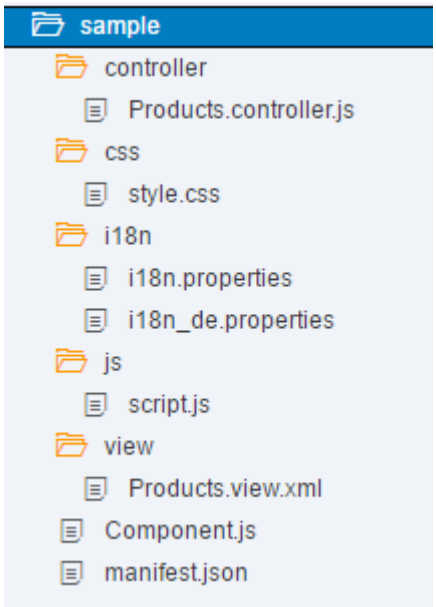
Structure of a Component

A component is organized in a unique namespace, the namespace of the component equals the component name.

Basically, a component consists of the component controller (`Component.js`) and a descriptor (`manifest.json`). Only the component controller is mandatory, but we recommend to also use the descriptor file. The descriptor then contains the component metadata, and also expresses the component dependencies and configuration (see Descriptor for Applications, Components, and Libraries [page 734]). All required and optional resources of the component have to be organized in the namespace of the component.

> **i Note**
>
> Optional resources are, for example, `CSS`, `js`, or `i18n` files, views, and controllers.

The following figure gives an example of a component folder structure.



The `ComponentContainer` control wraps a UI component. You use the `ComponentContainer` control in the SAPUI5 control tree in the same way as any other control.

Differentiation to Other Concepts in SAPUI5

The following list explains how other concepts used in SAPUI5 are distinguished from the SAPUI5 components concept:

- Composite controls
  Both concepts provide a set of controls behind a single interface. Composite controls are intended for reuse within control development and allow to include existing controls in a complex control whereas components are intended for reuse in application development.
- UI library
  The UI library is the deployable unit around controls: Controls are never deployed standalone, but as part of a control library. Components, however, are self-contained and should **not** be used to deploy controls.
- Notepad control
  A notepad control is another way to define a control. Notepad controls have all the characteristics of a control.
- MVC
  The MVC concept allows to define views and controllers and, thus, to structure and reuse parts within an application. As MVC can only be deployed separately and has no means to define dependent styles or scripts that are loaded together with a view, this concept is of limited use across different applications.

Related Information

Descriptor for Applications, Components, and Libraries [page 734]
API Reference: `sap.ui.core.Component`

Component.js File

The `Component.js` file is the component controller and provides the runtime metadata and the component methods.

A component controller is defined with the asynchronous module definition (AMD) syntax. In the `sap.ui.define` statement; the required dependencies can be declared which can be used in the controller.

To create an SAPUI5 component, you extend either the `Component` or `UIComponent` base class and pass the name of the module (namespace + `.Component`).

```
sap.ui.define(['jquery.sap.global', 'sap/ui/core/UIComponent'],
    function(jQuery, UIComponent) {
    "use strict";
    var Component = UIComponent.extend("samples.components.sample.Component", {
        metadata : {
            manifest : "json"
        }
    });
    return Component;
});
```

The metadata of the component controller should be used to declare the runtime metadata only (which are the properties, aggregations, associations and events).

We recommend to define the component metadata externally in the descriptor (`manifest.json`), because the descriptor for components is mandatory for modern components and allows performance optimizations.

Related Information

Using and Nesting Components [page 726]

Component Metadata

The component class provides specific metadata for components by extending the `ManagedObject` class. The `UIComponent` class provides additional metadata for the configuration of user interfaces or the navigation between views.

> **i** Note
>
> With the introduction of the descriptor for applications, components, and libraries, we recommend to migrate the component metadata to the descriptor. The descriptor is inspired by W3C's Web Application Manifest and provides comprehensive information for applications, components and libraries. For more information, see Descriptor for Applications, Components, and Libraries [page 734]. The metadata property `manifest` must be set to `json` to indicate that the `manifest.json` file should be loaded and used:
>
> ```
> sap.ui.core.Component.extend("some.sample.Component", {
>     "metadata": {
>         "manifest": "json"
>     }
> });
> ```
>
> You can also define the descriptor inline by just providing an object. However, we do **not** recommend this because this would prevent that the descriptor can be analyzed by tools.

The metadata defined in `Component.js` is common for faceless components and UI components. The following parameters are available:

- `manifest`: Specifies if your component uses the descriptor
- `abstract`: Specifies if your component class is an abstract class that serves as a base for other components
- `version`: Version of your component; this parameter belongs to the design time metadata and is currently not used; it may be used in the future in the design time repository
- `properties`, `aggregations`, `associations`, and `events`: Define these for your component in the same way as for a control. For more information, see Defining the Control Metadata [page 2188].
- `library`: Specifies the library to which your component belongs to

The following properties are deprecated and no longer needed if you use the descriptor:

- `includes`: Array of strings containing the paths to CSS and JavaScript resources for your component; will be added to the header of the HTML page and loaded by the browser. The resources will be resolved relative to the location of `Component.js`.

- `dependencies`: Used to specify all external dependencies, such as libraries or components. Like the includes for resources that are added to the application's HTML, the dependencies are loaded by SAPUI5 core before the component is initialized. Everything that is referenced here can be used in your component code right from the start. Specify here external dependences such as libraries or components, that will be loaded by SAPUI5 core in the initialization phase of your Component and can be used after it.
  - `libs`: Path to the libraries that should be loaded by SAPUI5 core to be used in your component
  - `components`: Full path to the components that should be loaded by SAPUI5 core to be used in your component
  - `ui5version`: Minimum version of SAPUI5 that the component requires; it helps to be ensure that the features of SAPUI5 runtime used in this component are available. As SAPUI5 currently does not enforce the use of the correct version, it is only used for information purposes.
- `config`: Static configuration; specify the name-value pairs that you need in the component
- `extensions`: Extensions for components and views, see Extending Apps [page 2143]
  - `sap.ui.viewExtensions`: Used for providing custom view content in a specified extension point in the standard application
  - `sap.ui.viewModifications`: Used for overriding control properties in the standard application
  - `sap.ui.viewReplacements`: Used for replacing a standard view with a custom view
  - `sap.ui.controllerExtensions`: Used for extending a controller in a delivered standard application with a custom controller
  - `sap.ui.controllerReplacements`: Used for replacing a controller in a delivered standard application with a custom controller

Example for metadata in `Component.js`:

```
sap.ui.core.Component.extend("some.sample.Component", {
    "metadata": {
        "manifest": "json", // Specifies that your Component class uses the
descriptor via the manifest.json file
        "abstract": true, // Specifies if your Component class is an abstract
one that serves as a base for your other components
        "library": "sap.ui.core", // Specifies the library the component belongs
to
        "version": "1.0", // Version of your Component
        "properties": { // Defined for components in the same way as for a
control or view
            "config": "any"
        }
    }
});
```

In addition to the common metadata for components, the `UIComponent` class provides the following metadata for UI compnents:

- `publicMethods`: Definition of public methods for your component
- `aggregations`: Defines aggregations for your component

The following properties are deprecated and no longer needed if you use the descriptor:

- `rootView`: Can be the view name as string or the view configuration object
- `routing`: Provides the default values for all views
  - `config`: Default values for routing that are applied, if no setting is specified by a route
    - `viewType`: View type of the view that is created, for example XML, JS or HTML
    - `viewPath`: Prefix that is preceding the view

- `targetParent`: ID of the view in which the `targetControl` is searched
- `targetControl`: ID of the control that contains the views
- `targetAggregation`: Name of the aggregation of the `targetControl` that contains views
- `clearTarget`: Boolean; if set to `true`, the aggregation should be cleared before adding the View to it

- `routes`: Contains the configuration objects
  - `name`: Mandatory parameter used for listening or navigating to the route
  - `pattern`: String that is matched against the hash. The {} means this segment of the URL is passed to a handler with the value it contains
  - `view`: Name of the view that is created

Example for UI component metadata:

```
sap.ui.core.UIComponent.extend("some.sample.UIComponent", {
    "metadata": {
        "publicMethods": [ "render" ],
        "aggregations": {
            "rootControl": {
                "type": "sap.ui.core.Control", multiple: false, visibility:
"hidden"
            }
        }
    }
}),
```

Properties Section in Component Metadata

You can add a properties section to the metadata for all properties that can adopt different values during runtime. The getters and setters for these properties are generated automatically, but you can overwrite them if you require additional functionality. The following example contains two properties at the end of the metadata section.

```
sap.ui.core.UIComponent.extend("samples.components.shell.Component", {
    "metadata": {
       "abstract": true,
       "version": "1.0",
[… omitting some lines to make the example shorter]
        "properties": {
          "appTitle": {
               "name":"appTitle",
                "type":"string",
                "defaultValue":"Default Value that will be replaced with
something meaningful through the setter for this property"
            },
            "someOtherProp": {
               "name":"myProperty",
                "type":"string",
                "defaultValue":"Some text"
            }
        }
    }
});
```

The getters and setters for these properties are generated automatically and can be overwritten if additional functionality is required.

Methods Controlling the Initial Instantiation

SAPUI5 provides two methods for the initial instantiation of the component.

You can use the following methods:

- `init`
  Overwrite this method for example to connect the model between the control and the component. This method is **not** called by the application directly, but called automatically when you create the instance of the component. The routing instance needs to be initialized here, see Initializing and Accessing a Routing Instance [page 1081].
- `createContent`
  By default, the UI component creates the `sap.ui5/rootView` declared in the manifest as the root control, see Descriptor Dependencies to Libraries and Components [page 778].
  Alternatively, you can overwrite this method and programmatically create the root control:

```
sap.ui.define(["sap/ui/core/UIComponent", "sap/m/Label"],
function(UIComponent, Label) {
    return UIComponent.extend("my.app.Component", {
        metadata: {
            manifest: "json"
        },
        createContent: function () {
            return new Label({ text: "Hello!" });
        }
    });
});
```

> **i Note**
>
> The configuration properties for a component, that is, the settings given in the constructor or the `sap.ui.core.Component.create` or `sap.ui.component` call, are not available in the `Init` and `createContent` methods. Use `componentData` instead. For more information, see `sap.ui.core.Component.create`.

You can also overwrite the getters and setters for component properties in the `Component.js` file.

Using and Nesting Components

You can use a `ComponentContainer` to wrap a `UIComponent` and reuse it anywhere within the SAPUI5 control tree. With the `ComponentContainer` you can nest components inside other components.

Component Containers

To render UI components, you must wrap them in a `sap/ui/core/ComponentContainer`. You **cannot** use the `placeAt` method to place UI components directly in a page. A `ComponentContainer` carries specific settings and also contains the lifecycle methods of a regular control, such as the `onBeforeRendering` and

`onAfterRendering` methods. The lifecycle methods of the `ComponentContainer` are forwarded to the corresponding methods of the nested component.

The `ComponentContainer` separates the application and the nested component. The control tree and data binding of the inner component are decoupled from the outer component.

If you want to share data with the inner component, you can use the `propagateModel` property on the `ComponentContainer` to forward models and binding contexts to the inner component.

You load and create a `UIComponent` in one of the following ways:

- Load the component asynchronously in "manifest first" mode by specifying the component name:

```
    // "ComponentContainer" required from module "sap/ui/core/
ComponentContainer"
    var oContainer = new sap.ui.core.ComponentContainer({
        name: "samples.components.sample",
        manifest: true,
        async: true
    });
    oContainer.placeAt("target");
```

- Load the component asynchronously before creating the container:

```
    // "Component" required from module "sap/ui/core/Component"
    // "ComponentContainer" required from module "sap/ui/core/
ComponentContainer"
    Component.load({
        name: "samples.components.sample",
    }).then(function(oComponent) {
        var oContainer = new ComponentContainer({
            component: oComponent
        });
        oContainer.placeAt("target");
    });
```

- Load the component asynchronously with "manifest first" mode by specifying the URL of the descriptor (`manifest.json`):

```
    // "Component" required from module "sap/ui/core/Component"
    // "ComponentContainer" required from module "sap/ui/core/
ComponentContainer"
    Component.load({
        manifest: "samples/components/sample/manifest.json",
    }).then(function(oComponent) {
        var oContainer = new ComponentContainer({
            component: oComponent
        });
        oContainer.placeAt("target");
    });
```

> **i Note**
>
> You can use the `lifecycle` property to determine whether the container or your application code will take care of destroying the component.
>
> See `ComponentContainer` for a detailed explanation of the lifecycle property and its possible values.

Using a Component Container to Load Components from a Different Location

You may want to load components from a location that is different from the location where the SAPUI5 libraries are located or a location that is not registered as a resource root in the SAPUI5 bootstrap.

You can do so by defining the URL of the additional components as a setting for the component factory or the component container.

- Loading the component asynchronously before creating the container:

```
    // "Component" required from module "sap/ui/core/Component"
    // "ComponentLifecycle" required from module "sap/ui/core/
ComponentLifecycle"
    Component.load({
        name: "samples.components.sample",
        url: "./myComponents"
    }).then(function(oComponent) {
        var oContainer = new ComponentContainer({
            component: oComponent
        });
        oContainer.placeAt("target");
    });
```

- Loading the component asynchronously when creating the container:

```
    // "ComponentContainer" required from module "sap/ui/core/
ComponentContainer"
    // "coreLibrary" required from module "sap/ui/core/library"
    var oContainer = new ComponentContainer({
        name: "samples.components.sample",
        lifecycle: coreLibrary.ComponentLifecycle.Container,
        async: true,
        url: "./myComponents"
    });
    oContainer.placeAt("target");
```

  Here you use the `lifecycle` property to make sure that the component is destroyed when the container is destroyed.

Reuse Components

To be able to reuse a component, the component has to be declared in the `componentUsages` section of the `manifest.json` descriptor file as follows:

```
"sap.ui5": {
  "componentUsages": {
    "myreuse": {
      "name": "sap.reuse.component",
      "settings": {},
      "componentData": {},
      "lazy": false
    }
  }
}
```

The reuse component is declared via its `componentUsage` ID as the key and the supported values are `name` (name of the component), `settings`, `componentData` and `lazy`. The values defined in the `manifest.json` file will be merged with the values specified in the instance-specific component factory function An exception

to this is the lazy flag which is an indicator for the Component factory function how to handle the dependency. Allowed values in the instance-specific factory function are `settings`, `componentData`, `async`, and `id`.

The `lazy` flag is used to indicate whether the Component should be already preloaded or not. By default, the Components defined in the usage are lazy. A Component preloaded with the flag `lazy: false` has to be explicitly maintained in the `manifest.json`.

For more information, see Descriptor for Applications, Components, and Libraries [page 734].

If you want to exchange the reuse component, for example, to extend an app, you simple exchange the reuse component in the `manifest.json` descriptor file.

The application index can also access the information in the `manifest.json` file and optimize the determination of dependencies when loading components.

Reuse components that are embedded by a library must have an explicit entry in the `manifest.json` in the `sap.app/embeddedBy` section:

```
"sap.app": {
  "embeddedBy": "../"
}
```

Under `embeddedBy`, you specify the relative path to the namespace root of the library. This ensures that tools like the application index can discover embedded libraries and won't include them in the transitive scope (otherwise you would get unwanted 404 requests). Additionally tools should declare a library dependency to the embedding library. This will ensure that the library containing the component preload will be loaded automatically instead of the trying to load the component preload by itself.

Instantiation

To instantiate the reuse component in the current component, you use an instance-specific factory function. The factory function requires at least the `componentUsage ID` as a parameter (simplified usage) or a configuration object that contains the `usage` and optionally `settings` and `componentData` (extended usage).

- Example for simplified usage (Async):

  ```
  this.createComponent("myreuse").then(function(oComponent) {
    // ...
  });
  ```

- Example for extended usage (Async):

  ```
  var oComponentPromise = this.createComponent({
    usage: "myreuse"
    settings: {},
    componentData: {},
    async: true
  });
  ```

Declarative Usage

You can also declare a reuse component directly, for example, in your JavaScript or XML code. In an XML view, the local service factory can only be used via the `ComponentContainer` that has a superordinate component.

```
<View ...>
  <ComponentContainer usage="myreuse" async="true"></ComponentContainer>
</View>
```

Migration

If you have been reusing components before we introduced the reuse feature described above, we recommend that you refactor your code and implement the new logic.

If you use a component that is embedded in a library, and the application declares a dependency to that library, remove the dependency to the library from the embedding application. Make sure that the application code does not contain any direct references to the component or the embedding application.

Old Code

`manifest.json` with dependency declaration only:

```
{
  "sap.ui5": {
    "dependencies": {
      "components": {
        "sap.reuse.component": {}
      }
    }
  }
}
```

Recommended Code

`manifest.json` with declaration of reuse components:

```
{
  "sap.ui5": {
    "dependencies": {
      "components": {
        "sap.reuse.component": {}
      }
    },
    "componentUsages": {
      "reuse": {
        "name": "sap.reuse.component",
        "lazy": false
      }
    }
  }
}
```

> **i Note**
>
> As of version 1.56 it is sufficient to declare the component usage and to indicate whether the component should be lazy loaded or not. The declaration of the component dependencies can and should be avoided in this case.

`Component.js` with nested reuse component:

```
createContent: function() {

  var oReuseComponent =
sap.ui.component({
    "name": "sap.reuse.component"
  });

}
```

`Component.js` that loads the reuse component

```
createContent: function() {

  var oReuseComponentPromise =
this.createComponent({ /* this =
Component instance */
    "usage": "reuse"
  });

}
```

Related Information

Enabling Routing in Nested Components [page 1086]

API Reference: `sap.ui.core.ComponentContainer`

API Reference: `sap.ui.core.ComponentContainer.setLifecycle`

Descriptor for Applications, Components, and Libraries [page 734]

Declarative API for Initial Components

The declarative API enables you to define the initially started component directly in the HTML markup.

Using the `ComponentSupport` Module

With the declarative `sap/ui/core/ComponentSupport` API it is possible to define the initially started component directly in the HTML markup instead of the imperative way using JavaScript. The declarative `ComponentSupport` is not activated by default, but must be enabled via the bootstrap:

```
<!-- index.html -->
<script id="sap-ui-bootstrap"
    src="/resources/sap-ui-core.js"
    ...
    data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
    ...>
</script>
```

This module scans the DOM for HTML elements containing a special data attribute named `data-sap-ui-component`. All DOM elements marked with this data attribute will be regarded as container elements into which a `sap/ui/core/ComponentContainer` is inserted. Additional data attributes are then used to define the constructor arguments of the created `ComponentContainer` instance, e.g. `data-name` for the name of the component which should be instantiated:

```
<!-- index.html -->
<body id="content" class="sapUiBody sapUiSizeCompact" role="application">
    ...
    <div data-sap-ui-component
        data-id="container"
        data-name="sap.ui.core.samples.formatting"
        ...
        data-handle-validation="true"
        ...>
    </div>
    ...
</body>
```

Declarative Configuration of `ComponentContainer`

As HTML is case-insensitive, in order to define a property with upper-case characters, you have to "escape" them with the hyphen character. This is similar to CSS attributes. In the following sample the `handleValidation` argument of the `ComponentContainer` constructor is used:

```
<div data-sap-ui-component ... data-handle-validation="true" ...></div>
```

Asynchronouos loading with `ComponentSupport`

The `ComponentSupport` module enforces asynchronous module loading of the component with "manifest first". This means, that the `manifest.json` file is loaded before evaluating the component to optimize loading behavior. In this way libraries and other dependencies can be loaded asynchronously and in parallel. To achieve this, the following settings for the ComponentContainer are applied by default:

- `async` {*boolean*} (forced to `true`)
- `manifest` {*boolean|string*} (forced to `true` if no string is provided to ensure manifest first)
- `lifecycle` {*sap.ui.core.ComponentLifecycle*} (defaults to `Container`)
- `autoPrefixId` {*boolean*} (defaults to `true`)

For details on the manifest, see Descriptor for Applications, Components, and Libraries [page 734].

See also `ComponentSupport` and `ComponentContainer` for more information.

Delay the Initial Component Instantiation

In some cases, the component initialisation must wait until all pre-required modules have been loaded. If this is the case, the `ComponentSupport` module needs to be executed later, and you have to replace the `onInit` module execution in the bootstrap with a custom module:

```
<!-- index.html -->
<script id="sap-ui-bootstrap"
    src="resources/sap-ui-core.js"
    data-sap-ui-onInit="module:sap/ui/demo/myBootstrap"> <!-- Execute custom
module on init -->
</script>
```

The custom module can load dependencies and execute code before activating the `ComponentSupport` module:

```
// sap/ui/demo/myBootstrap.js
sap.ui.define(["sap/ui/demo/MyModule"], function(MyModule) {
    // Execute code which needs to be executed before component initialization
    MyModule.init().then(function() {
        // Requiring the ComponentSupport module automatically executes the
component initialisation for all declaratively defined components
        sap.ui.require(["sap/ui/core/ComponentSupport"]);
    });
});
```

Handling IDs in UI Components

Components are usually used with a root view and in this case, the component handles the prefixing of IDs of views, elements, or controls, with the component ID.

This works similar to the prefixing of control IDs in XML views, see Support for Unique IDs [page 814]. However, if you implement your own `createContent` function, you need to handle this yourself. The following two options exist:

- Set the `sap.ui5/autoPrefixId` attribute in the `manifest.json` file to `true`. This is the easiest option.
- Use the `createId` function of the UI component to prefix the respective ID of a view, element, or control yourself.

Use the `byId` function of the UI component to retrieve the views, controls, and elements that have been created in a UI component.

Advanced Concepts for SAPUI5 Components

Advanced concepts for components include routing and navigation and component data as well as the event bus.

The following advanced concepts for components exist.

- Routing and navigation
  UI components support the routing and navigation concept, see Initializing and Accessing a Routing Instance [page 1081].
- Extensibility and customizing
  The extensibility and customizing concept allows you to extend and modify components in order to replace and extend the views and controllers as well as to modify the views. A customization can be performed, for example, on a custom application that extends a delivered standard application.
  For more information, see Extending Apps [page 2143]
- Component data
  The JSON object `ComponentData` contains any initial values of parameters that can be used in the `createComponent()` method. Component data are already available for use in the `createComponent()` method, but not the parameters. The parameters are available in the `onBefore`, the `onAfterRendering` and the setter methods of the parameters.
  Component data is provided from outside and can be configured as desired. Configuration data is static and defined in the component. To change or extend the configuration, the component needs to be extended and a new configuration has to be created and merged with the configuration in the parent component.
- Event bus of the component
  The local event bus belongs to the component and can be used by all children of this component. Once a component instance is destroyed, the listeners registered in the event bus are destroyed automatically. For more information, see API Reference for the `getEventBus` method of `sap.ui.core.Component`.

Descriptor for Applications, Components, and Libraries

The descriptor for applications, components, and libraries is inspired by the Web Application Manifest concept introduced by the W3C. The descriptor provides a central, machine-readable and easy-to-access location for storing metadata associated with an application, an application component, or a library.

The data is stored in json format in the `manifest.json` file. The developer creates the file with attributes in different namespaces. It contains, for example, the app ID, the version, the data sources used, along with the required components and libraries. The existence of the `manifest.json` file must be declared in the component metadata, which is then delivered as part of the application archive. After delivery, the file is read-only.

General Information

Every new version of SAPUI5 implies a new version of the app descriptor. In the following table you can see how the SAPUI5 version is related to the descriptor version and the value of `_version`.

Table 11: AppDescriptor Release and SAPUI5 Version

AppDescriptor Release	SAPUI5 Version	_version
Version 2	>=1.30	1.1.0
Version 3	>=1.32	1.2.0
Version 4	>=1.34	1.3.0
Version 5	>=1.38	1.4.0
Version 6	>=1.42	1.5.0
Version 7	>=1.46	1.6.0
Version 8	>=1.48	1.7.0
Version 9	>=1.50	1.8.0
Version 10	>=1.52	1.9.0
Version 11	>=1.54	1.10.0
Version 12	>=1.56	1.11.0
Version 13	>=1.58	1.12.0
Version 14	>=1.61	1.13.0
Version 15	>=1.62	1.14.0
Version 16	>=1.66	1.15.0

AppDescriptor Release	SAPUI5 Version	_version
Version 17	>=1.70	1.16.0
Version 18	>=1.71	1.17.0

For more information on the new fields introduced in each version, check out Migration Information for Upgrading the Descriptor File [page 775]

Manifest First Function

The component factory function `Component.create`, as introduced with 1.58, loads the `manifest.json` by default before the component instance is created. With this, you can preload the dependencies (libraries and components) and, thus, improve the performance for loading the component. The preload is also available for models, which can be flagged for preload during component loading.

The `manifest` option allows to configure when and from where the descriptor is loaded:

- Default, equivalent to setting `manifest` to `true`.

```
// "Component" required from module "sap/ui/core/Component"
// load manifest.json from default location and evaluate it before creating
an instance of the component
Component.create({
  name: "sap.my.component",
});
```

- Specify an alternative URL as parameter for `manifest` for the component factory function:

```
// "Component" required from module "sap/ui/core/Component"
// load via manifest URL
Component.create({
  name: "sap.my.component",
  manifest: "any/location/sap/my/component/manifest.json"
});
```

- Suppress loading the manifest by setting the `manifest` flag to `false`, for example, when a legacy component does not have one:

```
// "Component" required from module "sap/ui/core/Component"
// load component without loading a manifest first
Component.create({
  name: "sap.my.component",
  manifest: false
});
```

> **i Note**
>
> When you enable `manifest`, all legacy component metadata needs to be migrated into the descriptor for applications/components. Only those entries in the descriptor for components will be respected by the component and all other entries will be ignored.

Descriptor Content

> **i Note**
>
> You can find an example `manifest.json` file with sample code for the descriptor content here [page 749].

The content for the descriptor is contained in the following namespaces: `without`, `sap.app`, `sap.ui`, `sap.ui5`, `sap.platform.abap`, `sap.platform.hcp` and `sap.fiori`. The following tables show the application-specific attributes provided by the respective namespaces:

No Namespace

Table 12: Attributes in the `without` namespace

Attribute	Description
`start_url`	Start page of your app, if available

`sap.app`

Table 13: Attributes in the mandatory `sap.app` namespace

Attribute	Description
`id`	Mandatory attribute: Unique identifier of the app, which must correspond to the component name **i Note** The ID must not exceed 70 characters. It must be unique and must correspond to the component name. This is checked in consistency check reports, for example, for the SAPUI5 application index which returns an error in case of duplicate IDs, see SAPUI5 Application Index [page 1525].
`type`	Possible values: • `application` • `component` • `library` • `card`

Attribute	Description
`i18n`	Relative URL to the properties file that contains the text symbols for the descriptor; default: `"i18n/i18n.properties"` **i Note** The path to the i18n file must not exceed 100 characters.
`applicationVersion`	Mandatory version of the app (semantic version with the following format `major.minor.patch`)
`embeds`	Array of relative paths to the nested `manifest.json` files; attribute is mandatory if a nested `manifest.json` exists
`embeddedBy`	Relative path back to the `manifest.json` file of an embedding component or library; attribute is mandatory for a nested `manifest.json`
`title`	Mandatory attribute: The entry is language-dependent and specified via `{{...}}` syntax
`subTitle`	Language-dependent entry for a subtitle; specified via `{{...}}` syntax
`shortTitle`	Short version of the title. Language-dependent entry has to be specified via `{{...}}` syntax
`info`	Needed for CDM (Common Data Model) conversion of tiles. Language-dependent entry has to be specified via `{{...}}` syntax
`description`	Description; language-dependent entry that is specified via `{{...}}` syntax
`tags`	Contains the following: • An array of language-dependent `keywords` that are specified via `{{...}}` syntax, for example `"keywords": ["{{keyWord1}}","{{keyWord2}}"]`. • An array of `technicalAttributes` (general technical attributes, for example, technical catalog, upper case and language-independent attributes).

Attribute	Description
`ach`	Application component hierarchy (SAP's component names for bug reports); attribute is mandatory for SAP apps, but is not used so far for apps developed outside SAP

`dataSources`	Unique key/alias for specifying the used data sources; contains the following information:
	• `uri`: Mandatory relative URL in the component; takes `embeddedBy` into account, if filled, or the server absolute of the data source, for example `"/sap/opu/odata/snce/PO_S_SRV;v=2/"`
	• `type`: OData `(default)or ODataAnnotation or INA or XML or JSON`
	• `settings`: Data source type-specific attributes (key, value pairs), which are:
	  ○ `odataVersion`: 2.0 (default), 4.0
	  ○ `localUri`: Relative URL to local metadata document or annotation uri
	  ○ `annotations`: Array of annotations which references an existing data source of type "ODataAnnotation" under **sap.app/dataSources**
	  ○ `maxAge`: Indicates the number of seconds the client is willing to accept with regard to the age of the data that is requested

`cdsViews`	Array of directly used CDS views This attribute is optional and only added if used via INA protocol directly, not if used via OData service.
`offline`	Indicates whether the app is running offline; default is `false` (online)
`sourceTemplate`	If an app has been generated from a template, this attribute is filled automatically by the generation tool (SAP Web IDE): • `id`: Mandatory ID of the template from which the app was generated • `version`: Mandatory version of the template from which the app was generated

Attribute	Description
`openSourceComponents`	Array of directly used open source libraries for documentation purposes; not used when open source libraries are used via SAPUI5 capsulation • `name`: Mandatory name of the open source component • `version`: Required if the open source component is part of the app; not required if the open source component is part of the SAPUI5 dist layer • `packagedWithMySelf`: Indicates if the open source component is part of the app (`true`) or not (`false`)
`provider`	Name of the provider that owns the application. Current supported enum value is `sfsf`.

Attribute	Description
`crossNavigation`	Cross-navigation for specifying inbounds and outbounds • `scopes`: Scope of a site `sapSite` • `inbounds`: Unique key or alias to specify inbounds (mandatory); contains: ○ `semanticObject` (mandatory) ○ `action` (mandatory) ○ `icon`: Used to overwrite `sap.ui/icons/icon` ○ `title`: Used to overwrite `sap.app/title` (language-dependent entry to be specified via {{...}} syntax) ○ `subTitle`: Used to overwrite `sap.app/subTitle` (language-dependent entry to be specified via {{...}} syntax) ○ `shortTitle`: Used to overwrite `sap.app/shortTitle` (language-dependent entry to be specified via {{...}} syntax) ○ `info`: Language-dependent entry to be specified via {{...}} syntax ○ `displayMode`: <`ContentMode` or `HeaderMode`> Display mode for an inbound which specifies what kind of tile is displayed. A static tile can be displayed in content mode or header mode. The tile in header mode is a text only tile without an icon which allows longer title and subtitle. ○ `hideLauncher` (`true/false`): Indicates that an inbound must not be represented as a tile/link ○ `indicatorDataSource`; specifies the data source; contains: ○ `dataSource`: reference to `sap.app/dataSources` (mandatory) ○ `path`: Relative path to `sap.app/dataSources uri` (mandatory) ○ `refresh`: Defines the refresh interval ○ `deviceTypes`: Contains objects with device types on which the app is running; if empty, use the default from `sap.ui/deviceTypes`; the following device types can be defined (`true/false`): ○ `desktop` ○ `tablet`

Attribute	Description
	○ `phone` ○ `signature`: Specifies the signature; contains: ○ `parameters` (mandatory): Contains parameter names with the following information: ○ `required` (`true/false`) ○ `filter`: Represents the filter only if the input parameter matches the filter; with a mandatory `value` and `format` attribute ("plain", "regexp", "reference") ○ `defaultValue`: Specifies the default value; has mandatory attributes `value` (depending on the format this is a verbatim default value) and `format` ("plain", "reference"). If the `format` is "reference", the syntax for the `value` is as follows: `"UserDefault.<parameterName>"` for single-value parameters, `"UserDefault.extended.<parameterName>"` for sets of values and value ranges, or `"User.env.<parameterName>"` for supported user-specific settings. ○ `renameTo`: Used for parameter mapping to specify the parameter name in legacy ABAP applications, for example, `RF05L_BUKRS` for the `CompanyCode` parameter ○ `launcherValue`: Represents a value to be used when creating an tile intent for this inbound with value and format ("plain", "array") ○ `additionalParameters` (mandatory): Indicates how additional parameters to the declared signature are handled; values can be, for example, "ignored", "notallowed", "allowed" • `outbounds`: Specifies outbounds with a unique key or alias containing: ○ `semanticObject` (mandatory) ○ `action` (mandatory) ○ `parameters`: Specifies the parameter name

Attribute	Description
	○ `value`: Represents a value to be used in the outbound; with `value` (verbatim value for format "plain", or not supplied, or a binding reference for format "binding") and `format` (indicates how value is to be interpreted, "plain", "binding") ○ `required`: Indicator whether paramter is required (`true`, `false`) ○ `additionalParameters`: Indicates whether additional context parameters are to be used: ○ `ignored`: Parameters are not used ○ `allowed`: Parameters are passed on to application

`sap.ui`

Table 14: Attributes in the mandatory `sap.ui` namespace

Attribute	Description
`technology`	Specifies the UI technology; value is `UI5`
`icons`	Contains object with app-specific icons, which are: • `icon`: Icon of the app, can be chosen from Icon Explorer . • `favIcon`: ICO file to be used inside the browser and for desktop shortcuts **i Note** `favIcon` is not set automatically by the framework. The icons can be set manually using the `sap/ui/util/Mobile` module and the `setIcons` function. • `phone`: 57x57 pixel version for non-retina iPhones • `phone@2`: 114x114 pixel version for retina iPhones • `tablet`: 72x72 pixel version for non-retina iPads • `tablet@2`: 144x144 pixel version for retina iPads

Attribute	Description
`deviceTypes`	Mandatory; contains objects with device types on which the app is running, such as: • `desktop`: Indicator for whether desktop devices are supported, `true` (default), `false` • `tablet`: Indicator for whether tablet devices are supported, `true` (default),`false` • `phone`: Indicator for whether phone devices are supported, `true` (default),`false`
`supportedThemes`	Optional; array of supported SAP themes, such as `sap_hcb`, `sap_belize`
`fullWidth`	Indicates whether an app shall run in full screen mode (`true`), or not (`false`)

`sap.ui5`

The `sap.ui5` namespace is aligned with the previous component metadata and contributes the following SAPUI5-specific attributes for the application descriptor, see Migrating from Component Metadata to Descriptor [page 756] for more details.

Table 15: Attributes in the `sap.ui5` namespace

Attribute	Description
`resources`	Relative URLs in the component, taking `embeddedBy` into account if filled, pointing to `js` (JavaScript) and `css` resources that are needed by the app for specifying the mandatory `uri` and an `id` (optional) for CSS. The JavaScript files are loaded by the `require` mechanism. The CSS files are added to the head of the HTML page as a link tag. The resources are resolved relative to the location of the `manifest.json` file.

Attribute	Description
`dependencies`	Mandatory; specifies the external dependencies that are loaded by the SAPUI5 core during the initialization phase of the component and used afterwards. These are the following libraries or components: • `minUI5Version`: Mandatory; Minimum version of SAPUI5 that your component requires; this information ensures that the features of the SAPUI5 runtime version of the component are available. As SAPUI5 does not currently enforce use of the correct version, the `minUI5Version` is used for information purposes only. If the minimum SAPUI5 version criteria is not fulfilled, a warning is issued in the console log. • `libs`: ID (namespace) of the libraries that the SAPUI5 core should load for use in the component. If your app requires a minimum version of the lib, specify the `minVersion` for information purposes. Specify `lazy` to indicate that the lib shall be lazy loaded. • `components`: ID (namespace) of the components that the SAPUI5 core should load for use in your component. If your app requires a minimum version of the component, specify the `minVersion` for information purposes. Specify `lazy` to indicate that the component shall be lazy loaded. For more information, see Descriptor Dependencies to Libraries and Components [page 778].
`componentUsages`	Specifies the used components with the a unique key/alias. Contains the following: • `name`: Mandatory name of the reuse component • `settings`: Settings of the component • `componentData`: Component data of the component • `lazy`: Indicates whether the component usage should be lazily loaded. Default value: `true` For more information see:Using and Nesting Components [page 726]

Attribute	Description
`models`	Defines models that should be created or destroyed along the component's lifecycle. The key represents the model name. Use an empty string ("") for the default model. • `type`: Model class name • `uri`: Relative URL in the component, taking `embeddedBy` into account if filled, or server for absolute model • `settings`: Object that is passed to the model constructor. **Example** You can overwrite the default binding mode with the `defaultBindingMode` attribute (enumeration of type `sap.ui.model.BindingMode`, with values. `Default`, `OneTime`, `OneWay`, `TwoWay`). For OData models constructor see the following: ○ sap.ui.model.odata.ODataModel ○ sap.ui.model.odata.v2.ODataModel ○ sap.ui.model.odata.v4.ODataModel For ResourceModel constructor see: ○ sap.ui.model.resource.ResourceModel The attribute `enhanceWith` can be specified with `bundleUrl`, `bundleUrlRelativeTo` (default: `component` or `manifest`) or `bundleName` to provide a list of additional resource bundle configurations to enhance the `ResourceModel` with. • `dataSource`: String of key or alias from `sap.app dataSources` to reference an existing data source; the `type`, `uri` and `settings` properties are set according to the data source's `type`, `uri` and `settings` (if not already defined). If the type under `sap.app dataSources` is `OData`, an OData Model V2 is created automatically. If you need an OData Model V1, specify the `type` as well. • `preload`: Optional; Boolean with `true`, `false` (default) Defines whether or not the model is initialized (preloaded) before the component instance is created and while loading the component preload and its dependencies. For more information, see Manifest Model Preload [page 781].

Attribute	Description
`rootView`	Specifies the root view that shall be opened; can be the view name as a string for XML views, or the view configuration object with `viewName` for the view name as a string and `type` for the type (enumeration of sap.ui.core.mvc.ViewType), **id**, **async** and other properties of `sap.ui.core.mvc.view`.
`autoPrefixId`	true, false (default), Enables the auto prefixing for the UIComponent for IDs of ManagedObjects (controls or elements) which are created in the context of the `createContent` function, or any other invocation of the `Component.prototype.runAsOwner()` function (for example a component's router uses this method when creating new views). In former SAPUI5 releases this prefixing of the ID needed to be done with `oComponent.createId` by overwriting the method `getAutoPrefixId`. The same can now be achieved declaratively by setting `autoPrefixId` to true.
`handleValidation`	Possible values: `true` or `false` (default); used to enable or disable validation handling by the message manager for this component, see Error, Warning, and Info Messages [page 1063]
`config`	Static configuration; specify the name-value pairs that you need in your component.
`routing`	Provides configuration parameters for route and router, see Routing and Navigation [page 1072]
`extends`	Used to extend another component. • `component`: ID (namespace) of the component being extended • `minVersion`: Specifies the minimum version of the component being extended, for information purposes if your app requires a minimum version of the component • `extensions`: Component or view extensions, which enable you to replace and extend views and controllers and also to modify the views, see Extending Apps [page 2143]

Attribute	Description
`contentDensities`	Mandatory; contains an object with the content density modes that the app supports, see Content Densities [page 1142] • `compact`: Mandatory; indicates whether compact mode is supported (`true`, `false`) • `cozy`: Mandatory; indicates whether cozy mode is supported (`true`, `false`)
`resourceRoots`	Map of URL locations keyed by a resource name prefix; only relative paths inside the component are allowed and no ".." characters This attribute is intended for actual sub-packages of the component only, meaning that it must not be used for the component namespace itself. **i Note** When loading with *manifest first* (by using the property `manifest`), the `resourceRoots` are evaluated before the component controller is loaded. Otherwise, the defined resource roots will be registered after the component controller is loaded and do not affect the modules being declared as dependencies in the component controller.
`componentName`	Name of the SAPUI5 component
`appVariantIdHierarchy`	Needed for an app variant scenario to reference UI flex changes from layers below. An array of `appVariantId` hierarchy with origin layer and version, calculated attribute and filled automatically during variant merge.
`i18n`	Determines if the library contains a i18n resource or not. If using a string instead of a boolean value, an alternative name for the i18n resource could be defined. **i Note** If your SAP Fiori library benefits from the new attribute `sap.ui5/library/i18n`, adoption is recommended. This is only the case if the main resource bundle (properties file) used by the SAP Fiori Library is different than the default name `messagebundle.properties`

Attribute	Description
`flexEnabled`	Determines whether the app is enabled for adaptation flex enabled (for example, using stable IDs); possible values are `true`, `false` or `undefined` (default) For more information, see SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450].
`commands`	Specifies provided commands with a unique key/alias. Contains: • `shortcut`: String that describes a key combination. When the user presses the key combination, the command is triggered.

`sap.platform.abap`

Table 16: Attributes in the `sap.platform.abap` namespace

Attribute	Description
`uri`	Specifies the app's URI in the ABAP system, for example `/sap/bc/ui5_ui5/sap/appName`; filled during deployment.

`sap.platform.hcp`

Table 17: Attributes in the `sap.platform.hcp` namespace

Attribute	Description
`uri`	Specifies the URI inside the SAP Cloud Platform HTML5 application; filled during deployment, default is ""
`providerAccount`	Specifies the name of the provider account; filled during deployment
`appName`	Specifies the name of the deployed HTML5 application; filled during deployment
`appVersion`	Specifies the version of the deployed HTML5 application; filled during deployment

`sap.fiori`

Table 18: Attributes in the `sap.fiori` namespace

Attribute	Description
`registrationIds`	Array of registration IDs, for example, the Fiori IDs for SAP Fiori apps

Attribute	Description
`archeType`	Mandatory archetype of the app, possible values `transactional`, `analytical`, `factsheet`, `reusecomponent`, `fpmwebdynpro`, `designstudio`

`sap.card`

Table 19: Attributes in the `sap.card` namespace

Attribute	Description
`type`	Describes the card type; possible values are `list` and `analytical`
`header`	Specifies the card's header area
`content`	Specifies the type-dependent card content

`_version`

- On root level (no namespace) Describes the descriptor format version (mandatory). Needs to be updated when migrating to a new descriptor format version, see Migrating from Component Metadata to Descriptor [page 756]
- Inside namespace: Describes the namespace format version (optional from version 1.38 on)

Example

Current version of the `manifest.json`

```
{
    "_version": "1.19.0",

    "start_url": "index.html",

    "sap.app": {
        "id": "sap.fiori.appName",
        "type": "application",
        "i18n": "",
        "applicationVersion": {
            "version": "1.2.2"
        },
        "embeds": ["mycomponent1", "subpath/mycomponent2"],
        "embeddedBy": "../../",
        "title": "{{title}}",
        "subTitle": "{{subtitle}}",
        "shortTitle": "{{shorttitle}}",
        "description": "{{description}}",
        "info": "{{info}}",
        "tags": {
            "keywords": ["{{keyWord1}}", "{{keyWord2}}"],
            "technicalAttributes": ["ATTRIBUTE1", "ATTRIBUTE2"]
        },
        "ach": "PA-FIO",
        "dataSources": {
```

```
            "equipment": {
                "uri": "/sap/opu/odata/snce/PO_S_SRV;v=2/",
                "type": "OData",
                "settings": {
                    "odataVersion": "2.0",
                    "annotations": ["equipmentanno"],
                    "localUri": "model/metadata.xml",
                    "maxAge": 360
                }
            },
            "equipmentanno": {
                "uri": "/sap/bc/bsp/sap/BSCBN_ANF_EAM/
BSCBN_EQUIPMENT_SRV.anno.XML",
                "type": "ODataAnnotation",
                "settings": {
                    "localUri": "model/annotations.xml"
                }
            }
        },
        "cdsViews": [
            "VIEW1", "VIEW2"
        ],
        "resources": "resources.json",
        "offline": true,
        "sourceTemplate": {
            "id": "sap.ui.ui5-template-plugin.1worklist",
            "version": "1.0.0"
        },
        "destination": {
            "name": "SAP_ERP_FIN"
        },
        "openSourceComponents": [{
            "name": "D3.js",
            "packagedWithMySelf": false
        }],
        "crossNavigation": {
            "scopes": {
                "sapSite": {
                    "value": "123"
                }
            },
            "inbounds": {
                "contactCreate": {
                    "semanticObject": "Contact",
                    "action": "create",
                    "icon": "sap-icon://add-contact",
                    "title": "{{title}}",
                    "subTitle": "{{subtitle}}",
                    "shortTitle": "{{shorttitle}}",
                    "info": "{{info}}",
                    "displayMode": "HeaderMode",
                    "indicatorDataSource": {
                        "dataSource": "equipment",
                        "path": "TaskListSet/$count",
                        "refresh": 5
                    },
                    "deviceTypes": {
                        "desktop": true,
                        "tablet": true,
                        "phone": false
                    },
                    "signature": {
                        "parameters": {
                            "id": {
                                "required": true
                            },
                            "ContactName": {
                                "defaultValue": {
```

```
                                "value": "anonymous"
                            },
                            "required": false,
                            "renameTo": "NAME2"
                        },
                        "Gender": {
                            "filter": {
                                "value": "(male)|(female)",
                                "format": "regexp"
                            },
                            "required": true,
                            "renameTo": "SEX",
                            "launcherValue": {
                                "value": "female",
                                "format": "plain"
                            }
                        }
                    },
                    "additionalParameters": "ignored"
                }
            },
            "contactDisplay": {
                "semanticObject": "Contact",
                "action": "display",
                "signature": {
                    "parameters": {
                        "id": {
                            "required": true
                        },
                        "Language": {
                            "filter": {
                                "value": "EN"
                            },
                            "required": true
                        },
                        "SomeValue": {
                            "filter": {
                                "value": "4711"
                            }
                        },
                        "GLAccount": {
                            "defaultValue": {
                                "value": "1000"
                            },
                            "filter": {
                                "value": "(1000)|(2000)",
                                "format": "regexp"
                            }
                        }
                    }
                }
            },
            "contactDisplayAlt": {
                "semanticObject": "Contact",
                "action": "display",
                "hideLauncher": true,
                "signature": {
                    "parameters": {
                        "GLAccount": {
                            "defaultValue": {
                                "value": "UserDefault.GLAccount",
                                "format": "reference"
                            },
                            "filter": {
                                "value": "\\d+",
                                "format": "regexp"
                            },
                            "required": true
```

```
                            },
                            "SomePar": {
                                "filter": {
                                    "value": "UserDefault.CostCenter",
                                    "format": "reference"
                                },
                                "required": true
                            }
                        }
                    }
                }
            },
            "outbounds": {
                "addressDisplay": {
                    "semanticObject": "Address",
                    "action": "display",
                    "additionalParameters": "ignored",
                    "parameters": {
                        "CompanyName": {
                            "value": {
                                "value": "companyName",
                                "format": "plain"
                            },
                            "required": true
                        }
                    }
                },
                "companyDisplay": {
                    "semanticObject": "Company",
                    "action": "display",
                    "additionalParameters": "allowed",
                    "parameters": {
                        "CompanyName": {
                            "value": {
                                "value": "companyName",
                                "format": "plain"
                            },
                            "required": true
                        }
                    }
                }
            }
        },

        "sap.ui": {
            "technology": "UI5",
            "icons": {
                "icon": "sap-icon://add-contact",
                "favIcon": "icon/F1373_Approve_Purchase_Orders.ico",
                "phone": "icon/launchicon/57_iPhone_Desktop_Launch.png",
                "phone@2": "icon/launchicon/114_iPhone-Retina_Web_Clip.png",
                "tablet": "icon/launchicon/72_iPad_Desktop_Launch.png",
                "tablet@2": "icon/launchicon/144_iPad_Retina_Web_Clip.png"
            },
            "deviceTypes": {
                "desktop": true,
                "tablet": true,
                "phone": false
            },
            "fullWidth": true
        },

        "sap.ui5": {
            "resources": {
                "js": [{
                    "uri": "component.js"
                }],
```

```
            "css": [{
                "uri": "component.css",
                "id": "componentcss"
            }]
        },
        "dependencies": {
            "minUI5Version": "1.66.0",
            "libs": {
                "sap.m": {
                    "minVersion": "1.34.0"
                },
                "sap.ui.commons": {
                    "minVersion": "1.34.0",
                    "lazy": true
                }
            },
            "components": {
                "sap.ui.app.other": {
                    "minVersion": "1.1.0",
                    "lazy": true
                }
            }
        },
        "componentUsages": {
            "myusage": {
                "name": "my.used",
                "lazy": false,
                "settings": {},
                "componentData": {}
            }
        },
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "uri": "i18n/i18n.properties",
                "settings": {
                    "enhanceWith": [{
                        "bundleUrl": "i18n/i18n.properties",
                        "bundleUrlRelativeTo": "manifest"
                    }]
                }
            },
            "equipment": {
                "preload": true,
                "dataSource": "equipment",
                "settings": {}
            }
        },
        "rootView": {
            "viewName": "sap.ui.test.view.Main",
            "id" : "rootView",
            "async": true,
            "type": "XML"
        },
        "handleValidation": true,
        "config": {

        },
        "routing": {

        },
        "extends": {
            "component": "sap.fiori.otherApp",
            "minVersion": "0.8.15",
            "extensions": {}
        },
        "contentDensities": {
            "compact": true,
```

```
            "cozy": false
        },
        "resourceRoots": {
            ".myname": "./myname"
        },
        "componentName": "sap.fiori.appName",
        "autoPrefixId": true,
        "appVariantId": "hcm.leaverequest.oil",
        "appVariantIdHierarchy": [
            {"layer": "VENDOR", "appVariantId": "abc", "version": "1.0.0"}
        ],
        "services": {
            "myLocalServiceAlias": {
                "factoryName": "sap.ushell.LaunchPadService",
                "optional": true
            }
        },
        "library": {
            "i18n": true
        },
                "flexEnabled": true,
        "commands": {
            "Save": {
                "shortcut": "Ctrl+S"
            }
        }
        },

    "sap.platform.abap": {
        "uri": "/sap/bc/ui5_ui5/sap/appName",
        "uriNwbc": ""
    },

    "sap.platform.hcp": {
        "uri": "",
        "uriNwbc": "",
        "providerAccount": "fiori",
        "appName": "sapfioriappName",
        "appVersion": "1.0.0",
        "multiVersionApp": true
    },

    "sap.fiori": {
        "registrationIds": [
            "F1234"
        ],
        "archeType": "transactional"
    },

    "sap.mobile": {},
    "sap.flp": {},
    "sap.ui.generic.app": {},
    "sap.ovp": {},
    "sap.ui.smartbusiness.app": {},
    "sap.wda": {},
    "sap.gui": {},
    "sap.cloud.portal": {},
    "sap.apf": {},
    "sap.platform.cf": {},
    "sap.copilot": {},
    "sap.map": {},
    "sap.fe": {},
    "sap.url": {},
    "sap.platform.sfsf": {},
    "sap.wcf": {},
    "sap.cloud": {},
    "sap.integration": {},
    "sap.card": {},
```

```
    "sap.platform.mobilecards": {}
}
```

For the following namespaces, the indicated teams are responsible:

- sap.mobile - in Mobile responsibility
- sap.flp - in SAP Fiori Launchpad responsibility
- sap.ui.generic.app - in SAP Fiori Elements responsibility
- sap.ovp - in Overview Page responsibility
- sap.ui.smartbusiness.app - in Smart Business responsibility
- sap.wda - in Web Dypro ABAP responsibility
- sap.gui - in SAP GUI responsibility
- sap.cloud.portal - in SAP Cloud Portal responsibility
- sap.apf - in Analysis Path Framework responsibility
- sap.platform.cf - in Cloud Foundry/XSA responsibility
- sap.map - in SAP Visual Business responsibility
- sap.fe - in SAP Fiori Elements responsibility
- sap.url - in SAP Fiori Launchpad responsibility
- sap.platform.sfsf - for SAP SuccessFactors specific attributes
- sap.wcf - for WCF Application specific attributes
- sap.cloud - for SAP Cloud Platform specific attributes
- sap.card - in SAPUI5 responsibility
- sap.platform.mobilecards - in Mobile Cards responsibility

Declaration in Component Metadata

The component declares the existence of the application descriptor by specifying `manifest: "json"` in the component metadata. Setting this flag makes the component load the `manifest.json` file and read the relevant entries for SAPUI5. This metadata is used to define the dependencies that need to be loaded in order to start the component. The following code snippet shows how to add the manifest link:

```
sap.ui.define(['sap/ui/core/UIComponent'], function(UIComponent) {

    return UIComponent.extend("sap.samples.Component", {
        metadata  : {
            manifest: "json"
        }
    });
});
```

SAPUI5 API

At runtime, the `manifest.json` content can be accessed from the component via the component metadata:

```
// get the component class
sap.ui.require(['sap/samples/Component'], function(SampleComponent) {
    // getting complete manifest from component metadata
```

```
    SampleComponent.getMetadata().getManifest();
    //or getting a namespace
    SampleComponent.getMetadata().getManifestEntry("sap.app");

});
```

Related Information

sap.ui.core.UIComponent
Component Metadata [page 723]

Migrating from Component Metadata to Descriptor

Overview, how the component metadata are mapped to the descriptor.

For compatibility reasons, the mapping to the `manifest.json` file is done automatically. If a metadata property has been defined, it can also be consumed via the corresponding property of the `manifest.json` file. For a detailed step-by-step guide, see Creating a Descriptor File for Existing Apps [page 764].

> **i Note**
>
> To benefit from the performance improvements that can be achieved by using “manifest first”, we recommend to migrate the component metadata to the descriptor (`manifest.json`). For more information about manifest first, see the *Manifest First Function* section in Descriptor for Applications, Components, and Libraries [page 734].

Table 20: Mapping Table

Metadata	Descriptor	Comment
Component namespace	`sap.app/id`	-
`version`	`sap.app/applicationVersion/version`	-
`config`	`sap.ui5/config`	-
`dependencies`	`sap.ui5/depedencies`	Different format, see *Dependencies* section below
`customizing`	`sap.ui5/extends/extensions`	-
`handleValidation`	`sap.ui5/handleValidation`	-
`includes`	`sap.ui5/resources`	Different format, see *Resources* section below

Metadata	Descriptor	Comment
`rootView`	`sap.ui5/rootView`	-
`routing`	`sap.ui5/routing`	-

Dependencies

Libraries and components are objects and not arrays. For the descriptor part, we use `ui5version` instead of `minUI5Version`.

Metadata

```
"dependencies": {
    "ui5version": "1.30.0",
    "libs": [
        "sap.m",
        "sap.ui.unified"
    ],
    "components": [ "sap.app.otherComponent" ]
}
```

Descriptor

```
"dependencies": {
    "minUI5Version": "1.30.0",
    "libs": {
        "sap.m": {},
        "sap.ui.unified": {}
    },
    "components": {
        "sap.app.otherComponent": {}
    }
}
```

Resources

Includes are renamed to resources and are objects and not an array.

Metadata

```
"includes": ["script.js", "style.css"]
```

Descriptor

```
"resources": {
    "js": [
        {
            "uri": "script.js"
        }
    ],
    "css": [
```

```
        {
            "uri": "style.css"
        }
    ]
}
```

Descriptor for Libraries

The descriptor for libraries contains a subset of the attributes in the descriptor for applications and components.

manifest.json	.library	Available for SAPUI5 dist libraries?	Comment
`sap.app/id`	`name`	x	
`sap.app/type`	-	x	Generated with value `library`
`sap.app/embeds`	-	x	Generated
`sap.app/i18n`	`appData/manifest/i18n`		New in `.library`
`sap.app/applicationVersion/version`	`version`	x	
`sap.app/title`	`title`	x	Text symbol syntax with leading curly brackets (`{{`) and trailing curly brackets (`}}`); new in `.library`
`sap.app/description`	`documentation`	x	Text symbol syntax with leading curly brackets (`{{`) and trailing curly brackets (`}}`)
`sap.app/ach`	`appData/ownership/component`	x	
`sap.app/openSourceComponents`	`appData/manifest/openSourceComponents`		New in `.library`
`sap.app/resources`	-	x	Generated with value `resources.json`

manifest.json	.library	Available for SAPUI5 dist libraries?	Comment
sap.app/offline	appData/manifest/offline	x	New in .library
sap.app/sourceTemplate	appData/manifest/sourceTemplate		New in .library, to be filled by SAP Web IDE only
sap.ui/technology	-	x	Generated with value UI5
sap.ui/deviceTypes	appData/manifest/deviceTypes		New in .library
sap.ui/supportedThemes	-	x	Generated and merged
sap.ui5/dependencies/minUI5Version	-	x	Generated
sap.ui5/dependencies/libs	dependencies	x	
sap.ui5/contentDensities	appData/manifest/contentDensities		New in .library
sap.platform.abap/uri	appData/manifest/sap.platform.abap/uri		New in .library
sap.platform.hcp/uri	appData/manifest/sap.platform.hcp/uri		New in .library
sap.fiori/registrationIds	appData/manifest/sap.fiori/registrationId		New in .library
sap.fiori/archeType	appData/manifest/sap.fiori/archeType		New in .library

Related Information

Creating a Descriptor File for Existing Apps [page 764]

Descriptor for Components (Inside Libraries)

The descriptor for components contains a subset of the attributes in the descriptor for applications

Table 21: Attributes in the `sap.app` namespace

Attribute	Comment
`id`	Mandatory
`type`	With value `component`; mandatory
`i18n`	Path relative to component; default is `"i18n/i18n.properties"` Path back to library is also possible, for example via `"../i18n/i18n.properties"`
`embeddedBy`	Mandatory, for example, `"../"`
`title`	Mandatory
`subTitle`	
`description`	
`ach`	
`dataSources`	
`cdsViews`	
`resources`	Mandatory; must have value `resources.json` as file; it is generated by the library build with this name
`offline`	
`sourceTemplate`	

Table 22: Attributes in the `sap.ui` namespace

Attribute	Comment
`technology`	With value `UI5`; mandatory
`deviceTypes`	
`supportedThemes`	

Table 23: Attributes in the `sap.ui5` namespace

Attribute	Comment
`resources`	
`dependencies`	`libs` `components`
`models`	
`rootView`	
`handleValidation`	
`config`	
`routing`	
`extends`	`component` `minVersion`
`contentDensities`	
`componentName`	

Table 24: Attributes in the `sap.mobile` namespace

Attribute	Comment
`definingRequests`	

Library Name Determination

SAPUI5 determines the library name by analyzing the component namespace (package) up to the part where the segment starts with a capitalized letter. If the library name that has been determined, does not fit your component, an additional library attribute needs to be filled in the component metadata in `Component.js` to specify the library your component belongs to.

Example:

```
sap.ui.core.UIComponent.extend("com.sap.fancylibrary.sub.CompLib.Component", {
    metadata : {
        "manifest" : "json",
        "library" : "com.sap.fancylibrary",
    ...
}
```

Resources.json File

The `Resources.json` file lists all resources in a component or library folder. It resides next to each `manifest.json` in the generated results.

The file is generated during build time and its main purpose is for mobile packaging, as `resources.json` mentions all files inside the application. If an app has a `resources.json` file, it is mentioned in the `manifest.json` under `sap.app/resources`.

> **i Note**
>
> This file is used by SAP Tools like the SAP Fiori Client Packager. It will be generated automatically when using SAP WebIDE.

The list of resources is stored in an array in the `resources` property of the top level JSON object. The top level object can also contain the `_version` property, which can be omitted if the value is `1.0.0`. For each resource, the following entries are possible:

Property	Type	Description
`name`	`string`	Relative path of the resource as accessible in a server; starts with the first name segment, for example `Component.js` (mandatory)
`isDebug`	`Boolean`	When set to `true`, the resource is a debug source, the SAPUI5 build derives the flag from the naming convention (`-dbg(.controller .view .fragment).js`) (optional)
`locale`	`string`	Locale of the resource for known i18n resources; the SAPUI5 build derives the locale from the naming convention (`*_[locale].properties`) (optional)
`raw`	`string`	Name of the corresponding resource in the raw (developer) language for known i18n resources; for `messagebundle.en.properties`, for example, the corresponding raw file is `messagebundle.properties` (optional)
`merged`	`boolean`	Indicates whether the resource is a merged resource (optional)

Property	Type	Description
		By default, the SAPUI5 build determines this from naming conventions (`library-preload.json`, `library-all.js`, `Component-preload.js`), but it also allows to add more merged files by manual configuration of the build step. SAP Web IDE may use other knowledge for this; it knows, for example, that it merges the `Component-preload.js`.
`theme`	`string`	Indicates a theme-dependant resource (optional) The SAPUI5 build determines this from the naming convention `**themes<theme>/ **`

Example

```
{
    "resources":[
        {
            "name": ".library"
        },
        {
            "name": ".theming"
        },
        {
            "name": "DynamicSideContent-dbg.js",
            "isDebug":true
        },
        {
            "name": "DynamicSideContent.js"
        },
        {
            "name": "DynamicSideContentRenderer-dbg.js",
            "isDebug":true
        },

        ...

        {
            "name": "library-preload.json",
            "merged":true
        },

        ...

        {
            "name": "messagebundle_de.properties",
```

```
            "raw":"messagebundle.properties",
            "locale":"de"
        },

        ...

        {
            "name": "themes/sap_belize/library.less",
            "theme":"sap_belize"
        }

        ...

    ]
}
```

Creating a Descriptor File for Existing Apps

Detailed description of the steps needed to create a descriptor V2 for applications file for an existing transactional app created by the customer based on SAP Fiori.

1. Create the `manifest.json` file.
   You create the file in the web context root of your app on the same level as the `Component.js` file, using the content according to the instructions described from step 2 onwards. You can use the following code sample as a template. Make sure that you exchange or remove all placeholders (`<...>`) according to the instructions below.

   ```
   {
       "_version": "1.1.0",

       "start_url": "<startUrl>",

       "sap.app": {
           "_version": "1.1.0",
           "id": "<id>",
           "type": "application",
           "i18n": "<i18nPathRelativeToManifest>",
           "applicationVersion": {
               "version": "<version>"
           },
           "title": "{{<title>}}",
           "tags": {
               "keywords": [
                   "{{<keyword1>}}", "{{<keyword2>}}"
               ]
           },
           "dataSources": {
               "<dataSourceAlias>": {
                   "uri": "<uri>",
                   "settings": {
                       "localUri": "<localUri>"
                   }
               }
           }
       },
       "sap.ui": {
           "_version": "1.1.0",
           "icons": {
               "icon": "<icon>",
               "favIcon": "<favIcon>",
               "phone": "<phone>",
   ```

```
            "phone@2": "<phone@2>",
            "tablet": "<tablet>",
            "tablet@2": "<tablet@2>"
        },
        "deviceTypes": {
            "desktop": true,
            "tablet": true,
            "phone": true
        },
        "supportedThemes": [
            "sap_hcb",
            "sap_belize"
        ]
    },
    "sap.ui5": {
        "_version": "1.1.0",
        "resources": {
            "js": [
                {
                    "uri": "<uri>"
                }
            ],
            "css": [
                {
                    "uri": "<uri>",
                    "id": "<id>"
                }
            ]
        },
        "dependencies": {
            "minUI5Version": "<minUI5Version>",
            "libs": {
                "<ui5lib1>": {
                    "minVersion": "<minVersion1>"
                },
                "<ui5lib2>": {
                    "minVersion": "<minVersion2>"
                }
            },
            "components": {
                "<ui5component1>": {
                    "minVersion": "<minComp1Version>"
                }
            }
        },
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "uri": "<uriRelativeToManifest>"
            },
            "": {
                "dataSource": "<dataSourceAlias>",
                "settings": {}
            }
        },
        "rootView": "<rootView>",
        "handleValidation": <true|false>,
        "config": {

        },
        "routing": {

        },
        "extends": {
            "component" : "<extendedComponentId>",
            "minVersion": "<minComp1Version>",
            "extensions": {}
        }
```

```
        "contentDensities": {
            "compact": <true|false>,
            "cozy": <true|false>
        }
    },
    "sap.platform.abap": {
        "_version": "1.1.0",
        "uri": "<uri>"
    },
    "sap.platform.hcp": {
        "_version": "1.1.0",
        "uri": "<uri>"
    }
}
```

2. Fill the `start_url` (W3C namespace).
   If applicable, replace the `<start_url>` placeholder with the start URL of your app, for example `index.html`. If no start URL is shipped, remove the `"start_url"` section in the `manifest.json` file.

```
{
    "start_url": "index.html",
    ...
}
```

3. Fill the `id` and `applicationVersion/version` attributes of the `sap.app` namespace.

> ⚠ Caution
>
> `id` in the `sap.app` namespace must correspond to the component name in the `Component.js` file, for example `jQuery.sap.declare("cust.emp.myleaverequests.Component");`.

   To fill the ID and version information, open the `Component.js` file of your app and add the ID / namespace and version information:

```
jQuery.sap.declare("cust.emp.myleaverequests.Component");
...
    metadata :  {
        "name" : "My Leave Requests",
        "version" : "1.2.6"
```

   Open the `manifest.json` file and enter the values from the `Component.js` file as follows:

   - Replace the `<id>` placeholder with the the id / namespace value from `jQuery.sap.declare ("cust.emp.myleaverequests.Component"` in the example above).
   - Replace the `<version>` placeholder with the `version` value (`"1.2.6"` in the example above).

   Example: `sap.app/id` **and** `sap.app/applicationVersion/version` **in the** `manifest.json` **file**:

```
"sap.app": {
    "_version": "1.1.0",
    ...
    "id": "cust.emp.myleaverequests",
    ...
    "applicationVersion": {
        "version": "1.2.6"
    },
```

4. Fill the `i18n` and `title` attributes of the `sap.app` namespace.
   You find the respective information in the `Component.js` file under `resourceBundle` for the `i18n` attribute, and under `titleResource` for the `title` attribute:

```
"config" : {
```

```
    "titleResource": "app.Identity",
    "resourceBundle": "i18n/i18n.properties",
```

Open the `manifest.json` file and enter the values from the `Component.js` file as follows:

- Replace the `<title>` placeholder with the `titleResource` value (`"app.Identity"` in the example above)
- Replace the `<i18nPathRelativeToManifest>` placeholder with the `resourceBundle` value (`"i18n/i18n.properties"` in the example above).

Example: `sap.app/i18n` **and** `sap.app/title` **in the** `manifest.json` **file**

```
"sap.app": {
    "_version": "1.1.0",
    ...
    "i18n": "i18n/i18n.properties",
    ...
    "title": "{{app.Identity}}",
```

5. Fill the `tags/keywords` attribute of the `sap.app` namespace.
   If you maintain keywords for the SAP Fiori launchpad tile configuration (optional), enter one or more text symbols from the `sap.app/i18n` file in the `keywords` attribute of the `manifest.json` file. If not, remove the tags/keywords section from the `manifest.json` file.
   Example: `sap.app/tags/keywords` **in the** `manifest.json` **file**

```
"sap.app": {
    "_version": "1.1.0",
    ...
    "tags": {
        "keywords": [
            "{{Leave}}"
        ]
    },
```

6. Fill the `dataSource` attribute of the `sap.app` namespace with the data source you use for your app.
   For this, open the location where the service URL and the mock data source is defined.
   - Open the `Component.js` file of your app to see the data source under `serviceUrl`, see the following example for `name`, `serviceUrl` and mock data URL in `Component.js`:

```
metadata : {
    ...
    "config" : {
        ...
        "serviceConfig" : {
            name: "LEAVEREQUEST",
            serviceUrl: "/sap/opu/odata/GBHCM/LEAVEREQUEST;v=2/"
        }
    },
...

init : function() {
    ...
        oMockServer.simulate(rootPath + "/model/metadata.xml", rootPath +
"/model/");
```

   Return to the `manifest.json` file and do the following:
   - Enter the `name` value in the placeholder for `<dataSourceAlias>`.
   - Enter the value from the `serviceUrl` in the placeholder for `<uri>` to fill the value for the URI attribute.

- Enter the value from the URI of `oMockServer.simulate...` in the `Component.js` file in the placeholder for `<localUri>` to fill the value for the `localUri` attribute.

Example: `dataSources` **with alias and URI in the** `sap.app` **namespace of the** `manifest.json` **file**

```
"sap.app": {
    "_version": "1.1.0",
    ...
    "dataSources": {
        "LEAVEREQUEST": {
            "uri": "/sap/opu/odata/GBHCM/LEAVEREQUEST;v=2/",
            "settings": {
                "localUri": "model/metadata.xml"
            }
        }
    }
```

7. Fill the `icons` attribute of the `sap.ui` namespace.
   Open the `Component.js` file of your app to see the `icons` in the `config` section.
   Example: `icons` **in the** `Component.js` **file:**

```
"config" : {
    ...
    "icon": "sap-icon://Fiori2/F0394",
    "favIcon": "./resources/sap/ca/ui/themes/base/img/favicon/
My_Leave_Requests.ico",
    "homeScreenIconPhone": "./resources/sap/ca/ui/themes/base/img/launchicon/
My_Leave_Requests/57_iPhone_Desktop_Launch.png",
    "homeScreenIconPhone@2": "./resources/sap/ca/ui/themes/base/img/
launchicon/My_Leave_Requests/114_iPhone-Retina_Web_Clip.png",
    "homeScreenIconTablet": "./resources/sap/ca/ui/themes/base/img/launchicon/
My_Leave_Requests/72_iPad_Desktop_Launch.png",
    "homeScreenIconTablet@2": "./resources/sap/ca/ui/themes/base/img/
launchicon/My_Leave_Requests/144_iPad_Retina_Web_Clip.png"
},
```

   Return to the `manifest.json` file:
   - Enter the `icon` value in the `<icon>` placeholder.
   - Enter the `favIcon` value in the `<favIcon>` placeholder.
   - Enter the `homeScreenIconPhone` value in the `<phone>` placeholder. Do the same for the `<phone@2>`, `<tablet>` and `<tablet@2>` placeholders.

   Example: `icons` **in the** `sap.ui` **namespace of the** `manifest.json` **file**

```
"sap.ui": {
    "_version": "1.1.0",
    ...
    "icons": {
        "icon": "sap-icon://Fiori2/F0394",
        "favIcon": "./resources/sap/ca/ui/themes/base/img/favicon/
My_Leave_Requests.ico",
        "phone": "./resources/sap/ca/ui/themes/base/img/launchicon/
My_Leave_Requests/57_iPhone_Desktop_Launch.png",
        "phone@2": "./resources/sap/ca/ui/themes/base/img/launchicon/
My_Leave_Requests/114_iPhone-Retina_Web_Clip.png",
        "tablet": "./resources/sap/ca/ui/themes/base/img/launchicon/
My_Leave_Requests/72_iPad_Desktop_Launch.png",
        "tablet@2": "./resources/sap/ca/ui/themes/base/img/launchicon/
My_Leave_Requests/144_iPad_Retina_Web_Clip.png"
    },
```

   If your app does not have icons, remove the icons section or the corresponding icon attributes from the `manifest.json` file.

8. Fill the `deviceTypes` and `supportedThemes` attributes in the `sap.ui` namespace in the `manifest.json` file.
   Return to the `manifest.json` file and ensure that the `deviceTypes` and `supportedThemes` attributes in the `manifest.json` are correct for your application. If not, adapt the entries accordingly.
   Example: `deviceTypes` **and** `supportedThemes` **in the** `sap.ui` **namespace in the** `manifest.json` **file**

```
"sap.ui": {
    "_version": "1.1.0",
    ...
    "deviceTypes": {
        "desktop": true,
        "tablet": true,
        "phone": true
    },
    "supportedThemes": [
        "sap_hcb",
        "sap_belize"
    ]
```

9. Fill the `resources` attribute in the `sap.ui5` namespace.
   Open the `Component.js` file of your app to see the js and CSS resources under `includes`.
   Example: `includes` **in the** `Component.js` **file**

```
"includes": ["css/shopStyles.css", "myfile.js"],
```

   Return to the `manifest.json` file:
   - Enter the js resource value under `"js"` in the `<uri>` placeholder.
   - Enter the CSS resource value under `"css"` in the `<uri>` placeholder.

> ⚠ Caution
>
> The format in the `Component.js` file is an array, whereas the format in the `manifest.json` file is a map.

   Example: `resources` **attribute in the** `sap.ui` **namespace in the** `manifest.json` **file**

```
"sap.ui5": {
    "_version": "1.1.0",
    ...
    "resources": {
        "js": [
            {
                "uri": "myfile.js"
            }
        ],
        "css": [
            {
                "uri": "css/shopStyles.css"
            }
        ]
    },
```

   If your app does not include resources, remove the `resources` section from the `manifest.json` file.
10. Fill the `dependecies` attribute of the `sap.ui5` namespace with the SAPUI5 dependencies that are used.
    Open the `Component.js` file of your app to see the dependencies for the `ui5` libs and components.
    Example: `dependencies` **in the** `Component.js` **file**

```
"dependencies": {
    "libs": [
```

```
        "sap.m",
        "sap.me"
    ],
    "components": ["sap.app.otherComponent"]
}
```

Return to the `manifest.json` file and fill the corresponding entries in the `manifest.json`. Enter a value for the minimum SAPUI5 version in the `<ui5Version>` placeholder.

> ⚠ Caution
>
> The format in the `Component.js` file is an array, whereas the format in the `manifest.json` file is a map. Ensure that **all** of the SAPUI5 libraries used by your app are mentioned under libs. Also make sure that all of the SAPUI5 components used by your app are mentioned under `components`. If there are no dependent components, remove the `components` entry.

Example: `dependencies` **in the** `sap.ui5` **namespace in the** `manifest.json` **file**

```
"sap.ui5": {
    "_version": "1.1.0",
    ...
    "dependencies": {
        "minUI5Version": "1.30",
        "libs": {
            "sap.m": {
                "minVersion": "1.30"
            },
            "sap.me": {
                "minVersion": "1.30"
            }
        },
        "components": {
            "sap.app.otherComponent": {
                "minVersion": "1.2.0"
            }
        }
    },
```

If your app requires a minimum version of a lib or component, specify the version under `minVersion` for information purposes. If not, remove the `minVersion` attribute.

11. Fill the `models` attribute of the `sap.ui5` namespace.
    If a `model` is entered in `sap.ui5/models` in the `manifest.json` file, SAPUI5 creates the model automatically and the coding for model creation inside the app can be removed.
    Example: `model` **creation in** `Component.js`**:**

```
init : function() {
    ...
    // set i18n model
    var i18nModel = new sap.ui.model.resource.ResourceModel({
        bundleUrl : rootPath + "/i18n/i18n.properties"
    });
    this.setModel(i18nModel, "i18n");

    // set data model
    var m = new sap.ui.model.odata.v2.ODataModel(sServiceUrl);
    this.setModel(m);
```

    Return to the `manifest.json` file:
    - i18n model

Use the same model name as in the `Component.js` file, for example "i18n", and the type `sap.ui.model.resource.ResourceModel`. Enter the URI from the `Component.js` file in the `<uriRelativeToManifest>` placeholder relative to `manifest.json`, for example, `i18n/i18n.properties`

- OData model
  Use the same model name as in the `Component.js` file, for example "leave" or "" for the default model. Enter a reference to a data source from `sap.app/dataSource` in the `<dataSourceAlias>` placeholder; if needed, enhance it with more settings for SAPUI5.

Example: Models in the `sap.ui5` namespace in the `manifest.json` file

```
"sap.ui5": {
    "_version": "1.1.0",
    ...
    "models": {
        "i18n": {
            "type": "sap.ui.model.resource.ResourceModel",
            "uri": "i18n/i18n.properties"
        },
        "": {
            "dataSource": "LEAVEREQUEST",
            "settings": {
            }
        }
    },
```

12. Fill the `rootView`, `handleValidation`, `config` and `routing` attributes in the `sap.ui5` namespace.
    Open the `Component.js` file of your app to see the `rootView`, `handleValidation`, `routing`, `config` in the component metadata section.
    Example: `rootView`, `handleValidation`, `config`, `routing` in `sap.ui5` namespace of the `manifest.json` file:

    ```
    ...
    "rootView": "myRootView",
    "handleValidation": true,
    "config": {
        ...
    },
    "routing": {
        ...
    }
    ```

    Return to the `manifest.json` file and copy this metadata from the `Component.js` file to the `sap.ui5` namespace in the `manifest.json` file.
    Only transfer those config parameters in the `config` section to the `manifest.json` file that have not yet been transferred in the steps before. In other words, do not transfer `resourceBundle`, `titleResource`, `icon`, `favicon`, `homeScreenIconPhone`, `homeScreenIconPhone2`, `homeScreenIconTablet` and `homeScreenIconTablet2`.
    Example: `rootView`, `handleValidation`, `config` and `routing` in the `sap.ui5` namespace of the `manifest.json` file

    ```
    "sap.ui5": {
        "_version": "1.1.0",
        ...
        "rootView": "myRootView",
        "handleValidation": true,
        "config": {
            ...
        },
    ```

```
    "routing": {
        ...
    },
```

If there is no corresponding entry in the `Component.js` file, remove the section in the `manifest.json` file.

13. Fill the `extends` attribute of the `sap.ui5` namespace.
    Open the `Component.js` file of your app to see the component which your app extends:

```
hcm.emp.myleaverequests.Component.extend("cust.emp.myleaverequests.Component",
 {
```

Return to the `manifest.json` file and enter the value from the `component` namespace in the `<extendedComponentId>` placeholder, for example `hcm.emp.myleaverequests`.
Example: `extends/component` **in** `sap.ui5` **namespace in** `manifest.json` **file**

```
"sap.ui5": {
    "_version": "1.1.0",
    ...
    "extends": {
        "component": "hcm.emp.myleaverequests",
        "minVersion": "1.1.0"
    }
```

If your app requires a minimum version of a component, specify the version under `minVersion` for information purposes, otherwise remove the attribute. If your app uses the SAPUI5 extension concept with a `customizing` entry under component metadata in the `Component.js` file, move the content of that entry to `sap.ui5/extends/extensions` in the `manifest.json` file, or remove the `customizing` entry. If your app does **not** extend another component, remove the `extends` section from the `manifest.json` file.

14. Fill the `contentDensities` attribute of the `sap.ui5` namespace.
    Enter the correct values for the `compact` and `cozy` attributes (`true` or `false`) under `contentDensities` in the `manifest.json` file. The attributes specify the content density modes that your app supports, see Content Densities [page 1142].
    Example: `contentDensities` **in** `sap.ui5` **namespace of the** `manifest.json` **file:**

```
"sap.ui5": {
    "_version": "1.1.0",
    ...
    "contentDensities": {
        "compact": true,
        "cozy": true
    }
```

15. Verify that **no** placeholders exist.
    Return to the `manifest.json` file and make sure there are **no more** placeholders within it (`<...>`). If the file still contains placeholders, remove the corresponding sections.

Code Changes

1. Adapt the `Component.js` file.
   Example: `Component.js` **before making changes**

```
jQuery.sap.declare("cust.emp.myleaverequests.Component");
jQuery.sap.require("cust.emp.myleaverequests.Configuration");

hcm.emp.myleaverequests.Component.extend("cust.emp.myleaverequests.Component",
 {
    metadata :  {
        "name" : "My Leave Requests",
        "version" : "...",
        "library" : "cust.emp.myleaverequests",
        "includes" : [],
        "dependencies" : {
            "libs" : ["sap.m", "sap.me"],
            "components" : ["sap.app.otherComponent"]
        },
        "rootView": ...,
        "handleValidation": ...,
        "config": {
            ...
        },
        "routing": {
            ...
        },
        "config" : {
            "titleResource": "app.Identity",
            "resourceBundle": "i18n/i18n.properties",
            "icon": "sap-icon://Fiori2/F0394",
            "favIcon": "./resources/sap/ca/ui/themes/base/img/favicon/
My_Leave_Requests.ico",
            "homeScreenIconPhone": "./resources/sap/ca/ui/themes/base/img/
launchicon/My_Leave_Requests/57_iPhone_Desktop_Launch.png",
            "homeScreenIconPhone@2": "./resources/sap/ca/ui/themes/base/img/
launchicon/My_Leave_Requests/114_iPhone-Retina_Web_Clip.png",
            "homeScreenIconTablet": "./resources/sap/ca/ui/themes/base/img/
launchicon/My_Leave_Requests/72_iPad_Desktop_Launch.png",
            "homeScreenIconTablet@2": "./resources/sap/ca/ui/themes/base/img/
launchicon/My_Leave_Requests/144_iPad_Retina_Web_Clip.png"
        });
        });
```

Apply the following changes:

- Comment or remove the line for the `require` statement for configuration (if available) `sap.ui.require("cust.emp.myleaverequests.Configuration");`
- Add the manifest reference to the metadata: `"manifest": "json"`.
- Remove the `name` section.
- Remove the `library` section.
- Remove the `version` section.
- Remove the `includes` section.
- Remove the `dependencies` section.
- Remove the `rootView` section.
- Remove the `handleValidation` section.
- Remove the `routing` section.
- Remove the `config` section.

Example: `Component.js` **after making changes**

```
jQuery.sap.declare("cust.emp.myleaverequests.Component");
//jQuery.sap.require("cust.emp.myleaverequests.Configuration");

hcm.emp.myleaverequests.Component.extend("cust.emp.myleaverequests.Component",
 {
    metadata : {
```

```
        "manifest": "json",
        ...
});
```

2. Adapt the data source reference in the `Component.js` file.
   Example: Data source reference in `Component.js` file before making changes

```
metadata : {
    ...
    "config" : {
        ...
        "serviceConfig" : {
            name: "LEAVEREQUEST",
            serviceUrl: "/sap/opu/odata/GBHCM/LEAVEREQUEST;v=2/"
        }
    },
...

init : function() {
    ...
    var oServiceConfig = this.getMetadata().getConfig()["serviceConfig"];
    var sServiceUrl = oServiceConfig.serviceUrl;
    ...
    oMockServer.simulate(rootPath + "/model/metadata.xml", rootPath + "/
model/");
```

   Apply the following changes:
   - Remove `serviceConfig` under `config` in the component metadata.
   - If you are still using the service URL in your coding for purposes other than model creation, change the lines for getting the service config / url and read the URI from the manifest via your component metadata, for example, `this.getMetadata().getManifestEntry("sap.app")...`; otherwise, remove that coding.
   - Change the line for `oMockServer.simulate...` and read the URI from the manifest via your component metadata, for example, `this.getMetadata().getManifestEntry("sap.app")...`

   Example: Data source reference in `Component.js` file after making changes

```
metadata : {
    "manifest": "json",
    ...

init : function() {
    ...
    var sServiceUrl =
this.getMetadata().getManifestEntry("sap.app").dataSources["LEAVEREQUEST"].uri
;
    ...
    oMockServer.simulate(rootPath + "/" +
this.getMetadata().getManifestEntry("sap.app").dataSources["LEAVEREQUEST"].set
tings.localUri, rootPath + "/model/");
```

3. Remove the SAPUI5 model creation in the `Component.js` file.
   Example: `Component.js` file before making changes

```
init : function() {
    ...
    // set i18n model
    var i18nModel = new sap.ui.model.resource.ResourceModel({
        bundleUrl : rootPath + "/i18n/i18n.properties"
    });
    this.setModel(i18nModel, "i18n");

    // set data model
```

```
    var m = new sap.ui.model.odata.v2.ODataModel(sServiceUrl);
    this.setModel(m);
```

Apply the following changes:

- Delete the lines for the i18n model creation and model setting.
- Delete the lines for the data model creation and model setting.

Smoke Test

To verify that your app works as before, perform checks to make sure the following is true:

- OData service works as before
- Mock data works as before
- Title, icons in SAP Fiori launchpad work as before
- Navigation works as before

Migration Information for Upgrading the Descriptor File

Information how to add new attributes of descriptor versions higher than V2 (SAPUI5 1.30) to the descriptor file.

Attribute	Version*	Description	Example
`_version`	V3 (1.32)	Needs to be updated in the `manifest.json` file when migrating to a new descriptor version: • `_version` for V3 is 1.2.0 • `_version` for V4 is 1.3.0 • `_version` for V5 is 1.4.0 (see example)	`{ "_version": "1.4.0", "sap.app": { ...`
`sap.app/ crossNavigation`	V3 (1.32)	Contains navigation information and is a mandatory attribute in the `manifest.json` file for SAP Fiori apps; the attribute contains two sections: • `sap.app/ crossNavigation/ inbounds` - Contains	

Attribute	Version*	Description	Example
		inbound intents and signature information • `sap.app/crossNavigation/outbounds` - Contains required intents that are called explicitely by the app, for example, if a business process is split among different apps A and B. If A calls B, A has outbound the intent to address B.	
`sap.app/subTitle`	V4 (1.34)	Added to the `manifest.json` file by using the `{{...}}` syntax **i Note** Text symbols must be part of the properties file which is defined in `sap.app/i18n` (default `"i18n/i18n.properties"`) .	`"sap.app": { "_version": "1.3.0", ... "title": "{{title}}", "subTitle": "{{subtitle}}",`
`sap.app/crossNavigation/inbounds/<inboundname>/subTitle`	V4 (1.34)	Used to overwrite the `subTitle` attribute per inbound; use the `{{...}}` syntax to add the attribute to the `manifest.json` file **i Note** Text symbols must be part of the properties file which is defined in `sap.app/i18n` (default `"i18n/i18n.properties"`) .	`"sap.app": { "_version": "1.3.0", ... "crossNavigation" : { "inbounds": { "contactCreate": { "semanticObject": "Contact", "action": "create", "icon": "sap-icon://add-contact",`

Attribute	Version*	Description	Example

			<pre>"title": "{{title}}",
			
			"subTitle": "{{subtitleOther}}",</pre>

sap.ui/fullWidth	V4 (1.34)	Indicates whether an app shall run in full screen mode (true)	<pre>"sap.ui": {
			"_version": "1.3.0",
			
			"technology": "UI5",
			...
			"fullWidth": true</pre>

sap.ui5/dependencies/components/<componentname>/lazy and dependencies/libs/<libname>/lazy	V4 (1.34)	If dependencies/components/<componentname>/lazy and dependencies/libs/<libname>/lazy are set to true, the attribute indicates in an SAP Fiori app that a dependency shall be lazy loaded (default is false), see the example for manifest.json for the SAP Fiori app.	Example for manifest.json for the SAP Fiori app:
			<pre>"sap.ui5": {
			"_version": "1.2.0",
			...
			
			"dependencies": {
			
			"minUI5Version": "1.34.0",
			"libs": {
			
			"sap.m": {
			
			"minVersion": "1.34.0"
			},
			
			"sap.ui.commons": {
			
			"minVersion": "1.34.0",
			
			"lazy": true
			}
			},
			
			"components": {
			
			"sap.ui.app.other": {
			
			"minVersion": "1.1.0"
			
			"lazy": true
			}</pre>

Attribute	Version*	Description	Example
			} },

sapui5/routing/config/async	V4 (1.34)	General setting for routing that indicates how the views are loaded; if set to true, the views are loaded asynchronously (default is false)
		
		For performance reasons, we recommend to always use the async setting. This recommendation implies that you have followed the SAPUI5 programming model in general and do **not** rely on any sync-execution depending event-orders.
			"sap.ui5": {
			"_version": "1.2.0",
			...
			"routing": {
			
			"config": {
			
			"viewType": "XML",
			
			"async": true
			...
			},
			...

sap.ui5/models/preload	V5 (1.38)	Defines whether or not the model is initialized (preloaded) before the component instance is created and while loading the component preload and its dependencies	"equipment": {
			"preload": true,
			
			"dataSource": "equipment",
			...
			}

* Available as of descriptor version (SAPUI5 version)

Descriptor Dependencies to Libraries and Components

Description of the performance-relevant attributes that are available for the descriptor for applications, components and libraries

The performance-relevant attributes have been introduced with the version 3 of the descriptor for applications, components, and libraries.

Dependencies to Libraries

The following dependencies to libraries can be implemented:

- To benefit from the asynchronous library preload, add the mandatory libraries to sap.ui5/dependencies/libs.

- To expose the necessary dependencies for offline packages for mobile devices, add optional libraries to `sap.ui5/dependencies/libs` and flag them as `lazy`.

For **applications and components**, modify the `manifest.json` as follows:

```
"sap.ui5": {
    ...
    "dependencies": {
      ...
      "libs": {
        "sap.m": {},
        "sap.suite.ui.commons": {
          "lazy": true
        }
      }
      ...
    },
    ...
```

For **libraries**, modify the `.library` file as shown in the follown code sample. This file is available because the `manifest.json` for libraries is generated based on this metadata.

```
<dependencies>
    <dependency>
       <libraryName>sap.m</libraryName>
    </dependency>
    <dependency>
       <libraryName>sap.suite.ui.commons</libraryName>
       <lazy>true</lazy>
    </dependency>
    ...
```

In a second step, modify the `library.js` file as follows:

```
sap.ui.getCore().initLibrary({
   ...
   dependencies : ["sap.ui.core","sap.m"], // lazy libs are not declared here
```

ℹ Note

In all cases, the lazy libraries need to be loaded manually in the application or library via the `loadLibrary` API:

```
// lazy lib loaded synchronously (avoid if possible!)
sap.ui.getCore().loadLibrary("sap.suite.ui.commons");
// lazy lib loaded asynchronously (the preferred way!)
sap.ui.getCore().loadLibrary("sap.suite.ui.commons", { async:
true }).then(...);
```

→ Tip

Execute the `loadLibrary` before any resource of the library is required to preload the complete library instead of loading each resource individually.

Always use the async API as this is the preferred and performant way. Only use the sync API as an exception if your coding relies on synchronous loading.

Dependencies to Components

Scenario 1: UI library contains multiple components

In this scenario, the library is the leading container and **no** component preload is available. This means, that you maintain the library dependency as described above. This is true for all kinds of component dependencies, also for `sap.ui5/extends/component`. If the extended component originates in a library, do **not** use `sap.ui5.extends/component`, but only declare the library dependency. Otherwise, the component dependency causes a 404 request.

For loading lazy components inside a library, proceed with the library mechanisms as described above:

```
// lazy lib loaded synchronously (avoid if possible!)
sap.ui.getCore().loadLibrary("sap.suite.ui.commons");
// lazy lib loaded asynchronously (the preferred way!!!)
sap.ui.getCore().loadLibrary("sap.suite.ui.commons", { async: true }).then(...);
```

Scenario 2: Standalone component

In this scenario, you only maintain a dependency to the component. The component preload is available for this scenario:

- To benefit from the asynchronous components preload, add the mandatory components to `sap.ui5/dependencies/components`
- Add the **optional** components to `sap.ui5/dependencies/components` and flag them as `lazy`.

For applications and components, modify the `manifest.json` as follows:

```
"sap.ui5": {
    ...
    "dependencies": {
      ...
      "libs": {
        ...
      },
      "components": {
        "samples.components.sample": {},
        "samples.components.samplelazy": {
          "lazy": true
        }
        ...
      }
    },
    ...
```

For loading/instantiating the lazy standalone components, use the component factory functions:

```
// "Component" required from module "sap/ui/core/Component"
// Asynchronously loads a component class without instantiating it.
Component.load({
  name: "..."
}).then(function(ComponentClass) {
  ...
});
// Asynchronously creates a new component instance from the given configuration.
// If necessary the component class is loaded.
Component.create({
  name: "..."
}).then(function(oComponentInstance) {
  ...
});
```

Related Information

loadLibrary
Component

Manifest Model Preload

The `preload` flag enables a preload mode for a model, thus improving the startup performance of an app or component.

The `preload` flag is located in `manifest.json` under `sap.ui5/models`:

```
"sap.ui5": {
  ...
  "models": {
      "mymodel": {
          "preload": true,
          ...
```

The flag is not active by default, as there are some prerequisites:

- `sap.ui.component` is set to `"async=true"` and `manifest` (API parameter name of `sap.ui.component`).
- `Model` implementation class is loaded before `sap.ui.component` is called; otherwise the model will not be created.
- As model events (for example `attachMetadataLoaded`) may be missed because they are fired before the component coding runs, we recommend using the `Promise` API (e.g. `metadataLoaded`) instead, depending on the model type.
- Use the model preload flag for `sap.ui.model.resource.ResourceModel` if one of the following applies:
  - There is no component preload.
  - The corresponding resource files are not part of the component preload.

This means: The preload flag only makes sense for models which load their data from other locations than the component itself. Local JSON, XML or resource model does not make sense as it interferes with the component preload which will result in loading the model data twice and should be omitted. But for the V2 OData model, for example, using preload speeds up the performance as the OData metadata can already be loaded in parallel to the component preload.

Before enabling the preload for the V2 ODataModel, make sure that you listen properly to metadata loaded by using the `Promise` API instead of the `Event` API (`metadataLoaded`) since the preload could have loaded the metadata already before the application code is executed. The Promise will be executed even if the metadata loaded event has been raised already.

Listen properly to metadata loaded by using the Promise:

```
oModel.metadataLoaded().then(function() {  /* TODO: add the event handling here!
*/  });
```

Enabling the Automatic SAP Fiori 2.0 Header Adaptation in the Descriptor

Application developers can enable automatic adaptation of their existing applications from the manifest.json app descriptor. This helps to easily convert applications to the new look-and-feel of SAP Fiori 2.0.

SAP Fiori 2.0

SAP Fiori 2.0 is the next evolution step of the SAP Fiori UX. SAP Fiori 2.0 features new themes, a more unified user experience, and smoother, more intuitive application interactions.

Application headers, written based on older SAP Fiori design guidelines, can now be easily adapted to the new SAP Fiori 2.0 look-and-feel by using the automatic adaptation mechanism in the app descriptor.

The SAP Fiori 2.0 Header

The SAP Fiori 2.0 design concept requires changes with regards to the headers of applications and the SAP Fiori launchpad (FLP). If your application has a header, it needs to be merged into the standardized SAP Fiori 2.0 header. SAPUI5 offers an adapter mechanism to let existing apps automatically adjust their header layout according to the SAP Fiori 2.0 guidelines.

> **i Note**
>
> The screenshots in this topic are mockups and are used to visually outline the adaptations. The final apps will look somewhat different.

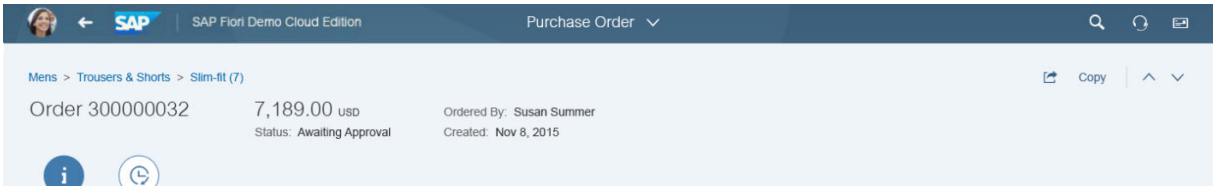


Figure 210: SAP Fiori 2.0 header of a fullscreen application

The complete adaptation of a fullscreen app to SAP Fiori 2.0 consists of five main steps:

1. Remove the app-specific header bar. The header is made transparent and collapsed if there is no content in it after the adaptation.
2. Display the title in the center of the FLP header
3. Move the action buttons from the app header to the header content area below the FLP header.
4. Move the *Back* button from the app-specific header to the FLP header.
5. Drill-down hierarchy levels can be added to the dropdown menu adjacent to the FLP title.

You can see how the elements are moved and transformed from the old SAP Fiori version (below) to the new SAP Fiori 2.0 design in the screenshot below.

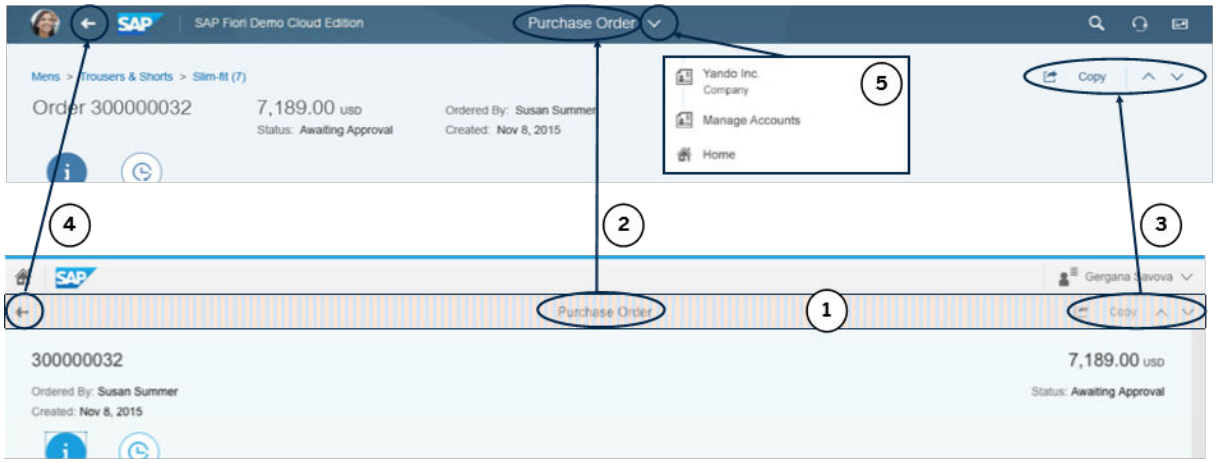


Figure 211: SAP Fiori 2.0 header adaptation

> **i** Note
>
> These adaptations are primarily valid only for **fullscreen** apps. Other floorplans, like Master-Detail, are affected differently and the adaptation there will not be the same.

Enabling the Adaptation in the App Descriptor

You can override the adapter default behavior for a single application by adding an entry in the app descriptor in the `sap.ui5/config` section. Setting `sapFiori2Adaptation` to **true** enables the full functionality of the SAP Fiori 2.0 Adapter.

```
"config": {
            ...
            "sapFiori2Adaptation": true,
            ...
}
```

Alternatively, you can use five fine-grained settings to enable only some of the adaptations. In the following example, you can see how to trigger transparent headers (`style` attribute) and title propagation to FLP (`title` attribute). The other adaptations are not applied.

```
"config": {
            ...
            "sapFiori2Adaptation": {
                  "style": true,
                  "collapse": false,
                  "title": true,
                  "back": false,
                  "hierarchy": false
            },
            ...
}
```

In the list below, you can see what each of the settings enables.

- `style` - Triggers header transparency

- `collapse` - Triggers collapsing of the header when empty
- `title` - Triggers moving the header to FLP
- `back` - Triggers the *Back* button visibility in the app
- `hierarchy` - Triggers propagation of the hierarchy to FLP

> **i** Note
>
> In rare cases this automatic adaptation of the header area may not work, due to the application structure or other reasons. In this case the headers will still appear in the old design, but the apps will continue to be usable.

Some old SAP Fiori applications do not have an app descriptor yet. If you consider the effort to provide proper app descriptors for all applications as too high, there is a second way to do this configuration. This alternative configuration is done in the `metadata` section of **`Component.js`** (the app's root component), which also has a `config` section. The configuration options can be done there in the same manner.

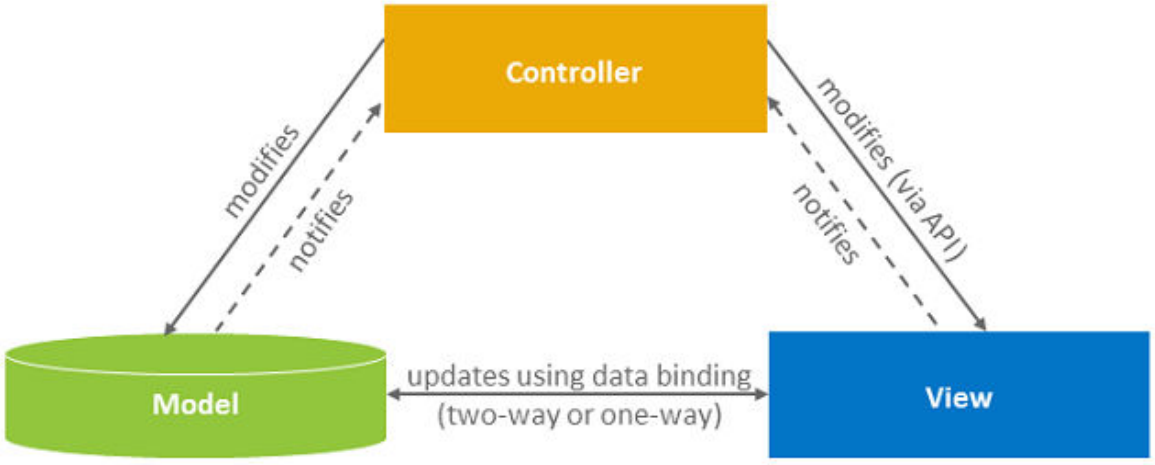
> **i** Note
>
> If both the metadata and manifest are configured, and contradict each other, the configuration in **manifest.json** is applied.

Model View Controller (MVC)

The Model View Controller (MVC) concept is used in SAPUI5 to separate the representation of information from the user interaction. This separation facilitates development and the changing of parts independently.

Model, view, and controller are assigned the following roles:

- The **view** is responsible for defining and rendering the UI.
- The **model** manages the application data.
- The **controller** reacts to view events and user interaction by modifying the view and model.



- Models [page 882]

- Controller [page 807]
- Views [page 787]

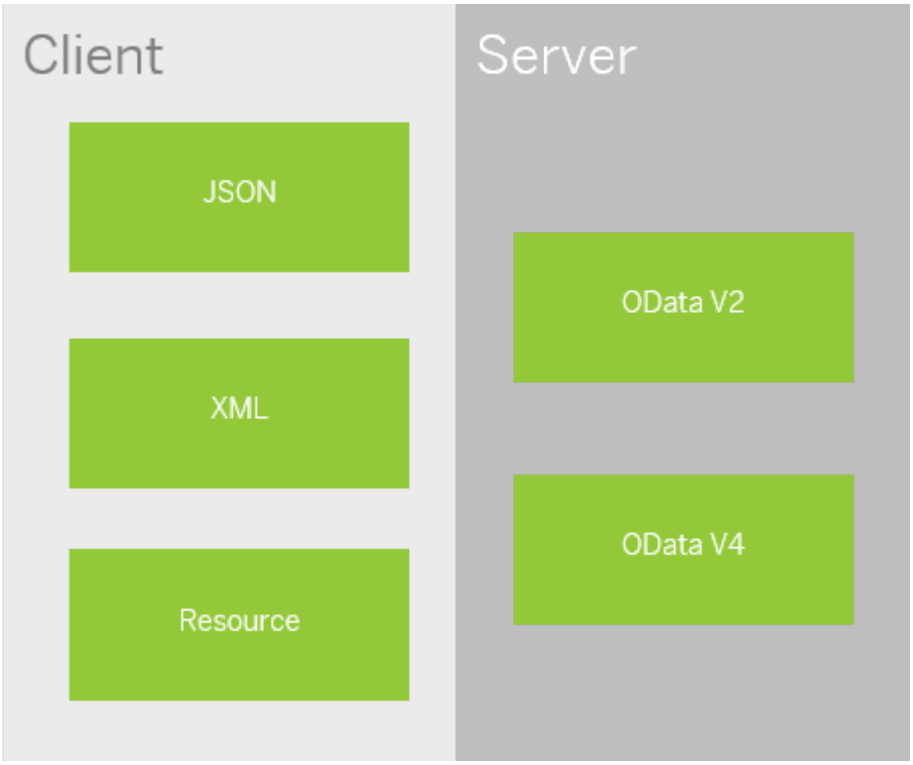
The purpose of data binding in the UI is to separate the definition of the user interface (view), the data visualized by the application (model), and the code for the business logic for processing the data (controller). The separation has the following advantages: It provides better readability, maintainability, and extensibility and it allows you to change the view without touching the underlying business logic and to define several views of the same data.

Views and controllers often form a 1:1 relationship, but it is also possible to have controllers without a UI, these controllers are called application controllers. It is also possible to create views without controllers. From a technical position, a view is a SAPUI5 control and can have or inherit a SAPUI5 model.

View and controller represent reusable units, and distributed development is highly supported.

Models

A model in the Model View Controller concept holds the data and provides methods to retrieve the data from the database and to set and update data.



- JSON Model [page 991]
- XML Model [page 993]
- Resource Model [page 995]

- OData V2 Model [page 883]
- OData V4 Model [page 918]

SAPUI5 provides the following predefined models:

- **OData model**: Enables binding of controls to data from OData services. The OData model supports two-way, one-way and one-time binding modes. However, two-way binding is currently only supported for properties, and not for aggregations.

  > **i** Note
  >
  > The OData model currently supports the following OData versions:
  >
  > - OData V2
  > - OData V4 (limited feature scope)

- **JSON model**: Can be used to bind controls to JavaScript object data, which is usually serialized in the JSON format. The JSON model is a client-side model and, therefore, intended for small data sets, which are completely available on the client. The JSON model supports two-way (default), one-way and one-time binding modes.
- **XML model**: A client-side model intended for small data sets, which are completely available on the client. The XML model does not contain mechanisms for server-based paging or loading of deltas. The XML model supports two-way (default), one-way and one-time binding modes.
- **Resource model**: Designed to handle data in resource bundles, mainly to provide texts in different languages. The resource model only supports one-time binding mode because it deals with static texts only.

The JSON model, XML model, and the resource model are **client-side models**, meaning that the model data is loaded completely and is available on the client. Operations such as sorting and filtering are executed on the client without further server requests.

The OData (V2 or V4) model is a **server-side model** and only loads the data requested by the user interface from the server.

You can not only define one model for your applications, but define different areas in your application with different models and assign single controls to a model. You can also define nested models, for example, a JSON model defined for the application and an OData model for a table control contained in the application.

A Web application should support several data sources, such as JSON, XML, Atom, or OData. However, the way in which data binding is defined and implemented within the UI controls should be independent of the respective data source. It is also possible to create a custom model implementation for data sources that are not yet covered by the framework or are domain-specific.

Related Information

API Reference: `sap.ui.model`

Views

The view in the Model View Controller concept is responsible for defining and rendering the UI. SAPUI5 supports predefined view types.

The following predefined view types are available:

- **XML view** (file or string in XML format); the `XMLView` type supports a mix of XML and plain HTML.
- **JSON view** (file or string in JSON format)
- **JS view**, constructed in a traditional manner
- **HTML view** (file or string in HTML format)

> **i** Note
>
> We recommend to use XML views, because XML views force a clear separation of the UI definition from the application logic (which has to be implemented in the controller). This makes the code more readable and easier to support.
>
> Therefore, we concentrate on XML views and only provide examples for XML views throughout this documentation.

Related Information

API Reference

XML View

The XML view type is defined in an XML file. The file name either ends with `.view.xml` or as an XML string. The file name and the folder structure together specify the name of the view that equals the SAPUI5 module name.

> **✤** Example
>
> For `resources/sap/hcm/Address.view.xml`, the view name is `sap.hcm.Address`. The application uses this view name for displaying an instance of this view. If you define the XML view by means of an XML string, no file or define/require is needed.
>
> The file looks as follows:

```
<mvc:View controllerName="sap.hcm.Address" xmlns="sap.m"
xmlns:mvc="sap.ui.core.mvc">
   <Panel>
      <Image src="http://www.sap.com/global/ui/images/global/sap-logo.png"/>
      <Button text="Press Me!"/>
   </Panel>
</mvc:View>
```

Nest the XML tags analogous to the nesting sequence of SAPUI5 controls and add the property values as attributes (see Namespaces in XML Views [page 788]).

Each control or element is represented by an XML tag with the name the control. If you, for example, want to create an instance of a `sap.m.Button`, you use tag `<Button>` with namespace `sap.m`. You can create a context binding for the control by using attribute `objectBinding` or `Binding`. For more information, see Context Binding (Element Binding) [page 824].

Related Information

API Reference: `sap.ui.html.xmlview`

Namespaces in XML Views

The names of the SAPUI5 control libraries and the related subpackages are mapped to XML namespaces.

One of the required namespaces can be defined as the default namespace (`xmlns="..."`). The control tags for this namespace do not need a prefix.

The `View` tag is required and in the example below, the `sap.ui.core.mvc` namespace is defined with alias `mvc`. Technically, you can define any alias for namespaces. However, the convention is to use the last part of the full package name.

A control can be located in a subpackage of a control library, for example `sap.ui.layout.form.Form` is located in the `sap.ui.layout` library, but the full package name is `sap.ui.layout.form`. You have to specify this subpackage as a separate XML namespace, even if `sap.ui.layout` is already defined as namespace.

```
<mvc:View
     xmlns:mvc="sap.ui.core.mvc"
     xmlns:layout="sap.ui.layout"
     xmlns:form="sap.ui.layout.form"
     xmlns="sap.m">
</mvc:View>
```

Aggregation Handling in XML Views

In XML views, aggregated child controls can be added as child tags.

Aggregations of XML Views

On root level, you can only define content for the default aggregation, e.g. without adding the `content` tag. If you want to specify content for another aggregation of a view like `dependents`, place it in a child control´s `dependents` aggregation or add it by using the `addDependent` method.

Aggregations of Controls Inside the View

Some controls have more than one content area, for example the shell control that has the main content area, a menu bar, a `headerItems` aggregation, a `worksetItems` aggregation, and so on. An aggregation tag usually serves as a direct child of a container and contains children. You can only add children directly if the container control has marked one of the child aggregations as default.

> **i Note**
>
> Some containers may not have default content, for example, the splitter container has two equally important content areas.
>
> The framework supports you by issuing error message in case of errors in the aggregation handling in XML views.

You fill aggregations as shown in the following example. The namespace of the parent control tag and the aggregation tag must be the same.

```
<mvc:View controllerName="sap.hcm.Address" xmlns="sap.m"
xmlns:mvc="sap.ui.core.mvc">
   <Panel>
      <content>  <!-- this is the general way of adding children: use the
aggregation name -->
         <Image src="http://www.sap.com/global/ui/images/global/sap-logo.png"/>
         <Button text="Press Me"/>
      </content>
   </Panel>
</mvc:View>
```

If an aggregation of **cardinality** `0..1` has an alternative type and if you want to specify a value of that alternative type, you have to do this as an attributes, not as a nested element.

> **Example**
>
> For the `noData` aggregation of the `sap.ui.comp.smarttable.SmartTable` control, you can either define a string or a nested text control.

String

```
<SmartTable noData="No data
available">
    ...
</SmartTable>
```

Nested text control

```
<SmartTable ....>
    <noData>
        <Text text="No data
available" icon="sap-icon://..." />
    </noData>
    ...
</SmartTable>
```

You can also use attributes to define binding information for aggregations with cardinality `0..n`. For more information, see *Declarative List Binding in XML Views* under List Binding (Aggregation Binding) [page 828].

Control Properties and Associations in XML Views

Properties

Property values for controls in XML views are specified as attributes of the XML element tag of the control. The name of the attribute corresponds to the name of the property in the API reference, for example, the property `text` of a the `sap.m.Button` control is specified as `text="value"`.

> **i** Note
>
> Escape characters that have a special meaning in XML (like `<`, or `&`) when they occur in a property value. Use XML entities instead (like `<` instead of a `<`, or `&` instead of `&`).

Attributes in XML views use the same binding syntax as constructors of controls. For example, `"{customerName}"` is used to bind a property against the model property with name `"customerName"`.

Associations

- Associations of **cardinality** `1`: Define the ID of the associated element in an attribute that has the same name as the association in the XML view.
- Associations of **cardinality** `0..n`: You can define multiple IDs separated by a blank.

Using Native HTML in XML Views

The use of native HTML in XML views depends on the XHTML feature set.

Context

When mixing XHTML and SAPUI5 controls, observe the following rules:

- XHTML elements can be used instead of the SAPUI5 type control, for example, in the root of an XML view or in the content aggregation of a layout container.
- When embedding XHTML in an aggregation of a SAPUI5 control, the XHTML must not consist of a single text node. The topmost node of an embedded XHTML tree must be an XHTML element. Embedding pure text into an aggregation is not supported.
- The XHTML nodes are converted 1:1 to HTML, the XML view does not deal with any differences between XHTML and HTML (for example rewriting and auto-closing tags)
- The created HTML DOM nodes are preserved during re-rendering of an XML view: Modifications to the DOM are not lost.

> **i** Note
>
> As an alternative to embedding XHTML, you can use the `sap.ui.core.HTML` control. As this requires content encoding it is, however, less convenient.

Procedure

To mix SAPUI5 controls with native XHTML, you only need the XHTML namespace to use (X)HTML:

```
<mvc:View controllerName="sap.hcm.Address" xmlns="sap.m"
xmlns:mvc="sap.ui.core.mvc"
          xmlns:html="http://www.w3.org/1999/xhtml">
   <Panel>
      <Button text="Press Me. I am an SAPUI5 Button"/>
      <html:button>No, press me. I am native HTML Button.</html:button>
   </Panel>
</mvc:View>
```

Using CSS Style Sheets in XML Views

Style sheets are included in XML views in the same way as plain HTML. To add further CSS classes to SAPUI5 controls, use the `class` attribute.

Context

The effect is the same as calling `myButton.addStyleClass(...)`.

> → Tip
>
> We recommend to carefully choose the elements that you style as the CSS always affects the whole page and is **not** restricted to the view.

Procedure

To add a style sheet, add the style definition.
To add a style class and define a button that uses it, add the following coding:

```
<mvc:View controllerName="sap.hcm.Address" xmlns="sap.m"
xmlns:mvc="sap.ui.core.mvc"
          xmlns:html="http://www.w3.org/1999/xhtml">
   <html:style>
      .mySuperRedButton {
         color: red;
```

```
      }
    </html:style>
    <Panel>
       <Button class="mySuperRedButton" text="Press Me"/>
    </Panel>
</mvc:View>
```

Handling Events in XML Views

XML views use event handlers as attributes: The attribute name is the event name, such as "press" for a button, and the attribute value is the event handler name.

Addressing the Event Handler

Depending on the syntax of its name, the event handler will be looked up by this name in different locations:

- Names starting with a dot ('.') are always assumed to represent a method in the controller. They are resolved by removing the leading dot and reading the property with the resulting name from the controller instance. These names are relative to the view/controller. For example, `press=".myLocalHandler"` is resolved by `attachPress(oController["myLocalHandler"], oController);`

  > **i Note**
  >
  > This syntax is by intention consistent to the complex binding syntax for formatter functions.

- Names defined in a `core:require` statement can be used to access static functions of the required modules. For example, `press= "Util.handler"` sets the static `handler` function of the required `Util` module as press handler for the respective control. For more information, see Require Modules in XML View and Fragment [page 799].
- Names containing a dot at a later position are assumed to represent:
  - Static functions from the modules which are loaded through the XML view required modules (See Require Modules in XML View and Fragment [page 799])
  - Global functions if the function cannot be resolved within the XML view require modules and are resolved by calling `ObjectPath.get` with the full name. For example, `name press= "some.global.handler"` is resolved by calling `attachPress(ObjectPath.get("some.global.handler"), oController);`.

    > **i Note**
    >
    > The use of globals is not recommended and they should be replaced, see Require Modules in XML View and Fragment [page 799].

- Names without dot are interpreted as a relative name; if nothing is found, they are interpreted as an absolute name. This variant is only supported for backward compatibility.

> **i Note**
>
> When specified without parameters, the event handler will be called with one argument, the event object. This object can be used to retrieve the event parameters documented by the control's respective event documentation.

The `"this"` Context

As long as no event handler parameters are specified and regardless of where the function was looked up, it will be executed with the controller as the context object (`this`). This is also true for global event handlers and makes the implementation of generic global handlers easier that may need an easy way back to the controller/ view in which they are actually used, for example, to call `createId` or `byId`. This should make the development of global event handlers more consistent with controller local event handlers.

Therefore, the following declaration is equivalent to a call of `controller.doSomething()` when the button is pressed:

```
<Button text="Press Me" press=".doSomething"/>
```

However, once event parameters are specified using the syntax described below, the `this` context is always the object on which the handler function is defined. For controller methods, the controller remains the `this` context, but for methods defined in the XML view required modules or on global objects, that owner object is used as `this` context. In case the controller is still required in such global handler functions, it can be explicitly passed as `$controller` parameter (see the *Passing Parameters* section below). Functions defined directly on the XML view required modules or on the `window` object have an undefined `this` context.

By invoking the special JavaScript function `.call(...)` on your event handler function, you can also provide a different `this` context. For example, you can still have the controller as `this` in an event handler in a global helper object, even when you pass parameters, by doing:

```
<Button core:require="{Helper:'path/to/Helper'}" text="Press Me"
press="Helper.doSomething.call($controller, 'Hello World')"/>
```

Passing Parameters

In XMLViews and JSONViews it is also possible to directly specify the parameters that should be passed into the event handler function. These parameters then are passed instead of the event object. The syntax mimics the JavaScript syntax for function calls:

```
<Button text="Press Me" press=".doSomething('Hello World')"/>
```

Any JavaScript literals including objects and arrays can be passed:

```
<Button text="Press Me" press=".doSomething('string', 0, 5.5, {key1: 'value1',
key2: 'value2'}, ['value1', 'value2'])"/>
```

It is also possible to access model properties. The syntax to be used is the one used within Expression Binding – binding paths are enclosed in **${...}** :

```
<Button text="Press Me" press=".doSomething(${products>unitPrice})"/>
```

The binding context from which relative binding paths are resolved is the context of the control which triggers the event. This means that for a control in a table row, relative binding paths like the one above, always pass the data from the table row where the event occurred. This is very convenient, because it is no longer required to find out the data element to which the table row is bound.

Complex binding syntax can also be used (to add formatters, types etc.), as well as all expressions allowed by Expression Binding [page 845]:

```
<Button text="Press Me" press=".doSomething(${path: 'products>unitPrice',
formatter: '.formatPrice'})" />
<Button text="Press Me" press=".doSomething(10 * ${products>unitPrice})" />
<Button text="Press Me" press=".doSomething(${products>type} === 'Laptop')" />
```

! Restriction

Even though complex bindings can have multiple parts (use multiple data properties) instead of just one data property path, this is not possible for the event parameters. Therefore, you cannot use `parts` in bindings.

Formatters are resolved the same way as the event handlers: a leading dot means the formatter is member of the controller.

i Note

While it seems like regular JavaScript can be written directly in the event handler specification, this is not the case. The entire expression is evaluated as expression binding and only the syntax elements allowed there can be used.

There are two special named models available in event handlers to make accessing certain values easier:

The first one is named `$parameters` and contains the event parameters:

```
<Select change=".doSomething(${$parameters>/selectedItem})" />
```

Here the event parameter `selectedItem` is passed into the event handler.

The other one is named `$source` and is a `ManagedObjectModel` which wraps the control firing the event:

```
<Button text="Press Me" press=".doSomething(${$source>/text})" />
```

Here the text of the pressed button is passed into the event handler.

There are also two special values which can be used as parameters.

The first special value is named `$event` and represents the original event object. This event object is no longer passed to event handlers, once parameters are specified. This is because of the `$parameters` model, which provides access to the event parameters, and it is not needed in most cases. However, when access to the event object is still needed in the event handler, it can be explicitly passed:

```
<Button text="Press Me" press=".doSomething($event)" />
```

This leads to the same result as specifying `.doSomething` without any parameters, but further parameters can of course be given.

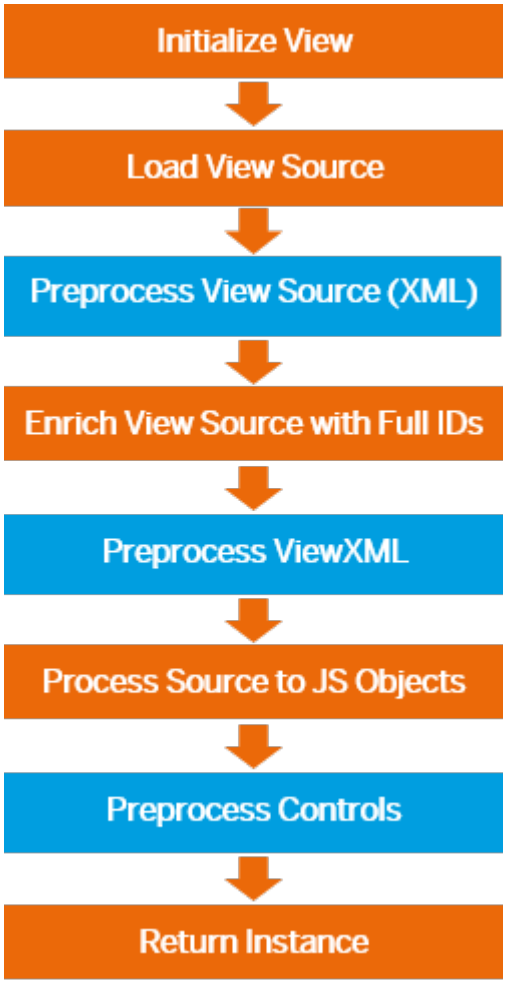
The second special value is `$controller`. As described above, as soon as parameters are specified, the `this` context inside the event handler function is always the object on which this function is defined. However, sometimes it is still required to access the controller even in a handler function which is NOT defined in the controller, but on some other helper object. In this case, the controller can be explicitly passed as one of the parameters:

```
<Button text="Press Me" press=".doSomething($controller)" />
```

Preprocessing XML Views

Applying preprocessing to views enables you to use features like XML templating. This feature is currently only enabled for XML views. On an abstract level, preprocessing means that a view can be modified during runtime before it is rendered. This makes it possible to use the templating syntax, as it is interpreted by the preprocessor. It also makes it possible to apply user customization.

The following figure shows the different stages of view initialization. There are three hooks, XML, ViewXML and controls, which differ mainly in the source that is passed to the preprocessor.



- XML
  The raw view source in the XML format is passed to the preprocessor. This enables example templating on XML level.
- ViewXML
  The raw view source with all XML preprocessing results gets enhanced with its full IDs for the controls and passed on to the preprocessor. Adaptions on a valid view source can now be made before the XML is being parsed into a control tree.
- Controls
  The view source is already processed to the JavaScript object and the control tree is available. This enables you to make changes to the view after design time, like special customizations, stored at some persistence layer or similar, the user has made previously. Or customizations depending on the user role which are not applicable during design time.

For preprocessor for view sources of type XML or ViewXML which create cacheable results, see VML View Cache: Preprocessor Integration [page 798].

Using the Preprocessor

First, you need a preprocessor implementation. SAPUI5 provides a default preprocessor for XML templating which can be enabled by default. You can also build your own preprocessor based on the `Preprocessor` interface. For more information, see the API Reference and the sample in the Demo Kit.

To use the preprocessor, you can choose one of the following options:

- Global availability
  Makes the preprocessor available to all XML views and processes it every time a View is initialized. This maybe the right case for essential operations you want to apply to every view in your application. Therefore you can make use of the `registerPreprocessor` method:

  ```
  // "XMLView" required from module "sap/ui/core/mvc/XMLView"
  XMLView.registerPreprocessor("xml", "sap.ui.sample.samplepreprocessor",
  /* bSyncSupport = */ false, {sampleInfo: "this is a global preprocessor",
  moreInfo: "…"});
  ```

  For more information, see API Reference: `sap.ui.core.mvc.XMLView.registerPreprocessor`.
- Local availability
  Make a preprocessor available to one instance. This can be achieved by passing the preprocessor to the view factory function, which then processes it for this special instance. This would be the choice for smaller tasks the preprocessor should perform.

  ```
  // "XMLView" required from module "sap/ui/core/mvc/XMLView"
  XMLView.create({
         viewName: "sap.ui.core.sample.View.preprocessor.Sample",
         preprocessors: {
                xml: [{
                      preprocessor: "sap.ui.sample.samplepreprocessor",
                      sampleInfo: "this is a local preprocessor",
                      moreInfo: "…"
                },
                {
                      preprocessor: "another.preprocessor"
                }]
         }
  }).then(function(oView) {
  ```

```
        oView.doSomething();
});
```

- On demand availability
  This enables the developer to activate a preprocessor for a view instance, without the need to provide an implementation, by just specifying a preprocessor, which has been registered globally earlier. This is currently used by the XML templating.

> **i Note**
>
> Preprocessors are per default only available for async views. Although it is possible to enable the preprocessor for sync views, we strongly recommend to only use them with async views.

> **i Note**
>
> The information that is provided when using a preprocessor locally will be passed to the preprocessor according to the `mSettings` of the register method.

Related Information

XML View Cache [page 797]

XML View Cache

To be able to speed up processing times of XML views that make heavy use of the preprocessor feature, you can use the view cache to store its processed XML source. Then according network requests for the source and all preprocessor runs that modify the XML source are skipped and the source is taken from the cache.

To make sure that the cache always contains the latest view data, the cache has to be invalidated whenever the data changes that is needed for preprocessing. When the cache is invalidated, all resources are processed again and the cache gets filled with new data.

> **⚠ Caution**
>
> Parts of this feature are currently still experimental. For more information, see API Reference: `sap.ui.xmlview`.

The following data changes are handled automatically by SAPUI5:

- SAPUI5 version changes
- Descriptor file (`manifest.json`)
- Locale (for each locale one cached resource is being created)

> **i Note**
>
> For each additional component that may have an influence on the preprocessing results of the view source, like OData metadata, you have to implement the invalidation by adding additional keys to the cache configuration.

Prerequisites

This feature is only enabled for the following browsers:

- Google Chrome as of version 49 for desktop
- Internet Explorer as of version 11 for desktop

The XML view has to be loaded asynchronously.

Cache Configuration

If you want to keep things simple, you can use the following cache configuration:

```
var sCalculatedCacheKey = oKeyProvider.getCacheKey();
sap.ui.xmlview({
    async: true,
    id: "cacheableView",
    viewName: "my.cacheableView",
    cache: {
        // one key is mandatory
        keys:[sCalculatedCacheKey]
    }
});
```

If you want to pass on multiple keys, for example strings or promises that resolve with a string, you use the following syntax:

```
var pCalculatedCacheKey = oKeyProvider1.getCacheKeyPromise();
var sAnotherKey = oKeyProvider2.getCacheKey();
sap.ui.xmlview({
    async: true,
    id: "cacheableView",
    viewName: "my.cacheableView",
    cache: {
        keys: [
        // several key providers, at least one
        pCalculatedCacheKeyPromise,
        sAnotherKey
        ]
    }
});
```

Preprocessor Integration

If you want to implement a preprocessor that has influence on the creation of views, for example, by changing the XML code, you can use function `getCacheKey`. With this function, the view can find out whether the preprocessor triggers changes that invalidate the cache. The function returns a cache key or a promise that resolves a cache key. For more information, see API Reference: `sap.ui.core.mvc.View.Preprocessor`.

```
// Example preprocessor implementation
sap.ui.define(['jquery.sap.global', 'sap/ui/base/Object'],
    function(jQuery, BaseObject) {
        'use strict';
```

```
        var TestPreprocessor = BaseObject.extend("smy.TestPreprocessor", {});
        TestPreprocessor.process = function(vSource, sCaller, mSettings) {
            return doSomething(vSource));
        };
        TestPreprocessor.getCacheKey = function(oViewInfo) {
            return sCacheKey;
        };
        return TestPreprocessor;
    }, /* bExport= */ true);
```

Related Information

API Reference: `sap.ui.xmlview`

Preprocessing XML Views [page 795]

Require Modules in XML View and Fragment

Modules can be required in XML views and fragments and assigned to aliases which can be used as variables in properties, event handlers, and bindings.

The `require` attribute with namespace URI `sap.ui.core` can be used to define the module aliases and paths. In the following sections we assume that the namespace prefix `core` is used to define the URI `sap.ui.core` which makes the attribute to be written as `core:require`. This attribute can be used at every element of an XML view or fragment. You can specify a list of required modules as Unified Resource Names, similar to `sap.ui.require`, and assign aliases to them using a JSON-like syntax.

```
<mvc:View xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc"
   core:require="{
      Box: 'sap/m/MessageBox',
      Toast: 'sap/m/MessageToast'
   }">
   ...
</mvc:View>
```

> **i Note**
>
> The modules defined in the `core:require` attribute are loaded first before any other attributes of the element with `core:require` are processed. Therefore, they can be used in the same element for bindings, event handlers, and so on.

`core:require` can only handle static imports which require the module path to be defined by using a string literal. It is not possible to use a binding or an expression for defining the module path. As `core:require` is not interpreted as a binding expression, it is not necessary to escape the curly braces in `core:require` which is different than in the other attributes.

The aliases can then be used to access the modules' static functions. The alias is valid for the element where the alias is defined and the subtree of that element.

> **i Note**
>
> When you use the view in combination with fragments, keep in mind that the alias does not work in embedded fragments. In this case, define a separate `core:require` inside the fragments.

Example With Event Handler

You can use the XML `require` to reference static functions of a module which can serve as event handlers. This works with static strings as well as with any model data. For a description how this is done, see Handling Events in XML Views [page 792]. As the `Box` module is defined on the root element, it can be used in the whole view.

```
<mvc:View controllerName="some.Controller" xmlns="sap.m"
      xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc"
      core:require="{Box:'sap/m/MessageBox'}">
   <Panel>
      <Image src="http://www.sap.com/global/ui/images/global/sap-logo.png"/>
      <Button text="Press Me!" press="Box.show('Hello!')"/>
   </Panel>
</mvc:View>
```

> **i Note**
>
> `$controller`, `$event`, `$parameters`, and `$source` are reserved keywords for resolving an event handler. Avoid using these keywords as aliases for the required modules with `core:require`.

Example With Data Binding

You can also use the `require` module with data binding. Formatters and factory functions can be defined with the `require` modules, as well as expression bindings. The following code extract also shows, that the `Factory` module, which is defined on the List element, can only be used there, and not in sibling or parent controls:

```
<mvc:View controllerName="some.Controller" xmlns="sap.m"
      xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc"
      core:require="{Util:'some/Util'}">
  <Panel>
      <Image src="http://www.sap.com/global/ui/images/global/sap-logo.png"/>
      <Text text="{formatter: 'Util.format', path: '/text'}"/>
      <List core:require="{
              Factory:'some/Factory'
          }" id="list" items="{path:'/items', factory:'Factory.createItem'}">
      </List>
   </Panel>
</mvc:View>
```

`core:require` in Fragments

`core:require` can be used in fragments and set on every element, including `FragmentDefinition`. However, `core:require` on `FragmentDefinition` node does not have any effect in the following use cases:

- The view where a fragment is used is preprocessed. For more information about preprocessing, see Preprocessing XML Views [page 795].
- A fragment is defined inline within a View.

```
<mvc:View controllerName="some.Controller" xmlns="sap.m"
      xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc">
  <FragmentDefinition>
      <!-- core:require can't be defined on the above FragmentDefinition -->
      ...
  </FragmentDefinition>
  ...
</mvc:View>
```

`core:require` does not work as expected in both of the above use cases because the `FragmentDefinition` node is **not** part of the resulting view. Therefore, the module information which is defined in the `core:require` on `FragmentDefinition` node is not available for its child nodes.

JSON View

The JSON view type is defined in a file. The file name has to either end with `.view.json` or as a JSON string. The file name and the folder structure together specify the name of the view that equals the SAPUI5 module name within the modularization concept.

For the file `resources/sap/hcm/Address.view.json`, the view name is `sap.hcm.Address`. The application uses this view name for displaying an instance of this view.

The file looks as follows:

```
{
   "Type":"sap.ui.core.mvc.JSONView",
   "controllerName":"sap.hcm.Address",
   "content": [{
      "Type":"sap.m.Image",
      "id":"MyImage",
      "src":"http://www.sap.com/global/ui/images/global/sap-logo.png"
   },
   {
      "Type":"sap.m.Button",
      "id":"MyButton",
      "text":"Press Me"
   }]
}
```

Nest the JSON objects analogous to the nesting of SAPUI5 controls and add the property values as attributes. The syntax is the same as the syntax of a JSON constructor for any control.

> **i Note**
>
> You can use strings, Boolean values, and null in your JSON view.

Aggregation Handling

You add child controls as arrays. This is shown in the example above where an image and a button have been added to the view content aggregation.

Event Handling

In JSON views, event handlers are bound as attributes with the attribute name as event name like `press` for a button and the attribute value as event handler name.

The following declaration causes `controller.doSomething()` to be executed when the button is pressed:

```
...
   {
      "Type":"sap.m.Button",
      "id":"MyButton",
      "text":"Press Me",
      "press":"doSomething"
   }
...
```

The location in which an event handler is looked up, are similar to XML views (see Handling Events in XML Views [page 792]).

Data Binding

You can bind data in JSON views. To bind the texts of a control to a language-dependent resource bundle, define the resource bundle via name (`resourceBundleName` property) or a URL (`resourceBundleUrl` property) and assign an alias (`resourceBundleAlias` property) for the bundle within the view definition. The binding path is the same for all other SAPUI5 data bindings.

Resource bundle content:

```
MY_TEXT=Hello World
```

Example

```
{   "Type": "sap.ui.core.JSONView",
    "controllerName":"my.own.views.test",
    "resourceBundleName":"myBundle",
    "resourceBundleAlias":"i18n",
    "content": [{
        "Type":"sap.m.Panel",
        "id":"myPanel",
        "content":[{
            "Type":"sap.m.Button",
            "id":"Button1",
            "text":"{i18n>MY_TEXT}",
            "press": "doIt"
        }]
    }]
}
```

The `ResourceModel` for binding this texts is created during view instantiation. The model is set as secondary model with the given alias to the view instance. To bind other properties to another model, create the model in the corresponding controller or HTML page and attach it to the view with another alias.

JS View

You create a JS (JavaScript) view in the same way as a controller and use the suffix `.view.js` for the file.

SAPUI5 provides the following two default methods for implementation:

- `getControllerName()`: Specifies the controller belonging to this view
  If this method is not implemented or returns `NULL`, the view has no controller.
- `createContent()`: Called initially once after the controller has been instantiated
  This method is used to create the UI. As the method knows the controller, it can directly attach the event handlers.

Example

```
   sap.ui.jsview("sap.hcm.Address", {  // this View file is called
Address.view.js

   getControllerName: function() {
      return "sap.hcm.Address";     // the Controller lives in
Address.controller.js
   },
   createContent: function(oController) {
      var oButton = new sap.m.Button({text:"Hello JS View"});
      oButton.attachPress(oController.handleButtonClicked);
      return oButton;
   }
});
```

The string in quotes denotes the view name that equals the SAPUI5 module name within the define/require concept.

Caution

In event handlers for controls "this" usually denotes the control itself. This is unexpected when it happens in event handlers that are implemented inside controllers: The controller would usually expected to be denoted as "this". This is no issue for declarative view types, but for JSViews the view developer may need to modify the "this" context as follows:

```
   ...
      oButton.attachPress(jQuery.proxy(oController.handleButtonClicked,
oController));
   ...
```

Alternatively, the view developer can give the event handler method in an array where the second element is the "this" object:

```
   ...
      var oButton = new sap.m.Button({
         text: "Hello JS View",
```

```
          press: [oController.handleButtonClicked, oController]
        });
    ...
```

⚠ Caution

If you want to define IDs for controls inside a JSView to guarantee their uniqueness when reusing views, you can **not** give hardcoded IDs, but have to give the view the opportunity to add its own instance ID as a prefix. This is done by using the `View.createId(...)` method. For the example above, this is done as follows:

```
var oButton = new sap.m.Button(this.createId("myButton"), {text:"Hello JS
View"});
```

This is not required for declarative view types as the view parser can manage this automatically, see Support for Unique IDs [page 814].

HTML View

An HTML View is defined by declarative HTML. Like the declarative support, the HTML view supports embedded HTML. The view file ends with `view.html`, for example `myview.view.html`.

Example

```
  <template data-controller-name="example.mvc.test">
   Hello
  <h1>Title</h1>
  <div>Embedded HTML</div>
  <div class="test test2 test3" data-sap-ui-type="sap.m.Panel" id="myPanel">
  <div class="test test2 test3" data-sap-ui-type="sap.m.Button" id="Button1"
data-text="Hello World" data-press="doIt"></div>
  <div data-sap-ui-type="sap.m.Button" id="Button2" data-text="Hello"></div>
  <div data-sap-ui-type="sap.ui.core.mvc.HTMLView" id="MyHTMLView" data-view-
name="example.mvc.test2"></div>
  <div data-sap-ui-type="sap.ui.core.mvc.JSView" id="MyJSView" data-view-
name="example.mvc.test2"></div>
  <div data-sap-ui-type="sap.ui.core.mvc.JSONView" id="MyJSONView" data-view-
name="example.mvc.test2"></div>
  <div data-sap-ui-type="sap.ui.core.mvc.XMLView" id="MyXMLView" data-view-
name="example.mvc.test2"></div>
  </div>
</template>
```

All view-specific properties can be added to the `<template>` tag as `data-*` attributes.

Related Information

Declarative Support [page 1057]

Instantiating Views

To instantiate views asychronously, SAPUI5 provides the factory method `View.create` defined in module `sap/ui/core/mvc/View`.

To pass the required information for the instantiation, use an object with the following properties:

- `type`: The type can be `JSON`, `JS`, `XML` or `HTML`. All possible types are declared in the enumeration `sap.ui.core.mvc.ViewType`.
- `viewName`: View name corresponding to the module concept
- `viewContent`: Only relevant for XML views and JSON views. Defines the XML or JSON string representation of the view definition. If `viewName` and `viewContent` are given, the `viewName` property is used to load the view definition.
- `Controller`: Any controller instance; the given controller instance overrides the controller defined in the view definition
- `viewData`: Only used for JS views; this property contains user-specific data that is available during the whole lifecycle of the view and the controller

All regular properties of a view (control) can be passed to the object as usual.

Loading Views

The default mode is the asynchronous loading of a view: The advantage of asynchronous loading compared to synchronous loading is that the UI does not freeze for the duration of the loading process and there is no blockage of functionalities during view initialization.

With the asynchronous loading of views, the instance is not fully available at the moment of creation, instead you may receive a `Promise` via the `View.prototype.loaded` method. The following code snippet shows how the view instance is available in the resolve function of the `promise`.

> **i** Note
>
> If you access the view in the controller's `onInit` callback, the view instance is available in any case. The behavior does not change.

```
// "View" required from "sap/ui/core/mvc/View"
// "coreLibrary" required from "sap/ui/core/library"
// "my.own.controller" was defined earlier
View.create({
    viewName: "my.own.view",
    controller: "my.own.controller",
    type: coreLibrary.mvc.ViewType.XML
}).then(function(oView) {
    // the instance is available in the callback function
    oView.placeAt("uiArea");
});
```

Synchronous Mode

> **i** Note
>
> We do **not** recommend this mode. Use the asynchronous mode instead.

The following code snippet creates a view instance, loads the view source, places the instance to the `uiArea`, and renders it later on.

```
var oController = sap.ui.controller("my.own.controller");
var oView = sap.ui.view({
    viewName: "my.own.view",
    controller: " my.own.controller",
    type: sap.ui.core.mvc.ViewType.XML
});
// the instance is available now
 oView.placeAt("uiArea");
...
```

Lazy Loading for XML Views

The following code snippet shows how to do a lazy loading for XML views:

```
<!-- File: view/CustomView.view.xml -->
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <Text text="Custom View loaded ..."/>
</mvc:View>
```

```
// File: controller/MainController.controller.js
sap.ui.require(["sap/ui/core/mvc/XMLView", "sap/ui/core/mvc/Controller"],
function(XMLView, Controller) {
    return Controller.extend("samples.controller.MainController", {
        // ...
        onSomeEventTriggered: function() {
            // instantiate view using create-factory
            XMLView.create({
                viewName: "samples.view.CustomView"
            })
            .then(function(oCustomView) {
                // View loaded ...
            })
        }
    });
 });
```

For an example, see the `sap.ui.core.sample.View.async/preview` sample in the Demo Kit.

View Cloning

For normal controls, view cloning bases on control settings that are described by SAPUI5 metadata, such as properties, aggregations, associations, and event handlers. The clone operation collects these settings and creates a new instance.

Another important aspect of SAPUI5 views is their cloning behavior. As you might know, SAPUI5 aggregation bindings can use template control to create a series of similar controls based on a collection of data, for example, items in a `RowRepeater` for each entry in a model array. The data binding uses a `ManagedObject.clone` operation to create multiple controls out of a single template.

For views there is a conflict between this basic, generic approach and the way how views usually define their content: via hooks (JSView) or via persisted XML or JSON files. Furthermore, it is allowed and documented

best practice to modify the view in the onInit hook of its controller. To avoid conflicts between the generic cloning and the MVC concepts, views implement a slightly modified clone operation: only a subset of the view settings are cloned, the remainder is re-created by calling the hook (JSView) or applying the external view description (XML or JSON file), depending on the view type.

Cloned in a generic way are the following settings:

- any models that have been set (`setModel()`)
- registered control event listeners (`attachSomeEvent`)
- registered browser event listeners (`attachBrowserEvent`)
- bindings (`bindProperty`, `bindAggregation`)

Not cloned, but recreated are all aggregations, namely the content.

In scenarios where the above clone approach still leads to undesirable behavior, factory functions can be used for the aggregation binding instead.

Related Information

List Binding (Aggregation Binding) [page 828]

Controller

A controller contains methods that define how models and views interact.

You define a simple controller as follows:

```
sap.ui.controller("sap.hcm.Address", {
   // controller logic goes here
});
```

The string in quotes specifies the controller name. The controller file's name should be named as the string in the quotes, `Address.controller.js`.

> **i Note**
>
> The suffix `.controller.js` is mandatory for controllers.

Lifecycle Hooks

SAPUI5 provides predefined lifecycle hooks for implementation. You can add event handlers or other functions to the controller and the controller can fire events, for which other controllers or entities can register.

SAPUI5 provides the following lifecycle hooks:

- `onInit()`: Called when a view is instantiated and its controls (if available) have already been created; used to modify the view before it is displayed to bind event handlers and do other one-time initialization

- `onExit()`: Called when the view is destroyed; used to free resources and finalize activities
- `onAfterRendering()`: Called when the view has been rendered, and therefore, its HTML is part of the document; used to do post-rendering manipulations of the HTML. SAPUI5 controls get this hook after being rendered.
- `onBeforeRendering()`: Called every time the view is rendered, before the renderer is called and the HTML is placed in the DOM tree.

> **i** Note
>
> For controllers without a view, no lifecycle hooks are called.

> **Example**
>
> ```
> sap.ui.controller("sap.hcm.Address", {
>    onInit: function() {
>       this.counter = 0;
>    }
> });
> ```

Event Handlers and Other Functions

In addition to lifecycle hooks, a controller can define additional methods that serve as event handlers or additional functionality offered by the controller.

> **Example**
>
> ```
> sap.ui.controller("sap.hcm.Address", {
>    increaseCounter: function() {
>       this.counter++;
>    }
> });
> ```

`Methods` Section in the Controller Metadata

By default, all methods that **do not** start with an underscore or with prefix "on", "init" or "exit" are public. You can get all public methods of a controller by using the `oController.getMetadata().getPublicMethods()` API.

When you use the new `methods` section in the controller metadata, only functions prefixed by "_" become private by default. In addition, you get the possibility to control the visibility, flag methods as final, or define an `overrideExecution` strategy. The same applies for the new controller extension metadata. This makes the definition of a public interface more flexible.

Only public methods and methods that are not flagged as final could be overridden by a controller extension.

Note

If you don't use the new `methods` definition for controllers, you could override the `onInit`, `onExit`, `onAfterRendering` and `onBeforeRendering` methods of the controller even if they are private by default.

The following sample code shows how to define an extension to an existing controller.

Example

Sample controller extension:

```
sap.ui.define(['sap/ui/core/mvc/Controller', 'sap/ui/core/mvc/
OverrideExecution'], function (Controller, OverrideExecution) {
    "use strict";
    return Controller.extend("sap.hcm.Address", {
        metadata: {
            // extension can declare the public methods
            // in general methods that start with "_" are private
            methods:{
                publicMethod: {public: true /*default*/, final: false /
*default*/, overrideExecution: OverrideExecution.Instead /*default*/},
                finalMethod: {final: true},
                onMyHook: {public: true /*default*/, final: false /
*default*/, overrideExecution: OverrideExecution.After},
                couldBePrivate: {public: false}
            }
        },
        // adding a private method, only accessible from this controller
        _privateMethod: function() {
        },
        // adding a public method, might be called from, but not overridden
by other controllers or controller extensions as well
        publicMethod: function() {
        },
        // adding a final method, might be called from, but not overridden by
other controllers or controller extensions as well
        finalMethod: function() {
        },
        // adding a hook method, might be called from, but not overridden by
a controller extension
        // override these method does not replace the implementation, but
executes after the original method
        onMyHook: function() {
        },
        // method public by default, but made private via metadata
        couldBePrivate: function() {
        }

    });
});
```

Controller Extensions Implementation Guidelines

All public methods need to stay compatible:

- Parameters of the method can be enhanced only with new optional parameters.
- It is recommended to use a JS object to pass the parameters. Extension can be done by adding an optional key, while working with parameters needs to stick to the sequence.

- Documentation should be maintained for all public methods.
- Use the `@since` version to tell the consumer on which version this method was introduced.

> **i Note**
>
> Within the methods of a controller extension, the reserved base member allows access to the public functions of the extended controller.
>
> Functionality can be called by using `this.base.basePublicMethod()`.

For more information on how to use controller extensions, see Using Controller Extension [page 810].

API Reference

sap.ui.core.mvc.Controller

Using Controller Extension

Controller extensions allow you to add functionality to existing applications. This can be used for extensibility purposes, for example a customer extending SAP-delivered applications, or as a reusable part that is added to the original application.

Overview

The following sample code shows how to define an extension for an existing controller.

```
sap.ui.define(['sap/ui/core/mvc/ControllerExtension', 'sap/ui/core/mvc/
OverrideExecution'], function(ControllerExtension, OverrideExecution) {
  "use strict";
  return ControllerExtension.extend("my.extension.SampleExtension", {
    metadata: {
      // extension can declare the public methods
      // in general methods that start with "_" are private
      methods: {
        publicMethod: {
          public: true /*default*/ ,
          final: false /*default*/ ,
          overrideExecution: OverrideExecution.Instead /*default*/
        },
        finalMethod: {
          final: true
        },
        onMyHook: {
          public: true /*default*/ ,
          final: false /*default*/ ,
          overrideExecution: OverrideExecution.After
        },
        couldBePrivate: {
          public: false
        }
```

```
      }
    },
    // adding a private method, only accessible from this controller extension
    _privateMethod: function() {},
    // adding a public method, might be called from or overridden by other
controller extensions as well
    publicMethod: function() {},
    // adding final public method, might be called from, but not overridden by
other controller extensions as well
    finalMethod: function() {},
    // adding a hook method, might be called from but not overridden by other
controller extensions
    // overriding these method does not replace the implementation, but executes
after the original method
    onMyHook: function() {},
    // method public by default, but made private via metadata
    couldBePrivate: function() {},
    // this section allows to extend lifecycle hooks or override public methods
of the base controller
    override: {
      // override onInit of base controller
      onInit: function() {},
      // override public method of the base controller
      basePublicMethod: function() {}
    }
  });
});
```

For more detailed information on lifecycle hooks and controller metadata, see Controller [page 807].

Custom Lifecycle Events

If you want to have additional lifecycle events like the standard `onInit` or `onExit` for developers extending the controller, you can define them as described in this section.

Controller extensions allow you to define custom lifecycle hooks. In the `Sample`, the `ReuseExtension.js` defines a custom lifecycle hook by specifying an `overrideExecution` function:

```
return ControllerExtension.extend("sap.my.ReuseExtension", {
        metadata: {
            methods: {
                "onFilterHook": {"public": true, "final": false,
overrideExecution: OverrideExecution.After}
            }
        },
        //...
        /**
         * @abstract
         */
        onFilterHook: function(aFilter) {
        }
    });
```

With `OverrideExecution.After`, the extensions are called in the order they are provided, with `OverrideExecution.Before` the order is reversed and the last extension is called first.

Final Methods in Controller Extensions

Adding `"final": true` metadata to the public method makes it available for execution (call) but not for overriding in the next controller extension.

```
return ControllerExtension.extend("sap.my.ReuseExtension", {
        metadata: {
            methods: {
```

```
                "myPublicMethod": {"public": true, "final": true}
            }
        },
        myPublicMethod: function() {
        }
    });
```

Accessing Controls in Controller Extensions

Only controls that belong to an extension are accessible by the `byId` function in a controller extension. These controls must be prefixed by the namespace of the controller extension. The namespace can be retrieved by calling `getMetadata().getNamespace()`. Here is an example of a valid ID: `my.controller.extension.MyControlId`.

Using the `byId` of the base controller allows the accessing of all controls of the corresponding view by calling `this.base.byId(myControlId)`.

Integrating Controller Extensions into Controllers

Controller extensions can serve for reuse purposes. You can achieve this by including a controller extension to your controller as a member.

```
sap.ui.define(['sap/ui/core/mvc/Controller', 'my/extension/SampleExtension'],
function(Controller, SampleExtension) {
    "use strict";
    return Controller.extend("sample.Main", {
        //include the extension
        sample: SampleExtension,

        _basePrivateMethod: function() {
            ...
        },
        basePublicMethod: function() {
     ...
    });
});
```

If the controller is instantiated, all members that have a `ControllerExtension` associated will create an instance of these controller extensions.

You can also override an extension directly in a controller.

Sample Code

```
sap.ui.define(['sap/ui/core/mvc/Controller', 'my/extension/SampleExtension'],
    function(Controller, SampleExtension) {
    "use strict";
    return MainController = Controller.extend("sample.Main", {
        //inline override of an extension. E.g. to provide a hook
implementation
        sample: SampleExtension.override({
            someHook: function() {},
            someOtherMethod: function() {}
        }),
        onLifecycleHook: function() {
        }
    });
});
```

File Names and Locations (View and Controller)

In SAPUI5, controllers and views are defined and instantiated via a name that equals an SAPUI5 module name within the define/require concept.

By default, all files have to be located in a subfolder of the `resources` folder of the Web application. If this location is not appropriate, deviations can be configured as described in the following example:

The following example assumes that your views and controllers are located on your local machine where the SAPUI5 runtime is loaded from another machine. When you instantiate a view or a controller, SAPUI5 runtime loads them in relation to the `resources` folder of the machine where SAPUI5 runtime was loaded. To inform SAPUI5 runtime that your views and controllers are located on your local machine, use the following code:

```
sap.ui.loader.config({
  paths: {
    "<moduleNamePrefix>": sUrl
  }
});
```

If your files are located at `http://<localhost:8080>/<myapp>/`, for example, you can use `sap.ui.loader.config` as follows:

```
sap.ui.loader.config({
  paths: {
    "my/app": "./myapp"
  }
});
```

All views and controllers with a name starting with `my.app`, for example `my.app.MyView`, will then be loaded from your local machine.

Related Information

Folder Structure: Where to Put Your Files [page 1428]

Typed Views and Controllers

More complex use cases may require a more formal way to define views and controllers. For this, typed views and controllers are used.

To create a controller that is a new type of its own, you need to write a boilerplate code and declare the functions of the new prototype:

```
/* boilerplate code for typed Controller */
jQuery.sap.declare({modName:"sap.hcm.AddressController", type:"controller"}); //
declaring a special type of module
```

```
sap.hcm.AddressController = function () { // the constructor
   sap.ui.core.mvc.Controller.apply(this, arguments);
};
jQuery.sap.require("sap.ui.core.mvc.Controller"); // this is currently required,
as the Controller is not loaded by default
sap.hcm.AddressController.prototype =
jQuery.sap.newObject(sap.ui.core.mvc.Controller.prototype); // chain the
prototypes
/* end of boilerplate code for typed Controller */

// to avoid the above we could in the future offer it behind a simple call to:
// sap.ui.defineController("sap.hcm.Address");

sap.hcm.AddressController.prototype.onInit = function() {
   // modify control tree - this is the regular lifecycle hook
};

// implement an event handler in the Controller
sap.hcm.AddressController.prototype.doSomething = function() {
   alert("Hello World");
};
```

Support for Unique IDs

Stable IDs are used to identify and modify the controls within the controller during runtime. However, if you reuse or nest these views, these stable IDs are no longer unique. To avoid ambiguity, each view adds its own ID as prefix to all its child controls.

If the ID is created during instantiation of the control, it is unique by default. If you create further controls during runtime, the controller creates a unique ID by calling the `oController.createId("ID")` method. These methods add the necessary prefix to the ID.

If you want to modify the control with the ID `<ID>`, you can call the `byId(<ID>)` method on your view to get the correct control directly. You do not have to handle all the prefix stuff on your own.

The following view defines a button with the stable ID `aButton` (in the `ButtonView`):

```
<mvc:View viewName="sap.hcm.ButtonView" controllerName="sap.hcm.myController"
xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
      <Button id="aButton" text="Click me"/><mvc:View>
```

The following view defines a view embedding the same view several times (`ContainerView`):

```
<mvc:View viewName="sap.hcm.ContainerView" controllerName="sap.hcm.Address"
xmlns="sap.ui.commons" xmlns:core="sap.ui.core"
           xmlns:html="http://www.w3.org/1999/xhtml">
      <mvc:View id="ButtonView1" viewName="sap.hcm.ButtonView"/>
      <mvc:View id="ButtonView2" viewName="sap.hcm.ButtonView"/>
<mvc:View>
```

The view is created as follows:

```
...
   // "View" required from module "sap/ui/core/mvc/View"
   View.create().then(function(oView) {/* code */});
...
```

The container view has the following IDs:

Both child view IDs have the prefix `myContainerView--`:

`myContainerView--ButtonView1`

`myContainerView--ButtonView2`

To get one of the child views, use the following code: Essentials [page 691]

```
...
var oButtonView1 = oView.byId("ButtonView1");
...
```

The button view has the following IDs:

`ButtonView1--aButton`

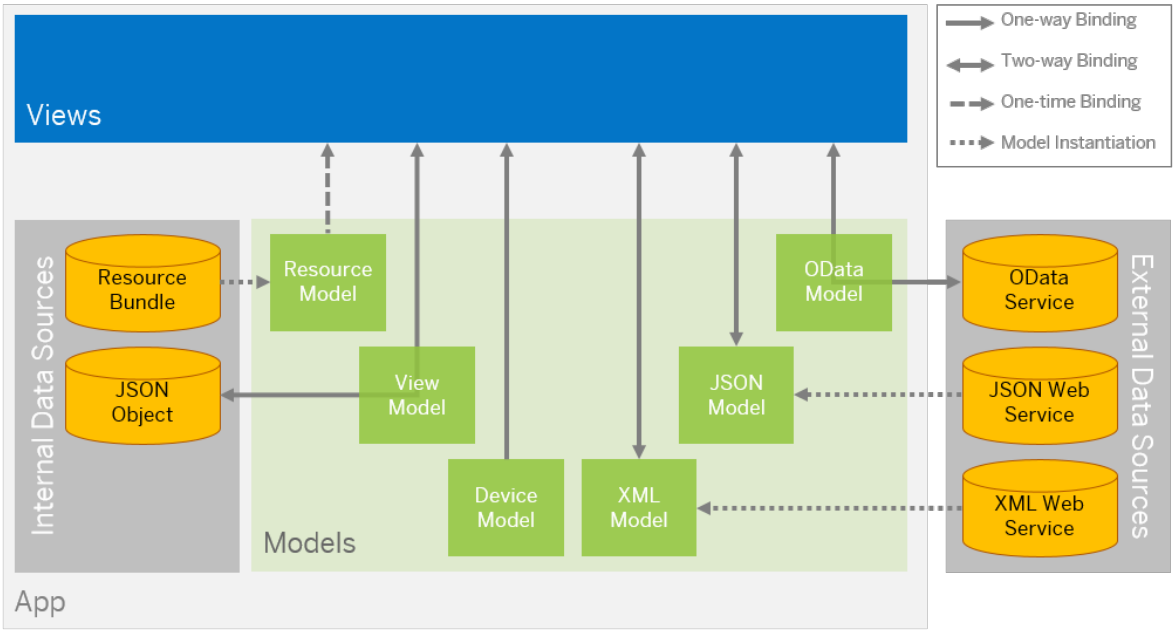
`ButtonView2--aButton`

To get the button control, use the following code:

```
...
   var oButton = oButtonView1.byId("aButton");
...
```

Data Binding

You use data binding to bind UI elements to data sources to keep the data in sync and allow data editing on the UI.



- Views [page 787]

- Models [page 882]
- Resource Bundles [page 1272]

SAPUI5 follows the "Model View Controller" (MVC) paradigm, which means that we clearly separate data sources (model), UI (view), and application logic (controller) from each other. Data binding defines how models and views communicate with each other.

Depending on which external data source you use, you can choose between different model types to represent it. SAPUI5 supports OData V4 (with restrictions), OData V2, JSON, and XML models.

There are also internal data sources that are defined in the app for specific purposes. For those, an app contains the following models:

- The **resource model** is used to communication with the resource bundle that contains translatable texts in multiple languages
- The **device model** is provided by the framework and defines device-specific settings
- **View models** can be, for example, JSON models that communicate with a corresponding JSON object. JSON data can also be edited in the app, but they are not stored - as soon as you refresh the browser or restart the app, the changes are reset.

Most of the models are client-side models. This means that all data is initially loaded to the model when the app is started. All actions performed on the data are only executed on the client, and only sent back to the data source when this is triggered by the app. Client-side models are therefore only recommended for small data sets.

The OData models (V2 and V4) are server-side models, which means that data is only requested on demand from the back end. Filtering, sorting, and paging actions are performed on the server. This means, for example, that you don't have to load a complete table on the UI to be able to sort the entries.

In the view, you bind data by specifying the **binding path** for a control. You can use **data types** and **formatters** to validate and format the data on the UI.

> **i** Note
>
> To learn more about data binding use the tutorial: Data Binding [page 219]

Binding Modes: One-time Binding, One-way Binding, and Two-way Binding

The binding mode defines how the data sources are bound. SAPUI5 provides the following binding modes:

- One-way binding means a binding from the model to the view; value changes in the model update all corresponding bindings and the view
- Two-way binding means a binding from the model to the view and from the view to the model, changes in the model and in the view fire events that automatically trigger updates all corresponding bindings and the view and model
- One-time binding means from model to view once.

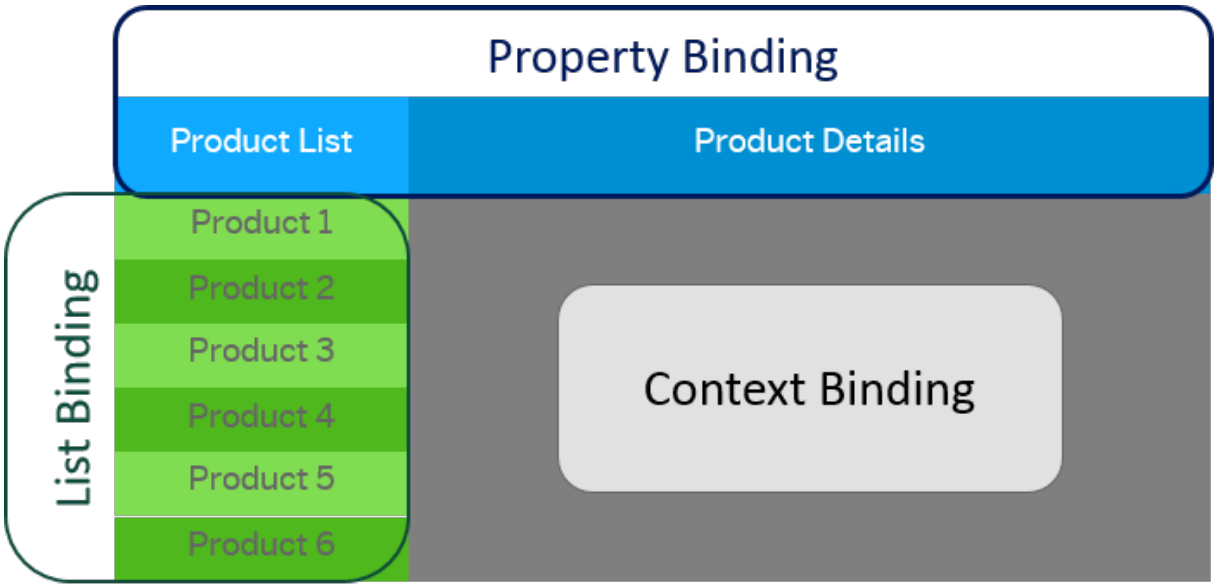
The following table shows which binding modes the respective binding models support:

Model	One-time	One-way	Two-way
OData V4 model	✓	✓	✓ (default)
OData V2 model	✓	✓ (default)	✓
ODataMetaModel V4	✓ (default)	✓	✗
ODataMetaModel	✓ (default)	✗	✗
JSON model	✓	✓	✓ (default)
XML model	✓	✓	✓ (default)
Resource model	✓	✓ (default)	✗

For more information, see API Reference: `sap.ui.model.BindingMode`.

Binding Types

Depending on the different use cases, you can use different binding types: Propety binding, context binding, and list binding.



- Property Binding [page 818]
- List Binding (Aggregation Binding) [page 828]

- Context Binding (Element Binding) [page 824]
- **Property binding** allows properties of the control to get automatically initialized and updated from model data. You can only bind control properties to model properties of a matching type, or you use a formatter or a data type to parse and convert the data as needed For more information, see Formatting, Parsing, and Validating Data [page 854].
- **Context binding** (or **"element binding"**) allows to bind elements to a specific object in the model that creates a binding context and allows relative binding within the control and all of its children. This is especially helpful in master-detail scenarios.
- **List binding** (or **"aggregation binding"**) can be used to automatically create child controls according to model. This can be done either by cloning a template control, or by using a factory function. Aggregations can only be bound to lists defined in the model, that is, to arrays in a JSON model or a collection in the OData model.

> **i** Note
>
> The model has a default size limit to avoid too much data being rendered on the UI. This size limit determines the number of entries used for the list bindings. The default size limit is 100 entries.
>
> This means that controls that don't support paging or don't request data in chunks (e.g. `sap.m.ComboBox`) only show 100 entries even though the model contains more items.
>
> To change this behavior, you can set a size limit in the model by using `oModel.setSizeLimit`.

Property Binding

With property binding, you can initialize properties of a control automatically and update them based on the data of the model.

To define property binding on a control, you have the following options:

- As part of the control's declaration in an XML view
- Using JavaScript, in the `settings` object in the constructor of a control, or in special cases, using the `bindProperty` method of a control

Once you have defined the property binding, the property is updated automatically every time the property value of the bound model is changed, and vice versa.

Let's say, we have the following JSON data:

```
{
"company" : {
        "name"   : "Acme Inc."
        "street": "23 Franklin St."
        "city"   : "Claremont"
        "state" : "New Hampshire"
        "zip"    : "03301"
        "revenue": "1833990"
          }
}
```

To define property binding in the control declaration in the **XML view**, just include the binding path within curly brackets (see also Binding Path [page 842]):

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input
        value="{/company/name}"
    />
</mvc:View>
```

In **JavaScript**, you can include the binding path within curly brackets as a string literal in the `settings` object:

```
// "Input" required from module "sap/m/Input"
var oInput = new sap.m.Input({
    value: "{/company/name}"
});
```

You can also use a complex syntax for property bindings. This complex syntax allows you to define additional binding information to be contained in the `settings` object, such as a formatter function.

If you are working with **XML views**, make sure that you've turned on complex binding syntax in your bootstrap script, as shown here:

```
<script
       id="sap-ui-bootstrap"
      src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js"
      data-sap-ui-theme="sap_belize"
     data-sap-ui-bindingSyntax="complex"
      data-sap-ui-async="true"
      data-sap-ui-onInit="module:sap/ui/sample/main"
       data-sap-ui-resourceRoots='{"sap.ui.sample": "./"}'
></script>
```

You can also use `data-sap-ui-compatVersion="edge"` to enable complex bindings.

You can then set the `bindingMode` or other additional properties like this:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input
        value="{
            path:'/company/name',
            mode: 'sap.ui.model.BindingMode.OneWay'
        }"
    />
</mvc:View>
```

In **JavaScript** views or controllers, you use a JS object instead of a string literal. This must contain a `path` property containing the binding path, and can contain additional properties:

```
// "Input" required from module "sap/m/Input"
// "BindingMode" required from module "sap/ui/model/BindingMode"
var oInput = new Input ({
    value: {
        path: "/company/name",
        mode: BindingMode.OneWay
    }
});
```

Depending on the use case, it may be useful to define the binding at a later time, using the `bindProperty` method:

```
oInput.bindProperty("value", "/company/name");
```

This option also allows you to use the same object literal that you used in the constructor to define the binding:

```
// "TypeInteger" required from module "sap/ui/model/type/Integer"
oInput.bindProperty("value", {
    path: "/company/name",
    type: new TypeInteger()
});
```

> **i** Note
>
> Some controls offer convenience methods for their main properties that are most likely to be bound by an application:
>
> ```
> oTextField.bindValue("/company/name");
> ```

To **remove** a property binding, you can use the `unbindProperty` method. The property binding is removed automatically whenever a control is destroyed:

```
oTextField.unbindProperty("value");
```

Formatting Property Values

Values in data are often represented in an internal format and need to be converted to an external format for visual representation, especially numbers, dates, and times with locale-dependent external formats. SAPUI5 provides two different options for converting data. You can use both options for each binding, you don't have to use one option consistently throughout your app:

- Formatter functions for one-way conversion
- Data types in two-way binding
  Data types can be used to parse user input in addition to formatting values.

Using a Formatter Function

If you define the property binding in the **XML view**, you need to define a formatter function (`roundToMillion`) in the view controller:

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel"
], function (Controller, JSONModel) {
    "use strict";
    return Controller.extend("sap.ui.sample.App", {
        ..............
        roundToMillion: function(fValue) {
            if (fValue) {
                return "> " + Math.floor(fValue/1000000) + "M";
            }
            return "0";
        }
```

```
    });
});
```

The `this` context of a formatter function is generally set to the control (or managed object) that owns the binding. However, in XML views, the reference to the formatter is done in the view controller by putting a dot (`.`) in front of the name of the formatter function (`{ formatter: '.myformatter' }`). In this case, the formatter's `this` context is bound to the controller.

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input
        value="{ path:'/company/revenue',
        formatter: '.roundToMillion'}"
    />
</mvc:View>
```

If you use **JavaScript**, you can pass the formatter function as a third parameter to the `bindProperty` method, or you can add the binding info with the `formatter` key. The `formatter` has a single parameter `value`, which is the value that is to be formatted, and is executed as a member of the control, meaning it can access additional control properties or model data.

```
//"Input" required from module sap/m/Input
oTextField.bindProperty("value", "/company/title", function(sValue) {
    return sValue && sValue.toUpperCase();
});
oControl = new Input({
    value: {
        path:"/company/revenue",
        formatter: function(fValue) {
            if (fValue) {
                return "> " + Math.floor(fValue/1000000) + "M";
            }
            return "0";
        }
    }
})
```

Because it can contain any JavaScript, the formatter function can be used for formatting a value and also for performing type conversions or calculating results, for example, to show a special traffic light image depending on a Boolean value:

```
oImage.bindProperty("src", "/company/trusted", function(bValue) {
    return bValue ? "green.png" : "red.png";
});
```

⚠ Caution

The framework only updates a binding when one of the properties included in the binding changes. If the formatter uses another property value that is not part of the binding definition, the framework won't know that the result depends on that additional property and could miss necessary updates. Therefore, make sure that you declare a composite binding referencing all necessary properties (maybe even from different models).

Using Data Types

The data type system enables you to format and parse data, as well as to validate whether the entered data lies within any defined constraints. SAPUI5 comes with several predefined and ready-to-use types, referred to as simple types. For more information, see Formatting, Parsing, and Validating Data [page 854].

Here's how you can use these types in an XML view:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input
        value="{ path:'/company/revenue',
        type: 'sap.ui.model.type.Integer'}"/>
</mvc:View>
```

You can also provide parameter values for some of the simple types in your XML view. These are declared as `formatOptions`, as you can see in the Float type sample below. Permitted `formatOptions` are properties of the corresponding data type. For more information, see the *API Reference* in the Demo Kit.

```
<mvc:View
   controllerName="sap.ui.sample.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">
   <Input
      value="{ path:'/company/revenue',
           type: 'sap.ui.model.type.Float',
           formatOptions: {
                   minFractionDigits: 2,
                   maxFractionDigits: 2
           }
      }"/>
</mvc:View>
```

Using JavaScript, you can define a type to be used for a property binding by passing it as a third parameter in `bindProperty` or by adding it to the binding information by using the key `type`, as shown here:

```
// "TypeString" required from module "sap/ui/model/type/String"
// "Input" required from module "sap/m/Input"
// "TypeFloat" required from module "sap/ui/model/type/Float"
oTextField.bindProperty("value", "/company/name", new
sap.ui.model.type.String());
oControl = new sap.m.Input({
    value: {
        path:"/company/revenue",
        type: new TypeFloat({
            minFractionDigits: 2,
            maxFractionDigits: 2
        })
    }
})
```

Predefined data types also offer visual feedback for erroneous user input. To turn this feature on, add the following line to your controller's `init` function:

```
sap.ui.getCore().getMessageManager().registerObject(this.getView(), true);
```

You can define **custom types** by inheriting from `sap.ui.model.SimpleType` and implementing the three methods `formatValue`, `parseValue`, and `validateValue`. `formatValue` is called whenever the value in the model is changed to convert it to the type of the control property it is bound to, and may throw a

`FormatException`. `parseValue` is called whenever the user has modified a value in the UI and the change is transported back into the model. It may throw a `ParseException` if the value cannot be converted. If parsing is successful, `validateValue` is called to check additional constraints, such as minimum or maximum value, and throws a `ValidateException` if any constraints are violated.

```
// "SimpleType" required from module "sap/ui/model/SimpleType"
// "ValidateException" required from module "sap/ui/model/ValidateException"
var Zipcode = SimpleType.extend("sap.ui.sample.Zipcode", {
    formatValue: function(oValue) {
        return oValue;
    },
    parseValue: function(oValue) {
        return oValue;
    },
    validateValue: function(oValue) {
       if (!/^(\d{5})?$/.test(oValue)) {
            throw new ValidateException("Zip code must have 5 digits!");
       }
    }
});
```

You can use your custom types in XML views or JavaScript in the same way as you would apply predefined types:

```
<mvc:View
   controllerName="sap.ui.sample.App"
   xmlns="sap.m"
   xmlns:mvc="sap.ui.core.mvc">

   <Input
      value="{ path:'/company/zip',
      type: 'sap.ui.sample.Zipcode'
     }"/>
</mvc:View>
```

Changing the Binding Mode

By default, all bindings of a model instance have the default binding mode of the model, but you can change this behavior if needed. When creating a `PropertyBinding`, you can specify a different binding mode, which is then used exclusively for this specific binding. Of course, a binding can only have a binding mode that is supported by the model in question.

```
// "JSONModel" required from module "sap/ui/model/json/JSONModel"
// "Input" required from module "sap/m/Input"
// "BindingMode" required from module "sap/ui/model/BindingMode"
    var oModel = new JSONModel();
    // default binding mode is two way
    oModel.setData(myData);
    sap.ui.getCore().setModel(oModel);
    var oInputFirstName = new Input ();

    // bind value property one way only
    // propertyname, formatter function, binding mode
    oInputFirstName.bindValue("/firstName", null, BindingMode.OneWay);
    oInputFirstName.placeAt("target1");
    oInputLastName = new Input();
    // bind value property two way (default)
    oInputLastName.bindValue("/lastName");
```

```
    oInputLastName.placeAt("target2");
```

In the example above, two `Input` fields are created and their `value` property is bound to the same property in the model. The first `Input` binding has a one-way binding mode, whereas the second `Input` has the default binding mode of the model instance, which is two-way. For this reason, when text is entered in the first `Input`, the value will **not** be changed in the model. This only happens if text is entered in the second `Input`. Then, of course, the value of the first `Input` will be updated as it has a one-way binding, that is, from model to view.

Related Information

Data Binding Tutorial Step 3: Create Property Binding [page 225]
API Reference: `sap.ui.base.ManagedObject.bindProperty`
Binding Syntax [page 840]
Formatting, Parsing, and Validating Data [page 854]

Context Binding (Element Binding)

Context binding (or element binding) allows you to bind elements to a specific object in the model data, which will create a binding context and allow relative binding within the control and all of its children. This is especially helpful in master-detail scenarios.

Let's assume we have the following JSON data:

```
{
    "company" : {
        "name"  : "Acme Inc."
        "street": "23 Franklin St."
        "city"  : "Claremont"
        "state" : "New Hampshire"
        "zip"    : "03301"
        "revenue": "1833990"
    }
}
```

Here's how you would use element binding in an XML view:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input id="companyInput"
        binding="{/company}"
        value="{name}"
        tooltip="The name of the company is '{name}'"/>
</mvc:View>
```

By setting `binding="{/company}"`, we can refer to `company` children without having to qualify the full binding path, when binding `Input` control's properties such as the `value`. Using plain property binding, our XML view would look like this:

```
<mvc:View
```

```
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input id="companyInput"
        value="{/company/name}"
        tooltip="The name of the company is '{/company/name}'}"/>
</mvc:View>
```

To define an element binding in JavaScript, for example in a controller, use the `bindElement` method on a control:

```
var oInput = this.byId("companyInput")
oInput.bindElement("/company");
oInput.bindProperty("value", "name");
```

Element binding is especially interesting for containers or layouts containing many controls that are all visualizing properties of the same model object. Here's an XML view with a `VerticalLayout` using element binding:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <l:VerticalLayout id="vLayout"
        binding="{/company}"
        width="100%">
            <Text text="{name}" />
            <Text text="{city}" />
            <Text text="{county}" />
    </l:VerticalLayout>
</mvc:View>
```

To realize this in JavaScript, proceed as follows in your controller:

```
var oVerticalLayout = this.getView().byId('vLayout');
oVerticalLayout.bindElement("/company");
oVerticalLayout.addContent(new Text({text: "{name}"}));
oVerticalLayout.addContent(new Text({text: "{city}"}));
oVerticalLayout.addContent(new Text({text: "{county}"})););
```

Given your XML view contains a `VerticalLayout`, it will look like this:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <l:VerticalLayout id="vLayout"
        width="100%"/>
</mvc:View>
```

Setting a New Context for the Binding (Master-Detail)

You create a new binding context for an element that is used to resolve bound properties or aggregations relative to the given path. You can use this method if the existing binding path changes or has not been provided before, for example in master-detail scenarios, as outlined below.

Let's look at the following JSON model featuring a company list:

```
{
    companies : [
        {
            name : "Acme Inc.",
            city: "Belmont",
            state: "NH",
            county: "Belknap",
            revenue : 123214125.34
        },{
            name : "Beam Hdg.",
            city: "Hancock",
            state: "NH",
            county: "Belknap"
            revenue : 3235235235.23
        },{
            name : "Carot Ltd.",
            city: "Cheshire",
            state: "NH",
            county: "Sullivan",
            revenue : "Not Disclosed"
        }]
}
```

Let's take this simple view, containing a single input control:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Input id="companyInput"
         value="{name}"/>
</mvc:View>
```

In your controller, you can now bind the input control as follows:

```
var oInput = this.byId("companyInput");
oInput.bindElement("/companies/0");
```

The XML view has bound the `value` of the input to the `name` property in the model. As the path to this property in the model is not set, this will not resolve. To resolve the binding, you use the `bindElement` method which creates a new context from the specified relative path.

To remove the current binding context, call the `unbindElement` method on the input control. By doing this, all bindings now resolve relative to the parent context again.

You can also use the `bindElement` method in conjunction with list binding. Let's consider the following extension of our JSON data:

```
{
    regions: [
        {
            name: "Americas",
            companies : [
            {
                name : "Acme Inc.",
                zip : "03301",
                city: "Belmont",
                county: "Belknap",
                state: "NH",
                revenue : 123214125.34,
                publ: true
```

```
            },
            {
                name : "Beam Hdg.",
                zip : "03451",
                city: "Hancock",
                county: "Sullivan",
                state: "NH",
                revenue : 3235235235.23,
                publ: true
            },
            {
                name : "Carot Ltd.",
                zip : "03251",
                city: "Cheshire",
                county: "Sullivan",
                state: "NH",
                revenue : "Not Disclosed",
                publ: false
            }]
        },{
            name: "DACH",
            companies : [
            {
                name : "Taubtrueb",
                zip : "89234",
                city: "Ginst",
                county: "Musenhain",
                state: "NRW",
                revenue : 2525,
                publ: true
            },
            {
                name : "Krawehl",
                zip : "45362",
                city: "Schlonz",
                county: "Humpf",
                state: "BW",
                revenue : 2342525,
                publ: true
            }]
        }
    ]
}
```

Say we want to display companies in a `sap.m.List` control. Here's what the XML view will look like:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
      <List id=”companyList” items="{companies}">
        <items>
            <StandardListItem
         title="{name}"
         description="{city}"
            />
        </items>
      </List>
</mvc:View>
```

Please note that `items="{companies}"` cannot be resolved initially, since it is a relative path. In your controller, you can now provide an element binding for the list control:

```
var oList = this.byId("companyList");
oList.bindElement("/regions/0");
```

This will display the companies for region **Americas**, while the code below displays all companies in the **DACH** region (Germany, Austria, Switzerland):

```
var oList = this.byId("companyList");
oList.bindElement("/regions/1");
```

API Reference

For more information, see the API Reference for the following methods:

- API Reference: `sap.ui.base.ManagedObject.bindObject`.
- API Reference: `sap.ui.base.ManagedObject.getObjectBinding`.
- API Reference: `sap.ui.base.ManagedObject.unbindObject`.
- API Reference: `sap.ui.core.Element.bindElement`.
- API Reference: `sap.ui.core.Element.getElementBinding`.
- API Reference: `sap.ui.core.Element.unbindObject`.

Related Information

Tutorial Step 13: Element Binding [page 250]
Binding Syntax [page 840]
Formatting, Parsing, and Validating Data [page 854]

List Binding (Aggregation Binding)

List binding (or aggregation binding) is used to automatically create child controls according to model data.

Let's say we would like to display the following JSON model data in a `sap.m.List`:

```
{
    companies : [
        {
            name : "Acme Inc.",
            city: "Belmont",
            state: "NH",
            county: "Belknap",
            revenue : "123214125.34"
        },{
            name : "Beam Hdg.",
            city: "Hancock",
            state: "NH",
            county: "Belknap"
            revenue : "3235235235.23"
        },{
            name : "Carot Ltd.",
            city: "Cheshire",
            state: "NH",
            county: "Sullivan",
```

```
            revenue : "Not Disclosed"
        }]
}
```

Declarative List Binding in XML Views

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <List id=”companyList” items="{path: '/companies', templateShareable:false}">
        <items>
            <StandardListItem
                title="{name}"
                description="{city}"
            />
        </items>
    </List>
</mvc:View>
```

The `List` element has both an `items` attribute and a nested `items` element:

- The attribute `items="{path: '/companies', templateShareable:false}"` binds the children of our json model's `companies` array to the list. This by itself is not enough to display the companies, instead it sets the parent path for the binding of all contained list items and their descendants. In addition you need to declare a nested element.
- The nested `items` element in our case contains a `StandardListItem`. This serves as a template for creating the individual list rows.

> **i Note**
>
> The binding paths of `StandardListItem` for properties `title` and `description` are relative to `companies`. This means that instead of having to write the whole binding path `title={/companies/name}`, you can simply write `title={name}`. By omitting the slash ‘/’ at the beginning, `{name}` is marked as a relative binding path.

Instead of using a `StandardListItem` as a list row template, you can also use any other `sap.m.` list item, such as:

- ActionListItem
- DisplayListItem
- CustomListItem
- ObjectListItem

For more examples and details on when to use which list item control, see the various list items in the Samples in the Demo Kit.

> **i Note**
>
> The model has a default size limit to avoid too much data being rendered on the UI. This size limit determines the number of entries used for the list bindings. The default size limit is 100 entries.
>
> This means that controls that don't support paging or don't request data in chunks (e.g. `sap.m.ComboBox`) only show 100 entries even though the model contains more items.

To change this behavior, you can set a size limit in the model by using `oModel.setSizeLimit`.

List Binding in the JavaScript Code

You can define list binding directly in JavaScript either in the `settings` object in the constructor or by calling the `bindAggregation` method. List binding requires the definition of a template, which is cloned for each bound entry of the list. For each clone that is created, the binding context is set to the respective list entry, so that all bindings of the template are resolved relative to the entry. The aggregated elements are destroyed and recreated whenever the bound list in the data model is changed.

To bind a list, you create a template or provide a factory function, which is then passed when defining the list binding itself. In the `settings` object, this looks as follows:

```
var oItemTemplate = new sap.ui.core.ListItem({text:"{name}"});
oComboBox = new sap.m.ComboBox({
    items: {
        path: "/companies",      //no curly brackets here!
        template: oItemTemplate
        templateShareable: false
    }
});
```

A template is not necessarily a single control as shown in the example above, but can also be a tree of controls. For each list entry, a deep clone of the template is created and added to the bound list.

You can also define the list binding by using the `bindAggregation` method of a control:

```
var oItemTemplate = new sap.ui.core.ListItem({text:"{name}"});
oComboBox.bindAggregation("items", {
path: "/companies",
template: oItemTemplate,
templateShareable: false
});
```

In addition, some controls have a typed binding method for lists that are likely to be bound by the application:

```
var oComboBox.bindItems("/companies", oItemTemplate);
```

To remove a list binding, you can use the `unbindAggregation` method:

```
oComboBox.unbindAggregation("items");
```

Controls with typed binding methods also provide a typed unbind:

```
oComboBox.unbindItems();
```

When a list is unbound, its aggregated controls are removed and destroyed by default. If you would like to keep the items in your `ComboBox`, for example, you can do so by using:

```
oComboBox.unbindAggregation("items", true);
```

Related Information

Tutorial Step 12: Aggregation Binding Using Templates [page 244]
Binding Syntax [page 840]
Formatting, Parsing, and Validating Data [page 854]

Using Factory Functions

The factory function is a more powerful approach for creating controls from model data. The factory function is called for each entry of a control's aggregation, and the developer can decide whether each entry shall be represented by the same control with different properties or even by a completely different control for each entry.

The factory function comes with the parameters `sId`, which should be used as an ID for the new control, and `oContext`, which is for accessing the model data of the entry. The returned object must be of type `sap.ui.core.Element`. Here's how this scenario can be realized in an XML view and a controller using our JSON model data:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    xmlns:mvc="sap.ui.core.mvc">
    <l:VerticalLayout
        content="{ path: '/companies', factory: '.createContent'}"
        class="sapUiContentPadding"
        width="100%"/>
</mvc:View>
```

Please note the `'.'` in `factory: '.createContent'`. The class `App.controller.js` contains the implementation of our factory method:

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/ui/model/type/String",
    "sap/ui/model/type/Float",
    "sap/m/Input",
    "sap/m/Text",
    "sap/m/CheckBox"
], function (Controller, JSONModel, StringType, Float, Input, Text, CheckBox ) {
    "use strict";
    return Controller.extend("sap.ui.sample.App", {
        onInit : function () {
        …
        },
        createContent: function (sId, oContext) {
        var oRevenue = oContext.getProperty("revenue");
            switch(typeof oRevenue) {
                case "string":
                    return new Text(sId, {
                        text: {
                            path: "revenue",
                            type: new StringType()
                        }
                    });
```

```
                case "number":
                    return new Input(sId, {
                        value: {
                            path: "revenue",
                            type: new Float()
                        }
                    });

                case "boolean":
                    return new CheckBox(sId, {
                        checked: {
                            path: "revenue"
                        }
                    });
            }
        },
    });
});
```

If you would like to avoid using the XML view, you would proceed as follows:

```
oVerticalLayout.bindAggregation("content", "/companies", function (sId,
oContext) {
    var oRevenue = oContext.getProperty("revenue");
    switch(typeof oRevenue) {
            case "string":
                return new sap.m.Text(sId, {
                    text: {
                        path: "revenue",
                        type: new sap.ui.model.type.String()
                    }
                });

            case "number":
                return new sap.m.Input(sId, {
                    value: {
                        path: "revenue",
                        type: new sap.ui.model.type.Float()
                    }
                });

            case "boolean":
                return new sap.m.CheckBox(sId, {
                    checked: {
                        path: "revenue"
                    }
                });
            }
        }
});
```

Related Information

Tutorial Step 15: Aggregation Binding Using a Factory Function [page 257]

Sorting, Grouping, and Filtering for List Binding

Initial Sorting, Grouping and Filtering for List Binding

To provide initial **sorting and grouping** in an XML view, proceed as follows:

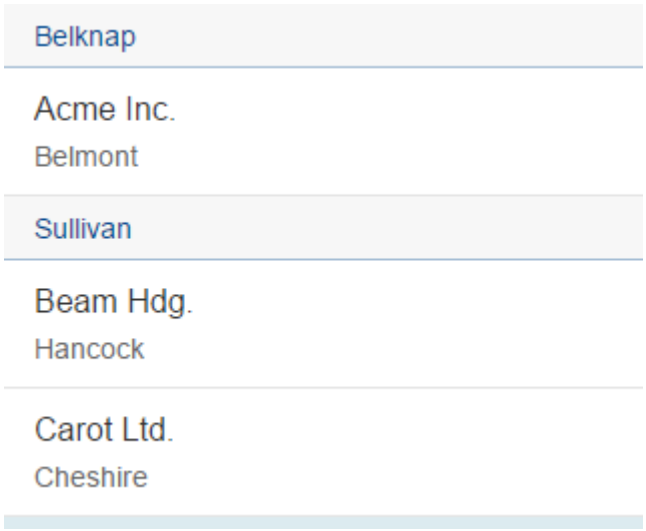
```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    xmlns:mvc="sap.ui.core.mvc">
    <List items="{ path: '/companies',
        sorter: { path: 'county', descending: false, group: '.getCounty'},
        groupHeaderFactory: '.getGroupHeader'}">
        <items>
            <StandardListItem
                title="{name}"
                description="{city}"
            />
        </items>
    </List>

</mvc:View>
```

The `this` context of a group header factory function is generally set to the control (or managed object) that owns the binding. However, in XML views, the reference to the group header factory is done in the view controller by putting a dot (`.`) in front of the name of the group header factory function (`{ groupHeaderFactory:'.myGroupHeader' }`). In this case, the group header factory's this context is bound to the controller.

The list uses a sorter which sorts the list of companies in ascending order by the `county` column. It also groups its rows using the App.controller's `getCounty` method to provide the captions and the `getGroupHeader` function to provide non-standard group header controls, as shown here:

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "sap/m/GroupHeaderListItem "
], function (Controller, JSONModel, GroupHeaderListItem) {
    "use strict";
    return Controller.extend("sap.ui.sample.App", {
        onInit : function () {
            …
        },
        getCounty: function(oContext) {
            return oContext.getProperty('county');
        },

        getGroupHeader: function(oGroup) {
            return new GroupHeaderListItem({
                title : oGroup.key
            }
        );
    },
});
```

As you can see, `getCounty` generates the group caption, which in this case is the county of the current companies. `getGroupHeader` serves as a group header factory function. After sorting and grouping, the company list looks like this:



The following XML snippet provides initial filtering:

```
<mvc:View
    controllerName="sap.ui.sample.App"
    xmlns="sap.m"
    xmlns:l="sap.ui.layout"
    xmlns:mvc="sap.ui.core.mvc">
    <List items="{ path: '/companies',
        filters: [{path: 'city', operator: 'StartsWith', value1: 'B'},
            {path: 'revenue', operator: 'LT', value1: 150000000}]}">
        <items>
            <StandardListItem
                title="{name}"
                description="{city}"
            />
        </items>
    </List>
</mvc:View>
```

The example shown here will only display companies whose city name begins with a ‘b’ and whose revenue is less than 150 million. As you can see, you can provide more than one filter, each of which may refer to different columns using different filter operators. For a complete list of permitted filter operators, see sap.ui.model.FilterOperator in the *API Reference* part of the Demo Kit.

As shown below, initial sorting, grouping and filtering can of course also be provided using JavaScript.

You can define a sorter and/or filters:

```
sap.ui.define([
    "sap/ui/model/Sorter",
    "sap/ui/model/Filter"
], function(Sorter, Filter) {
//returns group header captions
var fnGetCounty = function(oContext) {
    return oContext.getProperty('county');
}
var oSorter = new Sorter({
    path: 'county',
    descending: false,
```

```
    group: fnGetCounty});
var oFilterCity = new Filter("city",
    sap.ui.model.FilterOperator.StartsWith, "B"),
    oFilterRevenue = new sap.ui.model.Filter("revenue",
        sap.ui.model.FilterOperator.LT, 150000000);
);
});
```

You can pass sorters and filters to the list binding:

```
var oList = new sap.m.List({
    items: {path: "/companies", template: oItemTemplate,
        sorter: oSorter, filters:[oFilterCity, oFilterRevenue]
    }
});
```

You can also use the other list binding possibilities (for example `bindAggregation` or `bindItems`) and provide the sorter and filters as parameters.

Manual Sorting and Filtering for List Binding

You can sort or filter data manually after the list binding is complete by getting the corresponding binding and calling the sort/filter function:

```
// manual sorting
oList.getBinding("items").sort(oSorter);
// manual filtering
oList.getBinding("items").filter([oFilterCity, oFilterRevenue]);
```

> **i Note**
>
> `getBinding` requires the name of the bound list. In this example, we are looking at the `items` of the `sap.m.List` control.

For more information about the various sorting and filter methods and operators, see the documentation for Filter, Sorter, and Filter operations under sap.ui.model in the *API Reference* part of the Demo Kit.

Using Complex Syntax to Add Filters and Sorters

Complex syntax can be used to add filters and sorters for list binding. One or multiple objects can be defined.

```
<table:Table rows="{
    path: '/table',
    filters: [{
        path: 'field3',
        operator: 'EQ',
        value1: 'test'
    }],
    sorter: [{
        path: 'field1',
        descending: false
    }, {
        path: 'field2',
```

```
        descending: true
    }]
 }">
...
</table:Table>
```

Lifecycle of Binding Templates

The lifecycle of the binding templates differs from the lifecycle of controls that are contained in an aggregation. Whenever a control object is destroyed, any aggregating object is destroyed as well. For list binding templates, you specify the behavior by using the additional property `templateShareable` in the parameter of the `bindAggregation` method of class `sap.ui.base.ManagedObject`.

In **XML views**, you can also use the `templateShareable` property by adding it to the binding info as follows:

```
<Table id="EmployeeEquipments" headerText="Employee Equipments" items="{
    path: 'EMPLOYEE_2_EQUIPMENTS',
    templateShareable: false
}">
    <columns>
        <!-- ... -->
    </columns>
    <items>
        <ColumnListItem>
            <cells>
                <Text text="{ID}"/>
            </cells>
            <cells>
                <Text text="{EQUIPMENT_2_PRODUCT/Name}"/>
            </cells>
            <cells>
                <Text text="{Category}"/>
            </cells>
            <cells>
                <!-- Name="PRODUCT_2_CATEGORY" Type="Collection(...)" -->
                <List items="{
                    path: 'EQUIPMENT_2_PRODUCT/PRODUCT_2_CATEGORY',
                    templateShareable: true
                }">
                    <StandardListItem title="{CategoryName}"/>
                </List>
            </cells>
            <cells>
                <Text text="{EQUIPMENT_2_PRODUCT/PRODUCT_2_SUPPLIER/
Supplier_Name}"/>
            </cells>
        </ColumnListItem>
    </items>
</Table>
```

- `templateShareable = "false"` (preferred setting)
  If you set the parameter to `false` the lifecycle is controlled by the framework. It will destroy the template when the binding is removed (`unbindAggregation`, `unbindItems`)
- `templateShareable = "true"`
  If you set the parameter to `true` the template is **not** destroyed when (the binding of) the aggregated object is destroyed. Use this option in the following cases only:
  - The template is reused in your app to define an additional list binding.

Since the template is not destroyed, this could also affect some other aggregation that uses the same template at a later point in time.

- The parent control that contains the list binding with the template is cloned. The binding info is used in the clone as well.
  This means, when `templateShareable` is set to `true`, the template will not be cloned, when it is set to `false` it will be cloned when the parent is cloned.

In these cases, the app has to make sure that the templates are properly cleaned up at some point in time - at the latest when the corresponding controller or component is destroyed.

- If the parameter is undefined, (neither `true` nor `false`), the framework checks at several points in time whether all list bindings are removed. If there are no bindings, the templates is marked as `candidate for destroy()`, but it is not immediately destroyed. The candidate is destroyed in the following cases:
  - A **new object with the same ID** is created.
  - The component that owns the objects is destroyed.

  If the framework determines that a "candidate for destroy" is still in use in another binding or in a clone operation, the framework makes sure that the candidate is not destroyed by implicitly setting `templateShareable` to `true` (as this best reflects how the app deals with the template). But now the template is not destroyed at all (an error message is issued), and the app implementation needs to make sure that the binding template is destroyed as soon as it is no longer needed.

> **i Note**
>
> The error messages are:
> - *A binding template that is marked as 'candidate for destroy' is reused in a binding.*
> - *During a clone operation, a template was found that neither was marked with 'templateShareable:true' nor 'templateShareable:false'.*

> **⚠ Caution**
>
> To leave the parameter undefined is **very error-prone**, therefore we don't recommend this! Always set the parameter explicitly to `true` or `false`.

Extended Change Detection

Extended change detection offers fine-grained information on the actual data changes. This can be used, for example, to only update the DOM when really necessary and avoid complete rerendering of a huge list whenever data is changed.

The binding base class already offers a `Change` event, which is fired whenever the bound data has been changed. This is sufficient for bindings like property and context binding. Since lists can contain a huge amount of data, you need more detailed information on the changes to avoid a complete rerendering of the whole list each time data has been changed on the UI.

Calculation of Differences

When extended change detection is enabled, an algorithm is executed to compare the last returned context array with the current context array and the differences is attached to the array of contexts as an additional property named `diff` whenever the `getContexts` method is called. The following results are possible:

- There is no `diff` property on the context array
  The data was completely changed or a difference could not be calculated. In this case there is no possibility for fine-grained update, a complete recreation or rerendering is necessary.
- The `diff` property returns an empty array
  The algorithm has been executed, but could not find any differences between the initial and the current state. This may occur if data within the list has been changed, but detection of updates have not been enabled for the extended change detection.
- The `diff` property returns an array of different entries
  The difference has been calculated and can be used by the control or application to update dependent structures in a fine grained manner

The difference between the state when the list was initially loaded and the current state is provided to the control as an array that contains `insert` and `remove` entries that contain the actual changes.

Example

Old State	New State
`["one", "two", "three", "four", "five"]`	`["one", "three", "four", "five", "six"]`

Difference
`[{index: 1, type: "delete"}, {index: 4, type: "insert}]`

The algorithm is implemented in the utility method `jQuery.sap.arraySymbolDiff`, which tries to calculate the smallest possible difference for the transition from old to the new state. The indexes are calculated in a way that they are valid after all previous steps have been applied, so it can be used in a loop to update an existing array, without any additional index shift needed.

Caution

- Extended change detection calculates the difference between the context arrays returned by calling `getContexts`. This means, it is completely independent from the `startIndex` and `length` parameters. Any additional call to `getContexts`, either by the app or the control itself, may trigger a difference calculation and may cause update problems. If you want to access the current context of a list binding, you should use `getCurrentContexts` in your app instead.
- When a `ListBinding` is firing a `Refresh` event, the call to `getContexts` caused by this event is used to inform the `ListBinding` on the `startIndex` and `length` of entries requested by the control. No difference calculation is done on this specific call, as controls do not use the result of this call but instead wait for the data returned by the server.

Using Extended Change Detection in App Development

If a control you want to use in your app to visualize list entries supports extended change detection, you should make sure that each entity of your model has a unique key to improve performance.

- For OData models, the unique keys are automatically provided.
- For all other models (like a JSON model), you have to define the keys either by using a key property or by using a function that calculates the key in the binding info of their list binding as in the following example:

`key` **property**

```
oControl.bindItems({
  path: "/listData",
  key: "id"
});
```

`key` **function**

```
oControl.bindItems({
  path: "/listData",
  key: function(oContext) {
    return
oContext.getProperty("user") +
oContext.getProperty("timestamp");
  }
});
```

Using Extended Change Detection in Control Development

Extended change detection is disabled by default. If your control is meant to have only a few children like a toolbar with buttons, you should not activate extended change detection because a copy of the previous state would then always be kept unnecessarily in the binding.

With extended change detection the control uses specific `insert` and `remove` calls only for elements that need to be added or removed instead of recreating all elements of an aggregation or setting new binding contexts on all aggregated elements.

You activate extended change detection for your control by setting the `bUseExtendedChangeDetection` property either on the control prototype or a specific control instance. The `ManagedObject` class takes care of reading and applying the information about the differences to aggregations with the `enableExtendedChangeDetection` method. The method has the following parameters:

- `bDetectUpdates`
  Defines whether data changes within the same entity should also be contained in the `diff`. This is especially relevant when a factory function is used to create child controls, so depending on the data a different control may be created by the same entity.
- `vKey`
  Defines how a unique symbol for each row is calculated, which is then used for the calculation of differences. This can either be a property name (in case the data already has something like a `key` property) or a function that is able to calculate such a unique key from the entity data.

You also have to implement the specific aggregation modifier methods to avoid the rerendering of the complete UI and only to a fine-grained DOM update.

> **i Note**
>
> If your control has a custom `updateAggregation` method (that means control takes care for updating the aggregation) you have to make sure in your implementation that the difference information is interpreted and applied.

For more information, see the API Reference: `sap.ui.base.ManagedObject.bindAggregation`.

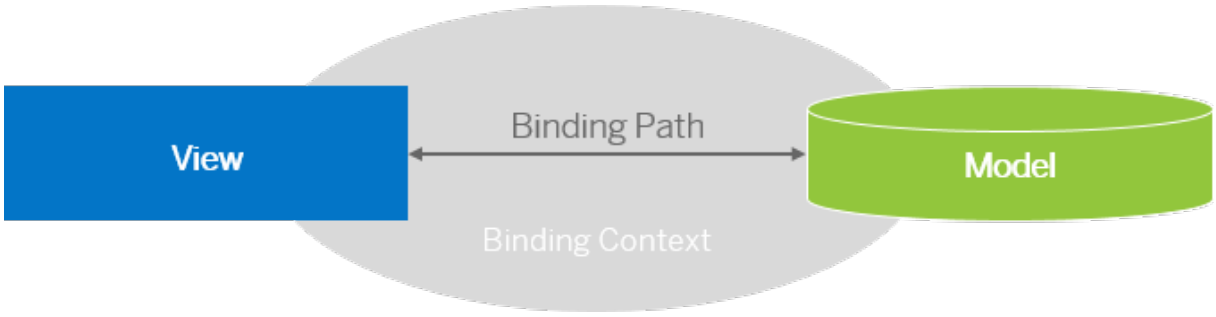
Related Information

API Reference: `sap.ui.base.ManagedObject.bindAggregation`
API Reference: `sap.ui.Model.ListBinding.getContexts`

Binding Syntax

You bind UI elements to data of a data source by defining a binding path to the model that represents the data source in the app.

When defining a binding path for a control, a binding context is created which connects this control to a data model. The UI control then gets the data through that context and displays it on the screen.



- Views [page 787]
- Binding Path [page 842]
- Models [page 882]

Simple Binding

To reference model data in a view , you can use the simple binding syntax "`{`*/path/to/data*`}`":

```
<Input value="{/firstName}"/>
```

You can add other properties like formatters or data types:

- Data type:

```
<Input value="{path: '/firstName', type: 'sap.ui.model.type.String'}"/>
```

- Formatter:

```
<Input value="{path: '/firstName', formatter:'my.globalFormatter'}"/>
```

For more information, see Binding Path [page 842].

For more information about data types and formatters, see Formatting, Parsing, and Validating Data [page 854].

Composite Binding

If a control requires data from multiple different model properties, you use a `parts` array of `path`s to define composite binding paths:

```
<TextField value="{
    parts: [
        {path:'birthday/day'},
        {path:'birthday/month'},
        {path:'birthday/year'}
    ],
    formatter:'my.globalFormatter'
}"/>
```

For more information, see Composite Binding [page 843] and Examples for Data Binding in Different View Types [page 851].

Expression Binding in XML Views

Expression binding is a simple way to calculate values directly in the view. For example, if you want to change the color of the price depending on whether it is above or below some threshold. With expression binding you don't have to declare a separate formatter:

```
<ObjectStatus state=="{= ${products>UnitPrice}  > ${/priceThreshold} ? 'Error' :
'Success' }"/>
```

For more information, see Expression Binding [page 845].

Property Metadata Binding for OData Services

With metadata binding, you can bind properties of a control to the corresponding property that is defined in the metadata of an OData service:

```
<Input maxLength="{/#Company/ZipCode/@maxLength}"/>
```

For more information, see Property Metadata Binding [page 851].

Related Information

API Reference: `sap.ui.base.ManagedObject.bindProperty`
API Reference: `sap.ui.base.ManagedObject.bindAggregation`
API Reference: `sap.ui.base.ManagedObject.bindObject`

Binding Path

Binding paths address the different properties and lists in a model and define how a node in the hierarchical data tree can be found.

A binding path consists of a number of name tokens, which are separated by a separator char. In all models provided by the framework, the separator char is the slash "/".

A binding path can either be absolute or relative: Absolute binding paths start with a slash, relative binding paths start with a name token and are resolved relative to the context of the control that is bound. A context exists either for each entry of the aggregation in case of aggregation binding or can be set explicitly for a control by using the `setBindingContext` method.

When you use multiple models, specify the model name within the binding path to address the correct model. The same applies for setting a binding context for such a model. The binding path must start with the model name followed by a '>' as shown in the following example for setting a binding context.

```
oControl.setBindingContext(oContext );
oControl.setBindingContext(oContext,"myModelName");
```

Binding path examples:

```
'/Products/0/ProductName'
'/Products(0)/ProductName'
'ProductName'
//with model name
'myModelName>/Products/0/ProductName'
'myModelName>/Products(0)/ProductName'
'myModelName>ProductName'
```

Related Information

OData V2 Model [page 883]
OData V4 Model: Bindings [page 922]
Binding Path Syntax for JSON Models [page 992]
Binding Path Syntax for XML Models [page 995]
Binding Path Syntax for Resource Models [page 996]

Composite Binding

Calculated fields enable the binding of multiple properties in different models to a single property of a control.

The value property of a text field, for example, may be bound to a property `firstName` and a property `lastName` in a model. The application can access these values in a formatter function and can decide how they should be processed or combined together. If no formatter function is specified, the values are joined together by default. You can use the `useRawValues` property to specify if the parameter values in the formatter function are formatted according to the type of the property or not.

The multiple property bindings are stored in a `CompositeBinding` and can be accessed by calling the `getBindings` function. You can access the composite binding, for example, by using the `getBinding('value')` function of the control. The composite binding has no path, model, context, and type because it contains multiple property bindings containing the necessary information. A composite binding may, for example, store two property bindings which belong to different models and have different types.

If you have specified a formatter function, it is also available in the composite binding.

There are several options to create multiple bindings for a control. The syntax is very similar to the normal single binding declaration.

Each binding is created by the specified parts and assigned information. A part must contain the path to the property in the model and may contain additional information for the binding, for example a type.

Constructor Declaration

1. Use binding objects to add additional parameters, for example the type:

```
oTxt = new sap.m.Input({
    value: {
        parts: [
                {path: "/firstName", type: new sap.ui.model.type.String()},
                {path: "/lastName"},
                {path: "myModel2>/amount", type: new
sap.ui.model.type.Float()} // path to property in another model
               ]
         }
});
```

2. Use strings which only take the path:

```
oTxt = new sap.m.Input({
    value: {
        parts: [
                "/firstName",
                "/lastName",
                "myModel2>/fraud" // path to property in another model
               ]
         }
});
```

Bind Property Declaration

1. Use binding objects to add additional parameters, for example the type:

```
oTxt.bindValue({
    parts: [
             {path: "/firstName", type: new sap.ui.model.type.String()},
             {path: "/lastName"}
            ]
});
```

2. Use strings which only take the path:

```
oTxt.bindValue({
    parts: [
             "/firstName",
             "/lastName"
            ]
});
```

These samples also work with a relative binding path, when you use them as a template in a list binding.

Complex Syntax for Calculated Fields

Complex (or "extended") syntax can be used for calculated fields in declarative views, such as HTML and XML views.

To use the feature in your SAPUI5 application, set the configuration flag `bindingSyntax` in the bootstrap as follows:

```
<script id="sap-ui-bootstrap"
      ...
        data-sap-ui-bindingSyntax="complex">
    </script>
```

The following examples show how to use the feature:

- You can mix text with calculated fields as follows:

```
<Label text="Hello Mr. {
                         path:'/singleEntry/firstName',
                         formatter: '.myFormatter'
                        },
                        {
                         /singleEntry/lastName
                        }
 " />
```

> **i Note**
>
> Use translatable text in your application.

- Use a syntax with leading quotation marks ("...") if you use MVC and your formatter or type is located in the controller. In the following example, the existing type or formatter function in the controller is used:

```
<TextField value="{
```

```
                    path:'gender',
                    formatter:'.myGenderFormatter'
                   }
                   {firstName},
                   {lastName}
"/>
```

- If you have a global formatter function, use the following syntax:

```
<TextField value="{
              parts: [
                      {path:'birthday/day'},
                      {path:'birthday/month'},
                      {path:'birthday/year'}
                     ],
              formatter:'my.globalFormatter'
}"/>
```

- For a global type that is created with the specified format options, see the following example:

```
<Label text="A type test: {
                           path:'/singleEntry/amount',
                           type:'sap.ui.model.type.Float',
                           formatOptions: { minFractionDigits: 1}
                          } EUR
"/>
```

Expression Binding

Expression binding is an enhancement of the SAPUI5 binding syntax, which allows for providing expressions instead of custom formatter functions.

Using expression binding saves the overhead of defining a function and is recommended if the formatter function has a trivial implementation like a comparison of values. Expression binding is especially useful in the context of SAPUI5 XML templating where XML views with templating are preprocessed and the SAPUI5 controller as the natural place to put custom formatter functions is not available.

To use expression binding, you need to enable complex binding syntax by using configuration setting `bindingSyntax` to `complex`.

> **i Note**
>
> Complex syntax is automatically activated when the `compatVersion` is set to `edge` or to version `1.28` or higher. For more information, see Configuration Options and URL Parameters [page 703].

An expression binding is specified in an XML view by one of the following two options:

- `{=`*expression*`}`
  This variant uses one-way binding. This allows the automatic recalculation if the model values change.
- `{:=`*expression*`}`
  This variant uses one-time binding, meaning that the value is calculated only once. This variant needs less resources because no change listeners to the model have to be maintained.

The syntax of the *expression* is similar to JavaScript syntax, but you can only use a subset of the JavaScript expression syntax as defined in the table below. Additionally, you can embed values from the model layer into an expression as additional bindings by using one of the following syntaxes:

- `${`*binding*`}`
- `%{`*binding*`}`

binding can either be a simple path, or a complex binding. The embedded binding `${`*binding*`}` delivers a value formatted according to the target type of the control property the expression binding applies to, for example, "boolean" in case of `<Icon src="sap-icon://message-warning" visible="{= ${status} === 'critical' }">`. This can be undesirable or even lead to errors, for example, if OData V4 automatically adds the correct type for the "status" property which is string-like, not boolean. In such cases, use the syntax `%{binding}` instead. It is just a shortcut for `${path : 'binding', targetType : 'any'}`. In rare cases, you might also want to specify a different "targetType", for example "string", "boolean", "int" or "float". For more information how these values relate to OData types, see the sap.ui.model.odata.type API documentation or explore the XML Templating: UI5 OData Types sample in the Demo Kit. For more information about `targetType`, see the sap.ui.base.ManagedObject#bindProperty API documentation in the Demo Kit.

i Note

Expression binding can also be used with JavaScript. For example:

```
new Text({"visible" : "{= ${status} === 'critical' && ${amount} > 10000 }"});
```

or

```
new Icon({color : "'{= encodeURIComponent(${/ID}) }'"});
```

i Note

An expression binding does **not** validate binding paths. As a result, an expression binding will **not** detect incorrect or misspelled binding paths. But if you use an OData V4 model and try to bind data that does **not** exist in the model, a warning is logged in the console.

To embed a path containing a closing curly brace into an expression binding, use a complex binding syntax: `${path:'...'}`, for example `"{:= ${path:'target>extensions/[${name} === \'semantics\']/value'} === 'email'}"`. You can use this also to avoid variable replacement by build tools like Maven for special names like "Description" or "Name".

Syntax Element	Symbol
Literal	number, for example `42`, `6.022e+23` or `-273.15`
	object, for example `{foo: 'bar'}`
	string, for example `'foo'`
	`null`
	`true`
	`false`

Syntax Element	Symbol
Grouping	(...), for example `3 * (4 + 10)`

Unary operator	`!`
	`+`
	`-`
	`typeof`

Multiplicative operator	`*`
	`/`
	`%`

Additive operator	`+` `-`

Relational operator	`<`
	`>`
	`<=`
	`>=`

Strict equality operator	`===` `!==`
Binary logical operator	`&&` `||`
Conditional operator	`?`
Member access operator with the `.` operator	**Note** With these, you can use members and member methods on standard types such as string, array, number, and so on. Example: `${message>/}.length >0` or `${/firstName}.indexOf('S').`

Syntax Element	Symbol
Function call	`<function name>(...)` Example: • `text="{= Math.max(${/value1}, ${/value2}, ${/value3}) }"` **Note** You can use functions that are available via global symbols, such as `Math.max(...)` or `isNaN(...)`.
Array literals	`[...]`, for example `[2,3,5,7,11]`
Property/array access	`o[...]`, for example `'foo/bar'.split('/')[1]`
`in` operator	`'PI' in Math` (true) or `0 in []` (false)
Global symbol	`Array`, `Boolean`, `Date`, `encodeURIComponent`, `Infinity`, `isFinite`, `isNaN`, `JSON`, `Math`, `NaN`, `Number`, `Object`, `parseFloat`, `parseInt`, `RegExp`, `String`, `undefined`

Simple Example

Note

With expression binding you only need the XML view but no controller logic.

The following example shows how you use the custom formatter function to map an XML view to an expression binding in the XML view without controller logic.

The icon is only displayed if the status property in the view's default model has the value `critical`. You can use expression binding to replace the formatter function `myFormatter` in the controller with an expression binding in the XML view. You no longer need to implement any formatter function.

The application version without expression binding consists of the XML view (`sample.App.view.xml`) and the controller:

XML view (`sample.App.view.xml`)

```
<mvc:View controllerName="sample.App" xmlns="sap.ui.core"
xmlns:mvc="sap.ui.core.mvc">
...
  <Icon src="sap-icon://message-warning" visible="{path:'status',
formatter:'.myFormatter'}">
...

</mvc:View>
```

Controller (`sample.App.controller.js`)

```
...
myFormatter: function(sStatus) {
  return sStatus === "critical";
}
...
```

When using expression binding, however, you only need the XML view without controller logic (`sample.App.view.xml`):

```
<mvc:View controllerName="sample.app" xmlns="sap.ui.core"
xmlns:mvc="sap.ui.core.mvc">
...
  <Icon src="sap-icon://message-warning" visible="{= ${status} === 'critical' }">
...
</mvc:View>
```

> **i Note**
>
> Some symbols need to be escaped in XML views, for example `&&` needs to be escaped with `&&`.

More Complex Expressions

With the expression syntax sketched above it is possible to create more complex expressions as shown in the examples below.

> **i Note**
>
> We recommend to use formatter functions instead of very complex and hard-to-read expressions. Some characters that are used by operators, however, need to be escaped in XML, for example the left angle bracket (<) and the ampersand (&). Escaping makes it more difficult to read the expression. To avoid escaping, use one of the following options:
>
> - Rephrase the expression to make it more readable, for example, use `a > b` instead of `b < a`.
> - Use a custom formatter function.
>
> For more information about escaping in XML, see the W3C XML specification at http://www.w3.org/TR/xml/#syntax .

Examples for more complex expressions:

```
<!-- Set to visible if the status is critical and the amount
is above the threshold (note escaping of &&). -->
visible="{= ${status} === 'critical' &amp;&amp; ${amount} > 10000 }"
```

```
<!-- Text for amount level using language-dependent texts
from the resource model. -->
text="{= ${/amount} > 10000 ? ${i18n>/high} : ${i18n>/normal} }"
```

```
<!-- Set to visible if the rating is VIP, ignoring case
or if the order amount is greater than 10,000. -->
visible="{= ${/rating}.toUpperCase() === 'VIP' || ${/orderAmount} > 10000 }"
```

```
<!-- Set to visible if the rating contains VIP, ignoring
 the case. -->
visible={= RegExp('vip', 'i').test(${/rating}) }
```

```
<!-- Text is maximum of three values. -->
text="{= Math.max(${/value1}, ${/value2}, ${/value3}) }"
```

```
<!-- Control is enabled only if the order status is set. -->
enabled="{= ${/orderStatus} !== null }"
```

```
<!-- Set text to the second string 'middle', access second
element in the array generated via 'split'. -->
text="{= 'small@middle@long'.split('@')[1] }"
```

```
<!-- Concatenate literal strings and expression bindings
or bindings. -->
text="Hello {=${gender}==='male' ? 'Mr.' : 'Mrs.'} {lastName}"
```

```
<!-- Control such as a button in the toolbar of a table is
enabled only if there are items in the table. -->
enabled="{= ${/items}.length>0 }"
```

```
<!-- Set text by using a composite binding that combines
several values in a formatter defined by a parameterized
entry of an i18n language resource. -->
<!-- i18n language resource -->
successMsg=Message is available from {0} until {1}
errorMsg=Message is too short
<!-- View -->
<mvc:View controllerName="sample.App" xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
...
   <Text text="{= ${/data/message}.length &lt; 20
      ? ${i18n>errorMsg}
      : ${parts: [
         {path: 'i18n>successMsg'},
         {path: '/data/today', type:'sap.ui.model.type.Date', constraints:
{displayFormat:'Date'}},
         {path: '/data/tomorrow', type:'sap.ui.model.type.Date', constraints:
{displayFormat:'Date'}}
      ], formatter: '.formatMessage'}}" />
...
</mvc:View>
<!-- Controller -->
sap.ui.define(["sap/base/strings/formatMessage"], function(formatMessage) {
   sap.ui.controller("sample.App", {
      onInit: function() {
         ...
      },
      formatMessage : formatMessage,
      ...
   });
});
```

Related Information

Composite Binding [page 843]
Examples for Data Binding in Different View Types [page 851]

XML Templating [page 1018]
Configuration of the SAPUI5 Runtime [page 699]

Property Metadata Binding

The extended syntax makes it possible to access the metadata for certain properties of an entity in OData services, such as heading, label, and precision.

The extended data binding syntax is only valid for `PropertyBindings`. The annotations can be addressed either absolute or relative to a data path.

The consumption of label and description within an application is an example for a possible integration. Instead of copying the corresponding label text to a properties file, which in turn will be translated, a developer can bind a label against the label metadata field for the respective input field.

The binding must know the metadata part of the binding expression. The path to metadata must therefore start with `/#`.

- Absolute bindings
  An absolute binding path starts with the entity name followed by the property name. Property attributes can be accessed with `@` + `propertyName`, nodes can be accesses with the node name only.
  Example:

```
var myLabel = new sap.m.Label({text:"{/#Company/CompanyName/@sap:label}"});
```

- Relative bindings
  A relative binding path can be resolved relative to a data path/context.
  Example:

```
var myLabel = new sap.m.Label({text:"{/Companies(1)/CompanyCode/
#@sap:label}"});
var myLabel2 = new sap.m.Label({text:"{City/#@sap:label}"});
myLabel2.bindElement("Companies(1)");
```

Examples for Data Binding in Different View Types

Examples how complex syntax can be used for calculated fields in XML, HTML, and JS views.

XML View (Recommended)

```
<mvc:View controllerName="testdata.complexsyntax" xmlns:core="sap.ui.core"
    xmlns:mvc="sap.ui.core.mvc" xmlns="sap.ui.commons" xmlns:table="sap.ui.table"
    xmlns:html="http://www.w3.org/1999/xhtml">
    <html:h2>
        <Label text="Hello Mr. {path:'/singleEntry/firstName',
formatter:'.myFormatter'}, {/singleEntry/lastName}"></Label>
    </html:h2>
    <table:Table rows="{/table}">
```

```
        <table:columns>
            <table:Column>
                <Label text="Name"></Label>
                <table:template>
                    <TextField value="{path:'gender',
formatter:'.myGenderFormatter'} {firstName}, {lastName}"></TextField>
                </table:template>
            </table:Column>
            <table:Column>
                <Label text="Birthday"></Label>
                <table:template>
                    <TextField value="{parts:[{path:'birthday/day'},
{path:'birthday/month'},{path:'birthday/year'}],
formatter:'my.globalFormatter'}"></TextField>
                </table:template>
            </table:Column>
        </table:columns>
    </table:Table>
    <html:h2>
        <Label text="A type test: {path:'/singleEntry/amount',
type:'sap.ui.model.type.Float', formatOptions: { minFractionDigits: 1}} EUR"></
Label>
    </html:h2>
</mvc:View>
```

HTML View

```
<template data-controller-name="testdata.complexsyntax">
    <div>
        <h2><div data-sap-ui-type="sap.ui.core.HTML" id="MyHTMLControl" data-
content="<div>Hello Mr. {/singleEntry/firstName}, {/singleEntry/lastName}</
div>"></div></h2>
        <div data-sap-ui-type="sap.ui.table.Table" id="MyTable" data-rows="{/
table}">
            <div data-sap-ui-aggregation="columns">
                <div data-sap-ui-type="sap.ui.table.Column">
                    <div data-sap-ui-type="sap.ui.commons.Label" data-
text="Name"></div>
                    <div data-sap-ui-aggregation="template">
                        <div data-sap-ui-type="sap.ui.commons.TextField" data-
value="{firstName}, {lastName}"></div>
                    </div>
                </div>
                <div data-sap-ui-type="sap.ui.table.Column">
                    <div data-sap-ui-type="sap.ui.commons.Label" data-
text="Birthday"></div>
                    <div data-sap-ui-aggregation="template">
                        <div data-sap-ui-type="sap.ui.commons.TextField" data-
value="{parts:[{path:'birthday/day'},{path:'birthday/month'},{path:'birthday/
year'}], formatter:'my.globalFormatter'}"></div>
                    </div>
                </div>
            </div>
        </div>
        <h2><div data-sap-ui-type="sap.ui.commons.Label" id="MyLabelType" data-
text="A type test: {path:'/singleEntry/amount', type:'sap.ui.model.type.Float',
formatOptions: { minFractionDigits: 1}} EUR"></div></h2>
    </div>
</template>
```

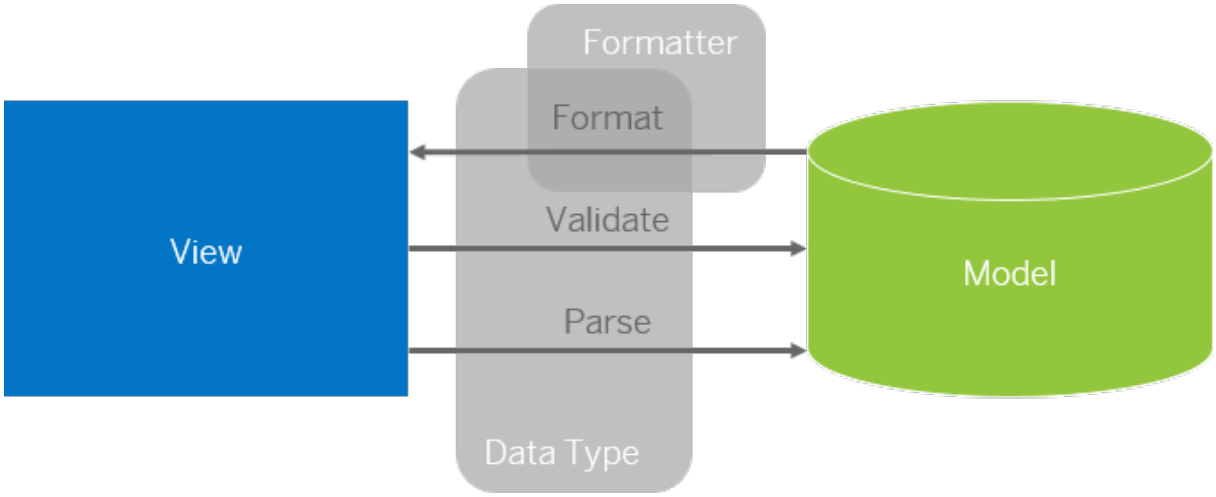
JS View

A JS view is not declarative view, but you may use the same syntax as in XML- and HTMLView, just do not forget to pass a controller instance as a parameter:

```
sap.ui.jsview("testdata.complexsyntax", {

    getControllerName: function() {
        return "testdata.complexsyntax";
    },
    /**
     *
     * @param oController may be null
     * @returns {sap.ui.cre.Control}
     */
    createContent: function(oController) {
        var c = sap.ui.commons;
        var aControls = [];
        var oLabel = new c.Label({text:"Hello Mr. {path:'/singleEntry/
firstName', formatter:'.myFormatter'}, {/singleEntry/lastName}"}, oController);
        aControls.push(oLabel);

        var oTable = new sap.ui.table.Table({rows:"{/table}"});
        var oColumn = new sap.ui.table.Column();
        var oLabel2 = new c.Label({text:"Name"});
        var oTextField = new c.TextField({value:"{path:'gender',
formatter:'.myGenderFormatter'} {firstName}, {lastName}"}, oController);
        oColumn.setLabel(oLabel2);
        oColumn.setTemplate(oTextField);
        oTable.addColumn(oColumn);
        aControls.push(oTable);
        var oLabel2 = new c.Label({text:"{path:'/singleEntry/amount',
type:'sap.ui.model.type.Float'}"});
        aControls.push(oLabel2);
        return aControls;
    }
});
```

Formatting, Parsing, and Validating Data

Data that is presented on the UI often has to be converted so that is human readable and fits to the locale of the user. On the other hand, data entered by the user has to be parsed and validated to be understood by the data source. For this purpose, you use formatters and data types.



- Views [page 787]
- Models [page 882]

Formatters are used to define the formatting of data on the UI while data types work in both directions: they format the data on the UI and parse and validate user input that is entered.

You can either use standard formatters and data types or define your own custom objects. SAPUI5 provides standard formatter classes that can be used to define custom data types and custom formatters.

If an error occurs during formatting or parsing, the following exception occurs: `sap/ui/model/FormatException` / `sap/ui/model/ParseException`.

> **i** Note
>
> For some controls like `sap/m/Input` you can also use API properties that define the data type and add additional features like restricted input options, for example, `<Input type="Number"/>`.

Formatters

A simple formatter can be defined directly in the controller. For example, you can format name data with the first letter in upper case:

```
myFormatter: function(sName) {
    return sName.charAt(0).toUpperCase() + sName.slice(1);
}
```

> **i Note**
>
> We recommend to use a separate `formatter.js` file that groups the formatters and makes them globally available in your app. You can then load the formatters in any controller by defining a dependency and instantiating the formatter file in a `formatter` variable. For more information, see Step 23: Custom Formatters [page 128] in the *Walkthrough* tutorial.

When the formatter is defined in the controller, you can use it, for example, in an XML view:

```
<Text text="{
    path : 'person/name',
    formatter : '.myFormatter'
}" />
```

> **i Note**
>
> You can also use predefined formatter functions for standard uses cases, like `formatMessage` from module `sap/base/strings/formatMessage`.

> **⚠ Caution**
>
> The automatic type determination for OData V4 interacts with `targetType` and can, thus, influence a formatter's input values. For more information on type determination in OData V4, see Type Determination [page 931].

Data Types

Simple Types

If you also want to validate and parse input values, you use data types. All data types inherit from the abstract `sap.ui.model.Type` class.

A subclass of this class is `sap.ui.model.SimpleType`. The currently available types inherit from `SimpleType` class.

For simple data types, you can generate the following parameters in the constructor:

- `formatOptions`: Format options define how a value is formatted and displayed in the UI.
- `constraints`: Constraints are optional and define how an input value entered in the UI should look like. During parsing the value is validated against these constraints. For example, an `Integer` type has a constraint for `maximum` that is automatically validated when parsing the input values.

```
<Input value="{
    path: '/number',
    type: 'sap.ui.model.type.Integer',
    formatOptions: {
        minIntegerDigits: 3
    },
        constraints: {
    maximum: 1000
    }
}" />
```

For a complete list of all simple types, see API Reference: `sap.ui.model.Type`.

OData Types

These types support OData V2 and V4 including relevant property facets as constraints. The OData types represent the OData EDM primitive types. For more information, see Primitive Data Types in the OData documentation .

For a complete list of all OData types, see API Reference: `sap.ui.model.odata.type`.

> **i Note**
>
> Also see the information on automatic type determination in OData V4 under Type Determination [page 931].

Custom Data Types

You can also define a custom data type based on `sap.ui.model.SimpleType` by specifying a custom implementation for `formatValue`, `parseValue`, and `validateValue`:

```
sap.ui.define([
    "sap/ui/model/SimpleType"
], function (SimpleType) {
    "use strict";
    return SimpleType.extend("sap.ui.demo.myCustomType", {
        formatValue: ...
        parseValue: ...
        validateValue: ...
    });
});
```

> **Example**
>
> Step 5: Adding a Flag Button [page 389] of the *Testing* tutorial shows how to implement a custom data type.

Related Information

Binding Syntax [page 840]

Simple Data Types

For a complete list of all simple types, see API Reference: `sap.ui.model.Type`.

sap.ui.model.type.Boolean

The `Boolean` data type represents a string.

The source value (value given in the model) must be given as boolean and is transformed into the type of the bound control property:

- `boolean`: No transformation needed
- `string`: "true" or "X" are interpreted as true, "false" and "" as false

The `Boolean` type has **no** format or validation constraint options.

Example how a `Boolean` type can be initialized:

```
// "TypeBoolean" required from module "sap/ui/model/type/Boolean"
// The source value is given as boolean.
var oType = new TypeBoolean();
```

sap.ui.model.type.Date

The `Date` data type represents a date (without time).

This type transforms a source value (given value in the model) into a formatted date string and the other way round.

The format patterns must be defined in LDML Date Format notation. For the output, the use of a style ("short, "medium", "long" or "full") instead of a pattern is preferred, as it will automatically use a locale-dependent date pattern.

Examples how a `Date` type can be initialized:

```
// "TypeDate" required  from "sap/ui/model/type/Date"
// The source value is given as Javascript Date object. The used output pattern
depends on the locale settings (default).
var oType = new TypeDate();
// The source value is given as Javascript Date object. The used output pattern
is "yy-MM-dd": e.g. 09-11-27
oType = new TypeDate({pattern: "yy-MM-dd"});
// The source value is given as string in "yyyy/MM/dd" format. The used output
style is "long". The styles are language dependent.
// The following styles are possible: short, medium (default), long, full
// This might be the common use case.
oType = new TypeDate({source: {pattern: "yyyy/MM/dd"}, style: "long"});
// The source value is given as string in "yyyy/MM/dd" format. The used output
pattern is "EEEE, MMMM d, yyyy": e.g. Saturday, August 22, 2043
oType = new TypeDate({source: {pattern: "yyyy/MM/dd"}, pattern: "EEEE, MMMM d,
yyyy"});
// The source value is given as timestamp. The used output pattern is
"dd.MM.yyyy": e.g. 22.12.2010
oType = new TypeDate({source: {pattern: "timestamp"}, pattern: "dd.MM.yyyy"});
// The source value is given as string. The used input pattern depends on the
locale settings (default). The used output pattern is "dd '|' MM '|' yyyy": e.g.
22 | 12 | 2010
oType = new TypeDate({source: {}, pattern: "dd.MM.yyyy"});
```

The `Date` type supports the following validation constraints:

- `maximum` (expects a date presented in the source-pattern format)
- `minimum` (expects a date presented in the source-pattern format)

sap.ui.model.type.DateTime

The `DateTime` data type represents an exact point of time (date and time).

This data type transforms a source value (given value in the model) into a formatted date+time string and the other way round.

The format patterns must be defined in LDML Date Format notation. For the output, the use of a style ("short, "medium", "long" or "full") instead of a pattern is preferred, as it will automatically use a locale-dependent date and time pattern.

> ⚠ Caution
>
> When talking about exact points in time, time zones are imported. The formatted output of the `DateTime` type currently shows the "local" time which equals the time settings of the machine on which the browser runs. If the source value is given as a JavaScript Date object or as a timestamp, the exact moment is sufficiently defined. For string source values this value is interpreted in "local" time if it does not explicitly have a time zone. Currently, all accepted time zone notations must be based on GMT/UTC.

Examples how a `DateTime` type can be initialized:

```
// "TypeDateTime" required from "sap/ui/model/type/DateTime"
// The source value is given as JavaScript Date object. The used output pattern
depends on the locale settings (default).
var oType = new TypeDateTime();
// The source value is given as JavaScript Date object. The used output pattern
is "yyyy/MM/dd HH:mm:ss": e.g. 2011/04/11 09:11:27
oType = new TypeDateTime({pattern: "yyyy/MM/dd HH:mm:ss"});
// The source value is given as string in "yyyy/MM/dd HH:mm:ss" format. The used
output style is "full". The styles are language dependent.
// The following styles are possible: short, medium (default), long, full
// This usecase might be the common one.
oType = new TypeDate({source: {pattern: "yyyy/MM/dd HH:mm:ss"}, style: "full"});
// The source value is given as string in "dd.MM.yyyy HH:mm:ss" format (no
timezone given). The used output pattern is "MMMM d, yyyy, HH:mm:ss.SSS": e.g.
August 22, 2043, 18:48:48.374
oType = new TypeDateTime({source: {pattern: "dd.MM.yyyy HH:mm:ss"}, pattern:
"MMMM d, yyyy, HH:mm:ss.SSS"});
// The source value is given as timestamp. The used output pattern is
"dd.MM.yyyy HH:mm": e.g. 22.12.2010 13:15
oType = new TypeDateTime({source: {pattern: "timestamp"}, pattern: "dd.MMM.yyyy
HH:mm"});
// The source value is given as string. The used input pattern depends on the
locale settings (default). The used output pattern is "hh-mm-ss '/' yy-MM-dd":
e.g. 06-48-48 / 43-08-22
oType = new TypeDateTime({source: {}, pattern: "hh-mm-ss '/' yy-MM-dd"});
// The source value is given as string in "dd.MM.yyyy HH:mm:ss X" format
(timezone is defined in ISO8601 format, e.g. "+02:00"). The used output pattern
depends on the locale settings (default).
oType = new TypeDateTime({source: {pattern: "dd.MM.yyyy HH:mm:ss X"}});
// The source value is given as string in "dd.MM.yyyy HH:mm:ss Z" format
(timezone is defined in RFC822 format, e.g. "+0200"). The used output pattern
depends on the locale settings (default).
oType = new TypeDateTime({source: {pattern: "dd.MM.yyyy HH:mm:ss Z"}});
```

```
// The source value is given as string in "dd.MM.yyyy HH:mm:ss z" format
(timezone is currently defined as e.g. "GMT+02:00", "UTC+02:00", "UT+02:00" or
"Z" (shortcut for "UTC+00:00")).
// The used output pattern depends on the locale settings (default).
oType = new TypeDateTime({source: {pattern: "dd.MM.yyyy HH:mm:ss z"}});
```

The `DateTime` type supports the following validation constraints:

- `maximum` (expects a `dateTime` presented in the source-pattern format)
- `minimum` (expects a `dateTime` presented in the source-pattern format)

sap.ui.model.type.Float

The `Float` data type represents a float value.

The source value for this data type, that is, the value given in the model must be given as a number and is transformed into the type of the bound control property:

- `float`: No transformation needed
- `integer`: Value is rounded using `Math.floor`
- `string`: Value is formatted or parsed according to the given output pattern

Examples how a `Float` type can be initialized:

```
// "TypeFloat" required from module "sap/ui/model/type/Float"
// The source value is given as JavaScript number. Output is transformed into
the type of the bound control property.
// If this type is "string" (e.g. the value property of the TextField control)
the used default output pattern parameters depend on locale and fixed settings.
var oType = new TypeFloat();
// The source value is given as JavaScript number. Output is transformed into
the type of the bound control property.
// If this type is "string" (e.g. the value property of the TextField control)
the given output pattern is used (parameters which are not specified are taken
from the default pattern)
oType = new TypeFloat({
  minIntegerDigits: 1, // minimal number of non-fraction digits
  maxIntegerDigits: 99, // maximal number of non-fraction digits
  minFractionDigits: 0, // minimal number of fraction digits
  maxFractionDigits: 99, // maximal number of fraction digits
  groupingEnabled: true, // enable grouping (show the grouping separators)
  groupingSeparator: ",", // the used grouping separator
  decimalSeparator: "." // the used decimal separator
});
```

The `Float` type supports the following validation constraints:

- maximum
- minimum

sap.ui.model.type.Integer

The `Integer` data type represents an integer value.

The source value for this data type, that is, the value given in the model, must be given as a number and is transformed into the type of the bound control property:

- `float`: Value is rounded using `Math.floor`
- `integer`: No transformation needed
- `string`: Value is formatted or parsed according to the given output pattern

Examples how an `Integer` type can be initialized:

```
// "TypeInteger" required from module "sap/ui/model/type/Integer"
// The source value is given as JavaScript number. Output is transformed into
the type of the bound control property.
// If this type is "string" (e.g. the value property of the TextField control)
the used default output pattern parameters depend on locale and fixed settings.
var oType = new TypeInteger();
// The source value is given as JavaScript number. Output is transformed into
the type of the bound control property.
// If this type is "string" (e.g. the value property of the TextField control)
the given output pattern is used (parameters which are not specified are taken
from the default pattern)
oType = new TypeInteger({
  minIntegerDigits: 1, // minimal number of non-fraction digits
  maxIntegerDigits: 99, // maximal number of non-fraction digits
  minFractionDigits: 0, // minimal number of fraction digits
  maxFractionDigits: 0, // maximal number of fraction digits
  groupingEnabled: false, // enable grouping (show the grouping separators)
  groupingSeparator: ",", // the used grouping separator
  decimalSeparator: "." // the used decimal separator
});
```

The `Integer` type supports the following validation constraints:

- maximum
- minimum

sap.ui.model.type.String

The `String` data type represents a string.

The source value (value given in the model) must be given as a number and is transformed into the type of the bound control property:

- `string`: No transformation needed
- `integer/float`: String is parsed accordingly
- `boolean`: "true" or "X" are interpreted as true, false, and " " as false

The `string` type does not have any format options.

Example how a `String` type can be initialized:

```
// "TypeString" required from module "sap/ui/model/type/String"
// The source value is given as string. The length of the string must not be
greater than 5.
```

```
var oType = new TypeString(null, {maxLength: 5});
```

The `String` type supports the following validation constraints:

- `maxLength` (expects an integer number)
- `minLength` (expects an integer number)
- `startsWith` (expects a string)
- `startsWithIgnoreCase` (expects a string)
- `endsWith` (expects a string)
- `endsWithIgnoreCase` (expects a string)
- `contains` (expects a string)
- `equals` (expects a string)
- `search` (expects a regular expression)

For more information, see API Reference: `sap.ui.model.type.String`.

sap.ui.model.type.Time

The `Time` data type represents a time (without date).

This type transforms a source value (given value in the model) into a formatted time string and the other way round.

The format patterns must be defined in LDML Date Format notation. For the output, the use of a style ("short, "medium", "long" or "full") instead of a pattern is preferred, as it will automatically use a locale dependent time pattern.

Examples how a `Time` type can be initialized:

```
// "TypeTime" required from module "sap/ui/model/type/Time"
// The source value is given as JavaScript Date object. The used output pattern
depends on the locale settings (default).
var oType = new TypeTime();
// The source value is given as JavaScript Date object. The used output pattern
is "hh-mm-ss": e.g. 09-11-27
oType = new TypeTime({pattern: "hh-mm-ss"});
// The source value is given as string in "hh-mm-ss" format. The used output
style is "short". The styles are language dependent.
// The following styles are possible: short, medium (default), long, full
// This might be the common use case.
oType = new TypeTime({source: {pattern: "hh-mm-ss"}, style: "short"});
// The source value is given as string in "hh/mm/ss/SSS" format. The used output
pattern is "HH:mm:ss '+' SSS 'ms'": e.g. 18:48:48 + 374 ms
oType = new TypeTime({source: {pattern: "hh/mm/ss/SSS"}, pattern: "HH:mm:ss '+'
SSS 'ms'"});
// The source value is given as timestamp. The used output pattern is "HH
'Hours' mm 'Minutes'": e.g. 18 Hours 48 Minutes
oType = new TypeTime({source: {pattern: "timestamp"}, pattern: "HH 'Hours' mm
'Minutes'"});
// The source value is given as string. The used input pattern depends on the
locale settings (default). The used output pattern is "hh:mm a": e.g. 06:48 PM
oType = new TypeTime({source: {}, pattern: "hh:mm a"});
```

The `Time` type supports the following validation constraints:

- `maximum` (expects a time presented in the source-pattern format)
- `minimum` (expects a time presented in the source-pattern format)

sap.ui.model.type.DateTimeInterval

The interval data types represent intervals between two date/time related properties.

Three new types are used to format two date related properties from a model for displaying in the UI. Additionally they are used to parse and validate the values in UI controls before they are saved back to the model. All of the them are subtypes of `sap.ui.model.CompositeType` and are supposed to be set with a composite binding. The new interval types are:

- `sap.ui.model.type.DateInterval` - represents a date interval (without time) which transforms the source values into a formatted date interval string and the other way around.
- `sap.ui.model.type.DateTimeInterval` - represents a date interval with the exact point of time (date and time) which transforms the source values into a formatted date+time interval string and the other way around
- `sap.ui.model.type.TimeInterval` - represents a time interval (without date) which transforms the source values into a formatted time interval string and the other way around

Usage with JSON, OData V2 or V4 models

> **i** Note
>
> The new date interval types can be used together with different types of model. However, there are some differences in the usage when the new date interval types get used with a JSON, OData V2 or V4 model.

The interval types need two JavaScript `Date` objects from the sub-bindings to format them as a date interval string. If the values which come from the sub-binding aren't instances of JavaScript `Date` object, they need to be converted to JavaScript `Date` objects before they are forwarded to the date interval types. The conversion can be done by setting a corresponding type on the sub-binding and this type knows how the value which comes directly from the model can be converted to a JavaScript `Date` object. A date interval type works together with the types on the sub-bindings to get the original value from the model converted to a JavaScript `Date` object.

> **!** Restriction
>
> One exception is with the OData V2 Model. Although the date fields are saved as string in the model, they get converted to JavaScript `Date` objects by the open source library which is used in `v2.ODataModel`. Therefore it's not needed to set an extra type on the sub-binding when the date interval types are used together with the OData V2 model.

The following example shows how this should be setup with a `sap/m/Table` which is bound to an OData V4 model. The table consists of four different columns whereas the `StartsAt` and `EndsAt` represent date field as string with Edm type `Edm.DateTimeOffset`. The date interval formatting is done by combining these two date fields together.

In order to let the date interval type correctly, get two JavaScript `Date` objects from the sub-bindings, a type `sap.ui.model.odata.type.DateTimeOffset` is set on each sub-binding to convert the date string into a JavaScript `Date` object.

```
// "ODataModel" required from module "sap/ui/model/odata/v4/ODataModel"
// "Table" required from module "sap/m/Table"
// "Text" required from module "sap/m/Text"
// "Column" required from module "sap/m/Column"
// "ColumnListItem" required from module "sap/m/ColumnListItem"
var urlV4 = "/databinding/proxy/http/services.odata.org/TripPinRESTierService/
(S(kqyippfvpypubsah2zi1enbi))/";
// Some OData V4 model configuration
var oModelConfig = {
    groupId : "$direct",
    serviceUrl: urlV4,
    synchronizationMode: "None",
    operationMode: "Server"
};
var oModel = new ODataModel(oModelConfig);
var oTable = new Table({
    growing: true,
    columns : [
        new Column({header: new Text({text: "Name"})}),
        new Column({header: new Text({text: "StartsAt"})}),
        new Column({header: new Text({text: "EndsAt"})}),
        new Column({header: new Text({text: "Interval"})}),
    ]
});
oTable.setModel(oModel);
oTable.bindItems({
    path:"/People('russellwhyte')/Trips",
    template: new ColumnListItem({
        cells: [
           new Text({text:"{Name}"}),
            new Text({text:"{StartsAt}"}),
            new Text({text:"{EndsAt}"}),
            new Text({text:{
                parts:[
                    {
                        path:'StartsAt',
                        // requires OData type to convert the date
                        // string into JavaScript Date object
                        type: "sap.ui.model.odata.type.DateTimeOffset"
                    },{
                        path:'EndsAt',
                        // requires OData type
                        type: "sap.ui.model.odata.type.DateTimeOffset"
                    }
                ],
                type: 'sap.ui.model.type.DateInterval',
                formatOptions: {format: "yMMMdd"}
            }})
        ]
    })
});
```

Formatter Classes

For a complete list of all formatter classes, see API Reference: `sap.ui.model.Type`.

Date Format

The `sap.ui.core.format.DateFormat` class can be used to parse a string representing a date, time, or the combination of date and time into a JavaScript date object and vice versa (also known as format). `DateFormat` formats and parses date and time values according to a set of format options. It can also be used to format intervals. A pattern base on Locale Data Markup Language (LDML) date format notation can be given and the date is formatted following the given pattern. `DateFormat` can also format the date and time into relative values on "day" level.

Instantiation

You instantiate of `sap.ui.core.format.DateFormat` by calling the `getter` defined on the `DateFormat` (and not by calling the constructor):

```
var oDateFormat = sap.ui.core.format.DateFormat.getDateInstance();
// or
var oDateTimeFormat = sap.ui.core.format.DateFormat.getDateTimeInstance();
// or
var oTimeFormat = sap.ui.core.format.DateFormat.getTimeInstance();
```

Parameters

There are several parameters which affect the final result of formatting and parsing a date. If no parameter is set, the default setting defined in the current locale is used to format and parse the date.

Date Pattern

`format` (recommended):

The format string does contain pattern symbols (e.g. `yMMMd` or `Hms`) and will be converted into the pattern in the used locale, which matches the wanted symbols best.

The symbols must be in canonical order, that is: Era (`G`), Year (`y/Y`), Quarter (`q/Q`), Month (`M/L`), Week (`w/W`), Day-Of-Week (`E/c`), Day (`d/D`), Hour (`h/H/k/K`), Minute (`m`), Second (`s`), Timezone (`z/Z/v/V/O/X/x`).

```
var oFormat = sap.ui.core.format.DateFormat.getInstance({
    format: "yMMMd"
});
oFormat.format(new Date()); //string in locale de "29. Jan. 2017"; string in
locale en "Jan 29, 2017"
```

`pattern`: A date pattern in LDML date format notation. The date is formatted based on the given pattern.

```
var oDateFormat = sap.ui.core.format.DateFormat.getDateInstance({
    pattern: "EEE, MMM d, yyyy"
});

var oNow = new Date();
oDateFormat.format(oNow); //string in the same format as "Thu, Jan 29, 2017"
```

> **i Note**
>
> If you define `format` and `pattern`, the `format` will be ignored!

The letters which can be included in this pattern are explained in the following table:

Table 25: Patterns

Letter	Replaced By
`G`	era string for the current date
`y`	year
`Y`	Same as `y`, but uses the ISO year-week calendar
`M`	month
`L`	month in stand-alone format
`w`	week number in year
`(W)`	(currently not supported) week number in month
`(D)`	(currently not supported) day number in year
`d`	day number in month
`Q`	quarter number
`q`	quarter number in stand-alone format
`(F)`	(currently not supported) day of week in month
`E`	day of week
`c`	day of week in stand-alone format
`u`	day number of week
`a`	AM or PM
`j`	Can only be used in the `format` option, not in the `pattern`. It will be replaced by `h`, `H`, `K` or `k` depending on the locale preferred time cycle type (12-hour or 24-hour).

Letter	Replaced By
J	Can only be used in the format option, not in the pattern. It will be replaced by h, H, K or k. However unlike j it requests no dayPeriod marker such as "am" or "pm". It is typically used where there is enough context that the day period not necessary. For example, with jmm, "18:00" could appear as "6:00 PM", while with Jmm, it would appear as "6:00" (no PM).
H	hour (0-23)
k	hour (1-24)
K	hour (0-11)
h	hour (1-12)
m	minute
s	second
S	fractional second
z	time zone
Z	time zone in RFC 822 format
X	time zone in ISO 8601 format

Style

This can be set with either empty, short, medium or long. If no pattern is given, a locale-dependent default date pattern of that style is used which is extracted from the current locale.

If in addition to style pattern or format is defined, the style is ignored.

If you use the datetime instance by calling getDateTimeInstance, you can define different styles for date and time. For example, medium/short defines medium style for the date and short style for the time.

Relative Format

Relative format on "day"' level is only supported by the date instance but not the date time or the time instance.

- relative: if this is set to true, the date is formatted relatively to the actual date if it's within the given date range.
- relativeRange: the day range used for relative formatting. The default is set to 6 which means only dates within the last six days, the acutal date, and the next six days are formatted relatively.
- relativeScale: the relative scale is chosen depending on the difference between the given date and now, possible relative scales are: year, month, week, day, hour, minute, second, and auto
  If auto is set, the scale is chosen dependent on the actual difference.

Interval

`interval`: If this is set to `true`, the `DateFormat` is capable to format two dates as an interval. The `format` method expects an array with two dates as the first argument.

If the format option is set with necessary symbols, the `DateFormat` displays the fields which have the same value between the two dates only once in the result string. For example, the interval "Jan 10, 2008 - Jan 12, 2008" will be formatted as "Jan 10-12, 2008". Otherwise the two given dates are formatted separately and concatenated with locale-dependent pattern.

```
var oFormat = sap.ui.core.format.DateFormat.getInstance({
    format: "yMMMd",
    interval: true
});
var oDate1 = new Date(2017, 3, 11);
var oDate2 = new Date(2017, 4, 11);
oFormat.format([oDate1, oDate2]);
// string in locale de "11. Apr. – 11. Mai 2017";
// string in locale en "Apr 11 – May 11, 2017"
```

Parsing

`strictParsing`: If this is set to `true`, the `date` string is validated during parsing. If it doesn't pass the validation, `null` is returned.

```
var oDateFormat = sap.ui.core.format.DateFormat.getDateInstance({
    relative: true
});

var nMS = 1000 * 60 * 60 * 24; //milliseconds in a day
var oNow = new Date();
var oDate = new Date(oNow.getTime() - nMS);
oDateFormat.format(oDate); //returns yesterday

oDate = new Date(oNow.getTime() + 7 * nMS);
oDateFormat.format(oDate); //isn't returned in relative format because the
default value of relativeRange is [6|-6,]
```

Related Information

API Reference: `sap.ui.core.format.DateFormat`

Number Format

The `sap.ui.core.format.NumberFormat` class can be used to parse a string representing a number (float or integer) into a JavaScript `number` and vice versa (also known as `format`).

`NumberFormat` uses the parameters defined for the current locale. These parameters can be overwritten on each instance by setting the format options.

There are four types of formatters defined in `NumberFormat`:

- Integer formatter: formats and parses only the integer digits; decimal digits are ignored

- Float formatter: formats and parses both integer and decimal digits.
- Percent formatter: formats the number into a string with percentage sign. It validates the number whether it contains the right percentage sign in its parser.
- Currency formatter: formats the number by using the parameters defined for the given currency code. Either currency symbol, currency code, or none of both can be included in the final formatted string. It parses the given string into an array which contains both the currency number and currency code.

Instantiation

The instantiation of `sap.ui.core.format.NumberFormat` is done by calling `getter` defined on `NumberFormat` (and not by using the constructor).

```
var oIntegerFormat = sap.ui.core.format.NumberFormat.getIntegerInstance();
// or
var oFloatFormat = sap.ui.core.format.NumberFormat.getFloatInstance();
// or
var oPercentFormat = sap.ui.core.format.NumberFormat.getPercentInstance();
// or
var oCurrencyFormat = sap.ui.core.format.NumberFormat.getCurrencyInstance();
```

Parameters

All parameters have their default value defined in the current locale. Therefore, if no parameter is given when instantiating the formatter instance, it fetches the parameters from the current locale. All parameters can be overwritten by giving a format option object in the `getter` of the formatter. There are a bunch of parameter defined for the four types of formatters. Most of them are shared among the types and the rest are specifically defined for a certain kind of formatter.

Integer and Decimal Digits

- `minIntegerDigits`: minimal number of non-fraction digits. If there are less integer digits in the number than the value here, `'0'(s)` is prepended in the final result.
- `maxIntegerDigits`: maximal number of non-fraction digits. If there are more digits in the number than the value here, all integer digits in the final result are replace by `?`.
- `minFractionDigits`: minimal number of fraction digits. If there are less decimal digits in the number than the value here, '0'(s) is appended in the final result.
- `maxFractionDigits`: maximal number of fraction digits. If there are more decimal digits in the number than the value here, those digits are discarded from the result and the least significant digit is calculated by using the given `roundingMode` parameter.
- `decimals`: number of decimal digits in the final result. Same result is achieved by setting both `minFractionDigits` and `maxFractionDigits` to this value.
- `precision`: number of digits used to display the number, for example with precision 5 a number could be 1.3456 or 134.45.

- `shortDecimals`: number of decimal digits in the shortifed number when parameter `style` is set to `short` or `long`. If this isn't set, the parameter `decimal` is used instead.

```
// Locale object is created and set to the format in order to have a user locale
independent formatter.
// In most of the cases, this locale object isn't needed.
var oLocale = new sap.ui.core.Locale("en-US");
var oFormatOptions = {
    minIntegerDigits: 3,
    maxIntegerDigits: 5,
    minFractionDigits: 2,
    maxFractionDigits: 4
};
var oFloatFormat =
sap.ui.core.format.NumberFormat.getFloatInstance(oFormatOptions, oLocale);
oFloatFormat.format(1.1); // returns 001.10
oFloatFormat.format(1234.567); // returns 1,234.567
oFloatFormat.format(123456.56789); // returns ??,???.5679
```

```
// Locale object is created and set to the format in order to have a user locale
independent formatter.
// In most of the cases, this locale object isn't needed.
var oLocale = new sap.ui.core.Locale("en-US");
var oFormatOptions = {
    style: "short",
    decimals: 1,
    shortDecimals: 2
};
var oFloatFormat =
sap.ui.core.format.NumberFormat.getFloatInstance(oFormatOptions, oLocale);
oFloatFormat.format(1234.56); // returns 1.23K (shortified number takes the
shortDecimals parameter)
oFloatFormat.format(123.456); // returns 123.5 (non-shortified number takes the
decimals parameter)
```

Separator and Signs

- `groupingEnabled` defines whether the integer digits are put into groups which are separated by the `groupingSeparator` parameter
- `groupingType` defines the type of grouping. Either `Arabic` or `Indian` can be set here.
- `groupingSeparator` defines the separator of grouping.
- `decimalSeparator` defines the symbol of decimal point.
- `groupingSize` only used if you don't want the locale-dependent grouping, for example 3 digits for de or en
- `groupingBaseSize` only used if your locale uses a specific group size for the first group (like Indian), and you don't want to use the standard
- `plusSign`
- `minusSign`

Compact Format

You can use compact format to format a number using a given scale. For example, 1000000 may be formatted under en-US locale as *1 Million*.

To format a number in compact format, set the option `style` to either `short` or `long`. These styles control which version of scale name is used. For example, 1000000 is formatted as *1M* with `short` and *1 Million* with `long`.

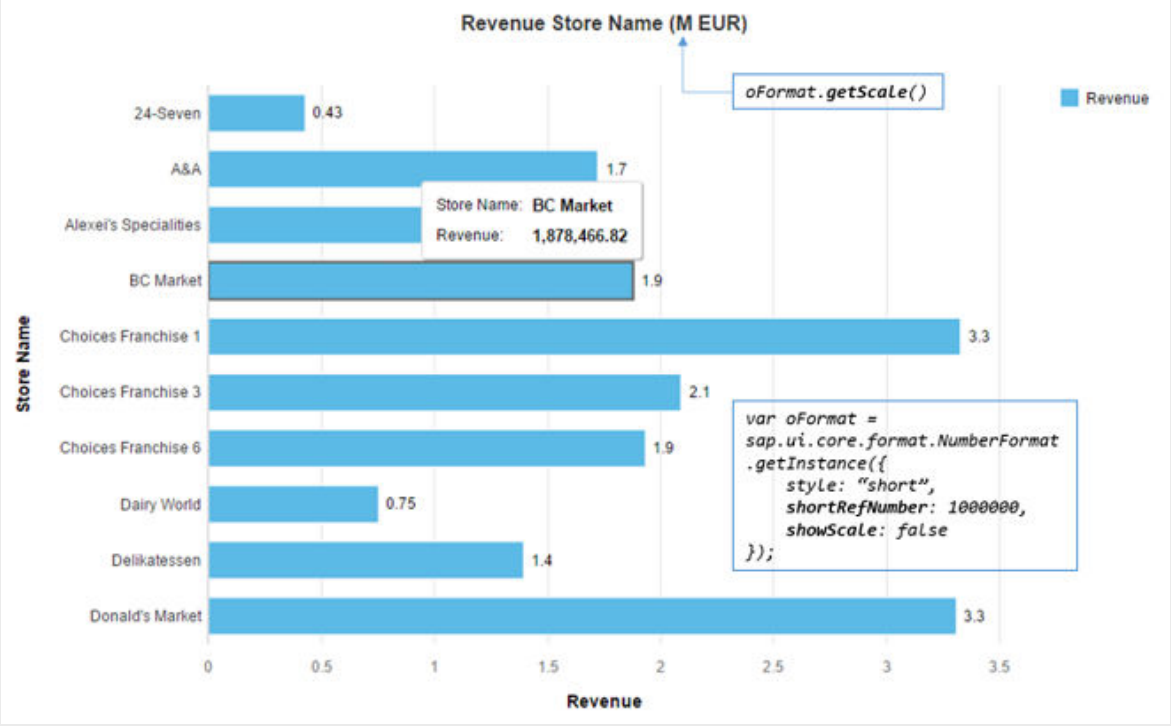
The scale can be selected automatically based on the given number, or you can set it explicitely by using `shortRefNumber`. You can set this option with a number which is then used for calculating the scaling factor for formatting all given numbers to this formatter.

To hide the scaling formatter from the formatted number and only be shown once on the screen, you can use option `showScale`. In order to get the scaling factor name of the number set to `shortRefNumber` under the current running locale, you use method `getScale`.

To control the start the starting point of numbers which should be displyed in compact format, you use `shortLimit`.

Example

In the following chart, all numbers both on the chart and axis should be formatted using the same scaling factor. The scaling factor should only appear in the chart title and be hidden from the formatted number. In order to achieve this, the option `shortRefNumber` is set to `1000000` and `showScale` is set to `false`. The corresponding scaling factor name is returned by calling the `getScale` method.



Miscellaneous

- `emptyString` defines what empty string (`""`) is parsed as and what is formatted as `emptyString`. The allowed values are only `NaN`, `null` or `0`. Default setting is `NaN`.
- `pattern`: a pattern which follows the CLDR syntax. The number is then formatted according to the given pattern.
- `roundingMode`: defines a rounding behavior for discarding the digits after the maximum decimal digits defined by `maxFractionDigits` or `decimals`. Rounding will only be applied, if the formatting value is of type number.

Table 26: Rounding Modes (with decimals parameter set to 1)

Number	FLOOR	CEILING	TOWARDS_ZERO	AWAY_FROM_ZERO	HALF_FLOOR	HALF_CEILING	HALF_TOWARDS_ZERO	HALF_AWAY_FROM_ZERO
2.21	2.2	2.3	2.2	2.3	2.2	2.2	2.2	2.2
2.25	2.2	2.3	2.2	2.3	2.2	2.3	2.2	2.3
2.29	2.2	2.3	2.2	2.3	2.3.	2.3	2.3	2.3
-2.21	-2.3	-2.2	-2.2	-2.2	-2.2	-2.2	-2.2	-2.2
-2.25	-2.3	-2.2	-2.2	-2.3	-2.3	-2.2	-2.2	-2.3
-2.29	-2.3	-2.2	-2.2	-2.3	-2.3	-2.3	-2.3	-2.3

Parsing

Parsing a number

A formatted number which contains a locale-dependent grouping separator, decimal point, or percentage sign can be parsed into a number object using `sap.ui.core.format.NumberFormat`. Those number string may not be correctly parsed by using `parseInt` or `parseFloat` in JavaScript.

```
// Locale object is created and set to the format in order to have a user locale
independent formatter.
// In most of the cases, this locale object isn't needed.
var oLocale = new sap.ui.core.Locale("en-US");
var oFloatFormat = sap.ui.core.format.NumberFormat.getFloatInstance(oLocale);
oFloatFormat.parse("1,234.567"); // returns 1234.567
oFloatFormat.parse("12.34%"); // returns 0.1234
```

Related Information

API Reference: `sap.ui.core.format.NumberFormat`
Unit Formatting [page 873]
Currency Formatting [page 878]

File Size Format

The `sap.ui.core.format.FileSizeFormat` class can be used to format a number into a string which contains both the number and the most appropriate size unit. It supports all parameters defined in

`sap.ui.core.format.NumberFormat`. It can also parse a file size string with or without unit into a number which represents the same size in unit byte.

Instantiation

The instantiation of `sap.ui.core.format.FileSizeFormat` is done by calling the `getter` defined on the `FileSizeFormat` (and not by calling the constructor):

```
var oFileSizeFormat = sap.ui.core.format.FileSizeFormat.getInstance();
```

Parameters

`binaryFilesize`: if this is set to `true`, base `2` is used which means 1 Kilobyte = 1024 Byte. Otherwise base 10 is used which means 1 Kilobyte = 1000 Byte. The default value is `false`.

```
var oFileSizeFormat =  sap.ui.core.format.FileSizeFormat.getInstance({
    binaryFilesize: true,
    decimals: 2
});

oFileSizeFormat.format(1023); //returns 1,023.00 Bytes because it's smaller than
1 KB (1024 Bytes)

oFileSizeFormat = sap.ui.core.format.FileSizeFormat.getInstance({
    binaryFilesize: false,
    decimals: 2
});
oFileSizeFormat.format(1023); //returns 1.02 KB because it's bigger than 1 KB
(1000 Bytes)
```

Parsing

`FileSizeFormat` can also parse a string which contains both number and unit of a number which represents the same amount of size in unit byte.

```
var oFileSizeFormat =  sap.ui.core.format.FileSizeFormat.getInstance();

oFileSizeFormat.parse("1.23 MiB"); //returns 1289748.48
oFileSizeFormat.parse("1 MB"); // returns 1230000
```

Related Information

API Reference: `sap.ui.core.format.FileSizeFormat`

Unit Formatting

SAPUI5 supports the formatting and parsing of units. These unit formats are by default taken from the CLDR. Besides the default units defined in the CLDR, you can also define custom units.

Introduction

Similar to the formatting of currencies, the new unit formatting feature allows you to combine a number value with a localized unit string. Still the actual numbers themselves can be formatted in all kinds of styles, for example, with different decimals or fraction digits.

With version 1.54 all language files will include the CLDR data for formatting. You can check out the public CLDR github repository for an overview of all supported languages and units: https://github.com/unicode-cldr/cldr-units-modern

The default CLDR units include a pattern for formatting a number value in the given unit and the set locale. Additionally, you can now parse an already formatted string back into the unit-format code from the CLDR and of course the raw number value. The same is true for your self-defined custom units.

In addition to the formatting and parsing patterns, the CLDR also provides a display-name for all units, as well as grammatical plural forms. The display-name for the unit `volume-cubic-inch` is *inches³*. The formatting output might however look like this: *12 in³*.

Usage

Simple Sample

Since the units and their formatting patterns are already included in the respective language JSON files, you can simply instantiate a new unit format instance via the `NumberFormat.getUnitInstance()` factory. The resulting instance supports formatting and parsing. While the `FormatOptions` are used to format the Number itself, the specific unit code is passed to the format/parse method.

```
// create a simple unit formatting instance, without any additional options
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   var oUnitFormat = NumberFormat.getUnitInstance();
   oUnitFormat.format(12345.678, "speed-mile-per-hour"); // output: 12,345.678
mph
   oUnitFormat.parse("12345.678 mph"); // output: [12345.678, "speed-mile-per-
hour"]
});
```

Complex Sample

Besides a simple formatting and parsing of units defined in the CLDR, you can also use the known format options for formatting numbers independent of the unit. The style of the number output format can be defined as either `short` or `long`.

Essentially, the new unit formatting can be combined together with the existing number formatting.

```
// new unit formatter, decimals are limited to 2, and the output style is set to
"short"
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   var en = new sap.ui.core.Locale("en");
   var oUnitFormat = NumberFormat.getUnitInstance({decimals:2, style:"short"},
en);
   console.log(oUnitFormat.format(12345.678, "speed-mile-per-hour")); //
output:  12.35K mph
   console.log(oUnitFormat.parse("12.35K mph")); // output: [12350, "speed-mile-
per-hour"]
});
// new unit formatter, decimals are limited to 2, and the output style is set to
"long"
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   var en = new sap.ui.core.Locale("en");
   var oUnitFormat = NumberFormat.getUnitInstance({decimals:2, style:"long"},
en);
   console.log(oUnitFormat.format(12345.678, "speed-mile-per-hour")); // output:
12.35 thousand mph
   console.log(oUnitFormat.parse("12.35 thousand mph")); // output: [12350,
"speed-mile-per-hour"]
});
```

The unit's `displayname` can also be retrieved based on the data from the CLDR.

```
sap.ui.require(["sap/ui/core/format/NumberFormat", "sap/ui/core/Locale", "sap/ui/
core/LocaleData"],
    function(NumberFormat, Locale, LocaleData) {
    console.log(LocaleData.getInstance(new
Locale("en")).getUnitDisplayName("speed-mile-per-hour")); // output: miles/hour
});
```

Plural Forms and RTL

Depending on the set locale/language, the output also correctly regards grammatical plural forms, as well as right-to-left orientation. In some Arabic languages, for example, there is a distinction between "many" and "one", with "one" being a single string without a number in it:

```
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   var ar = new sap.ui.core.Locale("ar");
   var oUnitFormat = NumberFormat.getUnitInstance({decimals:2, style:"long"},
ar);
   console.log(oUnitFormat.format(123456.789, "angle-revolution")); // 123.46 ألف
دورة
   console.log(oUnitFormat.format(1, "angle-revolution")); // دورة
});
```

And here's an example of right-to-left orientation in Hebrew:

```
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   var he = new sap.ui.core.Locale("he");
   var oUnitFormat = NumberFormat.getUnitInstance({decimals:2, style:"long"},
he);
   console.log(oUnitFormat.format(12345.678, "speed-mile-per-hour")); //  12.35
אלף mph
});
```

> **i Note**
>
> The right-to-left languages include a special whitespace character as a marker. This character is of course invisible, but you should take note of it, in case you intend to do string comparisons and other string operations. For example, in the Chrome debugger the RTL mark is visualized as a red dot.

Custom Units

Instance-Exclusive Units

The unit NumberFormat instance also allows you to specify custom units, which can be used for formatting, as well as parsing. All you have to do is add your custom units as an additional parameter in the `NumberFormat.getUnitInstance()` factory.

In the following example, you can see how this is done for a specific instance.

```
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {

	var oFormat = sap.ui.core.format.NumberFormat.getUnitInstance({
		customUnits: {
			"zomb": {
				"unitPattern-count-one": "{0} Zombie...",
				"unitPattern-count-other": "{0} Zombies!!"
			}
		}
	});

	console.log(oFormat.format(1, "zomb"));     // 1 Zombie...
	console.log(oFormat.format(9001, "zomb"));  // 9.001 Zombies!!

	console.log(oFormat.parse("12 Zombies!!")); // [12, "zomb"];
});
```

> ⚠ Caution
>
> The custom units defined on the number format instance will be exclusive to this instance. No other instances are affected. In addition, once you define custom units for an instance, only those units will be formatted and parsed by that instance. This also means that custom units defined via the Configuration are not taken into account for this specific instance.
>
> This is done to circumvent ambiguities and unit clashes with the CLDR units. So in the above example, only Zombies can be formatted, but no Gigawatt (CLDR key: `power-gigawatt`).
>
> > ✤ Example
> >
> > ```
> > 	// the previous Unit instance is used
> > 	// formatting/parsing Zombies is fine
> > 	console.log(oFormat.format(9001, "zomb"));  // 9.001 Zombies!!
> > 	console.log(oFormat.parse("12 Zombies!!")); // [12, "zomb"];
> >
> > 	// formatting/parsing Giga-Watt does not work (because of the
> > exclusivity of the custom units on the above instance)
> > 	console.log(oFormat.format(1.21, "power-gigawatt")); // "": results
> > in an empty string
> > ```

If you need both, CLDR units and custom units, you simply have to create two separate number format instances.

Globally Configured Units

You can also add custom units via the format settings in the Core configuration. Contrary to the custom units defined exclusively on a single unit-formatter instance, these custom units will be available in ALL unit-formatted instances for the current locale (except if they also define a set of custom units).

Adding a unit with a key which is already available in the CLDR, will overwrite the CLDR unit. This way you can overdefine single units, in case the CLDR provided formatting is not sufficient.

```
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   sap.ui.getCore().getConfiguration().getFormatSettings().addCustomUnits({
       "cats": {
          "displayName": "kitties",
          "unitPattern-count-one": "{0} kitty",
          "unitPattern-count-other": "{0} kitties"
       },
       "dogs": {
          "displayName": "puppies",
          "unitPattern-count-one": "{0} puppy",
          "unitPattern-count-other": "{0} puppies"
       },
       "power-horsepower": { // overwrite of an existing CLDR unit
          "displayName": "Horsepower",
          "unitPattern-count-one": "{0} AmazingHorse", // singular form
          "unitPattern-count-other": "{0} AmazingHorses" // plural form
       }
    });


    var oUnitFormat = NumberFormat.getUnitInstance({decimals:2, style:"long"});

    // formatting a custom unit
    console.log(oUnitFormat.format(12, "cats")); // 12,00 kitties


    // formatting and existing CLDR unit
    console.log(oUnitFormat.format(5, "speed-meter-per-second")); // 5,00 m/s


    // formatting and existing CLDR unit
    console.log(oUnitFormat.format(12, "power-horsepower")); // 12,00
AmazingHorses
});
```

Additionally, you can now define unit mappings, in order to use aliases for some units. A typical use-case is to map from an ISO unit code to a CLDR key. Of course you can also map to custom units as shown below.

```
sap.ui.getCore().getConfiguration().getFormatSettings().addUnitMappings({
   "kitties": "cats", // maps to a custom defined unit
   "mySpeedAlias": "speed-kilometer-per-hour" // maps to an existing the CLDR
unit
});
```

Additional Format Options

When using either instance, exclusive or globally configured custom units, you can also add two additional format options (`decimals` and `precision`) to the custom unit's definition block. In the following examples the `decimals` option is set.

> **Example**
>
> Globally configured custom units:
>
> ```
> sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
>    // define a new unit called Lux
>    sap.ui.getCore().getConfiguration().getFormatSettings().addCustomUnits({
>        "lux": {
>           "displayName": "Lux",
>           "unitPattern-count-one": "{0} lx",
>           "unitPattern-count-other": "{0} lx",
> ```

```
            "decimals": 2
        }
    });

    var oUnitFormat = NumberFormat.getUnitInstance({style:"long"});

    console.log(oUnitFormat.format(2.4, "lux")); // 2,40 lux (notice the
padded 0 after the 4, this is due to the decimals option)
});
```

Example

Instance exclusive custom unit definition:

```
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
    // define a new unit called Lux
    var oFormat = sap.ui.core.format.NumberFormat.getUnitInstance({
        customUnits: {
            "lux": {
                 "displayName": "Lux",
                 "unitPattern-count-one": "{0} lx",
                 "unitPattern-count-other": "{0} lx",
                 "decimals": 2
            }
        }
    });

    var oUnitFormat = NumberFormat.getUnitInstance({style:"long"});

    console.log(oUnitFormat.format(2.4, "lux")); // 2,40 lux (notice the
padded 0 after the 4, this is due to the decimals option)
});
```

Databinding: New Unit Type

Besides the NumberFormat instances, you now can also include the new Unit type in your application. Simply define it as the type for a property binding, and most of the formatting and parsing effort will be handled for you out of the box.

To demonstrate this, we can consider an example with electric meters. Typically they all measure the flow of energy in kilowatt hours (kWh). Yet some meters are more precise than others, some measure up to a few hundred wattseconds, others just cap it at full kilowatt hours. To simplify it for our example: the number of decimals might differ depending on the type of electric meter.

```
// defining a new custom Type as a subclass of the sap.ui.model.type.Unit type
sap.ui.require(["sap/ui/model/type/Unit", "sap/ui/core/format/NumberFormat"],
function(UnitType, NumberFormat) {

       UnitType.extend("sap.ui.core.samples.MeterType", {
              constructor: function(oFormatOptions, oConstraints){
                     // define the dynamic format options as the third argument
                     // 'aDynamicFormatOptionNames'
                     UnitType.apply(this, [oFormatOptions, oConstraints,
["decimals"]]);
              }
       });
});
```

In the example we defined a new `MeterType` to combine not only a number value and a unit, but the already mentioned optional dynamic format options in one single typed `PropertyBinding`.

```
<-- XML View snippet -->
<t:Table rows='energyModel>/meters'>
   ... <!-- here is more Table definition stuff, we cut this for simplicity -->


   <!-- the third part of the binding is the number of decimals for this meter
instance -->
   <m:Label text="{parts:['energyModel>value', 'energyModel>unit',
'energyModel>decimals'],type: 'sap.ui.core.samples.MeterType'}"/>


   ...
</t:Table>
```

With the new bindable dynamic format options of Unit type, you can pass the relevant meter formatting information in a generalized way through the cell's bindings.

The third argument of the base Unit type constructor is a list of dynamic format options. In our example, the binding context itself contains the information on how many-decimals should be used. Still, these dynamic format options can be bound to any value from any model.

> **i Note**
>
> If you use a combination of custom units on the Configuration and the Unit type, the format options from the type have priority.
>
> So if you define a `decimals` value for a custom unit in the Configuration, the bound values from the Unit type instance will still be taken for the formatting.

Related Information

API Reference: `sap.ui.core.format.NumberFormat`

Currency Formatting

Data formatting is one of the key features in SAPUI5 and enables applications to display data according to the user locale. For this, SAPUI5 uses the Common Locale Data Repository (CLDR), a third-party library that provides locale-specific patterns. SAPUI5 uses these patterns to adapt to the conventions of different languages.

One use case for data formatting is the ability to format and parse numbers including currency information. For this specific use case, the CLDR provides patterns with preconfigured currency information such as the number of decimals for a set of different currencies. It is also possible to define custom currencies by adding new custom currencies or reconfiguring existing currencies.

Available Functions for Currency Formatting

Use the `NumberFormat.getCurrencyInstance()` factory function for creating a currency format instance. On this instance, you can call the `format` and `parse` functions.

Creating a Currency Format Instance

The `sap.ui.core.format.NumberFormat.getCurrencyInstance()` function accepts two arguments:

- The first argument, `oFormatOptions`, is a set of format options that determines the output formatting. The format option `'decimals'`, for example, defines the number of decimal digits of the formatted value. For a full list of format options, see `NumberFormat`.
- The second argument, `oLocale`, defines the locale.

The following example shows how you create a simple currency formatter instance:

```
// create a simple currency formatting instance, without any additional options
sap.ui.require(["sap/ui/core/format/NumberFormat"], function(NumberFormat) {
   var oCurrencyFormat = NumberFormat.getCurrencyInstance();
});
```

> **i Note**
>
> All code examples are based on locale English. If no locale is defined explicitly on the currency formatter instance, the locale setting of the configuration is used.

For more information, see getCurrencyInstance.

`format` Function

The currency formatter instance allows you to combine a number value with a localized currency string.

```
// "NumberFormat" required from module "sap/ui/core/format/NumberFormat"
var oCurrencyFormat = NumberFormat.getCurrencyInstance();
oCurrencyFormat.format(12345.678, "EUR"); // output: EUR 12,345.68
```

For more information, see format

`parse` Function

The `'parse'` function turns a string containing a number and a currency code (EUR, USD) or symbol (€, $) back into its raw parts: the number value and the currency code. The results are returned in an array.

```
// "NumberFormat" required from module "sap/ui/core/format/NumberFormat"
var oCurrencyFormat = NumberFormat.getCurrencyInstance();
oCurrencyFormat.parse("EUR 12,345.678"); // output: [12345.678, "EUR"]
```

For more information, see parse

Formatting Options for Currency Formatting

The following formatting options for currency formatting are available:

- `currencyCode` defines whether the code or the symbol is used when `showMeasure` is set to true.
- `trailingCurrencyCode` defines whether the currency codes are always shown after the amount, independent of the locale.

- `currencyContext` defines the pattern that is used for formatting a currency number. It can be set to standard (default) or accounting.

Let's try out these format options and create a currency formatter that is able to format currency values with symbols:

```
// "NumberFormat" required from module "sap/ui/core/format/NumberFormat"
var oCurrencyFormat = NumberFormat.getCurrencyInstance({
    currencyCode: false
});
oCurrencyFormat.format(1234.567, "USD"); // returns $1,234.57
oCurrencyFormat.format(1234.567, "JPY"); // returns ¥1,235
oCurrencyFormat.parse("$1,234.57"); // returns [1234.57, "USD"]
oCurrencyFormat.parse("¥1,235"); // returns [1235, "JPY"]
```

Custom Currencies

As mentioned above, the Common Locale Data Repository (CLDR) provides patterns with preconfigured currency information according to the locale. It is possible, however, to add new custom currencies, or to reconfigure existing currencies. These custom currencies can be configured on currency format instances, or globally in the core configuration.

Custom Currencies Configuration on Currency Format Instances

The currency `NumberFormat` instance allows you to specify custom currencies which can be used for formatting and parsing. All you have to do is to add your custom currencies as an additional format option in the `sap.ui.core.format.NumberFormat.getCurrencyInstance()` factory.

The following example shows how this is done for a specific instance:

```
// "NumberFormat" required from module "sap/ui/core/format/NumberFormat"
var oCurrencyFormat = NumberFormat.getCurrencyInstance({
    customCurrencies: {
        "Bitcoin": {
            decimals: 5
         }
     }
});
oCurrencyFormat.format(10.1234567, "Bitcoin"); // 10,12346 Bitcoin
oCurrencyFormat.parse("12 Bitcoin"); // [12, "Bitcoin"];
```

If you want to define a custom currency that falls back on the currency symbol of an already existing currency, you can configure a respective currency code (also called ISO code):

```
// "NumberFormat" required from module "sap/ui/core/format/NumberFormat"
var oCurrencyFormat = NumberFormat.getCurrencyInstance({
    currencyCode: false,
    customCurrencies: {
        "MyEuro": {
            decimals: 5,
            isoCode: "EUR"
        }
    }
});
oCurrencyFormat.format(10.1234567, "MyEuro"); // €10.12346
```

The custom currencies defined on the `NumberFormat` instance are exclusive to this instance, meaning that no other instances are affected. In addition, once you define custom currencies for an instance, only those currencies are formatted and parsed by that instance.

This also means that custom currencies defined via the configuration are not taken into account for this specific instance. This is done to circumvent ambiguities and conflicts with the CLDR currencies. So, in the above example, only Bitcoin can be formatted, but not EUR.

In the following example, the currency instance from above is used. Formatting and parsing the currency 'Bitcoin' works fine, but the instance does not know about the currency 'EUR' because of the exclusivity of the custom currencies:

```
// formatting/parsing Bitcoin is fine
oCurrencyFormat.format(9001.987654, "Bitcoin");  // 9.001,98765 Bitcoin
oCurrencyFormat.parse("12 Bitcoin"); // [12, "Bitcoin"];
// formatting/parsing EUR does not work
oCurrencyFormat.format(1.21, "EUR"); // "": results in an empty string, as the
currency is unknown
```

If you need both, CLDR predefined currencies and custom currencies, you create two separate number format instances, or use the second approach to define custom currencies as described in the next section.

Global Custom Currencies Configuration in Core Configuration

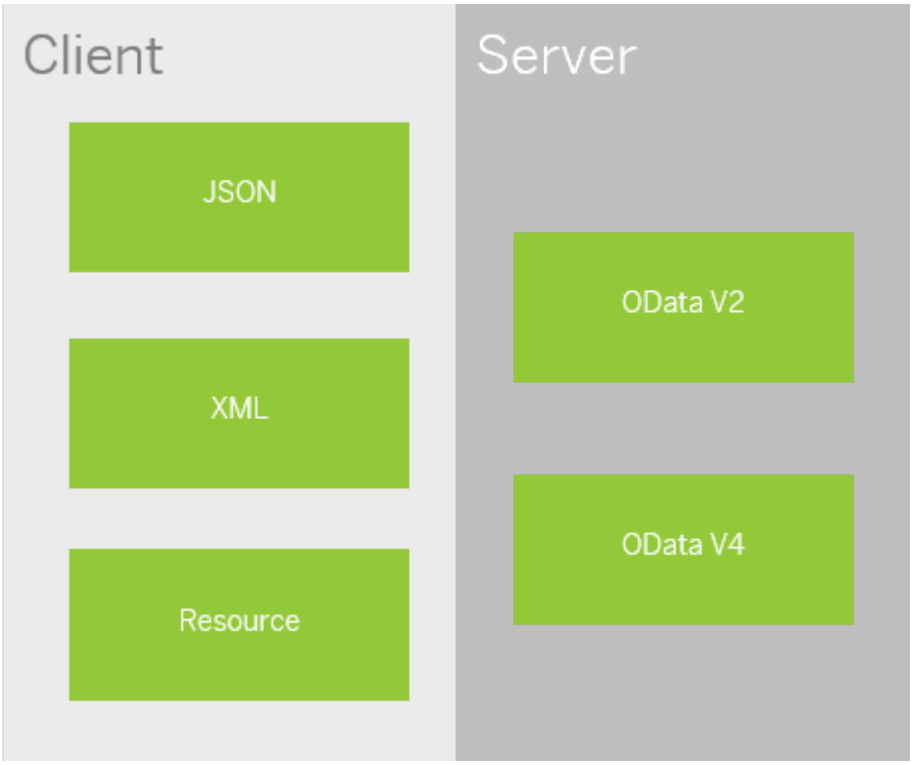
You can also add custom currencies via the formatting settings in the core configuration. Contrary to the custom currencies defined exclusively on a single currency-formatter instance, these custom currencies are available in all currency formatter instances for the current locale, except for if they also define a set of custom currencies as described in the previous section.

Adding a currency with a key which is already available in the CLDR will overwrite the CLDR currency. By this, you can overdefine single currencies, in case the CLDR provided formatting is not sufficient.

```
// "NumberFormat" required from module "sap/ui/core/format/NumberFormat"
sap.ui.getCore().getConfiguration().getFormatSettings().addCustomCurrencies({
         "MyCoin": {
            "symbol": "MC"
        },
        "Bitcoin": {
            "digits": 3
        },
        "USD": { // overwrite of an existing CLDR currency
            "digits": 5
        }
});
var oCurrencyFormat = NumberFormat.getCurrencyInstance();
// formatting a custom currency
oCurrencyFormat.format(12, "MyCoin"); // 12,00 MyCoin
// formatting an existing CLDR currency
oCurrencyFormat.format(5, "EUR"); // 5,00 EUR
// formatting an existing CLDR currency
oCurrencyFormat.format(12, "USD"); // 12,00000 USD // Default decimal setting
would have been two
```

Models

A model in the Model View Controller concept holds the data and provides methods to retrieve the data from the database and to set and update data.



- JSON Model [page 991]
- XML Model [page 993]
- Resource Model [page 995]
- OData V2 Model [page 883]
- OData V4 Model [page 918]

SAPUI5 provides the following predefined models:

- **OData model**: Enables binding of controls to data from OData services. The OData model supports two-way, one-way and one-time binding modes. However, two-way binding is currently only supported for properties, and not for aggregations.

  > **i** Note
  >
  > The OData model currently supports the following OData versions:
  >
  > - OData V2
  > - OData V4 (limited feature scope)

- **JSON model**: Can be used to bind controls to JavaScript object data, which is usually serialized in the JSON format. The JSON model is a client-side model and, therefore, intended for small data sets, which

are completely available on the client. The JSON model supports two-way (default), one-way and one-time binding modes.

- **XML model**: A client-side model intended for small data sets, which are completely available on the client. The XML model does not contain mechanisms for server-based paging or loading of deltas. The XML model supports two-way (default), one-way and one-time binding modes.
- **Resource model**: Designed to handle data in resource bundles, mainly to provide texts in different languages. The resource model only supports one-time binding mode because it deals with static texts only.

The JSON model, XML model, and the resource model are **client-side models**, meaning that the model data is loaded completely and is available on the client. Operations such as sorting and filtering are executed on the client without further server requests.

The OData (V2 or V4) model is a **server-side model** and only loads the data requested by the user interface from the server.

You can not only define one model for your applications, but define different areas in your application with different models and assign single controls to a model. You can also define nested models, for example, a JSON model defined for the application and an OData model for a table control contained in the application.

A Web application should support several data sources, such as JSON, XML, Atom, or OData. However, the way in which data binding is defined and implemented within the UI controls should be independent of the respective data source. It is also possible to create a custom model implementation for data sources that are not yet covered by the framework or are domain-specific.

Related Information

API Reference: `sap.ui.model`

OData V2 Model

The OData V2 Model enables binding of controls to data from OData services.

The OData model is a server-side model, meaning that the data set is only available on the server and the client only knows the currently visible (requested) data. Operations, such as sorting and filtering, are done on the server. The client sends a request to the server and shows the returned data.

> **i** Note
>
> Requests to the back end are triggered by list bindings (`ODataListBinding`), context bindings (`ODataContextBinding`), and CRUD functions provided by the `ODataModel`. Property bindings (`ODataPropertyBindings`) do not trigger requests.

The OData model currently supports OData version 2.0.

The following two versions of the OData model are implemented: `sap.ui.model.odata.ODataModel` and `sap.ui.model.odata.v2.ODataModel`. The `v2.ODataModel` has an improved feature set and new features will only be implemented in this model. `sap.ui.model.odata.ODataModel` is deprecated. We recommend to only use `v2.ODataModel`.

The following table shows the supported features for both OData models:

Feature	`sap.ui.model.odata.v2.ODataModel`	`sap.ui.model.odata.ODataModel`
OData version support	2.0	2.0
JSON format	Yes (default)	Yes
XML format	Yes	Yes (default)
Support of two-way binding mode	Yes; for property changes only, not yet implemented for aggregations	Experimental; only properties of one entity can be changed at the same time
Default binding mode	One-way binding	One-way binding
Client-side sorting and filtering	Yes For more information, see API Reference: sap.ui.model.odata.OperationMode.	No
$batch	Yes; all requests can be batched	Only manual batch requests are possible
Data cache in model	All data is cached in the model	Manually requested data is **not** cached
Automatic refresh	Yes (default)	Yes
Message handling	Yes, see Error, Warning, and Info Messages [page 1063]	No

> **i Note**
>
> Be aware of the Same-Origin-Policy security concept which prevents access to back ends on different domains or sites.

The requests to the service to fetch data are made automatically based on the data bindings that are defined for the controls.

Related Information

API Reference: `sap.ui.model.odata.v2.ODataModel`

Creating the Model Instance

One OData model instance can only cover one OData service. For accessing multiple services, you have to create multiple OData model instances.

The only mandatory parameter when creating an `ODataModel` instance is the service URL. It can be passed as first parameter or within the `mParameters` map to the constructor.

```
// "ODataModel" required from module "sap/ui/model/odata/v2/ODataModel"
var oModel = new ODataModel("http://services.odata.org/Northwind/
Northwind.svc/");
var oModel = new ODataModel({serviceUrl: "http://services.odata.org/Northwind/
Northwind.svc"});
```

When creating an `ODataModel` instance, a request is sent to retrieve the service metadata:

```
http://services.odata.org/Northwind/Northwind.svc/$metadata
```

Service Metadata

The service metadata is cached per service URL. Multiple OData models that are using the same service can share this metadata.

Only the first model instance triggers a `$metadata` request. A JSON representation of the service metadata can be accessed by calling the `getServiceMetadata()` method on an Odata model instance.

```
var oMetadata = oModel.getServiceMetadata();
```

> **i Note**
>
> In the `v2.ODataModel`, the service metadata is loaded asynchronously. It is not possible to load it synchronously. To get notified when the loading is finished, attach the `metadataLoaded` event.

Adding Additional URL Parameters

For OData services, you can use URL parameters for configuration. SAPUI5 sets most URL parameters automatically, according to the respective binding.

For authentication tokens or general configuration options, for example, you can add additional arguments to the request URL. Some of the parameters must not be included in every request, but should only be added to specific list or context bindings, such as `$expand` or `$select`. For this, the binding methods provide the option to pass a map of parameters, which are then included in all requests for this specific binding. The OData model currently only supports `$expand` and `$select`.

There are different ways to add URL parameters to the requests:

- Appending parameters to the service URL:

  ```
  // "ODataModel" required from module "sap/ui/model/odata/v2/ODataModel"
  ```

```
var oModel = new ODataModel("http://myserver/MyService.svc/?
myParam=value&myParam2=value");
```

These parameters will be included in every request sent to the OData server.

- Passing URL parameters with the `mparameters` map
  You can pass URL parameters that are used for `$metadata` requests only (`metadataUrlParams`) as well as URL parameters that are included only in data requests (`serviceUrlParams`). The parameters are passed as maps:

```
// "ODataModel" required from module "sap/ui/model/odata/v2/ODataModel"
var oModel = new ODataModel({
    serviceUrl: "http://services.odata.org/Northwind/Northwind.svc",
    serviceUrlParams: {
        myParam: "value1",
        myParam2: "value2"
    },
    metadataUrlParams: {
        myParam: "value1",
        myParam2: "value2"
    }
});
```

Custom HTTP Headers

You can add custom headers which are sent with each request.

To do this, provide a map of headers to the OData model constructor or use the `setHeaders()` function:

- Passing custom headers with the `mparameters` map

```
var oModel = new sap.ui.model.odata.v2.ODataModel({
    headers: {
        "myHeader1" : "value1",
        "myHeader2" : "value2"
    }
});
```

- Setting custom headers globally on a model instance

```
oModel.setHeaders({"myHeader1" : "value1", "myHeader2" : "value2"});
```

> **i Note**
>
> When you add custom headers, all previous custom headers are removed if not specified again in the headers map. Some headers are private, that is, they are set by the OData model internally and cannot be set:
>
> ```
> "accept"
> "accept-language"
> "maxdataserviceversion"
> "dataserviceversion"
> "x-csrf-token"
> ```
>
> For additional methods and parameters, see the API Reference: sap.ui.model.odata.v2.ODataModel.

Addressing Entities: Binding Path Syntax

The binding path syntax for OData models matches the URL path relative to the service URL used in OData to access specific entities or entity sets.

You access the data provided by the OData model according to the structure of the OData service as defined in the metadata of a service. URL parameters, such as filters, cannot be added to a binding path. A binding path can be absolute or relative. Absolute binding paths are resolved immediately. A relative path can only be resolved if it can be automatically converted into an absolute binding path. If, for example, a property is bound to a relative path and the parent control is then bound to an absolute path, the relative property path can be resolved to an absolute path.

The following binding samples within the `ODataModel` are taken from the Northwind demo service.

Absolute binding path (starting with a slash ('/')):

```
"/Customers"
"/Customers('ALFKI')/Address"
```

Relative binding paths that can be resolved with a context (for example `"/Customer('ALFKI')"`):

```
"CompanyName"
"Address"
"Orders"
```

Resolved to:

```
"/Customer('ALFKI')/CompanyName"
"/Customer('ALFKI')/Address"
"/Customer('ALFKI')/Orders"
```

Navigation properties, used to identify a single entity or a collection of entities:

```
"/Customers('ALFKI')/Orders"
"/Products(1)/Supplier"
```

For more information on addressing OData entries, see the URI conventions documentation on http://www.odata.org .

Accessing Data from an OData Model

The data requested from an OData service is cached in the OData model.

It can be accessed by the `getData()` and the `getProperty()` method, which returns the entity object or value. These methods do not request data from the backend, so you can only access already requested and cached entities:

```
oModel.getData("/Customer('ALFKI')");
oModel.getProperty("/Customer('ALFKI')/Address");
```

You can only access single entities and properties with these methods. To access entity sets, you can get the binding contexts of all read entities via a list binding. The values returned by these methods are copies of the data in the model, not references as in the JSONModel.

⚠ Caution

Do **not** modify objects or values inside the model manually; always use the provided API to change data in the model, or use two-way binding (see *Two-way Binding* section below).

ℹ Note

The ODataModel uses the `$skip` and `$top` URL parameters for paging. It is possible that data is modified between two paging requests, for eample, entities can be added orremoved and this may lead to data inconsistencies.

Creating Entities

To create entities for a specified entity set, call the `createEntry()` method. The method returns a context object that points to the newly created entity.

The application can bind against these objects and change the data by means of two-way binding. To store the entities in the OData backend, the application calls `submitChanges()`. To reset the changes, the application can call the `deleteCreatedEntry()` method.

The application can choose the properties that shall be included in the created object and can pass its own default values for these properties. Per default, all property values are empty, that is, undefined.

ℹ Note

The entity set and the passed properties must exist in the metadata definition of the OData service.

```
// create an entry of the Products collection with the specified properties
and values
var oContext = oModel.createEntry("/Products", { properties: { ID:99,
Name:"Product", Description:"new Product", ReleaseDate:new Date(),
Price:"10.1", Rating:1} });
// binding against this entity
oForm.setBindingContext(oContext);
// submit the changes (creates entity at the backend)
oModel.submitChanges({success: mySuccessHandler, error: myErrorHandler});
// delete the created entity
oModel.deleteCreatedEntry(oContext);
```

If created entities are submitted, the context is updated with the path returned from the creation request and the new data is imported into the model. So the context is still valid and points to the new created entity.

CRUD Operations

The OData model allows manual CRUD (create, read, update, delete) operations on the OData service. If a manual operation returns data, the data is imported into the data cache of the OData model. All operations require a mandatory `sPath` parameter as well as an optional `mParameters` map.

The `create` and `update` methods also require a mandatory `oData` parameter for passing the created or changed data object. Each operation returns an object containing a function abort, which can be used to abort

the request. If the request is aborted, the error handler is called. This ensures that the success or the error handler is executed for every request. It is also possible to pass additional header data, URL parameters, or an eTag.

- Creating entities
  The `create` function triggers a `POST` request to an OData service which was specified at creation of the OData model. The application has to specify the entity set, in which the new entity and the entity data is to be created.

```
var oData = {
    ProductId: 999,
    ProductName: "myProduct"
}
oModel.create("/Products", oData, {success: mySuccessHandler, error:
myErrorHandler});
```

- Reading entities
  The `read` function triggeres a `GET` request to a specified path. The path is retrieved from the OData service which was specified at creation of the OData model. The retrieved data is returned in the `success` callback handler function.

```
oModel.read("/Products(999)", {success: mySuccessHandler, error:
myErrorHandler});
```

- Updating entities
  The `update` function triggers a `PUT/MERGE` request to an OData service which was specified at creation of the OData model. After a successful request to update the bindings in the model, the refresh is triggered automatically.

```
var oData = {
    ProductId: 999,
    ProductName: "myProductUpdated"
}
oModel.update("/Products(999)", oData, {success: mySuccessHandler, error:
myErrorHandler});
```

- Deleting entities
  The `remove` function triggers a `DELETE` request to an OData service which was specified at creation of the OData model. The application has to specify the path to the entry which should be deleted.

```
oModel.remove("/Products(999)", {success: mySuccessHandler, error:
myErrorHandler});
```

- Refresh after change
  The model provides a mechanism to automatically refresh bindings that depend on changed entities. If you carry out a `create`, `update` or `remove` function, the model identifies the bindings and triggers a refresh for these bindings. If the model runs in batch mode, the refresh requests are bundled together with the changes in the same batch request. You can disable the auto refresh by calling `setRefreshAfterChange(false)`. If the auto refresh is disabled, the application has to take care of refreshing the respective bindings.

```
oModel.setRefreshAfterChange(false);
```

Concurrency Control and ETags

OData uses HTTP ETags for optimistic concurrency control. The service must be configured to provide them. The ETag can be passed within the parameters map for every CRUD request. If no ETag is passed, the ETag of the cached entity is used, if it is loaded already.

XSRF Token

To address cross-site request forgery, an OData service may require XSRF tokens for change requests by the client application. In this case, the client has to fetch a token from the server and send it with each change request to the server. The OData model fetches the XSRF token when reading the metadata and then automatically sends it with each write request header. If the token is no longer valid, a new token can be fetched by calling the `refreshSecurityToken` function on the OData model. The token is fetched with a request to the service root URL, which usually responds with the service document. To get a valid token, make sure that the response is **not** cached.

Refreshing the Model

The `refresh` function refreshes all data within an OData model. Each binding reloads its data from the server. For list or context bindings, a new request to the back end is triggered. If the XSRF token is no longer valid, it has to be fetched again with a `read` request to the service document. Data that has been imported via manual CRUD requests is **not** reloaded automatically.

Batch Processing

The `v2.ODataModel` supports batch processing (`$batch`) in two different ways:

- Default: All requests in a thread are collected and bundled in batch requests, meaning that request is sent in a timeout immediately after the current call stack is finished. This includes all manual CRUD requests as well as requests triggered by a binding.
- Deferred: The requests are stored and can be submitted with a manual `submitChanges()` call by the application. This also includes all manual CRUD requests as well as requests triggered by a binding.

The model cannot decide how to bundle the requests. For this, SAPUI5 provides the `groupId`. For each binding and each manual request, a `groupId` can be specified. All requests belonging to the same group are bundled into one batch request. Request without a `groupId` are bundled in the default batch group. You can use a `changeSetId` for changes. The same principle applies: Each change belonging to the same `changeSetId` is bundled into one `changeSet` in the batch request. Per default, all changes have their own `changeSet`.

You can use the `setDeferredGroups()` method to set a subset of previously defined groups to deferred.

> **i Note**
>
> The same is also valid for `setChangeGroups()` and `getChangeGroups()`.

All requests belonging to the `group` are then stored in a request queue. The deferred batch group must then be submitted manually by means of the `submitChanges()` method. If you do **not** specify a batch group ID when calling `submitChanges`, all deferred batch groups are submitted.

> **Example**
>
> Set a subset of groups to deferred:
>
> ```
> // "ODataModel" required from module "sap/ui/model/odata/v2/ODataModel"
> var oModel = new ODataModel(myServiceUrl);
> ```
>
> Pass the `groupId` to a binding:
>
> ```
> {path:"/myEntities", parameters: {groupId: "myId"}}
> ```
>
> Set the `groupId` to deferred:
>
> 1. Get the list of deferred groups:
>
>    ```
>    var aDeferredGroups = oModel.getDeferredGroups();
>    ```
> 2. Append your `groupId` to the list:
>
>    ```
>    aDeferredGroups=aDeferredGroups.concat(["myId"]);
>    ```
> 3. Set all groups to deferred:
>
>    ```
>    oModel.setDeferredGroups(aDeferredGroups);
>    ```
>
> Submit all deferred groups:
>
> ```
> oModel.submitChanges({success: mySuccessHandler, error: myErrorHandler});
> ```

Two-Way Binding

The `v2.ODataModel` enables two-way binding. Per default, all changes are collected in a batch group called "changes" which is set to deferred.

To submit the changes, use `submitChanges()`. The data changes are made on a data copy. This enables you to reset the changes without sending a new request to the backend to fetch the old data again. With `resetChanges()` you can reset all changes. You can also reset only specific entities by calling `resetChanges` with an array of entity paths.

> **i Note**
>
> Filtering and sorting is not possible if two-way changes are present as this would cause inconsistent data on the UI. Therefore, before you carry out sorting or filtering, you have to submit or reset the changes.

You can collect the changes for different entities or types in different batch groups. To configure this, use the `setChangeGroups()` method of the model:

```
// "ODataModel" required from module "sap/ui/model/odata/v2/ODataModel"
var oModel = new ODataModel(myServiceUrl);
oModel.setDeferredGroups(["myGroupId", "myGroupId2"]);
oModel.setChangeGroups({
    "EntityTypeName": {
        groupId: "myGroupId",
        [changeSetId: "ID",]
        [single: true/false,]
    }
});
oModel.submitChanges({groupId: "myGroupId", success: mySuccessHandler, error:
myErrorHandler});
```

To collect the changes for all entity types in the same batch group, use '*' as `EntityType`. If the change is not set to deferred, the changes are sent to the backend immediately. By setting the `single` parameter for `changeSet` to true or false, you define if each change results in its own change set (`true`) or if all changes are collected in one change set (`false`). The model only takes care of the `changeSetId` if `single` is set to `false`.

> **i Note**
>
> The first change of an entity defines the order in the change set.

Example

Reset changes:

```
// "ODataModel" required from module "sap/ui/model/odata/v2/ODataModel"
var oModel = new ODataModel(myServiceUrl);
//do a change
oModel.setProperty("/myEntity(0)", oValue);
//reset the change
oModel.resetChanges(["/myEntity(0)"]);
```

Binding-specific Parameters

The OData protocol specifies different URL parameters.

You can use these parameters in bindings in addition to the parameters described above:

- Expand parameter
  The `expand` parameter allows the application to read associated entities with their navigation properties:

  ```
  oControl.bindElement("/Category(1)", {expand: "Products"});
  oTable.bindRows({
      path: "/Products",
      parameters: {expand: "Category"}
  });
  ```

  In this example, all products of `"Category(1)"` are embedded inline in the server response and loaded in one request. The category for all `"Products"` is embedded inline in the response for each product.
- Select parameter

The `select` parameter allows the application to define a subset of properties that is read when requesting an entity.

```
oControl.bindElement("/Category(1)", {expand: "Products", select:
"Name,ID,Products"});
oTable.bindRows({
    path: "/Products",
    parameters: {select: "Name,Category"}
});
```

In this example, the properties `Name`, `ID` and `ofCategory(1)` as well as all properties of the embedded products are returned. The properties `Name` and `Category` are included for each product. The `Category` property contains a link to the related category entry.

- Custom query options
  You can use custom query options as input parameters for service operations. When creating the list binding, specify these custom parameter as follows:

```
oTable.bindRows({
        path: "/Products",
        parameters: {
        custom: {
            param1: "value1",
            param2: "value2"
        }
    },
    template: rowTemplate
});
```

If you use `bindElement`, you can specify custom parameters as follows:

```
oTextField.bindElement("/GetProducts", {
    custom: {
        "price" : "500"
    }
});
```

Optimizing Dependent Bindings

The ODataModel V2 supports a flag called "preliminaryContext". With this option set to true, the ODataModel is able to bundle the OData calls for dependent bindings together into fewer $batch requests.

Introduction

Two bindings are considered "dependent" if one cannot be resolved without the other being resolved first, for example a relative binding cannot be resolved without a resolved absolute binding.

If the `preliminaryContext` option is set to `false`, each binding will be resolved once its preceding binding has been resolved or if it is an absolute binding itself.

The `preliminaryContext` option can also be activated/deactivated per binding instance. This overwrites the default value set on the ODataModel instance.

Settings and Usage

ODataModel v2

The constructor parameter is named `preliminaryContext` (type Boolean) and has the following properties:

- Default value is `false`.
- It is used by the `ContextBinding` as a default value for `createPreliminaryContext` if not given in the constructor. For examples on its usage see "ContextBinding".
- It is used by the `ContextBinding` as a default value for `usePreliminaryContext` if not given in the constructor. For examples on its usage see "ContextBinding"..

ODataListBinding v2

The constructor parameter is named `usePreliminaryContext` (type Boolean) and has the following properties:

- Default value is `false`, as it is derived from the ODataModel's default.
- If set to true:
  - The `ODataListBinding` accepts preliminary contexts (for example. in a `setContext()` call).
  - The `ODataListBinding` fires a change event with `ChangeReason.Context`, if the binding is updated and a preliminary context was set.

ODataContextBinding

The `ODataContextBinding` supports two different parameters:

- `usePreliminaryContext` (same as a ODataListBinding v2)
- `createPreliminaryContext`
  - If the binding cannot be resolved, it still creates a preliminary binding context, which can be used by other subordinate dependent bindings, which have set the `usePreliminaryContext` option to `true`.
  - A change event with `ChangeReason.Context` is fired once the data is loaded for the currently preliminary `Context` instance. Afterwards, the existing `Context` instance is not considered "preliminary" anymore.

Relationship Between Binding and Model Settings

Default Behavior

To describe the preliminary context feature in more detail, we first have to look at the default Model/Binding behavior. Let's look at the simple example in the following graphic.

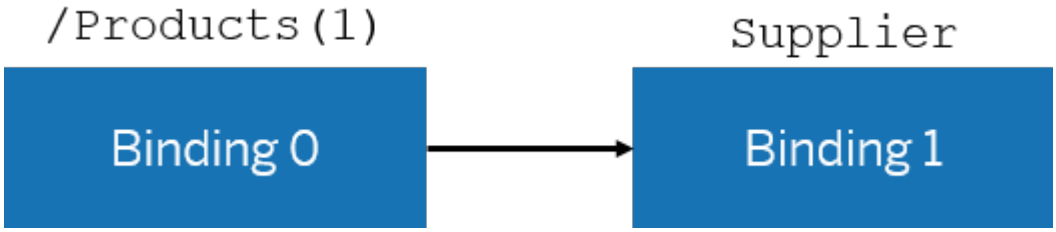


Figure 212: Simple Binding Example

Without using preliminary contexts, `Binding 1` resolves only after `Binding 0` is resolved.

For example, if `Binding 1` is a relative `ODataListBinding` on a `Table` control, its OData request will only be sent, once the data for the absolute `Binding 0` is available, for example by using an Element binding on a `Panel` control.

This leads to two subsequent OData requests, one for `Binding 0` and afterwards one for `Binding 1`, as shown in the following table:

Table 27: Simple Example: Binding Resolution

Request Number	Content
1	`GET Products(1)`
2	`GET Products(1)/Supplier`

Now let's look at a more complex example.

Figure 213: Complex Binding Example

In this example we add another binding, which will be resolved once `Binding 0` and `Binding 1` are resolved. This leads to the following three individual `$batch` requests:

Table 28: Complex Example: Binding Resolution

Request Number	Content
1	`GET Products(1)`
2	`GET Products(1)/Supplier`
3	`GET Suppliers(1)/Products`

Optimized Behavior

Let's look at the same simple example but with some optimizations.

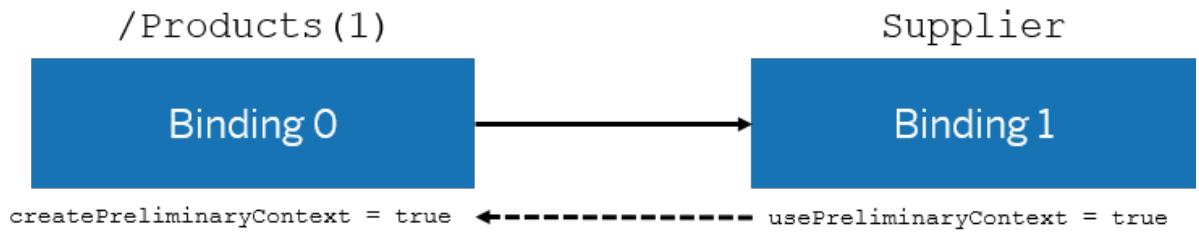


Figure 214: Simple Binding Example - Optimized

Here `Binding 1` uses the preliminary context created by `Binding 0`, and thus the request URL can directly be resolved.

This now leads to only a single `$batch` request:

Table 29: Simple Example: Binding Resolution Optimized

Request Number	Content
1	`GET Products(1)` `GET Products(1)/Supplier`

In this example `Binding 1` has set its `usePreliminaryContext` flag to `true`, and thus accepts preliminary contexts to be set.

> **i Note**
>
> If either `createPreliminaryContext` or `usePreliminaryContext` is set to `false`, the default behavior is active.

Now let's see how this works in the complex example.

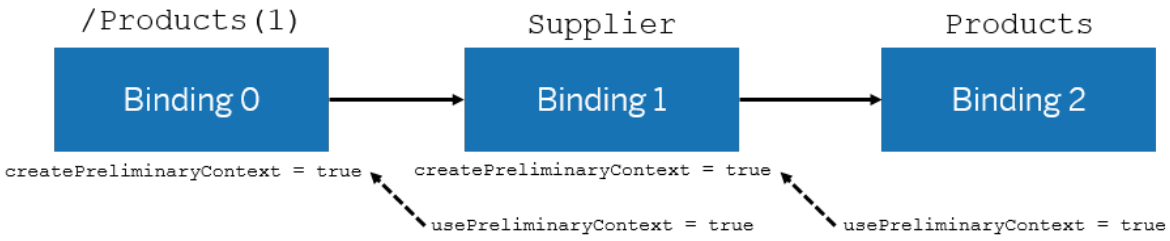


Figure 215: Complex Binding Example - Optimized

In this example we added another binding to the scenario. `Binding 2` is again a relative binding, which can only resolve once `Binding 1` is resolved. `Binding 1` behaves just as before.

In this case the single, generated request looks like this:

Table 30: Complex Example: Binding Resolution Optimized

Request Number	Content

1	`GET Products(1)`
	`GET Products(1)/Supplier`
	`GET Products(1)/Supplier/Products`

Results and Conclusion

Notice how the `Products` list of the `Supplier` is referenced through the entity `Products(1)`.This is a result of bundling all data requests into one single `$batch` request, without waiting for the `Products(1)` entity and its associated `Supplier` entity to be loaded.

As opposed to the default behavior, we do not require to have the `Products(1)` and `Supplier` entities loaded before sending the data request for the Supplier's `Products.Supplier`.So in this case we use a data path based on `Products(1)` and not the `ID` of the Supplier. You can compare that to the default behavior of the complex example described above.

Example

What would happen if one binding in the above chain does not set the `usePreliminaryContext` or the `createPreliminaryContext` option to true?

For example, if `Binding 2` sets its `usePreliminaryContext` option to `false`, the resolution chain is broken and we have a mixed scenario. Here one part is loaded optimized in one `$batch`, and the second part is loaded in a separate `$batch`:

Table 31: Complex Example: Binding Resolution Optimized

Request Number	Content
1	`GET Products(1)` `GET Products(1)/Supplier`
2	`GET Products(1)/Supplier/Products`

Note

With the `$expand` query option you can load all associated entities of another entity. In the previous examples we requested the `Product` list of a certain `Supplier` via a separate request. When using a `$expand` query instead, you could request the same information within one single request:

```
GET Products(1)?$expand=Supplier/Products
```

Even though you now also achieved to have less requests, using `$expand` has a couple of drawbacks. These can be circumvented by using the preliminary context feature, which does not have these limitations.

In OData V2, with a `$expand` you cannot use additional filters and sorters for the expanded entries. In addition, the `$expand` option always loads **ALL** associated entities, so paging with `$skip` or `$top` is also

not possible. Using the preliminary context feature, you get multiple sub-requests in a single `$batch`, yet you can easily include additional filters and sorters on the related subordinate entries.

Function Import

The `ODataModel` supports the invoking of function imports or actions by the `callFunction` method.

```
oModel.callFunction("/GetProductsByRating",{method:"GET", urlParameters:
{"rating":3}, success:fnSuccess, error: fnError})
```

If the `callFunction` request is deferred, it can be submitted via the `submitChangesmethod`.

> **i** Note
>
> Only "IN" parameters of function imports are currently supported.

Binding of Function Import Parameters

OData Model V2 supports the binding against function import parameters. This is similar to the `createEntry` method which supports binding against entity properties. The `callFunction` method returns a request handle that has a `promise`. This `promise` is resolved when the context to which it is bound is created successfully or is rejected if not:

```
var oHandle = oModel.callFunction("/GetProductsByRating", {urlParameters:
{rating:3}});
oHandle.contextCreated().then(function(oContext) {
      oView.setBindingContext(oContext);
});
```

If the function import returns result data, then the result data can be accessed and bound against in the `$result` property using the context:

```
<form:SimpleForm>
   <core:Title text="Parameters" />
   <Label text="Rating" />
   <Input value="{rating}" />
   <Button text="Submit" press=".submit" />
   <core:Title text="Result" />
   <List items="{$result}">
    <StandardListItem title="{Name}" />
   </List>
</form:SimpleForm>
```

Language

SAPUI5 uses the concept of a "current language" (see Identifying the Language Code / Locale [page 1269]). This language is automatically propagated to the OData service by the OData V2 model. For this reason,

applications must not hard code the language themselves, e.g. they must not specify the "sap-language" URL parameter as a custom query option.

Meta Model for OData V2

The implementation sap.ui.model.odata.ODataMetaModel offers a unified access to both OData Version 2.0 metadata and Version 4.0 annotations.

It uses the existing sap.ui.model.odata.ODataMetadata as a foundation and merges the OData Version 4.0 annotations from the existing sap.ui.model.odata.ODataAnnotations directly into the corresponding entity or property.

You can get an instance of sap.ui.model.odata.ODataMetaModel from an instance of sap.ui.model.odata.v2.ODataModel, see XML Templating [page 1018].

Basic Structure

The basic structure of sap.ui.model.odata.ODataMetadata is shown in the following code snippet. It shows you how the most important elements of the entity model are nested. Each of these elements (except *association set end*) can have extensions, that is, XML attribute values from some namespace. The code snippets below show how these extensions are stored and processed.

```
"dataServices": {
    "schema": [{
      "association": [{
        "end": []
      }],
      "complexType": [{
        "property": []
      }],
      "entityContainer": [{
        "associationSet": [{
          "end": []
        }],
        "entitySet": [],
        "functionImport": [{
          "parameter": []
        }]
      }],
      "entityType": [{
        "property": [],
        "navigationProperty": []
      }]
    }]
  }
}
```

The following code snippet gives a closer look and has more properties:

```
{
  "version": "1.0",
  "dataServices": {
    "dataServiceVersion": "2.0",
    "schema": [{
```

```
      "namespace": "GWSAMPLE_BASIC",
      "entityType": [{
        "name": "BusinessPartner",
        "key": {
          "propertyRef": [{
            "name": "BusinessPartnerID"
          }]
        },
        "property": [{
          "name": "BusinessPartnerID",
          "type": "Edm.String",
          "nullable": "false",
          "maxLength": "10"
        }],
        "navigationProperty": [{
          "name": "ToSalesOrders",
          "relationship": "GWSAMPLE_BASIC.Assoc_BusinessPartner_SalesOrders",
          "fromRole": "FromRole_Assoc_BusinessPartner_SalesOrders",
          "toRole": "ToRole_Assoc_BusinessPartner_SalesOrders"
        }]
      }],
      "complexType": [{
        "name": "CT_Address",
        "property": [{
          "name": "City",
          "type": "Edm.String",
          "maxLength": "40"
        }]
      }],
      "association": [{
        "name": "Assoc_BusinessPartner_SalesOrders",
        "end": [{
          "type": "GWSAMPLE_BASIC.BusinessPartner",
          "multiplicity": "1",
          "role": "FromRole_Assoc_BusinessPartner_SalesOrders"
        }, {
          "type": "GWSAMPLE_BASIC.SalesOrder",
          "multiplicity": "*",
          "role": "ToRole_Assoc_BusinessPartner_SalesOrders"
        }],
        "referentialConstraint": {
          "principal": {
            "role": "FromRole_Assoc_BusinessPartner_SalesOrders",
            "propertyRef": [{
              "name": "BusinessPartnerID"
            }]
          },
          "dependent": {
            "role": "ToRole_Assoc_BusinessPartner_SalesOrders",
            "propertyRef": [{
              "name": "CustomerID"
            }]
          }
        }
      }],
      "entityContainer": [{
        "name": "GWSAMPLE_BASIC_Entities",
        "isDefaultEntityContainer": "true",
        "entitySet": [{
          "name": "BusinessPartnerSet",
          "entityType": "GWSAMPLE_BASIC.BusinessPartner"
        }],
        "associationSet": [{
          "name": "Assoc_BusinessPartner_SalesOrders_AssocS",
          "association": "GWSAMPLE_BASIC.Assoc_BusinessPartner_SalesOrders",
          "end": [{
            "entitySet": "BusinessPartnerSet",
            "role": "FromRole_Assoc_BusinessPartner_SalesOrders"
```

```
          }, {
            "entitySet": "SalesOrderSet",
            "role": "ToRole_Assoc_BusinessPartner_SalesOrders"
          }]
        }],
        "functionImport": [{
          "name": "SalesOrder_Confirm",
          "returnType": "GWSAMPLE_BASIC.SalesOrder",
          "entitySet": "SalesOrderSet",
          "httpMethod": "POST",
          "parameter": [{
            "name": "SalesOrderID",
            "type": "Edm.String",
            "mode": "In",
            "maxLength": "10"
          }]
        }]
      }]
    }]
  }}
```

Accessing Objects and Properties

The objects in the OData meta model are arranged in arrays. `/dataServices/schema`, for example, is an array of schemas where each schema has an `entityType` property with an array of entity types, and so on. So, `/dataServices/schema/0/entityType/16` can be the path to the entity type with name "Order" in the schema with namespace "MySchema".

However, these paths are not stable: If an entity type with lower index is removed from the schema, the path to "Order" changes to `/dataServices/schema/0/entityType/15`. To avoid problems with changing indexes, `getObject` and `getProperty` support XPath-like queries for the indexes. Each index can be replaced by a query in square brackets. You can, for example, address the schema by using the path `/dataServices/schema/[${namespace}==='MySchema']` or address the entity by using the path `/dataServices/schema/[${namespace}==='MySchema']/entityType/[${name}==='Order']`.

The syntax inside the square brackets corresponds to the expression binding syntax. The query is executed for each object in the array until the result is true (truthy) for the first time. This object is then chosen. To embed such a path into an expression binding, use a complex binding syntax: `${path:'...'}`. Example: `{:= ${path:'target>extensions/[${name} === \'semantics\']/value'} === 'email'}`

Each of these queries is self-contained. The query can refer to properties of the current candidate via a relative path, for example `${name}`, but it **cannot** refer to variables such as `${meta>}` that are available in XML templating at that point.

Extensions

`extensions` array and transformed from objects into simple properties with an `sap:` prefix added to their name, see line number 8 in the following code snippet.

> **i Note**
>
> As this happens in addition, the following example shows both representations. By this, the respective annotations can be addressed via a simple relative path instead of searching an array.

```
{
  "name": "BusinessPartnerID",
  "extensions": [{
    "name": "label",
    "value": "Bus. Part. ID",
    "namespace": "http://www.sap.com/Protocols/SAPData"
  }],
  "sap:label": "Bus. Part. ID"
}
```

OData v4 Annotations

Each element of the entity model (except *association set end*) can be annotated. These annotations from the existing `sap.ui.model.odata.ODataAnnotations` are merged directly into the corresponding element. The following code snippet shows how the structure from the existing `sap.ui.model.odata.ODataMetadata`, as explained above and including extensions and constraints such as `nullable` or `maxLength`, is fleshed out with lifted v2 annotations and inlined v4 annotations, such as `Org.OData.Measures.V1.Unit` or `com.sap.vocabularies.UI.v1.Identification`. If you want to navigate the structure, for example for XML templating, it is important to understand this structure.

ODataMetaModel JSON Format:

```
"dataServices" : {
    "schema" : [{
        "namespace" : "GWSAMPLE_BASIC",
        "entityType" : [{
            "name" : "Product",
            "property" : [{
                "name" : "ProductID",
                "type" : "Edm.String",
                "nullable" : "false",
                "maxLength" : "10"
            }, {
                "name" : "SupplierName",
                "type" : "Edm.String",
                "maxLength" : "80",
                "extensions" : [{
                  "name" : "label",
                  "value" : "Company Name",
                  "namespace" : "http://www.sap.com/Protocols/SAPData"
                }, {
                  "name" : "creatable",
                  "value" : "false",
                  "namespace" : "http://www.sap.com/Protocols/SAPData"
                }, {
                  "name" : "updatable",
                  "value" : "false",
                  "namespace" : "http://www.sap.com/Protocols/SAPData"
                }],
                "sap:label" : "Company Name",
                "sap:creatable" : "false",
                "sap:updatable" : "false"
                "Org.OData.Core.V1.Computed" : {
```

```
                    "Bool" : "true"
                }
            }, {
                "name" : "WeightMeasure",
                "type" : "Edm.Decimal",
                "precision" : "13",
                "scale" : "3",
                "Org.OData.Measures.V1.Unit" : {
                    "Path" : "WeightUnit"
                }
            }, {
                "name" : "WeightUnit",
                "type" : "Edm.String",
                "maxLength" : "3"
            }],
            "com.sap.vocabularies.UI.v1.DataPoint" : {
                "Value" : {
                    "Path" : "WeightMeasure",
                    "EdmType" : "Edm.Decimal"
                }
            },
            "com.sap.vocabularies.UI.v1.Identification" : [{
                "Value" : {"Path" : "ProductID"}
            }, {
                "Value" : {"Path" : "SupplierName"}
            }, {
                "Value" : {"Path" : "WeightMeasure"}
            }]
        }]
    }]
}
```

Enhancement of the OData Meta Model

In addition to the easy access to the SAP-specific OData annotations, such as `sap:label`, corresponding vocabulary-based annotations are mixed in if they are not yet defined in the OData Version 4.0 annotations of the existing `sap.ui.model.odata.ODataAnnotations`.

> **i Note**
>
> Annotation terms are not merged, but replaced as a whole ("PUT" semantics). If the same annotation term with the same target is also contained in an annotation file, the complete OData V4 annotation converted from the OData V2 annotation is replaced by the one contained in the annotation file for the specified target. Converted annotations never use a qualifier and are only overwritten by the same annotation term without a qualifier.

The following tables show the transformations that are implemented with version 1.30 of SAPUI5 (variatons of this are marked accordingly). In the examples shown below, `AnyPath` is a path expression as defined in the OData Version 4.0 specification , section 14.5.12.

Transformations defined at `EntitySet`:

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:creatable = "false"`	`"Org.OData.Capabilities.V1.InsertRestrictions": { "Insertable" : { "Bool" : "false" } }`
`sap:deletable = "false"`	`"Org.OData.Capabilities.V1.DeleteRestrictions": { "Deletable" : { "Bool" : "false" } }` **i Note** If both, `sap:deletable` and `sap:deletable-path` are given, the service is broken and it is handled as `sap:deletable="false"`.
`sap:deletable-path = "AnyPath"` Where `AnyPath` is a path expression that identifies a Boolean property in the context of the entity type of the entity set. The value of this property indicates whether the entity can be deleted or not.	`"Org.OData.Capabilities.V1.DeleteRestrictions": { "Deletable" : { "Path" : "AnyPath" } }` **i Note** If both, `sap:deletable` and `sap:deletable-path` are given, the service is broken and it is handled as `sap:deletable="false"`.
`sap:label = "foo"` Where `foo` is any text.	`"com.sap.vocabularies.Common.v1.Label" : {"String" : "foo" }`
`sap:pageable = "false"`	`"Org.OData.Capabilities.V1.SkipSupported": {"Bool" : "false" }, "Org.OData.Capabilities.V1.TopSupported": {"Bool" : "false" }`
`sap:requires-filter = "true"`	`"Org.OData.Capabilities.V1.FilterRestrictions": { "RequiresFilter" : { "Bool" : "true" } }`
`sap:searchable = "false"` Alternatively, do not use the `sap:searchable` annotation.	`"Org.OData.Capabilities.V1.SearchRestrictions": { "Searchable" : { "Bool" : "false" } }`
`sap:topable = "false"`	`"Org.OData.Capabilities.V1.TopSupported": {"Bool" : "false" }`

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:updatable = "false"`	`"Org.OData.Capabilities.V1.UpdateRestrictions": { "Updatable" : { "Bool" : "false" } }` **Note** If both, `sap:updatable` and `sap:updatable-path` are given, the service is broken and it is handled as `sap:updatable="false"`.
`sap:updatable-path = "AnyPath"` Where `AnyPath` is a path expression that identifies a Boolean property in the context of the entity type of the entity set. The value of this property indicates whether the entity can be updated or not.	`"Org.OData.Capabilities.V1.UpdateRestrictions": { "Updatable" : { "Path" : "AnyPath" } }` **Note** If both, `sap:updatable` and `sap:updatable-path` are given, the service is broken and it is handled as `sap:updatable="false"`.

Transformations defined at `Property`:

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:label = "foo"` Where `foo` is any text.	`"com.sap.vocabularies.Common.v1.Label" : {"String" : "foo" }` **Note** The resulting annotation is added at different places, not to the `Property`.

`sap:creatable = "true"`
 	
and
 	
`sap:updatable = "false"`
	`"Org.OData.Core.V1.Immutable": { "Bool" : "true" }`

`sap:creatable = "false"`
 	
and
 	
`sap:updatable = "false"`
	`"Org.OData.Core.V1.Computed": { "Bool" : "true"}`

`sap:display-format = "NonNegative"`	`"com.sap.vocabularies.Common.v1.IsDigitSequence": { "Bool" : "true" }`

OData V2 SAP Extension	Resulting OData V4 Annotation
i Note `NonNegative` indicates that only non-negative numeric values are provided and persisted, other input leads to errors; intended for `Edm.String` fields that are internally stored as `NUMC`.	
`sap:display-format = "UpperCase"`	`"com.sap.vocabularies.Common.v1.IsUpperCase": { "Bool" : "true" }`
`sap:field-control = "AnyPath"` Where `AnyPath` is a path expression that identifies a property containing a numeric value that controls visibility..	`"com.sap.vocabularies.Common.v1.FieldControl": { "Path" : "AnyPath" }`
`sap:filterable = "false"`	`"Org.OData.Capabilities.V1.FilterRestrictions": { "NonFilterableProperties" : [ { "PropertyPath" : "PropA " }, { "PropertyPath" : "PropC " }] }` For example, if `sap:filterable` is set to `false` for properties `PropA` and `PropC`. **i Note** The resulting annotation is added to the `EntitySet`, not to the `Property`.
`sap:filter-restriction="multi-value"` For example, at a `BusinessPartnerID` property of a `BusinessPartner` type.	`"com.sap.vocabularies.Common.v1.FilterExpressionRestrictions": [{ "Property" : { "PropertyPath" : "BusinessPartnerID" }, "AllowedExpressions" : { "EnumMember": "com.sap.vocabularies.Common.v1.FilterExpressionType/MultiValue" } }]` At the corresponding entity set, for example, `BusinessPartnerSet.multi-value` is mapped to `MultiValue`, `single-value` is mapped to `SingleValue`, and `interval` is mapped to `SingleInterval`. **i Note** The resulting annotation is added to the `EntitySet`, not to the `Property`.

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:heading = "foo"` Where `foo` is any text.	`"com.sap.vocabularies.Common.v1.Heading": { "String" : "foo" }`
`sap:precision = "AnyPath"` Where `AnyPath` is a path expression that identifies a property in the context of the entity type containing the number of significant decimal places for a numeric value.	`"Org.OData.Measures.V1.Scale": { "Path" : "AnyPath" }`
`sap:quickinfo = "foo"` Where `foo` is any text.	`"com.sap.vocabularies.Common.v1.QuickInfo": { "String" : "foo" }`
`sap:required-in-filter = "true"`	If `sap:required-in-filter` is set to `TRUE` for the `PropA` and `PropC` properties: `"Org.OData.Capabilities.V1.FilterRestrictions": { "RequiredProperties" : [ { "PropertyPath" : "PropA " }, { "PropertyPath" : "PropC " }] }` **i Note** The resulting annotation is added to the `EntitySet`, not to the `Property`.
`sap:sortable = "false"`	If `sap:sortable` is set to `FALSE` for the `PropA` and `PropC` properties: `"Org.OData.Capabilities.V1.SortRestrictions": { "NonSortableProperties" : [ { "PropertyPath" : "PropA " }, { "PropertyPath" : "PropC " }]}` **i Note** The resulting annotation is added to the `EntitySet`, not to the `Property`.
`sap:text = "AnyPath"` Where `AnyPath` is a path expression that identifies a property in the context of the entity type containing a human-readable text for the value of this property.	`"com.sap.vocabularies.Common.v1.Text": { "Path" : "AnyPath" }`

OData V2 SAP Extension	Resulting OData V4 Annotation

`sap:unit="WeightUnit"`
 	
or
 	
`sap:unit="CurrencyCode"`
 	
Where `WeightUnit` and `CurrencyCode` are names of properties in the same entity and `WeightUnit` points to a property with `sap-semantics:unit-of-measure` and `CurrencyCode`points to a property with `sap-semantics:currency-code`.
	`"Org.OData.Measures.V1.Unit": { "Path" : "WeightUnit" }`
	
	or
	
	`"Org.OData.Measures.V1.ISOCurrency": { "Path" : "CurrencyCode" }`

`sap:visible="false"`	`"com.sap.vocabularies.UI.v1.Hidden" : { "Bool" : "true" }`
	
	Deprecated as of SAPUI5 1.44:
	
	`"com.sap.vocabularies.Common.v1.FieldControl": { "EnumMember" : "com.sap.vocabularies.Common.v1.FieldControlType/Hidden" }`

`sap:aggregation-role="dimension"`	`"com.sap.vocabularies.Analytics.v1.Dimension" : { "Bool" : "true" }` **i Note** Implemented with version 1.46.
`sap:aggregation-role="measure"`	`"com.sap.vocabularies.Analytics.v1.Measure" : { "Bool" : "true" }` **i Note** Implemented with version 1.46.
`sap:semantics="year"`	`"com.sap.vocabularies.Common.v1.IsCalendarYear" : {"Bool" : "true"}` **i Note** Implemented with version 1.50.
`sap:semantics="yearmonth"`	`"com.sap.vocabularies.Common.v1.IsCalendarYearMonth" : {"Bool" : "true"}`

OData V2 SAP Extension	Resulting OData V4 Annotation
	i Note Implemented with version 1.50.
`sap:semantics="yearmonthday"`	`"com.sap.vocabularies.Common.v1.IsCalendarDate" : {"Bool" : "true"}` **i Note** Implemented with version 1.50.
`sap:semantics = url`	`"Org.OData.Core.V1.IsURL" : { "Bool" : "true" }` **i Note** Implemented with version 1.52.
`sap:semantics="yearquarter"`	`"com.sap.vocabularies.Common.v1.IsCalendarYearQuarter" : {"Bool" : "true"}` **i Note** Implemented with version 1.54.
`sap:semantics="yearweek"`	`"com.sap.vocabularies.Common.v1.IsCalendarYearWeek" : {"Bool" : "true"}` **i Note** Implemented with version 1.54.
`sap:semantics="fiscalyear"`	`"com.sap.vocabularies.Common.v1.IsFiscalYear" : {"Bool" : "true"}` **i Note** Implemented with version 1.54.
`sap:semantics="fiscalyearperiod"`	`"com.sap.vocabularies.Common.v1.IsFiscalYearPeriod" : {"Bool" : "true"}` **i Note** Implemented with version 1.54.

Transformations defined at `NavigationProperty`:

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:filterable = "false"`	`"Org.OData.Capabilities.V1.FilterRestrictions": { "NonFilterableProperties" : [ { "PropertyPath" : "PropA " }, { "PropertyPath" : "PropC " } ] }` For example, if `sap:filterable` is set to `false` for properties `PropA` and `PropC` **i Note** The resulting annotation is added to the `EntitySet`, **not** to the `NavigationProperty`. **i Note** Implemented with version 1.42. **⚠ Caution** Deprecated with version 1.54. See entry below.
`sap:filterable = "false"`	`"Org.OData.Capabilities.V1.NavigationRestrictions": { "RestrictedProperties": [{ "FilterRestrictions": {"Filterable": false}, "NavigationProperty": {"NavigationPropertyPath": "NavPropA"} }, { "FilterRestrictions": {"Filterable": false}, "NavigationProperty": {"NavigationPropertyPath": "NavPropB"} }] }`  For example, if `sap:filterable` is set to false for navigation properties `NavPropA` and `NavPropB`.  **i Note** The resulting annotation is added to the `EntitySet`, **not** to the `NavigationProperty`.

OData V2 SAP Extension	Resulting OData V4 Annotation
	i Note Implemented with version 1.54.

`sap:creatable = "false"`	`"Org.OData.Capabilities.V1.InsertRestrictions": {`
	
	`"NonInsertableNavigationProperties" : [`
	`{ "NavigationPropertyPath" : "NavPropA " },`
	`{ "NavigationPropertyPath" : "NavPropC " }`
	`]`
	`}`
	
	For example, if `sap:creatable` is set to `false` for navigation properties `NavPropA` and `NavPropC`
	
	i Note
	The resulting annotation is added to the `EntitySet`, **not** to the `NavigationProperty`.
	
	i Note
	If `sap:creatable` and `sap:creatable-path` are given, the service is broken and it is handled as `sap:creatable="false"`.
	
	i Note
	Implemented with version 1.42.

`sap:creatable-path="Creatable"`	`"Org.OData.Capabilities.V1.InsertRestrictions": {`
	
	`"NonInsertableNavigationProperties" : [{`
	`"IF" : [{`
	`"Not" : {`
	`"Path" : "Creatable"`
	`}`
	`}, {`
	
	`"NavigationPropertyPath" : "NavPropA"`
	`}]`
	`}]`
	`}`

OData V2 SAP Extension	Resulting OData V4 Annotation
	Note The resulting annotation is added to the `EntitySet`, **not** to the `NavigationProperty`. **Note** If `sap:creatable` and `sap:creatable-path` are given, the service is broken and it is handled as `sap:creatable="false"`. **Note** Implemented with version 1.42.

Transformations defined at `Schema`:

OData V2 SAP Extension	Resulting OData V4 Annotation
`schema-version="foo"`	`"@Org.Odata.Core.V1.SchemaVersion" : "foo"` **Note** Implemented with version 1.54.

Depending on the value of the `sap:semantics` annotation, different vocabulary-based annotations are generated. The following transformations are implemented and defined at `property`. In the examples of the resulting JSON at the "defined at" object, `PROPERTY` is a placeholder for the name of the property at which the `sap:semantics` annotation is defined.

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:semantics = "currency-code"`	see `sap:unit` above
`sap:semantics = "unit-of-measure"`	see `sap:unit` above
`sap:semantics = "name"`	`"com.sap.vocabularies.Communication.v1.Contact" : { "fn" : { "Path" : "PROPERTY" } }`
`sap:semantics = "givenname"`	`"com.sap.vocabularies.Communication.v1.Contact" : { "n" : { "given" : { "Path" : "PROPERTY" } } }`

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:semantics = "middlename"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "n" : { "additional" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "familyname"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "n" : { "surname" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "nickname"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "nickname" : { "Path" : "PROPERTY" } }`
`sap:semantics = "honorific"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "n" : { "prefix" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "suffix"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "n" : { "suffix" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "note"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "note" : { "Path" : "PROPERTY" } }`
`sap:semantics = "photo"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "photo" : { "Path" : "PROPERTY" } }`
`sap:semantics = "city"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "adr" : { "locality" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "street"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "adr" : { "street" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "country"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "adr" : { "country" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "region"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "adr" : { "region" : { "Path" : "PROPERTY" } } }`

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:semantics = "zip"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "adr" : { "code" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "pobox"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "adr" : { "pobox" : { "Path" : "PROPERTY" } } }`
`sap:semantics = "org"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "org" : { "Path" : "PROPERTY" } }`
`sap:semantics = "org-unit"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "orgunit" : { "Path" : "PROPERTY" } }`
`sap:semantics = "org-role"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "role" : { "Path" : "PROPERTY" } }`
`sap:semantics = "title"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "title" : { "Path" : "PROPERTY" } }`
`sap:semantics = "bday"`	`"com.sap.vocabularies.Communication.v1 .Contact" : { "bday" : { "Path" : "PROPERTY" } }`
`sap:semantics = "dtstart"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "dtstart" : { "Path" : "PROPERTY" } }`
`sap:semantics = "dtend"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "dtend" : { "Path" : "PROPERTY" } }`
`sap:semantics = "duration"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "duration" : { "Path" : "PROPERTY" } }`
`sap:semantics = "class"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "class" : { "Path" : "PROPERTY" } }`

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:semantics = "status"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "status" : { "Path" : "PROPERTY" } }`
`sap:semantics = "transp"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "transp" : { "Path" : "PROPERTY" } }`
`sap:semantics = "fbtype"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "fbtype" : { "Path" : "PROPERTY" } }`
`sap:semantics = "wholeday"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "wholeday" : { "Path" : "PROPERTY" } }`
`sap:semantics = "location"`	`"com.sap.vocabularies.Communication.v1 .Event" : { "location" : { "Path" : "PROPERTY" } }`
`sap:semantics = "due"`	`"com.sap.vocabularies.Communication.v1 .Task" : { "due" : { "Path" : "PROPERTY" } }`
`sap:semantics = "completed"`	`"com.sap.vocabularies.Communication.v1 .Task" : { "completed" : { "Path" : "PROPERTY" } }`
`sap:semantics = "percent-complete"`	`"com.sap.vocabularies.Communication.v1 .Task" : { "percentcomplete" : { "Path" : "PROPERTY" } }`
`sap:semantics = "priority"`	`"com.sap.vocabularies.Communication.v1 .Task" : { "priority" : { "Path" : "PROPERTY" } }`
`sap:semantics = "from"`	`"com.sap.vocabularies.Communication.v1 .Message" : { "from" : { "Path" : "PROPERTY" } }`
`sap:semantics = "sender"`	`"com.sap.vocabularies.Communication.v1 .Message" : { "sender" : { "Path" : "PROPERTY" } }`

OData V2 SAP Extension	Resulting OData V4 Annotation
`sap:semantics = "subject"`	`"com.sap.vocabularies.Communication.v1 .Message" : { "subject" : { "Path" : "PROPERTY" } }`
`sap:semantics = "body"`	`"com.sap.vocabularies.Communication.v1 .Message" : { "body" : { "Path" : "PROPERTY" } }`
`sap:semantics = "received"`	`"com.sap.vocabularies.Communication.v1 .Message" : { "received" : { "Path" : "PROPERTY" } }`

`sap:semantics = "tel"`	At the `EntityType` or `ComplexType`:
	`"com.sap.vocabularies.Communication.v1 .Contact" : { "tel" : [{ "uri" : { "Path" : "ATTRIBUTE" } }]}`
	Where `ATTRIBUTE` is the name of the annotated attribute of an `EntityType` or `ComplexType`.
	At `Property`:
	`"com.sap.vocabularies.Communication.v1 .IsPhoneNumber" : { "Bool" : "true" }`

`sap:semantics = "tel";type=cell,work`	At the `EntityType` or `ComplexType`:
	`"com.sap.vocabularies.Communication.v1 .Contact" : { "tel" : [{ "type" : { "EnumMember": "com.sap.vocabularies.Communication.v1 .PhoneType/cell" + " com.sap.vocabularies.Communication.v1. PhoneType/work" }, "uri" : { "Path" : "ATTRIBUTE" } }]}`
	Where `ATTRIBUTE` is the name of the annotated attribute of an `EntityType` or `ComplexType`.
	At `Property`:
	`"com.sap.vocabularies.Communication.v1 .IsPhoneNumber" : { "Bool" : "true" }`

OData V2 SAP Extension	Resulting OData V4 Annotation

`sap:semantics = "email"`	At the `EntityType` or `ComplexType`:
	`"com.sap.vocabularies.Communication.v1 .Contact" : { "address" : [{ "uri" : { "Path" : "ATTRIBUTE" } }]}`
	Where `ATTRIBUTE` is the name of the annotated attribute of an `EntityType` or `ComplexType`.
	At `Property`:
	`"com.sap.vocabularies.Communication.v1 .IsEmailAddress" : { "Bool" : "true" }`

`sap:semantics = "email";type=work,pref`	At the `EntityType` or `ComplexType`:
	`"com.sap.vocabularies.Communication.v1 .Contact" : { "email" : [{ "address" : { "Path" : "ATTRIBUTE" }, "type" : { "EnumMember" : "com.sap.vocabularies.Communication.v1 .ContactInformationType/work" + "com.sap.vocabularies.Communication.v1 .ContactInformationType/preferred" } }]}`
	Where `ATTRIBUTE` is the name of the annotated attribute of an `EntityType` or `ComplexType`.
	At `Property`:
	`"com.sap.vocabularies.Communication.v1 .IsEmailAddress" : { "Bool" : "true" }`

Related Information

XML Templating [page 1018]
OData V2 Model [page 883]
Class `sap.ui.model.odata.ODataMetaModel`

OData V4 Model

The `sap.ui.model.odata.v4.ODataModel` is the model implementation for consuming an OData V4 service.

> **! Restriction**
>
> Due to the limited feature scope of this version of the SAPUI5 OData V4 model, check that all required features are in place before developing freestyle and Fiori elements applications. Double check the detailed documentation of the features, as certain parts of a feature may be missing. While we aim to be compatible with existing controls, some controls might not work due to small incompatibilities compared to `sap.ui.model.odata.(v2.)ODataModel`, or due to missing features in the model (such as tree binding). This also applies to smart controls (`sap.ui.comp` library) that do not support the SAPUI5 OData V4 model, as well as controls such as `TreeTable` and `AnalyticalTable`, which are not supported together with the SAPUI5 OData V4 model. The interface for applications has been changed for easier and more efficient use of the model. For a summary of these changes, see Changes Compared to OData V2 Model [page 971].

The OData V4 model supports the following:

- Read access
- Updating properties of OData entities (in entity sets and contained entities) via two-way-binding
- Deleting entities
- Operation (function and action) execution
- Grouping data requests in a batch request
- Server-side sorting and filtering

> **i Note**
>
> The OData V4 model documentation contains several code samples. These refer to the Sales Orders sample in the Demo Kit.

Related Information

Changes Compared to OData V2 Model [page 971]
sap.ui.model.odata.v4.ODataModel
Sample: Sales Orders

Model Instantiation and Data Access

One OData V4 model instance can only cover one OData service. This section describes the creation of a model instance in more detail.

The OData V4 model is primarily designed for OData V4 services. Nevertheless, OData V2 services may be used through an adapter as well. For more information see: Consuming OData V2 Services with the OData V4 Model [page 977]

When creating an OData V4 model instance, the only parameter you actually need is a map. This map must contain at least the properties `serviceUrl` and `synchronizationMode`. For more information, see the sap.ui.model.odata.v4.ODataModel constructor API documentation in the Demo Kit.

OData V4 model instantiation:

```
sap.ui.define(["sap/ui/model/odata/v4/ODataModel"], function (ODataModel) {
    var oModel = new ODataModel({
        serviceUrl : "/sap/opu/odata4/IWBEP/V4_SAMPLE/default/IWBEP/
V4_GW_SAMPLE_BASIC/0001/",
        synchronizationMode : "None"
    });
});
```

OData Custom Query Options

An OData service accepts query options placed in the service URL query part, as explained on the URL conventions page OData Version 4.0 Part 2: URL Conventions in chapter 2 *URL Components*. The OData V4 model accepts OData custom query options only as explained in section 5.2 *Custom Query Options* of the URL conventions page; you must not provide OData system query options (starting with "$") or OData parameter aliases (starting with "@") at model level, see sections 5.1 *System Query Options* and 5.3 *Parameter Aliases* in the URL conventions page.

> **i** Note
>
> Note that it's possible to specify certain system query options for OData V4 model bindings. For more information, see Bindings [page 922].

OData V4 model instantiation with service URL parameters:

```
sap.ui.define(["sap/ui/model/odata/v4/ODataModel"], function (ODataModel) {
    var oModel = new ODataModel({
        serviceUrl : "/sap/opu/odata4/IWBEP/V4_SAMPLE/default/IWBEP/
V4_GW_SAMPLE_BASIC/0001/?customParam=foo",
        synchronizationMode : "None"
    });
});
```

Default Groups for Batch Control

The OData V4 model allows you to specify whether or not requests are bundled and sent as a batch request, and when the requests are sent. For more information, see Batch Control [page 952].

The parameter `groupId` specifies the default batch group and defaults to `"$auto"`. You can use the parameter `updateGroupId` to set a batch group for update requests only. If you do not set this parameter, the `groupId` will be used.

The following code instantiates a model that bundles all update requests in the batch group `"myAppUpdateGroup"`; the batch request can then be sent using `oModel.submitBatch("myAppUpdateGroup")`.

OData V4 model with `updateGroupId`:

```
sap.ui.define(["sap/ui/model/odata/v4/ODataModel"], function (ODataModel) {
    var oModel = new ODataModel({
        serviceUrl : "/sap/opu/odata4/IWBEP/V4_SAMPLE/default/IWBEP/
V4_GW_SAMPLE_BASIC/0001/",
        synchronizationMode : "None",
        updateGroupId : "myAppUpdateGroup"
    });
});
```

Instantiating an OData V4 Model Using the Descriptor File (`manifest.json`)

The code sample below shows the parts of a Descriptor for Applications, Components, and Libraries [page 734] (`manifest.json`) that are relevant for instantiating an OData V4 model:

```
{
    "sap.app" : {
        "dataSources" : {
            "default" : {
                "uri" : "/sap/opu/odata4/IWBEP/V4_SAMPLE/default/IWBEP/
V4_GW_SAMPLE_BASIC/0001/",
                "type" : "OData",
                "settings" : {
                    "odataVersion" : "4.0"
                }
            }
        }
    },
    "sap.ui5" : {
        "models" : {
            "" : {
                "dataSource" : "default",
                "settings" : {
                    "synchronizationMode" : "None",
                    "updateGroupId" : "myAppUpdateGroup"
                }
            }
        }
    }
}
```

Data Access

The OData V4 model only supports data access using bindings. It does not provide any direct access to the data. For more information, see Unsupported Superclass Methods and Events [page 969]. One exception is sap.ui.model.odata.v4.Context#setProperty. It allows to update a property without using a property binding, even without reading the data first.

Language

SAPUI5 uses the concept of a "current language" (see Identifying the Language Code / Locale [page 1269]). This language is automatically propagated to the OData service by the OData V4 model. For this reason, applications must not hard code the language themselves, e.g. they must not specify the `"sap-language"` URL parameter as a custom query option.

Set HTTP Header Fields

You can set headers for HTTP requests sent by the OData V4 model: This is possible statically by adding them to the `manifest.json`, or dynamically with the method ODataModel#changeHttpHeaders. These headers are applied to data and metadata requests sent by the model. The `ODataModel` propagates its HTTP headers to value list models created via ODataMetaModel#createValueListInfo; when changing HTTP headers for the model, however, these changes are not applied to existing value list models: If value list models require the new headers, you have to additionally call `ODataModel#changeHttpHeaders` for each of them. For details, see ODataModel#changeHttpHeaders.

Sample: Set HTTP header `custom` in manifest.json:

```
{
    "sap.app" : {
        "dataSources" : {
            "default" : {
                "uri" : "/sap/opu/odata4/IWBEP/V4_SAMPLE/default/IWBEP/
V4_GW_SAMPLE_BASIC/0001/",
                "type" : "OData",
                "settings" : {
                    "odataVersion" : "4.0"
                }
            }
        }
    },
    "sap.ui5" : {
        "models" : {
            "" : {
                "dataSource" : "default",
                "settings" : {
                    "autoExpandSelect" : true,
                    "httpHeaders" : {
                        "custom" : "foo"
                    },
                    "synchronizationMode" : "None",

                }
            }
        }
    }
}
```

Related Information

Constructor: sap.ui.model.odata.v4.ODataModel

OData Version 4.0 Part 2: URL Conventions
Bindings [page 922]
Batch Control [page 952]
Descriptor for Applications, Components, and Libraries [page 734]
Unsupported Superclass Methods and Events [page 969]

Bindings

Bindings connect SAPUI5 view elements to model data, allowing changes in the model to be reflected in the view element and vice versa.

The OData V4 model supports the following types of binding:

- **List bindings**, which represent a collection (of OData entities, complex or primitive types) such as `/SalesOrderList` (see the sap.ui.model.odata.v4.ODataListBinding API documentation in the Demo Kit)
- **Context bindings**, which represent a single entity such as `/SalesOrderList('0500000000')` or a structural property with complex type (see the sap.ui.model.odata.v4.ODataContextBinding API documentation in the Demo Kit)
- **Property bindings**, which represent a single, primitive type property in an entity or complex type such as `/ProductList('HT-1000')/Name` (see the sap.ui.model.odata.v4.ODataPropertyBinding API documentation in the Demo Kit)

Related Information

sap.ui.model.odata.v4.ODataListBinding
sap.ui.model.odata.v4.ODataContextBinding
sap.ui.model.odata.v4.ODataPropertyBinding
Sample: Sales Orders
Binding Events [page 938]
Batch Control [page 952]
Filtering [page 939]
Sorting [page 942]
OData Version 4.0 Part 2: URL Conventions, 4 Resource Path
OData Version 4.0 Part 2: URL Conventions, 5 Query Options
OData Version 4.0 Part 2: URL Conventions, 5.2 Custom Query Options
OData Version 4.0 SimpleIdentifier

Creating Bindings

The OData V4 model offers the factory methods `bindList`, `bindContext` and `bindProperty` for creating bindings. Typically, these methods are not called directly in applications, but indirectly by the following:

- Binding an SAPUI5 control through an API such as `oForm.bindElement("{/SalesOrderList('0500000000')}");`
  This sample binds a form to a certain sales order so that form elements can be bound to display or change single properties of the sales order.

  > **i Note**
  >
  > Note that binding expressions can be complex so that they can take additional properties to specify the binding further in addition to the path:
  >
  > ```
  > oForm.bindElement({path : "/SalesOrderList('0500000000')", parameters :
  > {$expand : "SO_2_SOITEM", ...}, events : {dataReceived :
  > '.onDataEvents', ...}});
  > ```
  >
  > This allows you, for instance, to specify OData V4 binding parameters [page 928] such as `$expand` or attach to OData V4 Binding Events [page 938] such as `dataReceived`.

  For a complete example, see the `onSalesOrderSelect` method (file: `Main.controller.js`) in the SalesOrders sample in the Demo Kit.
- Declaring a binding for a control property in an XML view such as the following:

  ```
  <Table items="{path : '/SalesOrderList', parameters : { $expand : 'SO_2_BP',
  $filter : 'BuyerName ge \'M\'', ...}, events : {dataReceived :
  '.onDataEvents', ... } }">
  ```

  For a complete example, see the `Main.view.xml` file in the SalesOrders sample in the Demo Kit.

Path Syntax

According to the specification available under OData Version 4.0 Part 2: URL Conventions, 4 Resource Path, every resource path (relative to the service root URL, no query options) is a valid data binding path within this model if a leading slash is added. For example, you can use `"/EMPLOYEES('A%2FB%26C')"` to access an entity instance with key "A/B&C". Note that appropriate URI encoding is necessary.

Furthermore, the OData V4 model only supports OData SimpleIdentifier with characters from the US ASCII code character set.

Initialization and Read Requests

Bindings are called **absolute** if their path starts with a forward slash "/"; otherwise they are called **relative**. Relative bindings are initial meaning that they have no data as long as they have no context. They obtain a context either from a list binding where the context represents an entity for a certain index in an entity collection or from a context binding where the context represents the one entity of the context binding. The binding which created the context is called the **parent binding** of the relative binding; the relative binding is a **child binding** of its parent binding. Dependent bindings of a binding are the set of child bindings of the binding itself and the dependent bindings of its children. If the binding has no child bindings, it is the empty set.

An absolute binding creates a data service request to read data once data is requested by a bound control or a child control with a relative binding. The read URL path is the model's service URL concatenated with the binding's path. The read URL query options are the union of the binding's and model query options; query options specified for the binding overwrite model query options.

A relative list or context binding creates a data service request once it has a context in the following cases:

- The relative binding has parameters, such as OData query options like `$select`, or binding-specific parameters with a name starting with `$$` see the *Parameters* subsection below. If you want a relative binding to create its own data service request, use the binding-specific parameter `$$ownRequest`.
- You specify a dynamic filter or dynamic sorter for a list binding and use `sap.ui.model.odata.OperationMode.Server`, see sections Filtering [page 939] and Sorting [page 942].

> **i Note**
>
> If the model is configured to compute `$expand` and `$select` automatically, this behaviour changes, see Automatic determination of $expand and $select [page 937].

> **i Note**
>
> A relative property binding never creates a data service request; its binding parameters are ignored in this case.

In all other cases, a relative binding reads data from its parent binding that created the context. In case of an own data service request, the read URL path is the model's service URL concatenated with the path of the binding's context and the binding's path. Set the binding-specific parameter `$$canonicalPath` to `true` to use the canonical path computed from the context's path instead of the context's path in the read URL.

The point in time that is used to actually send the request is determined as explained in the section Batch Control [page 952]. Bindings which create own data service requests cache data from data service responses. They do not send a data service request if data can be served from this cache.

> **i Note**
>
> List bindings read data in pages, i.e. they only access a certain index range from their bound collection; they only trigger a new data service request if indexes are accessed which have not yet been read.

You can delete the cache for an absolute binding using its `refresh` method. The method also deletes the caches of child bindings of the absolute binding.

> **i Note**
>
> There must be no pending property changes for a binding and its child bindings when calling the `refresh` method. Use the binding's `hasPendingChanges` method to check for pending changes before you delete the cache.

You can refresh all bindings with `ODataModel.refresh`, see ODataModel.refresh in the Demo Kit.

Refresh a single entity

You can refresh a **single** entity by calling `sap.ui.model.odata.v4.Context#refresh` or the bound context or return value context of an `sap.ui.model.odata.v4.ODataListBinding` which corresponds to this entity. This also refreshes all dependent bindings of its contexts.

Example

Usage of Context#refresh for a context in a list binding

```
onRefreshSelectedSalesOrder : function () {
    // within a sap.m.Table bound to a OData V4 list binding get the OData V4
context for the selected entity
    var oSalesOrderContext =
this.byId("SalesOrders").getSelectedItem().getBindingContext();

    if (!oSalesOrderContext.hasPendingChanges()) {
        oSalesOrderContext.refresh();
    }
},
```

Note

- Contexts of an `sap.ui.model.odata.v4.ODataListBinding` and the bound context of an `sap.ui.model.odata.v4.ODataContextBinding` can only be refreshed if the binding is not relative to a `sap.ui.model.odata.v4.Contex` and if its root binding is not suspended.
- Refresh is only allowed if there are no pending changes for the context and all its dependent bindings. If you have a relative binding with changes and this binding loses its context, the former parent binding does not report pending changes: the changes are kept, but the relation between these bindings is lost. You can do the following:
  - To find out if there are pending changes, use `sap.ui.model.odata.v4.ODataModel#hasPendingChanges`.
  - To save the changes, use `sap.ui.model.odata.v4.ODataModel#submitBatch`, and to delete the changes, use `sap.ui.model.odata.v4.ODataModel#resetChanges`.
  - If you set a context at the relative binding, the new parent binding will report the pending changes again.

Allow removal of a single entity when refreshing

After updating an entity, it may no longer match the query options for the collection which loaded the entity, in particular `$filter`. You can decide whether the refresh on the context of a list binding should ignore the query options or not: The corresponding context may be removed from the list binding for the collection by setting the parameter `bAllowRemoval` to `true`.

Note that changes to the list like a different sort order require a refresh of the whole list.

An example can be seen in the SalesOrders application. The table has a filter applied to show only the sales orders with *Life Cycle Status = "New"*. When confirming a sales order, its status will change to *In Process* and does not match the filter anymore. This sales order is then refreshed and will be removed from the list as the `bAllowRemoval` flag is set to `true`. This is shown in the following code snippet:

Example

refresh with allow removal

```
oAction.execute("confirmSalesOrderActionGroup").then(function () {
    oConfirmedSalesOrderContext.refresh(undefined, true); // bAllowRemoval =
true
});
```

For details, see ODataListBinding.refresh, ODataContextBinding.refresh and ODataPropertyBinding.refresh in the Demo Kit.

Example: Absolute and relative bindings created by an XML view

```
<Table items="{
 path : '/SalesOrderList',
 parameters : {
  $expand : 'SO_2_BP',
  $select : 'BuyerName,CurrencyCode,GrossAmount,Note,SalesOrderID'
 }}">
 ...
 <items>
  <ColumnListItem>
   <cells>
    <Text text="{SalesOrderID}"/>
    <Text text="{SO_2_BP/CompanyName}"/>
    <Text text="{BillingStatus}"/>
   </cells>
  </ColumnListItem>
 </items>
</Table>
<Table items="{
 path : 'SO_2_SOITEM',
 parameters : {
    $select: "DeliveryDate,GrossAmount,SalesOrderID"
 }
>
...
</Table>
```

The above sample shows an absolute list binding: A table's `items` aggregation is bound to `/SalesOrderList` using the `$expand` and `$select` query options as binding parameters. The columns define relative bindings with paths `SalesOrderID`, `SO_2_BP/CompanyName`, and `BillingStatus` with the absolute list binding as parent binding.

> **i Note**
>
> The `BillingStatus` remains empty and logs an error to the browser console as this structural property is not part of the `$select` specified for the list binding.

The lower table for the line items has a relative binding. As it has parameters defined, it triggers its own data service request once it receives its binding context.

Property Binding to Metadata

You can use `##` in a property binding's path to branch from data into metadata.

Example: Determine label from the corresponding annotation for property `GrossAmount`

```
<SimpleForm binding="{/SalesOrderList('42')}">
    <Label text="{GrossAmount##@com.sap.vocabularies.Common.v1.Label}" />
    <Text text="{GrossAmount}" />
</SimpleForm>
```

For details, see `sap.ui.model.odata.v4.ODataModel#bindProperty`.

Property Binding With an Object Value

A property binding can have an object value, if the target type specified in the corresponding control property's binding info is "any" and the binding is relative or points to metadata. In case it points to metadata, the binding's mode has to be `OneTime`, see sap.ui.model.BindingMode

Example: Using the controller method `'formatPhoneNumbersAsCSV'` to show a comma-separated list of phone numbers for business partner contacts

```
<SimpleForm binding="{/BusinessPartnerList('42')}">
    <Label text="Phone number list" />
    <Text text="{path : 'BP_2_CONTACT', mode : 'OneTime', targetType : 'any',
formatter : '.formatPhoneNumbersAsCSV'}" />
</SimpleForm>
```

For details,

For details, see `sap.ui.model.odata.v4.ODataModel#bindProperty`.

Side Effects

Editing properties of an entity sometimes causes side effects on other properties within the same or a related entity. Normally, a `PATCH` request which sends the user's input to the server includes side effects for the same entity (if relevant for the UI) within its response. Sometimes, however, an application needs more control on how and when this happens, or needs side effects on related entities as well.

You can use sap.ui.model.odata.v4.Context#requestSideEffects to load side effects when implicit loading is switched off via the binding-specific parameter `$$patchWithoutSideEffects`. This method must only be called on the bound context of a context binding, or on the return value context of an operation binding. Collection-valued navigation properites are fully supported, so an efficient request is sent instead of a simple refresh. The event `validateFieldGroup` provides a suitable point in time to request side effects after a certain group of fields has been changed. The annotation `com.sap.vocabularies.Common.v1.SideEffects` describes side effects and the API strikes a balance between the generic use based on this annotation and specific hard-coded uses. When requested from the V4 OData meta model, the annotations value looks as follows:

```
{
    "SourceEntities" : [{
        "$NavigationPropertyPath" : ""
    }, ...],
    "SourceProperties" : [{
        "$PropertyPath" : "Name"
    }, ...],
    "TargetEntities" : [{
        "$NavigationPropertyPath" : "DraftAdministrativeData"
    }, ...],
    "TargetProperties" : [{
        "$PropertyPath" : "DraftAdministrativeData/InProcessByUser"
    }, ...]
}
```

The `sap.ui.model.odata.v4.Context#requestSideEffects` API requires a single array as parameter, namely the concatenation of `TargetEntities` and `TargetProperties`.

The binding-specific parameter `$$patchWithoutSideEffects` can be set on context bindings and list bindings. If the parameter is not specified in a relative binding, the parameter value from the parent binding is used.

Parameters

When creating a binding, you can provide a parameter map which can contain the following:

- OData query options; the values determine parameters for data service requests triggered by the binding. For more information about these options, see OData Version 4.0 Part 2: URL Conventions, 5 Query Options .
- Binding-specific parameters start with `"$$"` and influence the behavior of the binding as follows:
- - `$$canonicalPath`: Set to `true` to use the canonical path computed from the path of the binding's context in the read URL for data service requests. All values other than `true` lead to an error.
  - `$$groupId` and `$$updateGroupId`: see Batch Control [page 952]
  - `$$noPatch`: In a property binding, set to `true` to prevent changes of the value to appear in back-end requests.
  - `$$operationMode`: see Filtering [page 939] and Sorting [page 942]
  - `$$ownRequest`: Set to `true` to ensure the binding uses an own service request to read data. All values other than `true` lead to an error.

The binding's OData query options are combined with the query options passed to the OData V4 model; the binding's query options overwrite model query options with the same name. The resulting query options are appended to each data service request by this binding. The following query options are supported; all others are not allowed and lead to an error:

- OData custom query options except those with the name prefix `"sap-"`. For more information about these, see OData Version 4.0 Part 2: URL Conventions, 5.2 Custom Query Options .
- The list and context binding support the OData system query options `$apply`, `$count`, `$expand`, `$filter`, `$orderby`, `$search` and `$select`.

The query option `$count` must be specified as a boolean value with `true` or `false`. All other query options can be specified with a string value. In addition to strings, the following alternatives are possible:

- `$select` can be specified as an array of strings where each string specifies a select item, or the value `'*'` to select all properties. Normally, these items point to direct parts of the query result without further expanding into related entities.Further options are available with Automatic determination of $expand and $select [page 937].
- `$expand` can be an object where each object property corresponds to an expand item: the key is the complete expand path. The value can be set as follows:
  a) `true` or `null` if no `expand` options are required
  b) An object with query options for the `$expand`; numeric options (like `$levels`) may be given as numbers. If the option is `$expand` or `$select`, the value may again be an object or array.

Example: Binding with parameters in JavaScript

```
oView.byId("SalesOrderTable").bindItems({
    path : "/SalesOrderList",
    parameters : {
        "$count" : true,
```

```
        "$expand" : {
            "SO_2_SOITEM" : {
                "$orderby" : "ItemPosition",
                "$select" : ["ItemPosition", "Quantity", "QuantityUnit",
"SalesOrderID"]
            }
        },
        "$filter" : "BuyerName ge 'M'",
        "$orderby" : "GrossAmount desc",
        "$select" : ["BuyerName", "CurrencyCode", "GrossAmount", "Note",
"SalesOrderID"]
    }
});
```

Example: Binding with parameters in an XML view (`$select` and `$expand` values as string)

```
<Table growing="true" growingThreshold="5" id="SalesOrders"
    items="{
            path : '/SalesOrderList',
            parameters : {
                $count : true,
                $expand : 'SO_2_BP',
                $filter : 'BuyerName ge \'M\'',
                $orderby : 'GrossAmount desc',
                $select : 'BuyerName,CurrencyCode,GrossAmount,Note,SalesOrderID'
            },
        }">
```

Example: Binding with parameters in an XML view (`$select` and `$expand` values as object)

```
<Table growing="true" growingThreshold="5" id="SalesOrders"
    items="{
            path : '/SalesOrderList',
            parameters : {
                $count : true,
                $expand : {
                       'SO_2_SOITEM' : {
                              '$orderby' : 'ItemPosition',
                              '$select' :
['ItemPosition','Quantity','QuantityUnit','SalesOrderID']
                       }
                },
                $filter : 'BuyerName ge \'M\'',
                $orderby : 'GrossAmount desc',
                $select :
['BuyerName','CurrencyCode','GrossAmount','Note','SalesOrderID']
            },
        }">
```

`changeParameters` allows to change, add, or delete OData query options. This does **not** apply, however, to binding-specific parameters that start with `$$`.

The parameters are changed according to the given map of parameters: Parameters with an undefined value are removed, the other parameters are set, and missing parameters remain unchanged. Change, add or delete is possible at the same time. The binding is refreshed as soon as the parameter changes are applied.

Example: Change binding parameters in JavaScript

```
oView.byId("SalesOrderTable").getBinding("items").changeParameters({
    "$search" : '"mountain bike"',
    "$filter" : undefined
});
```

Binding Collection Inline Count

The OData V4 model allows for binding the inline count of the entity collection read by an `ODataListBinding` which has the parameter `$count` set to `true`. In the example below, the table title is bound to `"$count"`, thus representing the number of sales order entities in the collection bound to the table.

Example: Table with title bound to `$count`

```
<Table id="SalesOrders"
    items="{
        path : '/SalesOrderList',
        parameters : {
            $count : true,
        }
    }"
>
    <headerToolbar>
        <Toolbar>
            <content>
                <Title id="SalesOrdersTitle" text="{$count} Sales Orders"/>
            </content>
        </Toolbar>
    </headerToolbar>
    ...
</Table>
```

The `ODataListBinding` provides a header context which holds header information like inline count for the list. Set the binding context for the relative property binding with path `"$count"` to the header context, see ODataListBinding.getHeaderContext. With this, the property binding's value is the list's inline count.

Example: Setting the header context for a property binding to $count

```
setHeaderContext : function () {
    var oView = this.getView();
    oView.byId("SalesOrdersTitle").setBindingContext(
        oView.byId("SalesOrders").getBinding("items").getHeaderContext());
}
```

> **i Note**
>
> The header context needs to be set when the list binding has been resolved, for example after a relative binding has been given its context with `sap.ui.base.ManagedObject#setBindingContext` or by binding on a parent element with `sap.ui.core.Element#bindElement`. In case the list binding is resolved initially, it is sufficient to set the header context in `sap.ui.core.mvc.Controller#onBeforeRendering`.

> **i Note**
>
> A property binding bound to the path `"$count"` may be part of a control hierarchy. When a parent control receives a different binding context than the list's header context, this binding context is propagated to the property binding. The property binding then fails to read its value as the context is not the header context and a console error is written.
>
> Example: On selection in the *Sales Orders* table, the application controller code sets the binding context of the table for the corresponding *Sales Order Items* table with relative binding path `"SalesOrderItems"` to the context corresponding to the selected sales order. The title of the *Sales Order Items* table which is bound to `"$count"` fails to read its value.

To circumvent this issue, proceed as follows:

1. Define the `ODataModel` as an additional named model with a specific name, such as `"headerContext"`.
2. Bind the property binding to this model using the path `"headerContext>$count"`.
3. Set the property binding's context with the model name `"headerContext"`.

To see this solution for the above example, search for `"headerContext"` in the code of the *OData V4 "Sales Orders"* sample app.

Type Determination

The property binding automatically determines the appropriate type depending on the property's metadata, unless a type is specified explicitly. For example, the binding `"{DeliveryDate}"` will determine the type `sap.ui.model.odata.type.DateTimeOffset` (assuming the metadata specifies "Edm.DateTimeOffset" for this property), but `"{path : 'DeliveryDate', type : 'sap.ui.model.odata.type.String'}"` uses the hardcoded type `sap.ui.model.odata.type.String` instead (and does not require metadata). You cannot specify format options or constraints unless you also hardcode the type.

Automatic type determination will take constraints from metadata into account, namely the OData property facets "MaxLength", "Nullable", "Precision" and "Scale". In addition to the OData property facets, the following OData V4 annotations are considered to set type constraints on automatic type determination:

- `Org.OData.Validation.V1.Validation.Minimum`, `Org.OData.Validation.V1.Validation.Maximum` and `Org.OData.Validation.V1.Validation.Exclusive` are used to set the constraints `minimum`, `maximum`, `minimumExclusive` and `maximumExlusive` for `sap.ui.model.odata.type.Decimal`.
- `com.sap.vocabularies.Common.v1.IsDigitSequence` is used to set the constraint `isDigitSequence` for `sap.ui.model.odata.type.String`.

> **i Note**
>
> Only constant expressions are supported to determine the annotation value in this case.

Currently, the types "Edm.Boolean", "Edm.Byte", "Edm.Date", "Edm.DateTimeOffset", "Edm.Decimal", "Edm.Double", "Edm.Guid", "Edm.Int16", "Edm.Int32", "Edm.Int64", "Edm.SByte", "Edm.Single", "Edm.String" and "Edm.TimeOfDay" are supported and mapped to the corresponding type in the namespace `sap.ui.model.odata.type`. All other types, including collections, are mapped to the generic type `sap.ui.model.odata.type.Raw` which can only be used to access the raw model value "as is", but not to convert it to a human readable representation. This allows specialized controls to work with types that would otherwise not be supported.

For more information, see the sap.ui.model.odata.type and sap.ui.model.odata.type.Raw API documentation in the Demo Kit.

> **i Note**
>
> By default, a property binding delivers a value formatted according to the target type of the control property it applies to, for example, “boolean” in case of `<Icon src="sap-icon://message-warning"`

`visible="{path : 'DeliveryDate', formatter : '.isOverdue'}">`. This leads to errors because type determination adds the correct type for the `DeliveryDate` property which is `DateTimeOffset` and cannot format its value as a boolean value. In such cases, use `targetType : 'any'` as follows:

```
<Icon src="sap-icon://message-warning" visible="{path : 'DeliveryDate',
targetType : 'any', formatter : '.isOverdue'}">
```

In rare cases, you might also want to specify a different `targetType`, for example `string`, `boolean`, `int`, or `float`. For more information how these values relate to OData types, see the sap.ui.model.odata.type API documentation or explore the XML Templating: UI5 OData Types sample in the Demo Kit. For more information about `targetType`, see the sap.ui.base.ManagedObject#bindProperty API documentation in the Demo Kit.

Binding Modes

The OData V4 model supports one-time binding, one-way binding and two-way binding modes (see sap.ui.model.BindingMode in the Demo Kit). The two-way binding mode is the **default** binding mode of the OData V4 model. You can use the `setDefaultBindingMode` method on the model to change the binding mode. For more information, see setDefaultBindingMode in the Demo Kit.

Suspend and Resume

You can suspend a `list binding` or `context binding` with its suspend method. A suspended binding does not send data service requests nor does it fire change events. You can only suspend absolute bindings or bindings which are quasi-absolute. A quasi-absolute binding is a relative binding with a context which is not a `sap.ui.model.odata.v4.Context`. You can only suspend a binding which is not yet suspended. For a relative binding having a V4 context, you may suspend the **root binding** of its binding hierarchy which is the (quasi-) absolute ancestor binding of this binding. The binding's method `getRootBinding` provides the root binding; for (quasi-) absolute bindings it returns the binding itself.

You can resume a suspended list or context binding with its `resume` method.

Typical use cases for suspend and resume are:

1. **Trigger read requests for controls in the view later not when the view is initialized:**
   In some situations you may want to suppress OData requests and change events triggered by an OData V4 binding for a certain period of time. This is useful for value help dialogs, such as the value help for the `/BusinessPartnerList` when creating a sales order in the `SalesOrders OData V4 sample`.
2. **UI adaptation at runtime:**
   The UI is adapted by adding or removing a column to a table or a field to a form; the "auto-$expand/$select" feature recomputes the $expand and $select query options when the corresponding list binding or context binding is suspended before adaptation and resumed afterwards.

> **Note**
>
> The template for the aggregation in a UI5 control cannot be adapted afterwards. That's why the aggregation has to be "bound again" after applying changes to the table template. For example. for a `sap.m.Table`, you have to call its `bindItems` method.

3. **Avoid intermediate request when modifying the binding multiple times**
   You want to add a filter and change the sorting of a list binding. If the binding is not suspended, it will trigger a request after calling the `filter` method, and a second request after calling the `sort` method. If it is suspended, only one request with the updated filter and sort criteria is sent on `resume`.

The code below shows a snippet from the `SalesOrders OData V4 sample` which delays the request to `/BusinessPartnerList` until the *Create Sales Order* dialog is displayed.

> **Note**
>
> The `suspended` flag in the binding info triggers a call to the suspend method of the corresponding binding once it is created.

> **Example**
>
> View

```
<Dialog id="CreateSalesOrderDialog" title="Create New Sales Order">
...
    <Input id="NewBuyerID" suggestionItems="{path : '/BusinessPartnerList',
suspended : true}">
        <suggestionItems>
            <core:ListItem key="{BusinessPartnerID}"
additionalText="{CompanyName}"  text="{BusinessPartnerID}"/>
        </suggestionItems>
    </Input>
...
</Dialog>
```

The controller code to open the dialog resumes the list binding on `/BusinessPartnerList` and thus triggers the request.

> **Example**
>
> Controller

```
var oBPListBinding = this.byId("NewBuyerID").getBinding("suggestionItems");

if (oBPListBinding.isSuspended()) {
    oBPListBinding.resume();
}
```

When a binding is suspended, all methods which may trigger CRUD requests for this binding, for example `ODataListBinding.create` throw an error. This is also true for dependent bindings of a suspended binding. However methods that cause the binding to be refreshed completely are allowed. These methods are:

- `ODataContextBinding.changeParameters`
- `ODataContextBinding.refresh`
- `ODataListBinding.changeParameters`

- ODataListBinding.refresh
- ODataListBinding.filter
- ODataListBinding.sort
- ODataListBinding.setAggregation
- ODataListBinding.updateAnalyticalInfo

> ⚠ Caution
>
> It is not allowed to suspend operation bindings.

Context API

The OData V4 model's list and context bindings create `sap.ui.model.odata.v4.Context` objects, which enhance `sap.ui.model.Context` and provide the following methods:

- `getObject` or `getProperty` provide synchronous and `requestObject` or `requestProperty` provide asynchronous access to values; the methods to access a property can provide the value in internal or external format
- `getBinding` retrieves the binding which created the context
- `getIndex` returns the context's list index provided the context has been created by an `ODataListBinding`
- `created` returns a promise that resolves after the successful creation of the new entity in the back end
- `delete` deletes an entity
- `getCanonicalPath, requestCanonicalPath` determines the "canonical path" of an entity
- `hasPendingChanges` tests for pending changes
- `isTransient` tests whether a new entity exists on the client-side only
- `refresh` refreshes an entity
- `requestSideEffects` loads the side effects of a PATCH
- `setProperty` asynchronously sets a property value. This is even possible without reading data first, in which case `If-Match : *` is used. You can also set a property without sending a PATCH request by using `null` as a group ID, but only if data has been read before.

> ℹ Note
>
> For `getObject` and `requestObject`, the data is cloned if the given path points to a non-primitive type. This ensures that internal OData model values cannot be modified.
>
> When a property is read in external format, the format is solely determined by the type defined in the OData meta data of the property, and not by the type or formatter specified for the binding.

```
// assume oEvent is an event fired when a button is pressed in an item of a
table bound to /SalesOrderList
sOrderID =
oEvent.getSource().getBindingContext().getProperty("SalesOrderID"); // the
SalesOrderID in the same item
// get a value in external format e.g. "1.234,23" instead of 1234.23
sGrossAmount = oEvent.getSource().getBindingContext().getProperty("GrossAmount",
true);
```

For more information, see sap.ui.model.odata.v4.Context in the Demo Kit.

Accessing Data in Controller Code

In the OData V4 model, bindings are used to access and modify backend data also if the data is accessed or modified in controller code.

You can create bindings that are independent of controls using the factory methods sap.ui.model.odata.v4.ODataModel.bindContext, sap.ui.model.odata.v4.ODataModel.bindList, and sap.ui.model.odata.v4.ODataModel.bindProperty.

Contexts

sap.ui.model.odata.v4.Context is central for CRUD operations in the controller code. `sap.ui.model.odata.v4.Context` provides the following functions:

- `requestObject`: Returns a promise on the value for the given path relative to the context.
- `requestProperty`: Returns a promise on the property value for the given path relative to the context.
- `getObject`: Returns the value for the given path relative to this context
- `getProperty`: Returns the property value for the given path relative to this context.
- `setProperty`: Sets a new value for the property identified by the given path.

Lists

A list binding is obtained either with `sap.ui.model.odata.v4.ODataModel.bindList`, or by getting an existing list binding bound to a control. Entities of the list can be requested using sap.ui.model.odata.v4.ODataModel.ODataListBinding.requestContexts. The function returns a promise resolving with an array of the requested contexts. The data of each context can be accessed using the `requestObject`, `requestProperty`, `getObject`, and `getProperty` methods of `sap.ui.model.odata.v4.Context`. To modify the data, `sap.ui.model.odata.v4.Context.setProperty` can be used.

```
var oList = oModel.bindList("/SalesOrderList");
oList.requestContexts(10, 20).then(function (aContexts) {
    aContexts.forEach(function (oContext) {
        // As we have fetched the data already, we can access "Note" through
getProperty
        var sNote = oContext.getProperty("Note");
        if (!sNote) {
            oContext.setProperty("Note", "No notes");
        }
    });
});
```

New entities can be created with sap.ui.model.odata.v4.ODataModel.ODataListBinding.create. It is not required to read existing records before.

```
var oList = oModel.bindList("/SalesOrderList"),
    oNewContext = oList.create();
```

Single Entities

A context binding is obtained either with sap.ui.model.odata.v4.ODataModel.bindContext, or by getting an existing context binding from the control tree. Data can be accessed through the bound context, see sap.ui.model.odata.v4.ODataContextBinding.getBoundContext. Using `sap.ui.model.odata.v4.ODataContextBinding.requestObject` is a short cut for `oBinding.getBoundContext().requestObject()`. sap.ui.model.odata.v4.Context.setProperty is used to modify data.

> **i** Note
>
> Relative bindings need to be resolved meaning that a context must have been set before data access is possible.

```
var oContextBinding = oModel.bindContext("/SalesOrderList('0500000000')");
oContextBinding.requestObject("Note").then(function (sNote) {
    if (!sNote) {
        oContextBinding.getBoundContext().setProperty("Note", "No notes");
    }
});
```

Single Properties

The access to single properties may either be done using a context binding as described above in the *Single Entities* section, or by using a property binding. The property binding is obtained with sap.ui.model.odata.v4.ODataModel.bindProperty, or by getting an existing property binding bound to a control. The value can be requested using sap.ui.model.odata.v4.ODataPropertyBinding.requestValue. A synchronous access is possible with sap.ui.model.odata.v4.ODataPropertyBinding.getValue, if the value is already available.

The value of a property binding can be modified using `sap.ui.model.odata.v4.ODataPropertyBinding.setValue`.

```
var oNote = oModel.bindProperty("/SalesOrderList('0500000000')/Note");


oNote.requestValue().then(function (sValue) {
    // do something with sValue
    // Note: We cannot use setValue as oNote is an absolute property binding
});
```

Automatic determination of $expand and $select

With automatic determination of `$expand` and `$select` ("auto-$expand/$select" in short), the OData V4 Model computes `$expand` and `$select` query options for service requests from binding paths specified for control properties. This has the following advantages:

1. You don't have to add or change `$select` or `$expand` in the binding parameters yourself.
2. Auto-$expand/$select only selects data needed for the UI, so that you get a minimal response size and improved performance.

You switch on auto-$expand/$select by setting the flag `autoExpandSelect` during model construction.

It is still possible to specify `$expand` and `$select` in the binding parameters. This is useful if you need to access properties which are not bound on the UI. When auto-$expand/$select is switched on, you may add any path to a simple or structured property to `$select`, even if this path contains navigation properties. The binding converts this `$select` to a `$expand` if necessary. It is not possible to change `$expand` and `$select` via the binding's changeParameters API. You don't have to specify **key properties** in the binding's `$select` parameter if they aren't bound on the UI. These are selected automatically because keys are required in many scenarios, for example, to compute the edit-URL to update an entity.

In auto-$expand/$select mode, a parent binding aggregates the binding paths and query options of its child bindings in its `$select` and `$expand` options, so that they do not send own data services requests. This aggregation is only possible in the following cases:

1. If the request for the parent binding is **not sent** and the child binding is a list or context binding which has only OData system query options in its parameters, or is a property binding.
2. If the request for the parent binding is **already sent** and the request already contains the aggregation for the child binding in its `$expand` and `$select`.

In other cases the child binding is not aggregated and sends an own request.

The list binding for the table in the following sample leads to the following request (reduced to `$expand` and `$select` parameters):

```
SalesOrderList?$select=BuyerName,LifecycleStatus,Note,SalesOrderID&
$expand=SO_2_BP($select=BusinessPartnerID,CompanyName)
```

```
<Table items="{/SalesOrderList}"
  ...
  <items>
    <ColumnListItem>
      <cells> <Text text="{BuyerName}"/> </cells>
      <cells> <Text text="{SO_2_BP/CompanyName}"/> </cells>
      <cells> <Input enabled="{= %{LifecycleStatus} === 'N' }" value="{Note}"/>
</cells>
    </ColumnListItem>
  </items>
</Table>
```

If you use a list binding with factory function with auto-$expand/$select, you need to specify the binding parameters `$expand` and `$select` for all properties that may be needed by the factory function.

> **i Note**
>
> During automatic determination of `$expand` and `$select` the factory function is called with a "virtual" context, that returns `undefined` for `getProperty` calls.

For auto-$expand/$select the model metadata must be analyzed before sending the request. This allows further optimization of the request, also enabling access to the parent entity by reducing partner navigation properties in the path.

Example: A view shows a sales order together with its line items, with a line item binding that is relative to the sales order binding. Any property binding relative to the line item can then access a property of the sales order without causing a `$expand`. This even works if the property is needed only after the sales order data has been requested. This feature can be used to control the visibility or editability of a line item property based on the state of the sales order, or for value help at the line item.

This path reduction of partner navigation properties is also performed in `sap.ui.model.odata.v4.Context#requestSideEffects`, so that side effects on the sales order can be requested via the context of a line item.

Binding Events

The OData V4 model supports certain events intended for applications, and others that are to be used for controls, as outlined in this section.

Events for Applications

For applications, the OData V4 model supports the following events:

- The `dataRequested` and `dataReceived` events are typically used by applications to display and hide a busy indicator or to process a back-end error which happened when requesting data. The events are fired by `ODataPropertyBinding`, `ODataContextBinding` and `ODataListBinding` when reading data:
  - The `dataRequested` event is fired directly after data has been requested from a back end.
  - The `dataReceived` event is fired after the back-end data has been processed. Note that the `dataReceived` event is also fired after a back-end request has failed. The error of the failed request is passed to the event handler as an `error` parameter.

  For more details, see the corresponding API documentation for the specific bindings ODataPropertyBinding, ODataContextBinding and ODataListBinding in the Demo Kit.
- The `createSent` and `createCompleted` events at the `ODataListBinding` are typically used by applications to lock the UI for the created entity to avoid modifications while the data for the created entity is sent to the back end, but the response from the back end is not yet processed on the client. For each `createSent` event, a `createCompleted` event is fired.
  - The `createSent` event is fired each time a POST request that is triggered for an `ODataListBinding#create` is sent to the backend.
  - The `createCompleted` event is fired each time the backend has responded to a POST request triggered for an `ODataListBinding#create`.

  For more information, see `ODataListBinding#create`.
- The `patchSent` and `patchCompleted` events are typically used by applications that are using update groups with submit mode Auto (which is the default) and which need to be informed when PATCH requests are sent to the back end and when they are processed. For example, these events can be used to display a `DraftIndicator` to inform the user that his changes are being saved and when saving is finished.

The `patchSent` and `patchCompleted` events are fired by `ODataContextBinding` and `ODataListBinding` if they send their own service request:

- The `patchSent` event is fired when the first PATCH request for this binding is sent to the backend.
- The `patchCompleted` event is fired when the backend has responded to the last PATCH request for this binding.

If `ODataContextBinding` and `ODataListBinding` use the service request of a superordinate binding, the events are fired by the superordinate binding.
For more details, see the corresponding API documentation for the specific bindings ODataContextBinding and ODataListBinding in the Demo Kit.

Events for Controls

The events `change` and `refresh` are meant for controls only, and not available for app development. They indicate that the respective binding has new data which can be accessed by the control:

- When the binding is initialized, it fires a `change` event with the parameter `reason` set to `sap.ui.model.ChangeReason.Change`.
- When a relative binding gets a new context, it fires a `change` event with the parameter `reason` set to `sap.ui.model.ChangeReason.Context`.
- When a binding is refreshed, the event fired depends on the binding type, as follows:
  a) `ODataPropertyBinding` and `ODataContextBinding` fire a `change` event with the parameter `reason` set to `sap.ui.model.ChangeReason.Refresh`.
  b) `ODataListBinding` fires a `refresh` event.

For more details, see the corresponding API documentation for the specific bindings ODataPropertyBinding, ODataContextBinding and ODataListBinding, as well as sap.ui.model.ChangeReason in the Demo Kit.

Related Information

ODataPropertyBinding
ODataContextBinding
ODataListBinding
sap.ui.model.ChangeReason

Filtering

The OData V4 Model supports server side filtering on lists.

To use server side filtering, set the operation mode to sap.ui.model.odata.OperationMode.Server. This can be done as follows:

- For a single `ODataListBinding` instance, set the binding parameter `$$operationMode`
- For all list bindings of the model, set the model parameter `operationMode`.

Example: Operation mode set in `manifest.json` for the model

```
"models" : {
    "" : {
        "dataSource" : "default",
        "settings" : {
            "operationMode" : "Server",
            "synchronizationMode" : "None"
        }
    }
}
```

Example: Operation mode set as binding parameter for a specific list binding

```
<Table growing="true" growingThreshold="5" id="Equipments"
    items="{
            path : '/Equipments',
            parameters : {
                $$operationMode : 'Server',
                $filter : 'Category eq \'Electronics\'',
                $select : 'Category,EmployeeId,ID,Name'
            }
        }">
```

The `ODataListBinding` allows to set static and dynamic filters:

- To set a static filter, use the `$filter` system query option in the binding parameters. The static filter value is sent with every data service request for the binding; you may specify any filter value allowed in OData V4. The static filter cannot be overwritten for an existing binding.
- The dynamic filter is an instance of sap.ui.model.Filter , or an array thereof. For an array, the filters are combined with a logical AND. You can set the initial value for the dynamic filter in ODataModel.bindList or declaratively in an XML view with the `filters` property in an aggregation's binding information. To set the dynamic filter, use the ODataListBinding.filter method. This filter overwrites the initial value specified on binding construction.

The `ODataListBinding` combines the dynamic filter and static filter with a logical AND.

Examle: Dynamic and static filters

```
<Table growing="true" growingThreshold="5" id="Equipments"
    items="{
            path : '/Equipments',
            parameters : {
                $$operationMode : 'Server',
                $filter : 'Category eq \'Electronics
\'',                            <-- static filter
                $select : 'Category,EmployeeId,ID,Name'
            },
            filters : {                                          <-- dynamic filter
initial value
                path : 'EmployeeId',
                operator : 'GE',
                value1 : '0000'
            }
        }">
```

The example above filters the `Equipments` entity set by `Category` (static filter) and `EmployeeId` (dynamic filter, initial value).

Filtering with Any and All

The OData V4 model also supports the Lambda Operators `any` and `all` as defined in section 5.1.1.10 of the OData Version 4.0. Part 2: URL Conventions specification. They are represented by sap.ui.model.Filter objects with filter operators sap.ui.model.FilterOperator.Any and sap.ui.model.FilterOperator.All.

Example:

```
sap.ui.model.Filter({
        // the path of the collection for which the condition needs to be
evaluated
        path : "TEAM_2_EMPLOYEES",
        // either sap.ui.model.FilterOperator.Any or
sap.ui.model.FilterOperator.All
        operator : sap.ui.model.FilterOperator.Any,
        // any OData identifier which is a variable for the current element of
the collection referenced by path
        variable : "employee",
        // the filter condition; the path of the nested filter contains the
variable as prefix to reference current element of the collection
        condition : new sap.ui.model.Filter("employee/AGE",
sap.ui.model.FilterOperator.GT, 42)
    })
```

The path of the filter object is the path of the collection for which the boolean condition needs to be evaluated. The variable can be any OData identifier and it needs to be part of the path of a nested filter condition.

Filter Operator any

The filter operator Any applies the `boolean` filter condition to each member of the collection referenced by `path`. If the condition is true for **at least one** member of the collection, the any-filter matches. The filter with the Any operator without a filter condition matches only if the collection referenced by path is not empty.

Example 1: Get all teams that have at least one employee who is older than 42

```
oTeamsBinding.filter(
    new sap.ui.model.Filter({
        path : "TEAM_2_EMPLOYEES",
        operator : sap.ui.model.FilterOperator.Any,
        variable : "employee",
        condition : new sap.ui.model.Filter("employee/AGE",
sap.ui.model.FilterOperator.GT, 42)
    });
);
```

The resulting request would be: `http://host/service/TEAMS?$filter=TEAM_2_EMPLOYEES/any(employee:employee/AGE gt 42)`

Example 2: Get all teams that have at least one employee assigned

```
oTeamsBinding.filter(
    new sap.ui.model.Filter({
        path : "TEAM_2_EMPLOYEES",
        operator : sap.ui.model.FilterOperator.Any
    });
);
```

The resulting request would be: `http://host/service/TEAMS?$filter=TEAM_2_EMPLOYEES/any()`

Filter Operator all

The filter operator All applies the `boolean` filter condition to each member of the collection referenced by `path`. If the condition is true for **all** members of the collection, the all-filter matches.

Example: Get all teams for which all employees are older than 42.

```
oOrdersListBinding.filter(
    new sap.ui.model.Filter({
        path : "TEAM_2_EMPLOYEES",
        operator : sap.ui.model.FilterOperator.All,
        variable : "employee",
        condition : new sap.ui.model.Filter("employee/AGE",
sap.ui.model.FilterOperator.GT, 42)
    });
);
```

The resulting request would be: `http://host/service/TEAMS?$filter=TEAM_2_EMPLOYEES/all(employee:employee/AGE gt 42)`

Related Information

sap.ui.model.odata.OperationMode.Server

Sorting

The OData V4 model supports server side sorting on lists.

To use server side sorting, set the operation mode to sap.ui.model.odata.OperationMode.Server as described unter Filtering [page 939].

`ODataListBinding` allows to set static and dynamic sorters:

- For setting a static sorter, the `$orderby` system query option in the binding parameters is used. The static sorter value is sent with every data service request for the binding. The static sorter cannot be overwritten for an existing binding.
- The dynamic sorter is a sap.ui.model.odata.Sorter instance, or an array thereof in which case the sorters are concatenated. You can set the initial value for the dynamic sorter in ODataModel.bindList, or declaratively in an XML view with the sorter property in an aggregation's binding information. For setting the dynamic sorter, the ODataListBinding.sort method is used. The sorter that is given here overwrites the initial value specified on binding construction.

Dynamic sorters are transformed to an OData `$orderby` system query option value and the static sorters are always appended as secondary sort criterion. In this example, the equipments are first ordered by `Category` (dynamic sorter) and then by `Name` (secondary sort criterion, static sorter). For a description of the `group` property, see getGroup.

Example: Dynamic and static sorters

```
#js
<Table growing="true" growingThreshold="5" id="Equipments"
    items="{
```

```
            path : '/Equipments',
            parameters : {
                $$operationMode : 'Server',
                $orderby : 'Name',                                  <-- static sorter
                $select : 'Category,EmployeeId,ID,Name'
            },
            sorter : {                                          <-- dynamic sorter;
can be overwritten by calling sort on the list binding
                path : 'Category',
                group : true                                    <-- optional, see
parameter vGroup of sap.ui.model.Sorter
            }
        }">
```

In this example, the equipments are first ordered by `Category` (dynamic sorter) and then by `Name` (secondary sort criterion, static sorter).

Value Lists

The OData V4 model supports the access to value list metadata and data.

Value lists enable you to read the possible values for a given property, such as `Category` in the `Product` entity type. A value list is typically visualized as a dropdown list, or as a value help dialog, that is, a popup with additional features such as filters which help finding the correct value. For performance reasons, you can reduce the service `$metadata` document size by outsourcing value list information to value list services. Consequently, the value list information is accessed in two steps:

1. Determine the value list type that is available for a given property via `ODataPropertyBinding.requestValueListType` without loading the value list service. This is typically called to determine the visualization of this property.
   The `sap.ui.model.odata.v4.ValueListType`, that the promise delivers, can have the following values:
   - `None`: No value list exists.
   - `Fixed`: One enumeration of fixed values exists.
   - `Standard`: A dynamic value list with multiple queries including selection criteria exists.
2. Determine the value list detail information on demand via `ODataPropertyBinding.requestValueListInfo` which returns a map of all annotations `com.sap.vocabularies.Common.v1.ValueList` or `com.sap.vocabularies.Common.v1.ValueListMapping` by qualifier. Each mapping has the `ValueListMappingType` type as specified in the OData 4.0 Common Vocabulary, see OData 4.0 Vocabularies - SAP Common. Each mapping is enriched by a `$model` property of type `sap.ui.model.odata.v4.ODataModel` which can be used to access the value list metadata and retrieve value list data.
   For value lists of type `Fixed`, only one mapping is expected and the qualifier is ignored. The mapping is available with key "".

Additionally, you can use the synchronous method `ODataPropertyBinding.getValueListType` if the metadata for the property is already available. If this is not the case, an exception is thrown. The API is available in `sap.ui.model.odata.v4.ODataMetaModel` analogously for use cases where controls are not yet in place, for example, during XML templating.

Example: Retrieving the value list type for a property

```
#js
       onModelContextChange : function (oEvent) {
        var oBinding = this.getBinding("value");

        if (oBinding && oBinding.isResolved()) {
            oBinding.requestValueListType().then(function (sValueListType) {


                // render the control depending on the value list type and
attach the below
                // event handler onValueHelp which is invoked when the user
requests value help

                switch (sValueListType) {
                    case ValueListType.Standard:
                        ...
                        break;
                    case ValueListType.Fixed:
                        ...
                        break;
                    case ValueListType.None:
                        ...
                        break;
                }
                that.setAggregation("field", oField);
            });
        }
    },
...
    onValueHelp : function (oEvent) {
        var oBinding = this.getBinding("value");

        oBinding.requestValueListInfo().then(function (mValueListInfo) {
            // this assumes value list type "Fixed"
            var oValueListMapping = mValueListInfo[""],
                oValueListMetaModel = oValueListMapping.$model.getMetaModel();

            ...
        ]);
    },
...
```

The `ValueList` Annotation

There are two options to place the `ValueList` annotation:

- In the value list service (the preferred way): When adding a value list for a property, the OData service only contains an annotation with the property as target and the term com.sap.vocabularies.Common.v1.ValueListReferences pointing to the metadata of the value list service. The ValueList annotation itself is in the referenced service. It must not have the properties CollectionRoot and SearchSupported.
- In the OData service itself: In this case, the `ValueList` annotation must have the property `CollectionRoot` pointing to the metadata of the value list service. The annotation `com.sap.vocabularies.Common.v1.ValueListReferences` is not needed.
  The disadvantage of this solution is that the complete value list information for all properties of the service is preloaded when the application is initialized.

Related Information

OData 4.0 Vocabularies - SAP Common > ValueListType
OData 4.0 Vocabularies - SAP Common > ValueListMappingType
sap.ui.model.odata.v4.ValueListType
sap.ui.model.odata.v4.ODataPropertyBinding#getValueListType
sap.ui.model.odata.v4.ODataPropertyBinding#requestValueListInfo
sap.ui.model.odata.v4.ODataMetaModel#getValueListType
sap.ui.model.odata.v4.ODataMetaModel#requestValueListInfo

OData Operations

The OData V4 model supports OData operations (`ActionImport`, `FunctionImport`, bound `Actions` and bound `Functions`). Unbound parameters are limited to primitive values.

Simple Function Bindings

You gain access to a `FunctionImport` by binding it to a view element. If there are no parameters and there is no need to control the point in time when the function is called, you can simply bind the OData path like this:

```
<Text text="{path: '/GetNumberOfAvailableItems()', type:
'sap.ui.model.odata.type.Int16'}"/>
```

This binding path represents the function's return value. The model calls the function immediately when a control requests this value.

The type must be specified if the return value is a primitive type.

Deferred Operation Bindings

Often it is not feasible for the operation to be called immediately, for example if there are parameters that the user has to enter first. In such cases, use an ODataContextBinding as element binding at a layout element in the view, for example a `<Form>` or a `<VBox>` (see the ODataContextBinding API documentation in the Demo Kit). Mark the operation as **deferred** by inserting an ellipsis ("...") in the brackets, for example `GetNextAvailableItem(...)`. Access the return value from child elements using relative bindings. When used like this, the context binding is called an **operation binding** or more specifically, a **function binding** or **action binding** depending on the type of OData operation it is used for.

If the operation binding defers operation execution, you need to call its `execute` method to execute the operation. See below for an example.

View:

```
<Form id="getNextAvailableItem" binding="{/GetNextAvailableItem(...)}">
```

```
    <Label text="Description"/>
    <Text text="{Description}"/>
    <Button text="Call the function" press="onGetNextAvailableItem"/>
</Form>
```

Controller:

```
onGetNextAvailableItem : function (oEvent) {
    this.getView().byId("getNextAvailableItem").getObjectBinding().execute();
}
```

In the above example, the function import is bound to a form (which has an ID that we need later). The text field showing a property of the result is a child of this form. It has a relative binding to the property `"Description"`.

If the function returns a primitive value or a collection, the binding for the result must be `"{value}"` as shown in the 2 examples below:

View:

```
<Form id="getNumberOfAvailableItems" binding="{/GetNumberOfAvailableItems(...)}">
    <Label text="Number of available items:"/>
    <Text text="{value}"/>
    <Button text="Call the function" press="onGetNumberOfAvailableItems"/>
</Form>
```

```
<VBox id="getAvailableItems" binding="{path : '/GetAvailableItems(...)',
parameters : {$select : 'ProductName', 'ProductId'}}">
    <List id="xyz" items="{value}">
        <items>
            <ObjectListItem title="{ProductName}" />
        </items>
    </List>
</VBox>
```

`execute` returns a promise which is resolved if the operation was successful and rejected with an error if this was not the case. Note that the promise is **not** fulfilled with the action's result: Use dependent bindings to access the result.

`refresh` is silently ignored on a deferred function binding as long as it has not yet been executed. Afterwards, a `refresh` calls the function again.

Action Bindings

Action bindings must be deferred, otherwise the application cannot control when the action is executed. A deferred action binding is declared exactly like a deferred function binding:

View:

```
<Form id="Submit" binding="{/Submit(...)}">
    <Button text="Submit the action" press="onSubmit"/>
</Form>
```

You append "(...)" even though the action's resource URL does not contain them. However, they are needed to mark the binding as deferred. In `execute`, the binding uses the metadata to distinguish between action and function and to build the correct operation resource path.

`refresh` is always silently ignored on a deferred action binding to prevent the action from being executed accidentally (for example by calling the `refresh` method on the ODataModel instance `oModel.refresh()`).

Operation Parameters

You can use the parameters of a deferred operation binding inside an XML view.

The parameters are addressed by the path prefix "$Parameter". This can either be done by binding each control property via the path prefix "$Parameter" (Option 1) or by having an outer binding with a "$Parameter" path (Option 2).

> **i Note**
>
> The path "$Parameter" must not be added directly to the path of a deferred operation binding. A deferred operation binding is identified by an ellipsis at the end of the path.

This is how to bind each property without a "$Parameter" context:

View:

> **Example**
>
> Binding parameters to a dialog (Option 1)

```
<Dialog binding="{/ChangeTeamBudgetByID(...)}" id="operation1" title ="Change
Team Budget">
    <buttons>
        ...
    </buttons>
    <form:SimpleForm>
        <Label text="TeamID" />
        <Input value="{$Parameter/TeamID}" />
        <Label text="Budget" />
        <Input value="{$Parameter/Budget}" />
    </form:SimpleForm>
</Dialog>
```

Alternatively, you may bind the entire form to the `$Parameter` context:

View:

> **Example**
>
> Binding parameters to a dialog (Option 2)

```
<Dialog binding="{/ChangeTeamBudgetByID(...)}" id="operation2" title="Change
Team Budget">
    <buttons>
        ...
    </buttons>
    <form:SimpleForm binding="{$Parameter}">
        <Label text="TeamID" />
        <Input value="{TeamID}" />
        <Label text="Budget" />
        <Input value="{Budget}" />
    </form:SimpleForm>
</Dialog>
```

In either case, the values of the parameters are set using the model binding of the control, with no need to write any application code.

Alternatively, operation parameters can be set by calling the function `setParameter` on the operation binding, as shown in this example:

Controller:

```
onSubmit : function (oEvent) {
    this.getView().byId("Submit").getObjectBinding().setParameter("Comment",
sComment).execute();
}
```

The API method `getParameterContext` can be used to access parameters in controller code, see also Accessing Data in Controller Code [page 935]

The example below demonstrates how a budget may be modified depending on the `TeamID`:

Controller:

Example

Reading parameter values using the parameter context

```
adaptBudgetToTeam : function (){
    var oDialog = this.oView.byId("operation2"); // the second dialog in the
paragraph before
        oParameterContext = oDialog.getObjectBinding().getParameterContext();


    if (oParameterContext.getProperty("TeamID') === "STARTUP") {
        oParameterContext.setProperty("Budget", 555.55);
    else {
        oParameterContext.setProperty("Budget", 123.45);
    }
}
```

Note

The parameter context is only defined if the operation binding is resolved.

Bound Actions and Functions

So far, the examples always used operations at root level, addressed via an action import or function import. However, it is also possible to bind an action or a function to another resource of the service. This can be an entity, a collection of entities or an entity property.

Bound actions or functions are controlled in the same way as unbound operations; append `(...)` to the binding path for the control's property.

To call actions or functions bound to a single entity, entity property, or navigation property use a relative binding. The following sample calls the "invoice created" action on the sales order selected in the corresponding table:

```
var oModel = this.getView().getModel(),
```

```
    oTable = this.getView().byId("SalesOrders"),
    oSalesOrderContext = oTable.getSelectedItem().getBindingContext(),
    oAction = oMoodel.bindContext("name.space.InvoiceCreated(...)",
oSalesOrderContext);

oAction.execute().then(
    function () {
        MessageToast.show("Invoice created for sales order " +
oSalesOrderContext.getProperty("SalesOrderID"));
    },
    function (oError) {
        MessageBox.alert(oError.message, {
            icon : MessageBox.Icon.ERROR,
            title : "Error"});
        });
    }
);
```

To call actions or functions bound to a collection specified by an OData entity set, you can create a context binding with an absolute path, or with a relative path for the operation (for example `name.space.DestroyOutdated(...)`") and the header context of a list binding as parent context. The following sample shows a button press event handler which calls the `destroy outdated` action on the `LeaveRequests` entity set.

```
var oModel = this.getView().getModel();

oModel.bindContext("/LeaveRequests/name.space.DestroyOutdated(...)").execute();
```

The same example with a relative binding and the header context of the list binding as parent context:

```
var oModel = this.getView().getModel(),
    // assume there is a table with ID "leaveRequests" and its items aggregation
bound to "/LeaveRequests"
    oListBinding = this.byId("leaveRequests").getBinding("items"),
    oHeaderContext = oListBinding.getHeaderContext();
oModel.bindContext("name.space.DestroyOutdated(...)", oHeaderContext).execute();
```

> **i Note**
>
> - The path of an operation binding may also start with a navigation property.
>   Example: The operation binding has a relative path `BP_2_PRODUCT/name.space.Change(...)`. You set its binding context from the selected item in a table bound to `/BusinessPartners`. When you call `execute` on the operation binding, the "change" action is executed with the selected business partner's navigation property `BP_2_PRODUCT` as binding parameter.
> - The parent binding of a deferred operation must not be a deferred operation itself.

Advertised Operations

According to the OData 4.0 specification ("11.5.2 Advertising Available Operations within a Payload") services may return available actions and functions bound to a particular entity as part of the entity representation within the payload. Data for an advertised operation within an entity is sent as property starting with `#<namespace>.<action>` of that entity. If the entity does not advertise the operation, it does not contain this property. To access the advertised operation in a binding, the same format has to be used. See the following example:

Example

Enable a button to trigger an action `AcSetIsOccupied` available on entity type of entity set `EMPLOYEES` depending on advertisement of this action on the entity `EMPLOYEES('1')`

```
<FlexBox binding="{/EMPLOYEES('1')}">
    <Button text="Set occupied" enabled="{= !!%
{#com.sap.gateway.default.iwbep.tea_busi.v0001.AcSetIsOccupied} }"/>
</FlexBox>
```

Here a button is enabled only if the action `AcSetIsOccupied` is advertised for the entity `EMPLOYEES('1')`. The `%` operator is used to set the internal type to `any` because the advertised action is sent as an object. The double negation `!!` converts this object to a boolean value that is needed by the `enabled` control property.

If no advertised action was returned in the payload, `undefined` (or `null` in OData 4.01 in case of advertised non-availability) is returned as value for the binding. This translates to false in the expression above.

If there is an additional list of non-binding parameter names to identify a specific overload, then they need to be given in the binding path as well, for instance: `%{#Model.RemainingVacation(Year)}`.

Note

The bound action advertisement is added to $select automatically if the model parameter `autoExpandSelect` is set.

To access the metadata of an operation, the double hash (`##`) syntax has to be used as is illustrated in the next example:

Example

Binding against metadata of an action

```
var oContext = oModel.createBindingContext("/EMPLOYEES('1')/
##com.sap.gateway.default.iwbep.tea_busi.v0001.AcSetIsOccupied");
var oMetaModel = oContext.getModel();
oMetaModel.requestObject("0/$ReturnType/$Type", oContext).then(alert);
```

Here a context is created pointing to the metadata of the action and afterwards the type is accessed using this context.

This approach can also be used with XML templating where `createBindingContext` is called internally.

Access Operation Results

You can access the results of the operation by calling `getObject()` from the bound context.

```
// let oOperation be the operation's context binding
oOperation.execute().then(function () {
    // Note: execute does not deliver the results
    var oResults = oOperation.getBoundContext().getObject();
    ...
});
```

The promise returned by the operation binding's `execute` method may resolve with a *return value context* provided the conditions specified in `execute` are met. The operation binding may be bound to an entity or a collection of entities.

The typical use case for *return value context* is when you call a bound operation with a context `C1` defining its binding parameter and the bound operation returns a *different version* of the entity used as binding parameter. `C1` is the binding context of an "object page" container displaying properties of the corresponding entity. You need to replace `C1` as binding context of the object page by the *return value context*. This way, the *different version* of the entity is displayed without a further read request. If the bound operation returns the entity used as binding parameter, the changes will automatically be copied to the binding parameter.

If the operation binding fulfills the conditions for returning a context, you can set the parameter `$$inheritExpandSelect` for the binding: The request for the bound operation is then sent with the same `$expand` and `$select` query options used to load the operation's binding parameter. This way you guarantee that all fields of the object page are available in the operation response.

Sample object page to display an `Artist` entity

```
<form:SimpleForm id="objectPage">
    <Toolbar>
        <Button text="Edit" enabled="{IsActiveEntity}" press=".onEdit"/>
    </Toolbar>
    <Label text="ID"/> <Text text="{ArtistID}"/>
    <Label text="Is Active"/> <Text text="{IsActiveEntity}"/>
    <Label text="Name"/> <Input value="{Name}" />
    ...
</form:SimpleForm>
```

Controller code to display the active version of `Artist 42` initially and switch to draft version on *Edit*

```
// display "active" version of artist initially
onInit : function () {
    var oActiveArtistContext = oModel
            .bindContext("/Artists(ArtistID='42',IsActiveEntity=true)")
            .getBoundContext();
    this.byId("objectPage").setBindingContext(oActiveArtistContext);
},

// display the "inactive" version of the entity returned by the "EditAction"
onEdit : function () {
    var that = this;
    oModel.bindContext("name.space.EditAction(...)",
this.byId("objectPage").getBindingContext(), {$$inheritExpandSelect : true})
        .execute()
        .then(function (oInactiveArtistContext) {
            that.byId("objectPage").setBindingContext(oInactiveArtistContext);
        });
}
```

Related Information

OData Version 4.0 Part 1, 11.5 Operations
ODataContextBinding

Batch Control

OData V4 allows you to group multiple operations into a single HTTP request payload, as described in the official OData V4 specification Part 1, Batch Requests (see the link under Related Information for more details).

The OData V4 model sends requests in the following cases:

- **Implicit read requests** to retrieve data for a binding
  Example: A list binding with the absolute path `/SalesOrderList` triggers a `GET SalesOrderList` to read data.
- **Implicit update requests** via two-way binding
  Example: Update a sales order's note through a property binding with the relative path `Note`, which has a context with path `/SalesOrderList(SalesOrderID='42')` triggering `PATCH SalesOrderList(SalesOrderID='42')` with the note's value as JSON payload.
- **Explicit requests** triggered through API calls like `ODataListBinding.refresh` or `ODataContextBinding.execute`

For each of these cases, it is possible to specify a group ID of type `string`.

A group ID has one of the following submit modes to control the use of batch requests:

- `sap.ui.model.odata.v4.SubmitMode.API` - Requests associated with the group ID are sent in a batch request via `sap.ui.model.odata.v4.ODataModel#submitBatch` .
- `sap.ui.model.odata.v4.SubmitMode.Auto` - Requests associated with the group ID are sent in a batch request which is triggered automatically before rendering.
- `sap.ui.model.odata.v4.SubmitMode.Direct` - Requests associated with the group ID are sent directly without batch.

The following group IDs are possible:

- `"$auto"` and `"$auto.*"`: Predefined batch group ID which is the default if no group ID is specified. You can use different `$auto.*` group IDs to use different batch requests. The suffix can be any non-empty string consisting of alphanumeric characters from the basic Latin alphabet, including the underscore. They have the submit mode `sap.ui.model.odata.v4.SubmitMode.Auto`.
- `"$direct"`: Predefined batch group ID which has the submit mode `sap.ui.model.odata.v4.SubmitMode.Direct`. For more information, see Performance Aspects [page 967].
- An application group ID is a non-empty string consisting of alphanumeric characters from the basic Latin alphabet, including the underscore. By default, an application group has the submit mode `sap.ui.model.odata.v4.SubmitMode.API`. It is possible to use a different submit mode; for details see section Define submit mode for an application group ID [page 955].

To specify the group ID for implicit requests, use the parameters `$$groupId` (group ID for read requests) and `$$updateGroupId` (group ID for update requests) for the binding which triggers the request (see the ODataModel.bindList, ODataModel.bindContext and ODataModel.bindProperty API documentation).

Batch requests for update groups with a submit mode different from `$direct` are queued per group ID. A batch request with changes is only sent if the previous batch request for the same group ID is returned and processed. In this case, all submitted changes for that group ID are combined in one batch request; changes associated with different calls to ODataModel.submitBatch use different change sets inside the batch request.

Code example: Updates for the sales order note through two-way binding will use the group ID `"myGroup"`, whereas data is read with the group `"$auto"`.

Batch group usage for binding created via JavaScript:

```
sap.ui.define(["sap/ui/model/odata/v4/ODataModel"], function (ODataModel) {
    var oModel = new ODataModel({serviceUrl : "/myService/",
synchronizationMode : "None"}),
        oContextBinding = oModel.bindContext("/
SalesOrderList(SalesOrderID='42')", /*oContext*/ undefined, {$$updateGroupId :
"myGroup"}),
        oPropertyBinding = oModel.bindProperty("Note",
oContextBinding.getBoundContext());
});
```

XML view sample: Declares controls which create the context binding (in the `SimpleForm`) and the property binding (in the `Input`) as sketched in the above JavaScript code sample.

Batch group usage for bindings created via XML view:

```
<form:SimpleForm binding="{path : '/SalesOrderList(SalesOrderID=\'42\')',
parameters : {$$updateGroupId : 'myGroup'}}" editable="true" ...>
    <Label labelFor="Note" text="Note" />
    <Input id="Note" value="{Note}" />
    ...
</form:SimpleForm>
```

On instantiation of an OData V4 model, you can provide both a group ID and an update group ID; they are used as defaults if the corresponding binding parameter is not specified. The default for the group ID is `"$auto"`. The value of group ID is used as a default for the update group ID.

For explicit requests, the group ID can be specified as an optional parameter to the corresponding API method. The group ID or update group ID of the binding is used as a default. For more information, see the ODataContextBinding.execute, ODataContextBinding.refresh, ODataListBinding.refresh, ODataPropertyBinding.refresh and ODataPropertyBinding.setValue API documentation in the Demo Kit.

Change Sets and Order of Requests Inside a Batch Request

The OData V4 model automatically puts all non-GET requests into a single change set, which is located at the beginning of a batch request. All GET requests are put after it. If there is only a single request within the change set, it is replaced by that single request when submitting the batch group (saves overhead on the wire). PATCH requests for the same entity are merged into a single request.

Resetting Property Changes

You can set an update group ID for a binding so that property changes are collected in a batch queue. The `ODataModel.submitBatch` method sends all these changes for a given batch group at once and the `ODataModel.resetChanges` method resets the changes. With these methods, you can, for example, implement a *Save* and a *Cancel* button for a form: *Save* triggers `submitBatch`, and *Cancel* triggers *resetChanges*.

> **i Note**
>
> The `resetChanges` method only resets all implicit update requests via two-way binding for the given group, while read requests or requests from `ODataContextBinding.execute` remain in the queue and are sent when the `submitBatch` method is called.

The list and context binding also offer the `resetChanges` method which resets changes for the binding and its child bindings.

> **i Note**
>
> The promise returned by `submitBatch` can be used together with the `hasPendingChanges` method to check whether changes were successfully persisted. `hasPendingChanges` exists for the ODataModel as well as for ODataListBinding, ODataContextBinding and ODataPropertyBinding. Note that the promise returned by `submitBatch` is only rejected if the complete batch request has failed.

Example: View

```
<Toolbar design="Transparent">
    <content>
        <Button icon="sap-icon://save" press="onSaveSalesOrder"/>
        <Button icon="sap-icon://sys-cancel-2" press="onCancelSalesOrder"/>
    </content>
</Toolbar>
<form:SimpleForm id="mySimpleForm" binding="{path: '/SalesOrderList(ID=\'42\')',
$$updateGroupId: 'SalesOrderUpdateGroup'}">
    <Label text="Sales Order ID" />
    <Text text="{SalesOrderID}" />
    <Label labelFor="Note" text="Note" />
    <Input id="Note" value="{Note}" />
</form:SimpleForm>
```

Example: Controller

```
onCancelSalesOrder : function (oEvent) {
    this.getView().getModel().resetChanges("SalesOrderUpdateGroup");
},

onSaveSalesOrder : function (oEvent) {
    var that = this;

this.getView().getModel().submitBatch("SalesOrderUpdateGroup").then(function(){
        if (!
that.byId("mySimpleForm").getBindingContext().getBinding().hasPendingChanges()){
            // raise success message
        }
    });
},
```

Repeating Property Changes

The OData V4 model automatically repeats failed property changes (PATCH requests). If the update group ID has SubmitMode.API and the property change of the entity on the server fails, the change is repeated with the next call of ODataModel.submitBatch for this group. If the update group ID has SubmitMode.Auto or

SubmitMode.Direct and the change fails, the change is repeated automatically with the next update for the entity. Since 1.67.0, ODataModel.submitBatch can also be used for update group IDs with SubmitMode.Auto in order to repeat, independently of an update.

The same holds true for Creating an Entity [page 974].

Define submit mode for an application group ID

On construction of the model, it is possible to specify the submit mode for application group IDs. This is useful when you want to separate requests requiring short processing time on the server from those requiring long processing time, so that responses to "fast" requests are visible earlier on the UI.

The following example shows how to set the submit mode `sap.ui.model.odata.v4.SubmitMode.Auto` for the group IDs `fastGroup` and `slowGroup` in the manifest.

Example

Specify the submit mode for an application group in manifest.json

```
"models" : {
    "" : {
        "dataSource" : "default",
        "settings" : {
            "operationMode" : "Server",
            "synchronizationMode" : "None",
            "groupProperties" : {
                "fastGroup" : {"submit" : "Auto"},
                "slowGroup" : {"submit" : "Auto"}
            }
        }
    }
```

Related Information

ODataModel.submitBatch
ODataModel.bindList
ODataModel.bindContext
ODataModel.bindProperty
ODataContextBinding.execute
ODataContextBinding.refresh
ODataListBinding.refresh
ODataPropertyBinding.refresh
ODataPropertyBinding.setValue
OData V4 Specification Part 1, Batch Requests

Meta Model for OData V4

Each OData V4 model offers access via `getMetaModel` to a corresponding metadata model `sap.ui.model.odata.v4.ODataMetaModel`, which is read-only and offers access to OData V4 metadata in a streamlined JSON format (see links under Related Information for more details). Only one-time bindings are supported by this model because the metadata is immutable.

Synchronous vs. Asynchronous Access

Access to metadata is basically asynchronous (e.g. `requestObject`) to allow for dynamic loading of metadata. There is also a corresponding method for synchronous access (e.g. `getObject`) which returns `undefined` if metadata is not yet available. It should only be used in situations where metadata has already been loaded asynchronously before. Loading happens individually for each document, i.e. each $metadata document is loaded and processed as a whole and is available thereafter. Includes and references to other $metadata documents are not supported, only the service root's initial $metadata document can be used.

Path Syntax

The `requestObject` API documentation in the Demo Kit explains how metadata is accessed and the supported path syntax in great detail. The basic idea is that every path described in the specification OData Version 4.0 Part 3: Common Schema Definition Language, 14.2.1 Attribute Target is a valid absolute path within the metadata model if a leading slash is added; for example `"/" + "MySchema.MyEntityContainer/MyEntitySet/MyComplexProperty/MyNavigationProperty"`. For more information, see the requestObject API documentation in the Demo Kit.

Annotations

The main API for both programmatic access from JavaScript and declarative access from XML templating is `sap.ui.model.odata.v4.ODataMetaModel#getObject`. It works together with `sap.ui.model.odata.v4.ODataMetaModel#resolve` (for `<template:with>`) and `sap.ui.model.odata.v4.ODataMetaModel#bindList` (for `<template:repeat>`) in order to provide convenient access to annotations, inline as well as external targeting.

The OData meta model knows how to follow "14.2.1 Attribute Target" described in specification "OData Version 4.0 Part 3: Common Schema Definition Language " as well as "14.5.2 Expression edm:AnnotationPath", "14.5.11 Expression edm:NavigationPropertyPath", "14.5.12 Expression edm:Path", and "14.5.13 Expression edm:PropertyPath".

XML Templating [page 1018] still works the same as for V2, with some slight changes as outlined below:

- Metadata paths need to refer to the V4 metadata JSON structure.

- Note the difference between `"/TEAMS@Org.OData.Capabilities.V1.TopSupported"` and `"/TEAMS/@com.sap.vocabularies.Common.v1.Deletable"` (look closely at the slash!), see ODataMetaModel.requestObject.
- Use `sap.ui.model.odata.v4.AnnotationHelper` instead of `sap.ui.model.odata.AnnotationHelper`. The ability to follow a path has been built into the V4 OData meta model itself. See `field>Value/$Path@com.sap.vocabularies.Common.v1.Label` in the code example below. Instead of `sap.ui.model.odata.AnnotationHelper.format`, you can use `sap.ui.model.odata.v4.AnnotationHelper.value` or `sap.ui.model.odata.v4.AnnotationHelper.format`. You can use both as a computed annotation.
- Computed annotations start with "@@", for example `<Text text="{meta>Value/@@sap.ui.model.odata.v4.AnnotationHelper.value}" />`. Their name without the "@@" prefix refers to a function in the global namespace which computes an annotation value from the metadata addressed by the preceding path. For more information, see ODataMetaModel.requestObject .
- Ensure that the view is loaded asynchronously. In this case, there is no longer a need to preload metadata, because the template processor waits for every binding to be resolved before proceeding.
- Use a double hash ('##') or single hash ('#') separator to branch from the OData V4 model into metadata, see createBindingContext .

> **i Note**
>
> The single hash separator is deprecated since 1.52

> **→ Remember**
>
> An appropriate URI encoding is necessary for the data path (before the separator), but neither for the separator itself nor for the metadata path that follows it.

Example: `<template:with path="/Products('A%2FB%26C')/Name#@com.sap.vocabularies.Common.v1.Label" var="label">` or `<template:with path="data>/Products#/" var="productEntityType">`, etc.

Example of an OData V4 XML template:

```
<mvc:View
        template:require="{AnnotationHelper : 'sap/ui/model/odata/v4/
AnnotationHelper'}"
        xmlns="sap.m"
        xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
    <template:alias name="format" value="AnnotationHelper.format">
    <template:alias name="value" value="AnnotationHelper.value">
        <template:with path="meta>/BusinessPartnerList/" var="entityType">
          <template:with path="entityType>@com.sap.vocabularies.UI.v1.LineItem"
var="lineItem">
            <Table headerText="Business Partners"
              items="{path : '/BusinessPartnerList', length : 5}">
              <columns>
                <template:repeat list="{lineItem>}" var="field">
                  <Column>
                    <template:if test="{field>Label}">
                      <template:then>
                        <Label design="{:= $
{field>@com.sap.vocabularies.UI.v1.Importance/$EnumMember}
                          === 'com.sap.vocabularies.UI.v1.ImportanceType/High' ?
'Bold' : 'Standard'}"
                          text="{field>Label}"/>
                      </template:then>
```

```
                      <template:else>
                        <Text text="{field>Value/
$Path@com.sap.vocabularies.Common.v1.Label}"/>
                      </template:else>
                    </template:if>
                  </Column>
                </template:repeat>
              </columns>
              <items>
                <ColumnListItem>
                  <cells>
                    <template:repeat list="{lineItem>}" var="field">
                      <template:with path="field>Value/$Path" var="target">
                        <template:if test="{= $
{target>@@AnnotationHelper.getValueListType} === 'Standard' }">
                            <template:then>
                                <Input value="{path : 'field>Value/@@value'}"
showValueHelp="true", valueHelpRequest=".onValueHelp"}" />
                            </template:then>
                            <template:elseif test="{= $
{target>@@AnnotationHelper.getValueListType} === 'Fixed' }">
                                <ComboBox value="{path : 'field>Value/@@value'}"
loadItems=".onLoadItems" showValueHelp="true" />
                            </template:elseif>
                            <template:elseif
test="{target>@com.sap.vocabularies.Common.v1.Text}">
                                <!-- Note: TextFirst, TextLast, TextSeparate,
TextOnly -->
                                <template:if test="{= $
{target>@com.sap.vocabularies.Common.v1.Text@com.sap.vocabularies.UI.v1.TextArran
gement/$EnumMember}
                                    ===
'com.sap.vocabularies.UI.v1.TextArrangementType/TextLast' }">
                                    <!-- Text: "A descriptive text for values of
the annotated property.
                                        Value MUST be a dynamic expression when
used as metadata annotation." -->
                                    <Text text="{field>Value/@@value}
{target>@com.sap.vocabularies.Common.v1.Text/@@value}" />
                                </template:if>
                            </template:elseif>
                            <template:else>
                                <Text text="{field>Value/@@format}" />
                            </template:else>
                        </template:if>
                      </template:with>
                    </template:repeat>
                  </cells>
                </ColumnListItem>
              </items>
            </Table>
          </template:with>
        </template:with>
    </template:alias>
    </template:alias>
</mvc:View>
```

AnnotationHelper

The module `sap/ui/model/odata/v4/AnnotationHelper` delivers the following computed annotations; require it as shown in the example above:

- `value` helps to convert annotations into corresponding expression bindings or similar. The resulting binding does **not** contain type and constraint information; both are detected automatically. For examples, see sap.ui.model.odata.v4.AnnotationHelper.value.
- `format` helps to convert annotations into corresponding expression bindings, or similar. Compared to `value`, `format` adds type and constraints information to the resulting binding. This is useful, for example, if the XML of the view is cached.
  If you use `format` with a path containing a single "$AnnotationPath" or "$Path" segment, the value corresponding to that segment is considered as a data binding path prefix whenever a dynamic "14.5.12 Expression edm:Path" or "14.5.13 Expression edm:PropertyPath" is turned into a data binding.
  For examples, see `format`.
  If `format` finds a `Org.OData.Measures.V1.ISOCurrency` or a `Org.OData.Measures.V1.Unit` annotation at a property, a composite binding with a `sap.ui.model.odata.type.Currency` or a `sap.ui.model.odata.type.Unit` type is generated. For more information, see Currencies and Units [page 986].
- `label` - Returns the value for the label of a `com.sap.vocabularies.UI.v1.DataFieldAbstract` from the meta model.
- `getValueListType` - Determines which type of value list exists for the property. The function returns a value from the enumeration `sap.ui.model.odata.v4.ValueListType`. It can be called directly on a property:

```
<template:with path="/BusinessPartnerList/Role" var="property">
    <template:if test="{= ${property>@@AnnotationHelper.getValueListType} ===
'Fixed'}">
        ...
    </template:if>
</template:with>
```

Alternatively it can be called on an annotation holding an `edm:Path` to a property when it is called in the context of an entity type. This is typically the case when iterating over a `com.sap.vocabularies.UI.v1.LineItem` annotation of an entity type and asking for value help on the data fields. See the example regarding `LineItem` of `BusinessPartnerList` (the relevant parts are repeated here):

```
<template:with path="meta>/BusinessPartnerList/" var="entityType">
  <template:with path="entityType>@com.sap.vocabularies.UI.v1.LineItem"
var="lineItem">
...
            <template:repeat list="{lineItem>}" var="field">
              <template:with path="field>Value/$Path" var="target">
                <template:if test="{= $
{target>@@AnnotationHelper.getValueListType} === 'Standard' }">
...
```

The first `<template:with>` defines `entityType` to be the type of the set `BusinessPartnerList`. The `<template:repeat>` iterates over its annotation `com.sap.vocabularies.UI.v1.LineItem` (a collection of records with type `com.sap.vocabularies.UI.v1.DataField`). The record's property `Value` is assumed to be an `edm:Path` pointing to a property of the entity type. For this path the value list type is determined.

Related Information

OData V4 Metadata JSON Format [page 960]

getMetaModel
sap.ui.model.odata.v4.ODataMetaModel
requestObject
sap.ui.model.odata.v4.ODataMetaModel#getObject
sap.ui.model.odata.v4.ODataMetaModel#bindList
sap.ui.model.odata.ODataMetaModel#loaded

OData V4 Metadata JSON Format

The OData V4 model provides access to metadata in a streamlined JSON format which is described in the section below.

It is different to the $metadata service's JSON format (see OData JSON Format Version 4.0) and the OData JSON Format for Common Schema Definition Language (CSDL) Version 4.0 (see corresponding specification), intended to simplify client-side processing.

In the sections below, angled brackets indicate variable parts. The numbers next to each expression correspond to the numbered sections in the official specification, see OData Version 4.0 Part 3: Common Schema Definition Language (CSDL) Plus Errata 03 . Comments highlight optional properties, especially those that have certain default values.

Design Rationale

We have prefixed constant property names with `"$"` as this is a legal first character for JavaScript identifiers, but not for OData simple identifiers. This way, **inline annotations** can be added via `"@<14.3.1 Annotation Term>#<14.3.2 Annotation Qualifier>" : <value>` everywhere without resulting in any naming conflicts. This is shown as `"@..." : <value>` below.

We assume that schema **aliases** have been resolved. We add a trailing dot after a schema's namespace, meaning qualified name "A.B" cannot clash with schema namespace "A.B.", for example. This trailing dot is also present for `"$Include"`, `"$TermNamespace"` and `"$TargetNamespace"` values.

`$kind` has been added to each object with a (qualified) OData name and to almost each object which can be annotated via external targeting, but not to enum members. Actions and functions are arrays of overloads and `$kind` has been added to each overload.

We assume each **enum member** has a value via the fallback rule "If no values are specified, the members are assigned consecutive integer values in the order of their appearance, starting with zero for the first member."

Facets like `MaxLength`, `Precision` and `Scale` are represented as numbers if possible (`"$Scale" : "variable"` is the only exception). `DefaultValue` is represented as a string for lack of type information in the general case. `"$MaxLength" : "max"` is omitted and will be treated the same as an unspecified length on the client-side.

A "17.5 **TargetPath**" used as "13.4.1 Attribute Path" or "13.5.3/13.6.3 Attribute EntitySet" is normalized in the following sense: a simple identifier is used instead of a target path for entity sets (or singletons) within the same container.

Each annotation specifies a value. Accordingly, `$DefaultValue` has been omitted for the time being.

Normalization: For all EDM elements which allow both inline annotations and external targeting, only external targeting is used. This affects edm:ActionImport, edm:ComplexType, edm:EntityContainer, edm:EntitySet, edm:EntityType, edm:EnumType, edm:FunctionImport, edm:Member, edm:Singleton, edm:Term, edm:TypeDefinition, edm:NavigationProperty, edm:Property. The goal is to reduce cases that contain a mixture of inline annotations and external targeting to the bare minimum. External targeting is possible for edm:Action, edm:Function, edm:Parameter, and edm:ReturnType via 4.01 style annotation targets, either in a way that applies to all overloads of the action or function or all parameters of that name across all overloads, or in a way that identifies a single overload.

We use the `"<key>@<14.3.1 Annotation Term>#<14.3.2 Annotation Qualifier>" : <value>` syntax for inline annotations in the following cases to avoid explicit object representations:

- "7.2 Element ReferentialConstraint" with key `"<7.2.1 ReferentialConstraint Property>"`
- "7.3 Element OnDelete" with key `"$OnDelete"`
- "14.3 Annotation" with key `"@<14.3.1 Annotation Term>#<14.3.2 Annotation Qualifier>"` (yes, this does lead to a double at-sign `"@...#...@...#..."`)
- "14.5.14.2 Element PropertyValue" with key `"<14.5.14.2.1 PropertyValue Property>"`

Metadata JSON Structure

The following JSON file represents the metadata document which corresponds to `GET <serviceRoot>/$metadata`:

```
{
  "$Version" : "<3.1.1 Edmx Version>",
  "$Annotations" : {
    "<14.2.1 Annotations Target>" : {
      // Note: "<14.3.2 Annotation Qualifier>" defaults to "<14.2.2 Annotations
Qualifier>",
      // qualifiers are optional, "#" is omitted then
      "@<14.3.1 Annotation Term>#<14.3.2 Annotation Qualifier>" : <value> //
constant or dynamic expression
      "@<14.3.1 Annotation Term>#<14.3.2 Annotation Qualifier>@..." : <value> //
annotation of an annotation
    }
  },
  "$EntityContainer" : "<5.1.1 Schema Namespace>.<13.1.1 EntityContainer
Name>", // root entity container for this $metadata document
  "$Reference" : {
    // server-relative, dereferencable URLs (to $metadata) only!
    "<3.3.1 Reference Uri>" : { "@..." : <value>,
      "$Include" : ["<3.4.1 Include Namespace>.", ...], // optional
      "$IncludeAnnotations" : [{
        "$TermNamespace" : "<3.5.1 IncludeAnnotations TermNamespace>.",
        "$Qualifier" : "<3.5.2 IncludeAnnotations Qualifier>", // optional
        "$TargetNamespace" : "<3.5.3 IncludeAnnotations TargetNamespace>." //
optional
      }, ...] // optional
    }
  }, // optional
  "<5.1.1 Schema Namespace>" : {
    "$kind" : "Schema",
    "@..." : <value> // place inline annotations for schema itself here!
  },
  "<5.1.1 Schema Namespace>.<8.1.1 EntityType Name>" : {
```

```
    "$kind" : "EntityType",
    "$BaseType" : "<8.1.2 EntityType BaseType>", // optional
    "$Abstract" : true, // omit in case of default value: false
    "$OpenType" : true, // omit in case of default value: false
    "$HasStream" : true, // omit in case of default value: false
    "$Key" : [
      "<8.3.1 PropertyRef Name>", // in case no Alias is given
      {"<8.3.1 PropertyRef Alias>" : "<8.3.1 PropertyRef Name>"},
      ...
    ], // optional
    "<6.1.1 Property Name>" : {
      "$kind" : "Property",
      "$Type" : "<6.1.2 Property Type>",
      "$isCollection" : true, // omit in case of default value: false
      "$Nullable" : false, // omit in case of default value: true
      "$MaxLength" : <6.2.2 MaxLength>, // optional, number
      "$Precision" : <6.2.3 Precision>, // optional, number
      "$Scale" : <6.2.4 Scale> | "variable", // optional, number or fixed string
      "$Unicode" : false, // omit in case of default value: true
      "$SRID" : "<6.2.6 SRID>", // optional
      "$DefaultValue" : "<6.2.7 DefaultValue>" // optional
    },
    "<7.1.1 NavigationProperty Name>" : {
      "$kind" : "NavigationProperty",
      "$isCollection" : true, // omit in case of default value: false
      "$Type" : "<7.1.2 NavigationProperty Type>",
      "$Nullable" : false, // omit in case of default value: true
      "$Partner" : "<7.1.4 NavigationProperty Partner>", // optional
      "$ContainsTarget" : true, // omit in case of default value: false
      "$ReferentialConstraint" : {
        "<7.2.1 ReferentialConstraint Property>" : "<7.2.2 ReferentialConstraint
ReferencedProperty>"
      }, // optional
      "$OnDelete" : "<7.3.1. OnDelete Action>" // optional
    },
  },
  "<5.1.1 Schema Namespace>.<9.1.1 ComplexType Name>" : {
    "$kind" : "ComplexType",
    "$BaseType" : "<9.1.2 ComplexType BaseType>", // optional
    "$Abstract" : true, // omit in case of default value: false
    "$OpenType" : true, // omit in case of default value: false
    "<6.1.1 Property Name>" : {
      // see above
    },
    "<7.1.1 NavigationProperty Name>" : {
      // see above
    }
  },
  "<5.1.1 Schema Namespace>.<10.1.1 EnumType Name>" : {
    "$kind" : "EnumType",
    "$UnderlyingType" : "<10.1.2 EnumType UnderlyingType>", // omit in case of
default value: Edm.Int32
    "$IsFlags" : true, // omit in case of default value: false
    "<10.2.1 Member Name>" : "<10.2.2 Member Value>" // use string value in case
of base type Edm.Int64, else number
  },
  "<5.1.1 Schema Namespace>.<11.1.1 TypeDefinition Name>" : {
    "$kind" : "TypeDefinition",
    "$UnderlyingType" : "<11.1.2 TypeDefinition UnderlyingType>",
    "$MaxLength" : <11.1.3 MaxLength>, // optional, number
    "$Precision" : <11.1.3 Precision>, // optional, number
    "$Scale" : <11.1.3 Scale> | "variable", // optional, number or fixed string
    "$Unicode" : false, // omit in case of default value: true
    "$SRID" : "<11.1.3 SRID>" // optional
  },
  "<5.1.1 Schema Namespace>.<12.1.1 Action Name>" : [{
    "$kind" : "Action",
    "$IsBound" : true, // omit in case of default value: false
```

```
    "$EntitySetPath" : "<12.1.3 Action EntitySetPath>", // optional
    "$Parameter" : [{
      "$Name" : "<12.4.1 Parameter Name>",
      "$isCollection" : true, // omit in case of default value: false
      "$Type" : "<12.4.2 Parameter Type>",
      "$Nullable" : false, // omit in case of default value: true
      "$MaxLength" : <12.4.4 MaxLength>, // optional, number
      "$Precision" : <12.4.4 Precision>, // optional, number
      "$Scale" : <12.4.4 Scale> | "variable", // optional, number or fixed string
      "$SRID" : "<12.4.4 SRID>" // optional
    }, ...], // optional
    "$ReturnType" : {
      "$isCollection" : true, // omit in case of default value: false
      "$Type" : "<12.3.1 ReturnType Type>",
      "$Nullable" : false, // omit in case of default value: true
      "$MaxLength" : <11.1.3 MaxLength>, // optional, number
      "$Precision" : <11.1.3 Precision>, // optional, number
      "$Scale" : <11.1.3 Scale> | "variable", // optional, number or fixed string
      "$SRID" : "<11.1.3 SRID>" // optional
    } // optional
  }, ...],
  "<5.1.1 Schema Namespace>.<12.2.1 Function Name>" : [{
    "$kind" : "Function",
    "$IsBound" : true, // omit in case of default value: false
    "$IsComposable" : true, // omit in case of default value: false
    "$EntitySetPath" : "<12.2.4 Function EntitySetPath>", // optional
    "$Parameter" : [{
      // see above
    }, ...], // optional
    "$ReturnType" : {
      // see above
    }
  }, ...],
  "<5.1.1 Schema Namespace>.<13.1.1 EntityContainer Name>" : {
    "$kind" : "EntityContainer"
//      "$Extends" : "<13.1.2 EntityContainer Extends>", // not in the 1st step
    "<13.2.1 EntitySet Name>" : {
      "$kind" : "EntitySet",
      "$Type" : "<13.2.2 EntitySet EntityType>", // Note: renamed for
consistency!
      "$IncludeInServiceDocument" : false, // omit in case of default value: true
      "$NavigationPropertyBinding" : {
        "<13.4.1 NavigationPropertyBinding Path>" : "<13.4.2
NavigationPropertyBinding Target>" // normalized
      } // optional
    },
    "<13.3.1 Singleton Name>" : {
      "$kind" : "Singleton",
      "$Type" : "<13.3.2 Singleton Type>",
      "$NavigationPropertyBinding" : {
        "<13.4.1 NavigationPropertyBinding Path>" : "<13.4.2
NavigationPropertyBinding Target>" // normalized
      } // optional
    },
    "<13.5.1 ActionImport Name>" : {
      "$kind" : "ActionImport",
      "$Action" : "<13.5.2 ActionImport Action>",
      "$EntitySet" : "<13.5.3 ActionImport EntitySet>" // optional, normalized
    },
    "<13.6.1 FunctionImport Name>" : {
      "$kind" : "FunctionImport",
      "$Function" : "<13.6.2 FunctionImport Function>",
      "$EntitySet" : "<13.6.3 FunctionImport EntitySet>", // optional, normalized
      "$IncludeInServiceDocument" : true // omit in case of default value: false
    }
  },
  "<5.1.1 Schema Namespace>.<14.1.1 Term Name>" : {
    "$kind" : "Term",
```

```
    "$isCollection" : true, // omit in case of default value: false
    "$Type" : "<14.1.2 Term Type>",
    "$BaseTerm" : "<14.1.3 Term BaseTerm>", // optional
//    "$DefaultValue" : "<14.1.4 Term DefaultValue>", // omit in case of default
value: null
//    "$AppliesTo" : "<14.1.5 Term AppliesTo>", // JSON clients need not validate
    "$Nullable" : false, // omit in case of default value: true
    "$MaxLength" : <14.1.6 MaxLength>, // optional, number
    "$Precision" : <14.1.6 Precision>, // optional, number
    "$Scale" : <14.1.6 Scale> | "variable", // optional, number or fixed string
    "$SRID" : "<14.1.6 SRID>" // optional
  }
}
```

Constant and Dynamic Expressions

Constant and dynamic expressions are used as values for annotations. Their JSON representation is shown in the following two tables.

Table 32: Constant Expressions

Expression	Options	Additional Information

14.4 Constant Expressions	`{"$Binary" : "T0RhdGE"}`
	`{"$Date" : "2000-01-01"}`
	`{"$DateTimeOffset" : "2000-01-01T16:00:00.000-09:00"}`
	`{"$Decimal" : "3.14"}`
	`{"$Duration" : "P11D23H59M59.999999999999S"}`
	`{"$Guid" : "21EC2020-3AEA-1069-A2DD-08002B30309D"}`
	`{"$TimeOfDay" : "21:45:00"}`
		"Binary", "Date", "DateTimeOffset", "Decimal", "Duration", "Guid", "TimeOfDay" are objects with a single property that has a string value.

14.4.2 Expression `Bool`	`false` `true`	Is represented by the JavaScript boolean literals.
14.4.7 Expression `EnumMember`	`{"$EnumMember" : 42}` `{"$EnumMember" : "1234567890123456789"}`	Is represented like above object notation, but with a JavaScript number literal as long as the value is a safe integer, else with a string value.

Expression	Options	Additional Information

14.4.8 Expression `Float`	`3.1415926535`
	`{"$Float" : "-INF"}`
	`{"$Float" : "INF"}`
	`{"$Float" : "NaN"}`
		Is represented by a JavaScript number literal (except for the `nanInfinity` ABNF rule which needs an object notation with a string value).

14.4.10 Expression `Int`	`42` `{"$Int" : "1234567890123456789"}`	Is represented by a JavaScript number literal as long as the value is a safe integer. Else the above object notation is used.
14.4.11 Expression `String`	`"Product Catalog"`	Is represented by a JavaScript string literal.

Table 33: Dynamic Expressions

Expression	Options and Additional Information
14.5.1 Comparison and Logical Operators	edm:Not is written as `{"$Not" : <value>, "@..." : <value>}`. All others are written like `{"$And" : [<value>, <value>], "@..." : <value>}` because they require two child expressions.
14.5.2 Expression `AnnotationPath`	`{"$AnnotationPath" : "..."}`
14.5.3 Expression `Apply`	`{"$Apply" : [<value>, ...], "$Function" : "<14.5.3.1 Apply Function>", "@..." : <value>}`

14.5.4 Expression `Cast`	`{`
	`"$Cast" : <value>,`
	`"$isCollection" : true, // omit in case of default value: false`
	`"$Type" : "<14.5.4.1 Cast Type>",`
	`"$MaxLength" : <6.2.2 MaxLength>, // optional, number`
	`"$Precision" : <6.2.3 Precision>, // optional, number`
	`"$Scale" : <6.2.4 Scale> |`
	`"variable", // optional, number or fixed string`
	`"$SRID" : "<6.2.6 SRID>", // optional`
	`"@..." : <value>`
	`}`

14.5.5 Expression `Collection`	`[<value>, ...]` Simply an array. No additional properties, no annotations possible.

Expression	Options and Additional Information
14.5.6 Expression `If`	`{"$If" : [<value>, <value>, <value>], "@..." : <value>}` Condition, then, else (which is optional inside a "14.5.5 Expression Collection" only).
14.5.7 Expression `IsOf`	`{ "$IsOf" : <value>, "$isCollection" : true, // omit in case of default value: false "$Type" : "<14.5.7.1 IsOf Type>", "$MaxLength" : <6.2.2 MaxLength>, // optional, number "$Precision" : <6.2.3 Precision>, // optional, number "$Scale" : <6.2.4 Scale> | "variable", // optional, number or fixed string "$SRID" : "<6.2.6 SRID>", // optional "@..." : <value> }`
14.5.8 Expression `LabeledElement`	`{"$LabeledElement" : <value>, "$Name" : "<5.1.1 Schema Namespace>.<14.5.8.1 LabeledElement Name>", "@..." : <value>}`
14.5.9 Expression `LabeledElementReference`	`{"$LabeledElementReference" : "<QualifiedName name of a labeled element expression in scope>"}`

14.5.10 Expression `Null`	`null`
	`{"$Null" : null, "@..." : <value>}`
	The object notation is needed in case of inline annotations.

14.5.11 Expression `NavigationPropertyPath`	`{"$NavigationPropertyPath" : "..."}`
14.5.12 Expression `Path`	`{"$Path" : "..."}`
14.5.13 Expression `PropertyPath`	`{"$PropertyPath" : "..."}`
14.5.14 Expression `Record`	The record itself is a map: `{ "$Type" : "<14.5.14.1 Record Type>", "<14.5.14.2.1 PropertyValue Property>" : <value>, "@..." : <value> }`
14.5.15 Expression `UrlRef`	`{"$UrlRef" : <value>, "@..." : <value>}`

Related Information

OData JSON Format Version 4.0
OData JSON Format for Common Schema Definition Language (CSDL) Version 4.0
OData Version 4.0 Part 3: Common Schema Definition Language (CSDL) Plus Errata 02

Performance Aspects

The OData V4 model offers the features described below which influence performance.

`odata.metadata=minimal`

The OData V4 model uses an `odata.metadata=minimal` header in its requests to reduce the amount of data that is sent from server to client. For more information, see section "3.1.1 odata.metadata=minimal" in the OData JSON Format Version 4.0 specification.

`$expand` and `$select`

An application can either specify `$expand` and `$select` parameters to read all data to be displayed in one request, or create bindings dynamically to load only part of the data with one request per binding. The application needs to decide whether to have less roundtrips with a bigger payload or more roundtrips with smaller payload.

To reduce payload, applications should only select properties that are needed using `$select` (see the code sample in the Parameters [page 928] topic). Besides the properties needed on the UI, the binding must select key properties to support features such as read requests sent from a child binding, write requests, or bound operations on the respective entity.

Batching Requests or Not

By default, the OData V4 model collects all requests made to the OData service in a batch request to reduce the number of roundtrips. The disadvantage of a batch request is that it cannot be cached by the browser. If some of the requests (e.g. value help requests) are "cacheable", it is a good idea to request these resources directly and use the browser cache to improve the performance of the application. In such cases, use the `$direct` group as described in the section Batch Control [page 952].

Binding Caches

Absolute bindings and also relative bindings, which fulfill certain conditions, have a cache that is used for their data. Once data is read, all value requests (e.g. those made by dependent relative bindings) are served by this cached data. `ODataListBinding` additionally supports paging. For more information about this, see Bindings [page 922].

Calling `refresh` on an absolute binding clears its cache as well as the caches of its relative child bindings. Calling `refresh` on the model refreshes all bindings that have been created by that model.

> → Tip
>
> Relative bindings have an own cache store cache by context. When you change the context of the relative binding and then switch back to the previous context, the latter context change will not lead to a data request as the cache holding this data is reused. This is useful for master-detail scenarios where selection in the master leads to setting the corresponding context in the relative binding for the detail. Entries in the master that have already been selected won't lead to a data request.
>
> **Note:** For the above mechanism to work, you must not recreate a relative binding as you then lose the cache for the previously selected contexts. Keep the binding and just set its context.

Early requests for metadata and security token

The requests for the service's root `$metadata` document and annotation files and for the security token may be on the "critical execution path": By default, these requests are sent lazily when the SAPUI5 application starts, for example. only when the corresponding information is needed. This delays application startup until these requests have returned.

If you construct the model with parameter `earlyRequests`, the requests are sent as early as possible and application startup performance may improve.

> i Note
>
> Modern browsers typically can process up to six parallel requests. Therefore it strongly depends on the number of requests sent initially by the application, if and how much the performance improves.

> → Remember
>
> The default value for `earlyRequests` is false in SAPUI5 version 1.54. It may, however, change to true in later releases: Do not rely on the default value and explicitly set it to false to ensure the requests are not sent early.

Related Information

OData JSON Format Version 4.0
Bindings [page 922]

Batch Control [page 952]

Unsupported Superclass Methods and Events

Certain methods derived from SAPUI5 model and binding superclasses are not supported in OData V4 model classes or have limited support.

The following methods and events are affected by this. For more information, see the corresponding API documentation for each method and event in the Demo Kit.

Table 34: Unsupported Methods

Class	Method

`sap.ui.model.odata.v4.ODataMetaModel`
(See sap.ui.model.odata.v4.ODataMetaModel in the Demo Kit)
	`bindTree`
	`getOriginalProperty`
	`isList`
	`refresh`
	`setLegacySyntax`

`sap.ui.model.odata.v4.ODataModel`
(See sap.ui.model.odata.v4.ODataModel in the Demo Kit)
	`bindTree`
	`destroyBindingContext`
	`getObject`
	`getOriginalProperty`
	`getProperty`
	`isList`
	`setLegacySyntax`

`sap.ui.model.odata.v4.ODataContextBinding` (See sap.ui.model.odata.v4.ODataContextBinding in the Demo Kit)	`isInitial` `refresh` (limited support only) `resume` (limited support only) `suspend` (limited support only)

`sap.ui.model.odata.v4.ODataListBinding`
(See sap.ui.model.odata.v4.ODataListBinding in the Demo Kit)
	`getDistinctValues`
	`isInitial`
	`refresh` (limited support only)
	`resume` (limited support only)
	`suspend` (limited support only)

Class	Method

`sap.ui.model.odata.v4.ODataPropertyBinding`
(See sap.ui.model.odata.v4.ODataPropertyBinding in the Demo Kit)
	`isInitial`
	`refresh` (limited support only)
	`resume`
	`setValue` (limited support only)
	`suspend`

Table 35: Unsupported Events

Class	Event

`sap.ui.model.odata.v4.ODataMetaModel`
(See sap.ui.model.odata.v4.ODataMetaModel in the Demo Kit)
	`parseError`
	`propertyChange`
	`requestCompleted`
	`requestFailed`
	`requestSent`

`sap.ui.model.odata.v4.ODataModel`
(See sap.ui.model.odata.v4.ODataModel in the Demo Kit)
	`parseError`
	`propertyChange`
	`requestCompleted`
	`requestFailed`
	`requestSent`

Related Information

sap.ui.model.odata.v4.ODataMetaModel
sap.ui.model.odata.v4.ODataModel
sap.ui.model.odata.v4.ODataContextBinding
sap.ui.model.odata.v4.ODataListBinding
sap.ui.model.odata.v4.ODataPropertyBinding

Changes Compared to OData V2 Model

This section outlines the main differences between the OData V2 and OData V4 models.

While some of the differences between the OData V4 model and the OData V2 model are due to features that have not yet been implemented, many differences are due to the following:

- Protocol incompatibility between OData V4 and OData V2
- API cleanup and simplification
- Adherence to OData V4 standards regarding the names and terms used in APIs

These differences will therefore remain even after all features have been implemented. The table below gives you an overview of these changes, as well as the reason behind them and (if applicable) how the OData V2 model mechanism is supported in the OData V4 model.

Change	Reason
Binding parameter names: The binding parameter name for an OData system query option is identical to the system query option name: `$expand`, `$select`, ... (V2 uses `expand`, `select`).	Simplification: The OData V4 model simplifies the binding parameter structure to just one map where all entries in the map are OData query options, with the exception of entries that have a key starting with "$$" (binding-specific parameters). In all cases, the names of the binding parameters are exactly the same as in the OData URL sent to the server.
The model does not support the methods `getData`, `getObject`, `getOriginalProperty`, `getProperty`. For data access, use the context API instead of methods on the model.	OData requires asynchronous data retrieval: Synchronous data access requires that data has already been loaded from the server. This means there is no way of knowing whether this already happened, meaning the result of a synchronous access method is quite often unpredictable. The OData V4 context API offers ansynchronous and synchronous access to the data of a specific context. It is no longer necessary to construct a path for data access as needed by the methods on the model. For more information, see the section *Context API* in Bindings [page 922].
Minimize APIs required for batch control: Model does not support the methods `getChangeBatchGroups`, `getChangeGroups`, `getDeferredGroups`, `setChangeBatchGroups`, `setChangeGroups`, `setDeferredBatchGroups`, `setDeferredGroups`, `setUseBatch` (and corresponding model construction parameters).	Simplification: Batch groups are solely defined via binding parameters with the corresponding parameters on the model as default. Application groups are by default deferred; there is no need to set or get deferred groups. You just need the `submitBatch` method on the model to control execution of the batch. You can use the predefined batch group `"$direct"` to switch off batch either for the complete model or for a specific binding (only possible for the complete model in V2). For more information, see Batch Control [page 952].
OData operations executed via binding: Model does not support the method `callFunction`.	Simplification: Use an operation binding instead; it is now much easier to bind operation execution results to controls.

Change	Reason
No CRUD methods on model: Model does not support the methods `create`, `read`, `remove`, `update`.	Simplification: `read`, `update`, `create` and `remove` operations are available implicitly via the bindings. Bindings can also be used without controls. It is not possible to trigger requests for specific OData URLs. For more information, see Accessing Data in Controller Code [page 935].
No metadata access via model: Model does not support methods `getServiceAnnotations`, `getServiceMetadata`, `refreshMetadata` as well as methods corresponding to the events `metadataFailed`, `metadataLoaded`.	Simplification: Metadata is only accessed via `ODataMetaModel`. Metadata is only loaded when needed (e.g. for type detection or to compute URLs for write requests); the corresponding methods on the `v4.ODataMetaModel` use promises instead of events.
sap.ui.model.odata.AnnotationHelper is not supported for OData V4.	Simplification: Much of the functionality in sap.ui.model.odata.AnnotationHelper is provided by sap.ui.model.odata.v4.ODataMetaModel and sap.ui.model.odata.v4.ODataModel. **Example** The path syntax supported by the v4.ODataMetaModel, see sap.ui.model.odata.v4.ODataMetaModel, method `requestObject` allows for navigation in the model's metadata; there is no need to use `AnnotationHelper` methods for this. You can find the remaining functionality in the OData V4 specific sap.ui.model.odata.v4.AnnotationHelper.

Related Information

sap.ui.model.odata.AnnotationHelper
sap.ui.model.odata.v4.ODataMetaModel
sap.ui.model.odata.v4.ODataModel

Additional Annotation Files

The OData V4 model supports loading of additional annotation files.

The annotation files have to be given during creation of an `ODataModel` instance. Adding annotation files at a later point in time is **not** supported.

The format of the annotation file has to be the same as the metadata file of the service. Only XML files are supported. You can specify the annotation files in the descriptor for applications, components, and libraries (`manifest.json`).

Annotation files are specified in `manifest.json` as follows:

```
"dataSources" : {
    "default" : {
        "uri" : "/sap/opu/odata4/IWBEP/V4_SAMPLE/default/IWBEP/
V4_GW_SAMPLE_BASIC/0001/",
        "type" : "OData",
        "settings" : {
            "annotations": ["localAnnotations"],
            "odataVersion" : "4.0"
        }
    },
    "localAnnotations": {
        "uri": "data/annotations.xml",
        "type": "ODataAnnotation"
    }
},
```

Annotation files are usually defined as data sources in `manifest.json`. In the example above, the annotation file is located relative to the component. In the `dataSource` definition of the OData service, you can reference these annotation data sources in the `annotations` setting. The content of the annotation files are then merged into the service metadata in the given order (the last one wins). Every (target, term, qualifier)-tuple must appear at most once within `$metadata` documents, but can be overwritten by annotation files.

Annotation terms are not merged, but replaced as a whole (“PUT” semantics). For example, if you have defined the sort restriction annotation `Org.OData.Capabilities.V1.SortRestrictions` at the `BusinessPartnerSet` as shown in the example below, you have to repeat the term in your annotation file if you want to add, for example, the additional property `AscendingOnlyProperties`.

The annotation term is specified in the service metadata document:

```
<Annotations Target="GWSAMPLE_BASIC.GWSAMPLE_BASIC_Entities/BusinessPartnerSet">
    <Annotation Term="Org.OData.Capabilities.V1.SortRestrictions">
        <Record>
            <PropertyValue Property="NonSortableProperties">
                <Collection>
                    <PropertyPath>BusinessPartnerID</PropertyPath>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

If an additional property needs to be added, the term has to be repeated in the annotation file:

```
<Annotations Target="GWSAMPLE_BASIC.GWSAMPLE_BASIC_Entities/BusinessPartnerSet">
    <Annotation Term="Org.OData.Capabilities.V1.SortRestrictions">
        <Record>
            <PropertyValue Property="AscendingOnlyProperties">
                <Collection>
                    <PropertyPath>AnyPropertyPath</PropertyPath>
                </Collection>
            </PropertyValue>
            <PropertyValue Property="NonSortableProperties">
                <Collection>
                    <PropertyPath>BusinessPartnerID</PropertyPath>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

Creating an Entity

The `sap.ui.model.odata.v4.ODataListBinding#create` method creates a new entity. Users can interact with a newly created entity even before it has been sent to the server.

To create new entities, `ODataListBinding#create` uses the list binding's update group ID as group ID. For more information how this group ID is determined, see the documentation for the binding-specific parameter `$$updateGroupID` of `ODataModel#bindList`.

A newly created entity can be inserted at the start or at the end of the list. This new entity is transient until it is successfully submitted, see `Context#isTransient`. The initial data for the created entity can be supplied via the parameter `oInitialData` and modified via property bindings. Properties that are not part of the initial data show the default value from the service metadata on the UI, but they are not sent to the server. If there is no default value, null is used instead, even if the property is not nullable. Updates for the transient entity are collected and added to the POST request which creates the entity on the server.

Inserting an entity at the end of the list is done via the `bAtEnd` parameter in the `create` call. This is only possible, if the list's length has been requested via the system query option `$count`.

To delete transient entities, use `Context#delete`. Transient entities are also deleted when you reset the changes for the list binding on which the entity has been created, see `ODataListBinding#resetChanges` and `ODataModel#resetChanges`. The promise returned by `Context#created` is rejected in all cases where the created entity is deleted before it is created in the backend. As long as the list binding has a transient entity, `ODataListBinding#hasPendingChanges` returns `true` and the following methods of `ODataListBinding` raise an error: `refresh`, `filter`, and `sort`. The deletion of another entity of the same list binding is possible.

> **i Note**
>
> The position of the created entity may change after the methods `refresh`, `filter`, or `sort` of an `ODataListBinding`.

If you have called `ODataListBinding#create` on a list binding where the update group ID has `SubmitMode.API` and the creation of the entity on the server fails, the creation is repeated with the next call of `submitBatch` for this group. If the update group ID has `SubmitMode.Auto` or `SubmitMode.Direct` and the creation fails, the creation is repeated automatically with the next update for the entity. `submitBatch` can also be used for update group IDs with `SubmitMode.Auto` to repeat, independently of an update. The error returned by the server is passed to the `MessageManager` and the promise you get via `Context.created` is not rejected. Each time the data for the created entity is sent to the server, a `Context.createSent` event is fired. Each time the client receives a response for the creation, a `Context.creatCompleted` event is fired, independent of whether the creation was successful, or not.

If you have called `ODataListBinding#create` on a list binding with an application group ID, and the creation of the entity on the server fails, the creation is repeated with the next call of `ODataModel#submitBatch` for this group. If the update group ID is `$auto` or `$direct`, and the creation fails, the creation is repeated automatically with the next update for the entity. The error is passed to the `MessageManager` and the promise you get via `Context#created` is not rejected. Each time the data for the created entity is sent to the server, a `createSent` event is fired. Each time the client receives a response for the creation, a `createCompleted` event is fired, independent of whether the creation was successful, or not.

→ Recommendation

Lock the UI each time the (`POST`) request for the creation is sent to the server and unlock it, when the response from the server for that (`POST`) request is processed, because updates in between result in errors. If the update group ID is `SubmitMode.API`, you can lock the UI when calling `ODataModel#submitBatch` and unlock it again when the promise returned by `ODataModel#submitBatch` is resolved or rejected. However, if the update group ID is `SubmitMode.Auto` or `SubmitMode.Direct`, use the `createSent` event to lock the related UI and the `createCompleted` event to unlock it.

```
// suppose this list binding has no own update group; it uses the model's update
group instead (an application group)
...
        onCreateSalesOrder : function (oEvent) {
            var oContext = this.getView().byId("SalesOrders").getBinding("items")
                    .create({
                        "Note" : "My new Sales Order",
                        "NoteLanguage" : "E",
                        "BuyerID" : "0100000000",
                        "CurrencyCode" : "EUR"
                    });

            // Note: This promise fails only if the transient entity is deleted
            oContext.created().then(function () {
                    // sales order successfully created
                }, function (oError) {
                    // handle rejection of entity creation; if oError.canceled
=== true then the transient entity has been deleted
                });
        },

        onDeleteSalesOrder : function () {
            var oSalesOrderContext =
this.getView().byId("SalesOrders").getSelectedItem().getBindingContext();

            oSalesOrderContext.delete("$auto").then(function () {
                    // sales order successfully deleted
                }, function (oError) {
                    // do error handling
                });
        },

        onSaveSalesOrder : function () {
            var oView = this.getView();

            function resetBusy() {
                oView.setBusy(false);
            }

            // lock UI until submitBatch is resolved, to prevent errors caused
by updates while submitBatch is pending
            oView.setBusy(true);


oView.getModel().submitBatch(oView.getModel().getUpdateGroupId()).then(resetBusy,
 resetBusy);
        },
...
```

> **i Note**
>
> To ensure that for a list binding all expanded data is available as soon as the promise returned by `Context#created` is resolved, an additional single `GET` request for the newly created entity is sent automatically once the `POST` request has arrived.
>
> If you want to skip this additional single `GET` request, call `ODataListBinding#create` with parameter `bSkipRefresh=true`.

The `promise` returned by `Context#created` is resolved when the entity represented by this context has been created in the backend. Once the promise is resolved, `Context#getPath` returns a path including the key predicate of the new entity. For returning the path including the key predicates, all key properties need to be available.

Related Information

sap.ui.model.odata.v4.ODataListBinding
sap.ui.model.odata.v4.ODataModel
sap.ui.model.odata.v4.Context
sap.ui.core.message.MessageManager

Deleting an Entity

The `Context.delete` method deletes an entity on the server and updates the user interface accordingly.

When you delete the entity from a list binding, the corresponding row is removed. When you delete the entity from a context binding, the binding and all dependent bindings lose the reference.

Example: Delete From a Table

```
onDeleteSalesOrder : function () {
    var oTable = this.getView().byId("SalesOrders"),
        oSalesOrderContext = oTable.getSelectedItem().getBindingContext();

    oSalesOrderContext.delete("$auto").then(function () {
        oTable.removeSelections();
        MessageBox.alert("Deleted Sales Order",
            {icon : MessageBox.Icon.SUCCESS, title : "Success"});
    }, function (oError) {
        MessageBox.alert("Could not delete Sales Order: "
            + oError.message, {icon : MessageBox.Icon.ERROR, title : "Error"});
    });
},
```

Related Information

Context.delete

Consuming OData V2 Services with the OData V4 Model

The SAPUI5 framework can consume OData V2 Services in a non-intrusive way as if working directly with OData V4.

Overview

With SAPUI5 the OData V4 Model supports the consumption of OData V2 services. The framework takes care to convert the metadata and the data in a way that the application developer writes its application as he would do with an OData V4 model with a corresponding OData V4 service. As OData V4 supports various features that are not covered by OData V2, some features of the OData V4 model cannot be used. A list of the limitations is available below.

Here is an example of a manifest.json file, which shows how to configure your application to consume a V2 service with the V4 model.

```
{
    "_version" : "1.1.0",
    "sap.app" : {
        ...
        "dataSources" : {
            "default" : {
                "uri" : "<ODataV2 Service URL>",
                "type" : "OData",
                "settings" : {
                    "odataVersion" : "2.0"
                }
            }
        },
        ...
    },
    ...
    "sap.ui5" : {
        ...
        "dependencies" : {
            "minUI5Version" : "1.49",
            ...
        },
        "models" : {
            "" : {
                "dataSource" : "default",
                "settings" : {
                    "autoExpandSelect" : false,
                    "operationMode" : "Server",
                    "synchronizationMode" : "None"
                },
                "type" : "sap.ui.model.odata.v4.ODataModel"
            }
        },
        ...
    }
}
```

If you have an OData V2 service and you want to consume this service with an OData V4 model, you have to set `type` of the corresponding model to **`sap.ui.model.odata.v4.ODataModel`** and `odataVersion` of the corresponding data source to **`2.0`**. Both settings are needed. Additionally, the `minUI5Version` has to be at least **1.49**.

Type Mapping and Data Conversion

OData V4 has several types, which do not exist in OData V2 (e.g. `Edm.Date`, `Edm.TimeOfDay`) and the other way around `dm.DateTime`, `Edm.Time`), so we need a mapping between corresponding data types.

The following OData V2 types are supported and mapped:

- `Edm.Binary`, `Edm.Boolean`, `Edm.Byte`, `Edm.Decimal`, `Edm.Double`, `Edm.Guid`, `Edm.Int16`, `Edm.Int32`, `Edm.Int64`, `Edm.SByte`, `Edm.String` do not need a type mapping as they exist in both OData versions.
- `Edm.DateTime` is mapped to OData V4 type `Edm.Date` if the property has the OData V2 annotation `sap:display-format="date"` or otherwise to `Edm.DateTimeOffset` with UTC timezone.
- `Edm.Time` is mapped to OData V4 type `Edm.TimeOfDay`.

Some data types have different representation in OData V2 and OData V4. The application developer should only use the OData V4 values. The framework takes care that the values are converted before sending the request to the backend (e.g. as a value of a key property) and after receiving the response from the backend.

Here are some examples:

- OData V2 uses a different alphabet for the base-64 encoding for `Edm.Binary`. OData V2 uses '`+`' and '`/`' but OData V4 uses instead '`-`' and '`_`'.
- For `Edm.DateTimeOffset` OData V2 uses a JavaScript Date (i.e. `Date(1420529121547)`) and OData V4 a String (i.e. `"2015-01-06T12:25:21.547"`) representation.

Mapping of OData V4 Features

System Query Options

$expand / $select system query options

OData V4 supports "$expand with options", which means an $expand option can contain query options for the expanded navigation property such as $select, $orderby or $expand itself.. OData V2 does not support "$expand with options" but only $expand and $select with path values. An OData V4 $expand option, which contains only $select and $expand options, is transformed to the corresponding OData V2 $expand and $select options by "flattening" the OData V4 structure.

> ### Example
>
> OData V4 system query options to expand line items:
>
> ```
> $expand=SO_2_SOITEM($select=DeliveryDate,ItemPosition,SalesOrderID)
> $select=SalesOrderID,GrossAmount
> ```
>
> These options are converted into following OData V2 system query options:
>
> ```
> $expand=SO_2_SOITEM
> $select=SO_2_SOITEM/DeliveryDate,SO_2_SOITEM/ItemPosition,SO_2_SOITEM/
> SalesOrderID,SalesOrderID,GrossAmount
> ```

> **i Note**
>
> Because of the flat structure of $expand and $select, the URLs get longer than the URLs in OData V4. In some browsers that might be an issue if you use $direct as group ID. (for more information see: Batch Control [page 952]).

$orderby

OData V4 supports system query option $orderby also in $expand structures, but OData V2 supports only $orderby on top level. When consuming an OData V2 service with an OData V4 model, system query $orderby on top level is supported but an error is raised if $orderby is used in $expand.

`$orderby` can also work with expressions, but those need to be identical in V2 and V4.

$count

In OData V2 $count can only be used on top level. An error is raised if $count is used in $expand. On top level, $count is converted to V2 $inlinecount and propagated to the request.

$filter

In OData V2 $filter can only be used on top level. An error is raised if $filter is used in $expand. On top level, $filter is converted to V2 syntax and propagated to the request. $filter supports comparisons, `and`, `or`, `not` and the following built-in functions:

- String functions: `concat`, `contains` (converted to `substringof`), `endswith`, `indexof`, `length`, `startswith`, `tolower`, `toupper`, `trim`
- Date and time functions: `day`, `hour`, `minute`, `month`, `second`, `year`
- Arithmetic functions: `ceiling`, `floor`, `round`

The remaining functions are unsupported because they have no equivalent in V2.

> **⚠ Caution**
>
> Avoid passing literals to date, time and arithmetic functions, because the parameter to these functions can have different types and it cannot be decided which one is correct.
>
> > **Example**
> >
> > In floor(42) the 42 can be either an `Edm.Double` or an `Edm.Decimal`. Avoid comparing two literals like, for example `42 eq 42`, because the converter determines the type of a literal from the other operand in comparisons.

OData V2 Annotations

Following V2 (attributes with namespace http://www.sap.com/Protocols/SAPData) annotations are converted to corresponding V4 annotations (see OData V2 Model -> Meta Model for OData V2 ->Enhancement of the OData Meta Model [page 903]).

OData V2 Annotations defined at `EntitySet`:

- creatable, deletable, deletable-path, label, pageable, requires-filter, searchable, topable, updatable, updatable-path

OData V2 Annotations defined at `Property`:

- aggregation-role, creatable, creatable-path, display-format (with values `NonNegative` and `UpperCase`), field-control, filterable, filter-restriction, heading, label, precision, quickinfo, required-in-filter, sortable text, unit, updatable, visible

OData V2 Annotations defined at `Schema`:

- sap:schema-version="foo"

sap:semantics:

- bday, body, city, class, completed, country, currency-code dtend, dtstart, due, duration, familyname, fbtype, fiscalyear, fiscalyearperiode, from, givenname, honorific, location, middlename, name, nickname, note, org, org-role, org-unit, percent-complete, photo, pobox, priority, received, region, sender, status, street, subject, suffix, tel (with types `cell`, `work`, `fax`), title, transp, url, unit-of-measure, url, wholeday, year, yearmonth, yearmonthday, yearquarter, yearweek, zip

OData V2 Annotations defined at `NavigationProperty`:

- filterable, creatable, creatable-path

OData V2 Annotations defined at `FunctionImport`:

- action-for, label

Warnings are logged for all SAP attributes that have not been processed by the V2 converter.

Limitations

Not all OData V4 features are supported yet when consuming an OData V2 service. We have following limitations:

- The OData V4 model can consume an OData V2 service for read scenarios only.
- The OData V2 services has to provide inline type metadata in responses, i.e. property `__metadata.__type`. This information is needed to convert the data between the OData V2 and the OData V4 types.
- Supported data types are listed above (see Type Mapping and Data Conversion [page 978]).
- System query options $orderby, $filter and $count on top level and $expand and $select are supported. All other system query options raise an exception.
- Not all OData V2 annotations are converted yet. Supported OData V2 annotations are listed above (see OData V2 Annotations [page 979]).

> ⚠ Caution
>
> OData V4 validates that namespaces are always loaded from the same URI. Ensure to reference the metadata document (e.g. `.../IWBEP/GWSAMPLE_BASIC/$metadata`) in additional annotation files and not the service document (`.../IWBEP/GWSAMPLE_BASIC`).

Extension for Data Aggregation

The OData V4 Model supports features of the OData Extension for Data Aggregation V4.0 specification.

The binding parameter `$$aggregation` at `sap.ui.model.odata.v4.ODataModel#bindList` holds the information needed for data aggregation. It may be changed by `sap.ui.model.odata.v4.ODataListBinding#setAggregation`. It cannot be combined with an explicit system query option `$apply` because it implicitly derives `$apply`. You can find more information also in the OData Extension for Data Aggregation V4.0 specification .

Two scenarios are supported:

- You can provide properties for grouping and aggregation. An appropriate system query option `$apply` is derived from those and the list binding still provides a flat list of contexts ("rows"), but with additional aggregated properties ("columns"). In addition, you can request grand total values for aggregatable properties. In this case, an extra row appears at the beginning of the flat list of contexts and contains the grand total values as well as empty values for all other properties.

  ### Sample Code

  Example XML View With Grand Total

```
<table:Table fixedRowCount="1"
 rows="{
  path : '/BusinessPartners',
  parameters : {
   $$aggregation : {
    aggregate : {
     SalesAmountSum  : {
      grandTotal : true,
      name : 'SalesAmount',
      with : 'sap.unit_sum'
     }
    },
    group : {
     Region : {}
    }
   },
   $count : true,
   $filter : 'SalesAmountSum gt 1000000',
   $orderby : 'SalesAmountSum desc'
  }}">
 <table:Column template="Region">
  <Label text="Region"/>
 </table:Column>
 <table:Column hAlign="End">
  <Label text="Sales Amount"/>
  <table:template>
   <Text text="{path : 'SalesAmountSum', type :
'sap.ui.model.odata.type.Decimal'}" />
  </table:template>
 </table:Column>
 <table:Column>
  <Label text="Currency"/>
  <table:template>
   <Text text="{path : 'SalesAmountSum@Analytics.AggregatedAmountCurrency',
    type : 'sap.ui.model.odata.type.String'}" />
  </table:template>
 </table:Column>
</table:Table>
```

- You can provide group levels to determine a hierarchy of expandable group levels in addition to the leaf nodes determined by the `groupable` and `aggregatable` properties. Only a **single** group level is currently supported and it cannot be expanded yet. Group levels cannot be combined with filtering or with the system query option `$count : true`.

> **! Restriction**
>
> Multi-unit situations are not supported with data aggregation. An error is thrown if the `sap.ui.model.odata.v4.ODataListBinding` detects a multi-unit situation.
>
> Given a service with groupable properties `G1, ..., Gn`, an aggregatable property `A` with related unit property `U`. An aggregated OData request is said to return a multi-unit situation, if the following conditions are fulfilled for the requested aggregation levels `G1, ..., Gj` :
>
> 1. The result contains two or more entities with identical values for the groupable properties `G1, ..., Gj`.
> 2. The entities from **1.** have **different** values for the unit-property `U` and any value for `A`.

> **! Restriction**
>
> The grand total calculation is currently only supported for the standard aggregation functions `sum`, `min`, and `max` as well as for custom aggregates that use these functions. This also applies to displaying subtotals of group levels.

Filtering

Filters are provided to the list binding as described in Filtering [page 939]. The `Filter` objects are analyzed automatically to perform the filtering before the aggregation where possible using the `filter()` transformation. The remaining filters, including the provided `$filter` parameter of the binding, are applied after the aggregation either via the system query option `$filter` or within the system query option `$apply`, using again the `filter()` transformation.

Server Messages in OData V4 Model

The OData V4 model supports server messages sent via an OData V4 service.

> **⚠ Caution**
>
> This feature is experimental. For more information, see Compatibility Rules [page 17].

Messages transported via an OData V4 service response are parsed and reported to the message model `sap.ui.model.message.MessageModel`. An application can retrieve the messages and display them in a suitable control, for example in `sap.m.MessageView`.

End user messages contain the following information:

- `code` - language-independent message code

- `message`- language-dependent message text
- `target` - path to the target of the message detail
- `technicalDetails` - technical details of the message
- `transition` - specifies a message as a state (false) or a transition message (true)
- `numericSeverity` – classification of end user messages; allowed values: 1 (success), 2 (info), 3 (warning), 4 (error); `numericSeverity` is mapped to the specific `sap.ui.core.MessageType`
- `longtextUrl` – optional; is omitted, if there is no long text available for the corresponding message.

The use of the fields in specific cases is described in the sections below.

Messages can be either bound or unbound: Unbound messages are not related to OData entities and are, therefore, also not part of the OData success response in the HTTP body. Bound messages are related to OData entities and are modeled as OData resources.

Unbound Messages

Unbound messages are transported in the header field `sap-messages`, which is an array of messages. Unbound messages cannot be suppressed. They are always returned by the server and they always refer to the current request as described in the section about transition messages below. In case of successful requests, unbound messages are transported as an array in the HTTP header field `sap-messages`:

```
sap-messages:[
     {
          "code" : "SYS/42",
          "message" : "System will be down for maintenance next weekend.",
          "numericSeverity" : 2,
          "longtextUrl" : "Messages(3)/LongText/$value"
     }
]
```

Note

`longtextUrl` can be a relative or absolute path. Relative paths are treated as relative to the request URL. Absolute paths are treated as relative to the server.

Example

Request URL: `http://<server>:<port>/serviceroot.svc/BusinessPartners(42)/to_Address; longtextUrl: "Messages(3)/LongText/$value"`

Result: `http://<server>:<port>/serviceroot.svc/BusinessPartners(42)/Messages(3)/LongText/$value`

Request URL: `http://<server>:<port>/serviceroot.svc/BusinessPartners(42); longtextUrl: "/Messages(3)/LongText/$value"`

Result: `http://<server>:<port>/Messages(3)/LongText/$value`

Bound Messages

Bound messages are related to OData entities and are modeled as OData resources. An OData entity contains its bound messages as collection valued property of the complex type specified in the description of `com.sap.vocabularies.Common.v1.Messages`. Thus, bound messages are transported in the HTTP body. The target property specifies to which property the message is bound. The application needs to specify in the `$select` binding parameter whether messages should be returned by the server, or not.

```
<ComplexType Name="<name of message type>">
    <Property Name="code" Type="Edm.String" Nullable="false" />
    <Property Name="message" Type="Edm.String" Nullable="false" />
    <Property Name="target" Type="Edm.String" Nullable="true" />
    <Property Name="transition" Type="Edm.Boolean" Nullable="false" />
    <Property Name="numericSeverity" Type="Edm.Byte" Nullable="false" />
    <Property Name="longtextUrl" Type="Edm.String" Nullable="true" />
</ComplexType>
```

The `target` property may contain a path relative to the entity which contains the message. The target can, for example, refer to a property within that entity. This information is used to highlight UI elements such as input fields, if they are bound to properties referenced by the path contained in the `target` property. All responses are checked for bound messages. If there are messages, they are reported to the message model.

For bound messages, `longtextUrl` can be a relative or absolute path. Relative paths are treated as relative to the innermost context path (`@odata.context`) in the response, or to the request URL, if there is no context path. Absolute paths are treated as relative to the server.

Messages in Error Responses

Error messages are always reported in the error response in JSON format as described in the OData JSON Format Version 4.0 in section *19 Error Response* with the following additions:

- The instance annotation `com.sap.vocabularies.Common.v1.longtextUrl` can be used to provide a long text URL, which can be a relative or an absolute path. Relative paths are treated as relative to the request URL. Absolute paths are treated as relative to the server.
- `target` is relative to the requested resource.
- The error message type is always `sap.ui.core.MessageType.Error`. The instance annotation `com.sap.vocabularies.Common.v1.numericSeverity` determines the message type of the detail messages.
- The error message and all messages in details are transition messages.

State Messages and Transition Messages

Messages can be either state or transition messages:

- State messages refer to the state of the corresponding resource (OData entity instance). State messages are valid as long as the related business object is not changed. The OData V4 Model is responsible for the lifecycle of state messages and will remove state messages from the message model, if they are no longer sent by the server when the corresponding resource is requested.

- Transition messages refer to the current request and are not related to the state of a resource. They are only relevant for the request that was triggered, for example *System not available, business object could not be updated*. Optionally, transition messages can reference a business object, for example *Shipping address could not be changed due to missing authorization*. Transition messages are translated into persistent messages in the message model. The application is responsible for the lifecycle of such persistent messages. The OData V4 Model will not remove persistent messages from the message model.

Lifecycle Management for State Messages

The lifecycle management for state messages is optimized for a specific orchestration with the server. When bound messages are requested, the OData V4 server returns all bound messages for the respective entity and its subentities within the same business object. The business object is defined by the first path segment.

The following example uses a sales order with items and related products:

- A `GET` request for `/SalesOrder(´0815´)` returns all bound messages for the sales order and the items, even if the items themselves are not contained in the response. Messages to assigned products, business partners, and so on, that are not part of the `SalesOrder` business object will **not** be sent if the path starts within the `SalesOrder` business object.
- A `GET` request for a specific item with path `/SalesOrder(´0815´)/_Items(´010´)` returns all bound messages for this item.
- A `GET` request for the product related to an item using the deep path `/SalesOrder(´0815´)/_Items(´010´)/_Product` will not return any bound messages.

The OData V4 model checks whether the response contains the message property and removes all previous bound state messages from the message model, if their target path starts with the path of the entity.

This concept has the following consequences:

- When you display the information for the business object itself, you can also display the messages for all subentities of this business object.
- For displaying the entities within a business object, an application has to use deep paths, instead of canonical paths. Otherwise, messages will appear twice. In the object page of item `´010´`, for example, the binding needs to use the path `/SalesOrder(´0815´)/_Items(´010´)`. You can achieve this also with a relative binding using the context of the sales order.
- Binding entities outside the business object with the deep path means that no messages will be retrieved for this entity. Using the binding `/SalesOrder(´0815´)/_Items(´010´)/_Product` to display product information of item `010`, for example, will not return any product-specific bound message.
- As a consequence, it must also **not** be possible to change the entity that is bound with a path that starts with a different business object. If, for example, product information needs to be changed, we recommended to use the canonical path to bind the product assigned to item `010` to achieve that the server sends the bound messages of the product.

> **i Note**
>
> The SAPUI5 V4 ODataModel is agnostic to business objects. The application needs to take care of the proper setup.

Combining State/Transition and Bound/Unbound Messages

	State	Transition
Unbound	✗	✓
Bound	✓	✓

Message Severity

The table shows the supported severity values and their mapping to the specific `sap.ui.core.MessageType`.

numericSeverity	Type	Comment
1	`sap.ui.core.MessageType.Success`	Positive feedback - no action required
2	`sap.ui.core.MessageType.Information`	Additional information - no action required
3	`sap.ui.core.MessageType.Warning`	Warning - action may be required
4	`sap.ui.core.MessageType.Error`	Error - action is required

Accessing the Original Message

The attribute `technicalDetails.originalMessage` of the message in the message model allows you to access the original message from the back-end.

Related Information

https://wiki.scn.sap.com/wiki/display/EmTech/OData+4.0+Vocabularies+-+SAP+Common

Currencies and Units

> ⚠ Caution
>
> This feature is experimental. For more information, see Compatibility Rules [page 17].

For amounts or measures, you may sometimes need different currencies or units than defined in the CLDR. The data types `sap.ui.model.odata.type.Currency` and `sap.ui.model.odata.type.Unit` enable you to use code lists with customizing for currency codes and units. For code lists with customizing, you need to define the following annotations:

- Currencies: `com.sap.vocabularies.CodeList.v1.CurrencyCodes`
- Measures: `com.sap.vocabularies.CodeList.v1.UnitsOfMeasure`

Code list annotations for currency codes and measures in "metadata.xml"

```
<EntityType Name="Product">
    ...
   <Property Name="WeightMeasure" Type="Edm.Decimal" Nullable="false"
Precision="13" Scale="variable" />
   <Property Name="WeightUnit" Type="Edm.String" Nullable="false"
MaxLength="3" />
   <Property Name="CurrencyCode" Type="Edm.String" Nullable="false"
MaxLength="5" />
   <Property Name="Price" Type="Edm.Decimal" Nullable="false" Precision="15"
Scale="variable" />
    ...
</EntityType>
...
<Annotations Target="SAP__self.Container">
   <Annotation Term="com.sap.vocabularies.CodeList.v1.CurrencyCodes">
      <Record>
         <PropertyValue Property="Url" String="../../../../default/iwbep/common/
0001/$metadata" />
         <PropertyValue Property="CollectionPath" String="Currencies" />
      </Record>
   </Annotation>
</Annotations>
<Annotations Target="SAP__self.Container">
   <Annotation Term="com.sap.vocabularies.CodeList.v1.UnitsOfMeasure">
      <Record>
         <PropertyValue Property="Url" String="../../../../default/iwbep/common/
0001/$metadata" />
         <PropertyValue Property="CollectionPath" String="UnitsOfMeasure" />
      </Record>
   </Annotation>
</Annotations>

...
<Annotations Target="SAP__self.Product/Price">
    ...
   <Annotation Term="Org.OData.Measures.V1.ISOCurrency" Path="CurrencyCode" />
    ...
</Annotations>
...
<Annotations Target="SAP__self.Product/WeightMeasure">
    ...
   <Annotation Term="Org.OData.Measures.V1.Unit" Path="WeightUnit" />
    ...
</Annotations>
```

Code lists that are referenced by the `com.sap.vocabularies.CodeList.v1.CurrencyCodes` or `com.sap.vocabularies.CodeList.v1.UnitsOfMeasure` annotations need the following:

- The internal code as its only key property
- A language-dependent description
- A numeric property with the unit-specific number of significant fractional digits
- Optional: An external code that should be visualized instead of the internal code
- Optional: A standard code

The key property is annotated with:

- `com.sap.vocabularies.Common.v1.Text` pointing to the description property
- `com.sap.vocabularies.Common.v1.UnitSpecificScale` pointing to the numeric property
- Optional: `com.sap.vocabularies.CodeList.v1.StandardCode` pointing to the standard code property

The entity type is optionally annotated with `Org.OData.Core.V1.AlternateKeys` pointing to another property that should be used for visualization.

If the alternate key is available, the type uses the alternate key as the key of the currency or unit. In this case, the data of the actual service have to contain the alternate key representation in the currency or unit property. The key is used and expected in the data if no alternate key is annotated. Note that there must be a maximum of one alternate key, and that key and alternate key must have exactly one property.

The property annotated as `com.sap.vocabularies.CodeList.v1.StandardCode` is interpreted as the ISO code by `sap.ui.model.odata.type.Currency` and used to find currency symbols. The currency symbols may be used for entering data.

Example for the metadata of a code list service

```
...
  <EntityType Name="Currency">
    <Key>
      <PropertyRef Name="CurrencyCode" />
    </Key>
    <Property Name="CurrencyCode" Type="Edm.String" MaxLength="5" />
    <Property Name="ISOCode" Type="Edm.String" MaxLength="3" />
    <Property Name="Text" Type="Edm.String" MaxLength="15" />
    <Property Name="DecimalPlaces" Type="Edm.SByte" />
  </EntityType>

  <EntityType Name="UnitOfMeasure">
    <Key>
      <PropertyRef Name="UnitCode" />
    </Key>
    <Property Name="UnitCode" Type="Edm.String" MaxLength="3" />
    <Property Name="ISOCode" Type="Edm.String" MaxLength="3" />
    <Property Name="ExternalCode" Type="Edm.String" MaxLength="3" />
    <Property Name="Text" Type="Edm.String" MaxLength="30" />
    <Property Name="DecimalPlaces" Type="Edm.Int16" />
  </EntityType>

  <Annotations Target="SAP__self.Currency/CurrencyCode">
    <Annotation Term="Common.Text" Path="Text" />
    <Annotation Term="Common.UnitSpecificScale" Path="DecimalPlaces" />
    <Annotation Term="CodeList.StandardCode" Path="ISOCode" />
  </Annotations>

  <Annotations Target="SAP__self.UnitOfMeasure">
    <Annotation Term="Core.AlternateKeys">
      <Collection>
        <Record>
          <PropertyValue Property="Key">
            <Collection>
              <Record>
                <PropertyValue Property="Name" PropertyPath="ExternalCode" />
                <PropertyValue Property="Alias" String="ExternalCode" />
              <Record>
            </Collection>
          </PropertyValue>
        <Record>
      </Collection>
```

```
    </Annotation>
  </Annotations>

  <Annotations Target="SAP__self.UnitOfMeasure/UnitCode">
    <Annotation Term="Common.Text" Path="Text" />
    <Annotation Term="Common.UnitSpecificScale" Path="DecimalPlaces" />
    <Annotation Term="CodeList.StandardCode" PropertyPath="ISOCode" />
    <Annotation Term="CodeList.ExternalCode" PropertyPath="ExternalCode" />
  </Annotations>
...
```

With the metadata above, you can use the data types `sap.ui.model.odata.type.Currency` and the `sap.ui.model.odata.type.Unit` in an input field as shown in the following example. The data types use a complex binding with the amount or measure as first part, the currency code or unit as second part, and the information about the code list customizing that has to be used as third part.

Example how to use currency and unit types in a freestyle application

```
...
<Input value="{mode:'TwoWay', parts:['WeightMeasure', 'WeightUnit',
{mode:'OneTime', path:'/##@@requestUnitsOfMeasure', targetType:'any'}],
type:'sap.ui.model.odata.type.Unit'}"/>
...
<Input value="{mode:'TwoWay', parts:['Price', 'CurrencyCode', {mode:'OneTime',
path:'/##@@requestCurrencyCodes', targetType:'any'}],
type:'sap.ui.model.odata.type.Currency'}"/>
...
```

The code lists are automatically requested only once per browser session and code list URL.

If you use XML templating, you can use `sap.ui.model.odata.v4.AnnotationHelper.format` to generate the composite binding for an amount or measure property. To recognize a property as an amount or measure, the property needs to be annotated either with the `Org.OData.Measures.V1.ISOCurrency`, or with the `Org.OData.Measures.V1.Unit` annotation. For more informatio about XML templating, see XML Templating [page 1018].

Additional annotations when using XML templating

```
<!-- used in view template -->
<Annotations Target="SAP__self.Product">
   <Annotation Term="com.sap.vocabularies.UI.v1.LineItem">
      <Collection>
         ...
         <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Weight" />
            <PropertyValue Property="Value" Path="WeightMeasure" />
         </Record>
         <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Price" />
            <PropertyValue Property="Value" Path="Price" />
         </Record>
         ...
      </Collection>
   </Annotation>
</Annotations>
```

You can now use `sap.ui.model.odata.v4.AnnotationHelper.format` in the XML template view to generate the composite binding for `sap.ui.model.odata.type.Currency` and `sap.ui.model.odata.type.Unit types`.

How to use `AnnotationHelper.format` with currencies or units

```
<template:alias name="format"
value="sap.ui.model.odata.v4.AnnotationHelper.format">
<template:alias name="label"
value="sap.ui.model.odata.v4.AnnotationHelper.label">
<VBox>
   <template:with path="meta>/ProductList/@com.sap.vocabularies.UI.v1.LineItem"
var="lineItem">
      <Table items="{/ProductList}">
         <columns>
            <template:repeat list="{lineItem>}" var="field">
               <Column>
                  <Label text="{field>@@label}"/>
               </Column>
            </template:repeat>
         </columns>
         <ColumnListItem>
            <template:repeat list="{lineItem>}" var="field">
               <Input value="{field>Value/@@format}"/>
            </template:repeat>
         </ColumnListItem>
      </Table>
   </template:with>
</VBox>
</template:alias>
</template:alias>
```

Example: Templating output

```
<VBox>
    <Table items="{/ProductList}">
      <columns>
          ...
          <Column>
            <Label text="Weight"/>
          </Column>
          <Column>
            <Label text="Price"/>
          </Column>
          ...
      </columns>
      <ColumnListItem>
              ...
              <Input value="{mode:'TwoWay', parts:[{path:'WeightMeasure',
type:'sap.ui.model.odata.type.Decimal', constraints:{'precision':13,
'scale':'variable', 'nullable':false}}, {path:'WeightUnit',
type:'sap.ui.model.odata.type.String', constraints:{'maxLength':3,
'nullable':false}}, {mode:'OneTime', path:'/##@@requestUnitsOfMeasure',
targetType:'any'}], type:'sap.ui.model.odata.type.Unit'}"/>
              <Input value="{mode:'TwoWay', parts:[{path:'Price',
type:'sap.ui.model.odata.type.Decimal', constraints:{'precision':15,
'scale':'variable', 'nullable':false}}, {path:'CurrencyCode',
type:'sap.ui.model.odata.type.String', constraints:{'maxLength':5,
'nullable':false}}, {mode:'OneTime', path:'/##@@requestCurrencyCodes',
targetType:'any'}], type:'sap.ui.model.odata.type.Currency'}"/>
              ...
      </ColumnListItem>
    </Table>
</VBox>
```

JSON Model

The JSON model can be used to bind controls to JavaScript object data, which is usually serialized in the JSON format.

The JSON model is a client-side model and, therefore, intended for small data sets, which are completely available on the client. The JSON model does **not** support mechanisms for server-based paging or loading of deltas. It supports, however, two-way binding. Also, client-side models like the JSON model have **no** built-in support for sending data back to the server. The apps have to use, for example, `model.getData()` and `jQuery.ajax()` to send updated data to the server.

To instantiate a JSON model, use the following code:

```
var oModel = new sap.ui.model.json.JSONModel();
```

After the instance has been created, there are different options to get the data into the model.

The easiest option is to set data by using the `setData` method:

```
oModel.setData({
    firstName: "Peter",
    lastName: "Pan"
});
```

> **i** Note
>
> The correct JSON notation uses double quotes for the keys and string values.

Usually, you do not define your data inline in the application but load it from a server-side service using an XHR request. The JSON model, however, also has a `loadData` method, which loads the JSON data from the specified URL asynchronously and applies it to the model:

```
oModel.loadData("data.json");
```

Related Information

API Reference: `sap.ui.model.json.JSONModel`

Sorting and Filtering in JSON Models

If you use a JSON model for data binding, sorting and filtering is implemented in JavaScript because the data is available on the client. You can use custom sorting and filtering methods in the JSON model. To define custom

methods, set the `fnCompare` method on the Sorter object or the `fnTest` method on the filter object after creating it.

The `fnTest` method of the filter gets the value to test as the only parameter and returns, whether the row with the given value should be filtered or not.

```
var oFilter = new sap.ui.model.Filter("property", function(value) {
    return (value > 100);
});
```

The `fnCompare` method of the Sorter gets the two values to compare as parameters and returns -1, 0 or 1, dependent on which of the two values should be ordered before the other:

```
var oSorter = new sap.ui.model.Sorter("property");
oSorter.fnCompare = function(value1, value2) {
    if (value1 < value2) return -1;
    if (value1 == value2) return 0;
    if (value1 > value2) return 1;
};
```

Binding Path Syntax for JSON Models

The JSON model has a simple binding path syntax, because it consists of named objects, such as properties, arrays, or nested objects.

The following example shows a simple JSON model with the different binding paths:

```
{
    company: {
        name: "Treefish Inc",
        info: {
            employees: 3,
        },
        contacts: [
            {
                name: "Barbara",
                phone: "873"
            },
            {
                name: "Gerry",
                phone: "734"
            },
            {
                name: "Susan",
                phone: "275"
            }
        ]
    }
}
```

Absolute binding paths within this model:

```
/company/name
/company/info/employees
/company/contacts
```

Relative binding paths within the "/company" context:

```
name
info/employees
contacts
```

Relative binding paths within a list binding of "/company/contacts":

```
name
phone
```

XML Model

The XML model allows to bind controls to XML data. It is a client-side model intended for small datasets, which are completely available on the client. The XML model does not contain mechanisms for server-based paging or loading of deltas. It supports two-way binding.

To instantiate the model, use the following code:

```
var oModel = new sap.ui.model.xml.XMLModel();
```

The XML model allows to bind controls to XML data. It is a client-side model intended for small data sets, which are completely available on the client. The XML model does not contain mechanisms for server-based paging or loading of deltas. It supports two-way binding.

```
oModel.setData(oXMLDocument);
```

To create inline XML data or to get XML data as a string, the XML model provides a `setXML` method. This method takes XML in text format and uses the browser's XML parser to create a document.

```
oModel.setXML("<?xml version=\"1.0\"?><some><xml>data</xml></some>");
```

Usually, you load your data from the server using an HTTP-based service, so the `loadData` method provides an easy way to load XML data from the given URL:

```
oModel.loadData("data.xml");
```

For more information, see the API Reference in the Demo Kit.

Related Information

API Reference: `sap.ui.model.xml.XMLModel`

Sorting and Filtering in XML Models

If you use an XML model for data binding, sorting and filtering is implemented in JavaScript because all data is available on the client. You can use custom methods for sorting and filtering in an XML model. To define

custom methods, set the `fnCompare` method on the Sorter object or the `fnTest` method on the Filter object after creating it.

The `fnTest` method of the Filter gets the value to test as the only parameter and returns, whether the row with the given value should be filtered or not. To implement a filter, use the following code :

```
var oFilter = new sap.ui.model.Filter("property");
oFilter.fnFilter = function(value) {
    return (value > 100);
};
```

The `fnCompare` method of the Sorter gets the two values to compare as parameters and returns -1, 0 or 1, dependent which of the two values should be ordered before the other. To implement a sorter, use the following code:

```
var oSorter = new sap.ui.model.Sorter("property");
oSorter.fnCompare = function(value1, value2) {
    if (value1 < value2) return -1;
    if (value1 == value2) return 0;
    if (value1 > value2) return 1;
};
```

XML Namespace Support

The XML model supports documents using XML namespaces.

For this purpose, you must declare namespaces using the `setNameSpace` method. The namespace prefixes do not necessarily need to be the same as in the XML document, they only used in the binding paths which are used to address nodes in the document.

Assumed this sample XML document:

```
<data xmlns="http://tempuri.org/base" xmlns:ext="http://tempuri.org/ext">
     <ext:entry id="0" value="foo" />
     <ext:entry id="1" value="foo" />
</data>
```

The namespaces must be declared in the JavaScript like this, to be able to bind to them:

```
var oModel = new sap.ui.model.xml.XMLModel(oXMLDoc);
oModel.setNameSpace("http://tempuri.org/base");
oModel.setNameSpace("http://tempuri.org/ext", "e"); [...]
oTable.bindRows("/e:entry");
```

Binding Path Syntax for XML Models

XML models differentiate between attributes and content. XML has no arrays and defines lists as multiple elements with the same name instead. This makes the binding path syntax for XML models more difficult than for JSON or OData models.

For attributes, a special selector using the "@" character exists and "text()" can be used to reference the content text of an element. Lists are referenced by using the path to the multiple element.

> **i** Note
>
> For the XML model the root must **not** be included in the path.

```
<companies>
  <company name="Treefish Inc">
    <info>
       <employees>3</employees>
    </info>
    <contact phone="873">Barbara</contact>
    <contact phone="734">Gerry</contact>
    <contact phone="275">Susan</contact>
  </company>
</companies>
```

Absolute binding paths within this model:

```
/company/@name
/company/info/employees
```

Relative binding paths within the `/company` context:

```
@name
info/employees/text()
```

Relative binding paths within a list binding of `/company/contact`:

```
text()
@phone
```

> **i** Note
>
> In a similar JSON model you would use `/companies/company/locations` as binding path for the `locations` collection. In an XML model the respective collection binding path is: `/company/locations/location`.

Resource Model

The resource model is used as a wrapper for resource bundles. In data binding you use the resource model instance, for example, to bind texts of a control to language-dependent resource bundle properties.

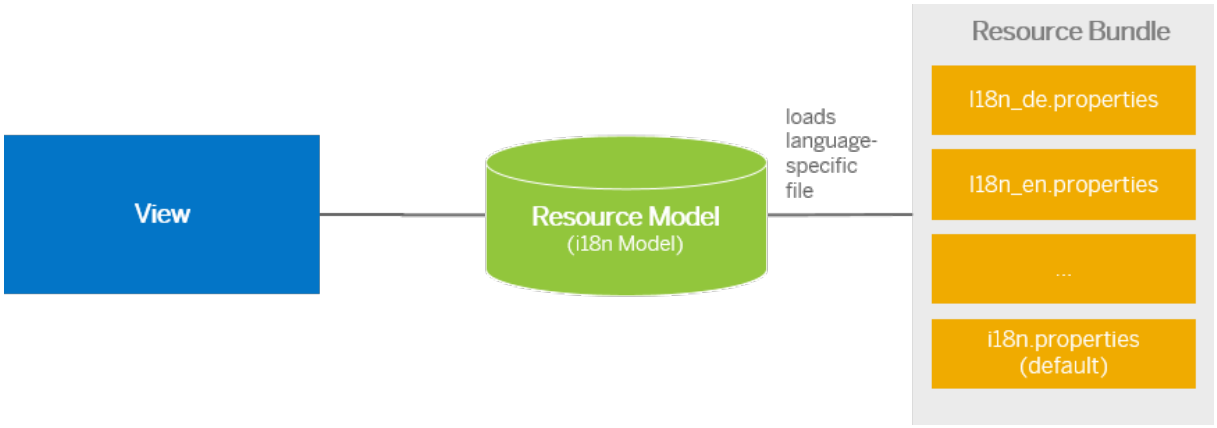
A resource model is instantiated with a `bundleName` or a `bundleURL`. The bundle name is the name of the resource bundle and equals a SAPUI5 module name within the define/require concept. The bundle URL points

to a resource bundle. If you use the bundle name, the file must have the `.properties` suffix. If you do not specify a `locale`, the system uses the login language: `var oModel = new sap.ui.model.resource.ResourceModel({bundleName:"myBundle",locale:"en"});`

In this resource model implementation you cannot pass parameters to your texts within the resource bundle. If you have to pass parameters, you must do this on your own. Therefore, you can load the bundle yourself or retrieve it from the model.

```
var myBundle = oModel.getResourceBundle();
```

After the resource model has been instantiated, you have a model containing the resource bundle texts as data.



- Views [page 787]
- Resource Bundles [page 1272]

Related Information

Modules and Dependencies [page 1094]
Localization [page 1269]
Resource Bundles [page 1272]
API Reference: `sap.ui.model.resource.ResourceModel`

Binding Path Syntax for Resource Models

The binding path syntax for the resource model only contains a flat list of properties.

The following example shows a simple resource model that illustrates the possible binding paths: Resource bundle content:

```
CLOSE_BUTTON_TEXT=Close
OPEN_BUTTON_TEXT=Open
```

```
CANCEL_BUTTON_TEXT=Cancel
```

Binding paths within the model:

```
CLOSE_BUTTON_TEXT
OPEN_BUTTON_TEXT
CANCEL_BUTTON_TEXT
```

Binding Texts to a Resource Bundle

If you don't want to use a component or descriptor file, or you want to use a more fine-grained resource bundle, you can declaratively instantiate a resource model in an XML or JSON view. To do so, you define the resource bundle by a name (`resourceBundleName` property) or a URL (`resourceBundleUrl` property) and assign an alias (`resourceBundleAlias` property) for the bundle in the view definition.

The `ResourceModel` required for binding these texts is created during view instantiation. The model is set as secondary model with the given alias to the view instance. If you want to bind other properties to another model, you have to create the model on your own in the corresponding controller or HTML page and attach it to the view with another alias. The binding itself behaves in the same way as every SAPUI5 data binding and as described above.

1. Define the following resource bundle content: `MY_TEXT=Hello World`
2. To bind this resource bundle content in XML views, insert the following code:

```
<core:View resourceBundleName="myBundle"
           resourceBundleAlias="i18n"
           controllerName="sap.hcm.Address" xmlns="sap.m"
xmlns:core="sap.ui.core"
           xmlns:html="http://www.w3.org/1999/xhtml">
   <Panel>
      <Button text="{i18n>MY_TEXT}"/>
   </Panel>
<core:View>
```

Custom Model

Custom models can be used if none of the models provided by SAPUI5 is suitable for the specific needs of an application.

To instantiate a custom model, proceed as follows:

1. Extend the `Model` class and specify the binding modes that the model should support (for example, two-way, one-way, one-time).
2. Extend the `Binding` class to suit your specific binding or reuse the existing specific binding implementations `PropertyBinding`, `ListBinding`, and/or `TreeBinding`.
3. To enable the filtering functionality, use the `Filter` class with `FilterOperator` enum in your binding implementation.
4. To enable the sorting functionality, use the `Sorter` class in your binding implementation.

You can find all necessary classes in the `sap.ui.model` namespace. As a starting point, take a look at the `JSONModel` implementation in `sap.ui.model.json.JSONModel`.

Assigning the Model to the UI

If you don't want to use a component or descriptor file, you have to assign the model instance manually to the UI, before you can bind controls to this model instance.

SAPUI5 provides a flexible and modularized concept in which you can not only define one model for your applications, but define different areas in your application with different models and assign single controls to a model. You can, for example, define a JSON model for the application and an OData model for a table contol that is contained in the application. You can also set multiple models for a control, a UI area, or the core by specifying a name for the model. These models can be accessed by their name.

```
var oJSONModel = new sap.ui.model.json.JSONModel();
var oODataModel  = new sap.ui.model.odata.v2.ODataModel("myServicelUrl");
var oControl  = new sap.m.Input();
oControl.setModel(oODataModel);
//set the JSONModel with the name 'myJSONModel' to the same control
oControl.setModel(oJSONModel,"myJSONModel");
```

When you set a model to a UI area or control, it will be propagated to all aggregated child controls. So if you set a model to a container control, for example, all controls that are contained (aggregated) in this container have access to this model. If one of the contained controls has its own model set (with the same name), the propagation stops. It is not possible to have two models with the same name set to one control instance.

Choose one of the following options:

- You can define a global model that can be accessed by all controls from all UI areas by using the `setModel` method on the SAPUI5 core object. This is useful for simple form applications or demo applications.

  ```
  sap.ui.getCore().setModel(oModel);
  ```

- You can also define a specific model for sections within a UI area, for example, inside a panel or for a table control. In this case, you can use the `setModel` method available on any control:

  ```
  var oTable = sap.ui.getCore().byId("table");
  oTable.setModel(oModel);
  ```

Related Information

Components [page 720]
Descriptor for Applications, Components, and Libraries [page 734]

Setting the Default Binding Mode

The default binding mode applies when a model instance is created. You can overwrite the default binding mode after model creation.

- To change the default binding mode, call the `setDefaultBindingMode` method on the model as follows:

  ```
  var oModel = new sap.ui.model.json.JSONModel();
  oModel.setDefaultBindingMode(sap.ui.model.BindingMode.OneWay);
  ```

  In this example, all new bindings for the model will have the one-way binding mode by default.
- You can, however, only set supported binding modes as default binding mode. You can check if a binding mode is supported as follows:

  ```
  var oModel = new sap.ui.model.json.JSONModel();
  if (oModel.isBindingModeSupported(sap.ui.model.BindingMode.OneTime)) { // true
      oModel.setDefaultBindingMode(sap.ui.model.BindingMode.OneTime);
  }
  ```

  > **i** Note
  >
  > When you change the binding mode of an existing model instance, the existing bindings are not updated with the newly set binding mode.

Using Data Binding for Data Export

Data binding supports the export of data in a specific format so that the data can be used in other programs.

To export data, load the necessary modules as shown in the following example:

```
sap.ui.require(["sap/ui/core/util/Export", "sap/ui/core/util/ExportTypeCSV"],
function(Export, ExportTypeCSV) {
    // ...
});
```

The following code snippet uses a JSON model as basis for the export. You can also use other models, such as the OData model.

```
// "JSONModel" required from module "sap/ui/model/json/JSONModel"
var oModel = JSONModel([
    {
        firstname: "Al",
        lastname: "Dente"
    },
    {
        firstname: "Andy",
        lastname: "Friese"
    },
    {
        firstname: "Anita",
        lastname: "Mann"
    },
    {
        firstname: "Doris",
        lastname: "Schutt"
    },
```

```
    {
        firstname: "Kenya",
        lastname: "Dewit"
    }
]);
```

In the next step, create the data export object and pass the required information for the export to the object:

```
// "Export" required from module "sap/ui/core/util/Export"
var oExport = new Export({
    // "ExportTypeCSV" required from module "sap/ui/core/util/ExportTypeCSV"
    // Type that will be used to generate the content. Own ExportType's can be
created to support other formats
    exportType: new ExportTypeCSV({
        separatorChar: ";"
    }),
    // Pass in the model created above
    models: oModel,
    // binding information for the rows aggregation
    rows: {
        path: "/"
    },
    // column definitions with column name and binding info for the content
    columns: [
        {
            name: "First name",
            template: {
                content: {
                    path: "firstname"
                }
            }
        },
        {
            name: "Last name",
            template: {
                content: {
                    path: "lastname"
                }
            }
        }
    ]
});
```

The export class provides a generate method that triggers the generation process and returns a jQuery Promise object. The done handler is called when the generation process has finished. If you use the OData model, this happens asynchronously. The always handler is also called when the generation has failed. When the generation has finished and the export object is no longer needed, destroy the export object.

```
oExport.generate().done(function(sContent) {
    console.log(sContent);
}).always(function() {
    this.destroy();
});
```

The above example provides the following output:

```
First name;Last name
Al;Dente
Andy;Friese
Anita;Mann
Doris;Schutt
Kenya;Dewit
```

You can directly save the file by triggering a download. This calls the generate method internally and uses the file util class (`sap/ui/core/util/File`) to trigger the download.

```
oExport.saveFile().always(function() {
    this.destroy();
});
```

Export Types

You can use the CSV export type out of the box, or define other export types. The concept is similar to custom notepad controls, see Developing Controls [page 2158]. The generate method is called and should return the created file as a string.

> **i Note**
>
> Make sure that all values are encoded or escaped properly.

```
// "ExportType" required from module "sap/ui/core/util/ExportType"
// "encodeXML" required from module "sap/base/security/encodeXML"
ExportType.extend("my.own.ExportType", {
    init: function() {
        // Set default values
        this.setProperty("fileExtension", "mytype", true);
        this.setProperty("mimeType", "text/mytype", true);
        this.setProperty("charset", "utf-8", true);
    },
    generate: function() {
        var aBuffer = [];
        var oColumns = this.columnGenerator(),
            oColumn;
        aBuffer.push("<columns>");
        while (!(oColumn = oColumns.next()).done) {
            aBuffer.push("<column>" + encodeXML(oColumn.value.name) + "</
column>");
        }
        aBuffer.push("</columns>");

        var oRows = this.rowGenerator(),
            oRow;
        aBuffer.push("<rows>");
        while (!(oRow = oRows.next()).done) {
            var oCells = oRow.value.cells,
                oCell;
            aBuffer.push("<row>");
            aBuffer.push("<cells>");
            while (!(oCell = oCells.next()).done) {
                aBuffer.push("<cell");
                if (oCell.value.customData.color) {
                    aBuffer.push(" color=\"" +
encodeXML(oCell.value.customData.color) + "\"");
                }
                aBuffer.push(">");
                aBuffer.push(encodeXML(oCell.value.content));
                aBuffer.push("</cell>");
            }

        aBuffer.push("</rows>");

        return aBuffer.join("");
    }
```

```
});
```

Custom Data

It is also possible to pass `customData` to the cell template. This can be used to provide additional metadata to the `ExportType` (see example code above).

```
// column definitions with column name, binding info for the content and
additional custom data
columns: [
    {
        name: "First name",
        template: {
            content: {
                path: "firstname"
            },
            customData: [
                {
                    "key": "color",
                    "value": {
                        path: "color"
                    }
                }
            ]
        }
    },
    {
        name: "Last name",
        template: {
            content: {
                path: "lastname"
            }
        }
    }
]
```

Model Data

```
// "JSONModel" required from module "sap/ui/model/json/JSONModel"
var oModel = new JSONModel([
    {
        firstname: "Al",
        lastname: "Dente",
        color: "red"
    },
    {
        firstname: "Andy",
        lastname: "Friese",
        color: "black"
    },
    {
        firstname: "Anita",
        lastname: "Mann",
        color: "yellow"
    },
    {
        firstname: "Doris",
        lastname: "Schutt",
        color: "green"
    },
    {
        firstname: "Kenya",
        lastname: "Dewit",
        color: "blue"
    }
]);
```

Output

```
<columns>
    <column>First name</column>
    <column>Last name</column>
</columns>
<rows>
    <row>
        <cells>
            <cell color="red">Al</cell>
            <cell>Dente</cell>
        </cells>
    </row>
    <row>
        <cells>
            <cell color="black">Andy</cell>
            <cell>Friese</cell>
        </cells>
    </row>
    <row>
        <cells>
            <cell color="yellow">Anita</cell>
            <cell>Mann</cell>
        </cells>
    </row>
    <row>
        <cells>
            <cell color="green">Doris</cell>
            <cell>Schutt</cell>
        </cells>
    </row>
    <row>
        <cells>
            <cell color="blue">Kenya</cell>
            <cell>Dewit</cell>
        </cells>
    </row>
</rows>
```

Integration in Controls

`sap/ui/table/Table`

The `exportData` method creates an export instance and fills the rows and columns with the table's rows/column definition, if not defined otherwise. This also includes filters and sorters that have been applied to the columns.

```
// "ExportTypeCSV" required from module "sap/ui/core/util/ExportTypeCSV"
oTable.exportData({
    exportType: ExportTypeCSV()
})
.saveFile()
.always(function() {
    this.destroy();
});
```

Reusing UI Parts: Fragments

Fragments are light-weight UI parts (UI sub-trees) which can be reused, defined similar to views, but do not have any controller or other behavior code involved.

UI parts which are to be used in several views cannot be easily defined. They either have to be created as new controls, or they have to be created as views. Creating them as new controls results in a development overhead, while creating them as separate views results in a runtime overhead. In the latter case they would have a separate controller instead of having the same controller as the view does. Also, views and popup controls like dialogs do not go well together. The dialog content can be defined as a view but the dialog control itself always has to be written in the program.

To solve these issues, fragments have been introduced. They can be reused and, if source code is required and for event handler methods, they can connect to existing controllers of the "owning" view. This means that one important characteristic of fragments is that they are independent of the model-view-controller (MVC) concept and can be used without using MVC. However, if fragments are used together with views and controllers, fragments can make use of them and integrate them neatly.

Similar to `DocumentFragments` in HTML, the fragment itself has no HTML representation when it is inserted into the UI tree. Instead, its content is inserted. This means that fragments are not like controls, but more like a factory creating the contained controls. They support reuse and view modularization without adding overhead.

SAPUI5 provides different types of fragments:

- XML fragments
- HTML fragments
- JS fragments

More fragment types can be implemented and plugged in.

Defining a fragment is similar to defining views within a separate file. The fragments simply end with `*.fragment` instead of `*.view`. Also, the same rules for file location apply.

Related Information

Model View Controller (MVC) [page 784]
Walkthrough Step 16: Dialogs and Fragments [page 106]

HTML Fragments

HTML fragments have a similar syntax as HTML views, but without the `<template>` tag.

You can define a simple HTML fragment like this:

```
<div data-sap-ui-type="sap.m.Button" data-press="doSomething" data-text="Hello
        World"></div>
```

The fragment is stored as …/my/useful/UiPartZ.fragment.html and referenced as Fragment my.useful.UiPartZ. HTML fragments cannot specify a controller to be instantiated. They can use a controller for binding event handler methods, but only if the code instantiating them passes a controller.

XML Fragments

XML fragments are similar to XML view, but have no <View> tag as root element. Instead, there is an SAPUI5 control.

You define a simple XML fragment as shown in the following code snippet:

```
<Button xmlns="sap.m" id="btnInFragment" text="Hello
    World"/>
```

This simple UI definition can, for instance, be located in a file named …/my/useful/VerySimpleUiPart.fragment.xml, be referenced by its name my.useful.VerySimpleUiPart, and can be found by the module loading mechanism.

A slightly more complex XML fragment can be defined as follows:

```
<VBox xmlns="sap.m">
        <Label text="My Label inside an XML fragment"/>
        <Button id="btnInFragment" text="Hello World" press="doSomething"/>
        <Button text="{/someText}"/>
    </VBox>
```

The event handler is bound to the doSomething method of a controller. This is expressed by the doSomething value of the press event attribute. This means that this fragment can only be instantiated with a controller if the controller has this method. If not, the code throws an error.

You can see how the data binding syntax is the same as that of XML views. Of course, this requires the fragment to be placed into a part of the UI tree where the model is available.

JS Fragments

The structure of JS fragments is similar to the structure of the respective views: They have a name and an object with a createContent() function.

You define a simple JS fragment named my.useful.UiPartX as shown in the following code snippet:

```
sap.ui.jsfragment("my.useful.UiPartX", {
    createContent: function(oController ) {
        var oButton = new sap.m.Button({
            text:"Hello World",
            press:oController.doSomething
        });
        return oButton;
    }
});
```

The createContent() function is responsible for the UI definition and has to return a control. The definition can be created either inline or in a separate file, for instance in …/my/useful/UiPartX.fragment.js. The

`oController` is either already defined or it is null. In the first case, its methods can be used for the event handlers of controls.

Despite the many similarities to views, there are also differences: First of all, there is no `getControllerName()` method. Fragments cannot specify whether they have a controller. Whether `oController` is defined or not is not a decision of the fragment itself. Instead, it is decided by the code instantiating the fragment. If that code is part of a controller, it can pass a reference to itself to the fragment. This means there can be a dependency between controllers and fragments: Fragments may expect a controller to exist and to have certain methods. And controllers may expect certain controls to be in the fragment. This is in line with the purpose of fragments - to be very light-weight re-use entities that provide little encapsulation. For more encapsulation, views or even components are better suited.

Related Information

Components [page 720]
Views [page 787]

Instantiation of Fragments

SAPUI5 provides the generic function `sap.ui.fragment()` to instantiate fragments.

Comparing fragments to views, there is one important difference: Fragments are no controls. While views are control instances which have their own HTML and their own set of properties and may contain other controls, fragments just consist of their content. Views contain their content controls, while fragments consist of their content controls.

For example, when a fragment containing a button is instantiated, the result is just this button.

The generic function `sap.ui.fragment()` can be called with either the name, the type, and optionally a controller, or with a configuration object and an optional controller. It either returns the root control contained in the fragment or an array of root controls, depending on the type of the fragment. This fragment type is usually known in advance. Therefore, a specific method for each fragment type can be used to programmatically instantiate a fragment. You find more information on the instantiation process in the respective topics linked in the related link section below.

The different methods used for the instantiation of a fragment have the following commonalities:

- A fragment name must be given. This name must be resolvable to the fragment file URL by the SAPUI5 module loading mechanism. In case of JS fragments the name may also be defined inline.
- A controller can be optionally given. Some fragments may require a controller and certain methods to be present in this controller.
- An ID can be optionally given.
  If no ID is given, any control IDs specified in the fragment are used as is. The repeated use of a fragment can lead to duplicate IDs. One way to avoid that problem is to specify a unique fragment ID. For more information see Unique IDs [page 1011]. This ID will then be used as prefix for all controls in this fragment instance.

Programmatically Instantiating JS Fragments

For each fragment type, SAPUI5 provides a method that can be used to programmatically instantiate a fragment.

Context

To give an example of a programmatic instantiation of a JS fragment, you first have to define one. The following code presents an example definition:

```
sap.ui.jsfragment ( "my.useful.UiPartX",{
    createContent: function (oController ) {
        var oButton  = new sap.m.Button({
            text: "Hello World" ,
            press:oController.doSomething
        });
        return oButton;
    }
});
```

This fragment can be instantiated from a controller as follows:

```
var myButton = sap.ui.jsfragment("my.useful.UiPartX",this); // assuming "this"
is the controller
```

This button can now be used as if it had been created in a standard way. Note how a controller instance is passed as second parameter. This is required because that particular fragment tries to bind the button press handler to the method `doSomething` in the given controller. With no controller given, this would cause an error.

For fragments that are used several times, an ID for the fragment can be given optionally, see Unique IDs [page 1011]:

```
var myButton = sap.ui.jsfragment("someId", "my.useful.UiPartX", this); //
assuming "this" is the Controller
```

Within a JS view's `createContent()` method the fragment content could be included like this:

```
...
    createContent: function (oController ) {
        var hLayout = new sap.m.HBox ();
        ...
        var myFragment = sap.ui.jsfragment( "my.useful.UiPartX" , oController );
        // here the fragment is instantiated
        hLayout.addContent (myFragment );
        ...
        return hLayout ;
    }
...
```

The fragment content (= the button) would be added to the layout which is the content of this JSView. Other fragments not requiring a controller can of course be instantiated without passing a controller. But it also does not hurt to pass the controller - it is only used for setting up the event handlers (or within the `createContent()` method, in case of JS fragments).

Programmatically Instantiating XML Fragments

For each fragment type, SAPUI5 provides a method that can be used to programmatically instantiate a fragment.

Context

To give an example of a programmatic instantiation of an XML fragment, you first have to define one. The following code presents an example definition:

```
<Button xmlns="sap.ui.commons" id="btnInFragment" text="Hello World" />
```

This fragment can be instantiated from a controller as follows:

```
sap.ui.require(["sap/ui/core/Fragment"], function(Fragment) {
    Fragment.load({
        name: "my.useful.VerySimpleUiPart"
    }).then(function(myButton) {
        // ...
    });
});
```

> **i** Note
>
> This specific fragment does not use a controller; if controls inside a fragment needs methods that are to be defined in a controller, the controller has to be referred to in an additional parameter.

Fragments can be instantiated from JSViews, as well. Fragments of any type can be used within views of any type.

If XML fragments are used within XML views, giving the view ID as fragment ID will allow calling `this.byId(…)` in the view's controller to retrieve controls inside the fragment. The following code inside the controller will instantiate the above fragment with the `Button` and then again retrieve the `Button`:

```
sap.ui.require(["sa/ui/core/Fragment"], function(Fragment) {
    Fragment.load({
        name: "my.useful.VerySimpleUiPart"
    }).then(function(myButton) {
        // ...
    });
});
var theSameButton = this.byId("btnInFragment"); // returns the button in the
fragment
```

Programmatically Instantiating HTML Fragments

For each fragment type, SAPUI5 provides a method that can be used to programmatically instantiate a fragment.

Context

To give an example of a programmatic instantiation of an HTML fragment, you first have to define one. The following code presents an example definition:

```
<div data-sap-ui-type="sap.m.Button" data-press="doSomething" data-text="Hello
World"></div>
```

This fragment can be instantiated from a controller as follows:

```
sap.ui.require(["sap/ui/core/Fragment"], function(Fragment) {
    Fragment.load({
        type: "HTML",
        name: "my.useful.UiPartZ",
        controller: oController // this specific fragment again needs a
controller
    }).then(function(myButton) {
        // ...
});
```

This instantiation can be done at any place in the code, given that a controller is available and the returned `button` can be used like any `button`.

Instantiating Fragments in Declarative Views

Example, how all three types of fragments can be instantiated in an XML view.

Context

In XML views, fragments are used like regular controls, or more precisely, like views.

The following code example shows an XML view that includes all three types of fragments, that is an XML fragment, a JS fragment and an HTML fragment. Each type is instantiated once without a given ID and once with a given ID. These fragment references basically work like import statements including the fragment content controls.

```
<mvc:View xmlns:mvc="sap.ui.core.mvc" xmlns:core="sap.ui.core"
controllerName="testdata.fragments.XMLViewController" >

    <core:Fragment               fragmentName="my.useful.SimpleUiPart"
type="XML" />
    <core:Fragment id="xmlInXml" fragmentName="my.useful.SimpleUiPart"
type="XML" />
```

```
    <core:Fragment              fragmentName="my.useful.UiPartX" type="JS" />
    <core:Fragment id="jsInXml" fragmentName="my.useful.UiPartX" type="JS" />

    <core:Fragment                fragmentName="my.useful.UiPartZ" type="HTML" />
    <core:Fragment id="htmlInXml" fragmentName="my.useful.UiPartZ" type="HTML" />
</mvc:View>
```

Related Information

Unique IDs [page 1011]
Using Other Objects Instead of Controllers [page 1010]

Using Other Objects Instead of Controllers

For the instantiation of fragments, the oController object must not necessarily be a controller. It can also be another object.

Context

The oController object given when instantiating a fragment does not need to be an object of type `sap.ui.core.mvc.Controller`. It is entirely up to the fragment what to expect from this object. This object is passed to the `createContent` method of JS fragments. In case of the declarative fragment types, that is XML, or HTML fragments, the event handler methods are searched on this object. This means that in most cases instead of a real controller object any JavaScript object could be given - provided it has the required methods.

The following example of an HTML fragment can be used in an environment where no MVC is used.

```
var oDummyController = {
    doSomething: function() {
        // do whatever should happen when the button in the fragment is pushed...
    }
};
var myButton  = sap.ui.htmlfragment( "my.useful.UiPartZ", oDummyController);
// this specific fragment needs a controller and gets a dummy controller here.
```

Inline Definition and Instantiation of Fragments

Instead of defining fragments externally in a separate file, they can also be defined inline and can be instantiated immediately.

The content definition and also the instantiation syntax are just the same compared to an instantiation in a program. However, instead of the "fragmentName" the "fragmentContent" needs to be given. This feature can

both be used for prototyping or for dynamic fragment composition or for loading fragment content from sources which are not accessible by the normal module loading mechanism. In general, inline definition of fragments plays only a minor role.

JS fragment definitions can be done both inline and within a separate file without any changes. Example inline definitions of XML and HTML fragments are displayed in the following code examples:

Example of an Inline XML Fragment

```
// define the XML fragment as a string (or load it from anywhere)
var myXml = '<Panel xmlns="sap.m" text="Hello World"><Button text="Hello
World"></Button></Panel>';
// use this XML string as "fragmentContent"
var oFragment = sap.ui.xmlfragment({fragmentContent:myXml}); // oFragment is now
the Panel Control
// put the Fragment content into the document
oFragment.placeAt('content');
```

Example of an Inline HTML Fragment

```
// define the HTML fragment as a string (or load it from anywhere)
var myHtml = '<div data-sap-ui-type="sap.m.Button" data-text="Hello World"></
div>';
// use this HTML string as "fragmentContent"
var oFragment = sap.ui.htmlfragment({fragmentContent:myHtml}); // oFragment is
now the Button Control
// put the Fragment content into the document
oFragment.placeAt('content');
```

Related Information

Programmatically Instantiating JS Fragments [page 1007]

Unique IDs

You can use a unique ID for a fragment that will be used as a prefix for all controls in a fragment instance.

In SAPUI5, IDs are either automatically generated or given as string constants by the application developer.

In MVC views, another type of reusable components, all IDs that are given as static strings inside the declarative views (XML, HTML, JSON) are automatically prefixed with the view ID. For JS views and for controls created in the controller code of declarative views, no automated prefixing exists. Instead, we recommend using the `View.createId()` method to prefix the ID and ensure there are no ID collisions even when the view is used multiple times within the same page.

The method `View.byId()` is used to handle these prefixed IDs.

Fragments, however, are meant to be a more light-weight concept of separating and reusing UI parts. What's more, there are no fragment instances involved which could have IDs and handle ID prefixing helper functions (in contrast to view instances). For this reason, stable IDs in fragments are used "as is" by default, meaning they are not prefixed to make them unique. If no ID is given, it will be generated. However, if a fragment is intended to be used more than once within one application, the prefixing mechanism can still be used by giving an ID when instantiating the fragment.

The basic principle is as follows: When a control ID is defined, declarative views add a prefix and the code in views should add a prefix. Fragments also add a prefix if they have a defined ID.

To get rid of the prefixes, the instance method `View.byId()` can be used with the static method `sap.ui.core.Fragment.createID()` if required, that is, if a fragment added a prefix.

> **i** Note
>
> Do **not** rely on the specific prefixing syntax because it may change at some point. Always use methods like `byId()` and `createId()`.

Related Information

IDs in Declarative XML or HTML Fragments [page 1012]
IDs in JS Fragments [page 1013]
IDs of Fragments in Views [page 1013]

IDs in Declarative XML or HTML Fragments

If a fragment with a control ID is instantiated twice without giving an ID, a duplicate ID error occurs.

Given the following XML fragment example:

```
<HBox xmlns="sap.m">
   <Button                    text="Hello World" />
   <Button id="btnInFragment" text="Hello World" />
</HBox>
```

The first button will always have a generated ID, as, for instance, `__button2`. This is regardless of how the fragment is instantiated or whether it resides inside a view.

The second button will either have the ID `btnInFragment`, in case the fragment is instantiated without giving an ID. This approach is easy to use, but implies the risk of ID collisions when instantiated multiple times:

```
sap.ui.htmlfragment("my.useful.UiPartZ"); // Button ID will not be prefixed
```

The other possible ID of the second button is `myFragment--btnInFragment`, in case the fragment is instantiated giving the ID `myFragment`. You should not rely on the exact syntax of this prefixing.

```
sap.ui.htmlfragment("myFragment", "my.useful.UiPartZ");
```

It is, however, possible that a containing view may add its prefix. For more information, see IDs of Fragments in Views [page 1013].

IDs in JS Fragments

The fragment logic of JS fragments cannot influence the IDs of controls that are created in the `createContent()` method.

This behavior is similar to how JS views behave regarding IDs. When a stable ID is given within a JS fragment, the `this.createId()` method should be used:

```
createContent: function(oController ) {
    var oButton  = new sap.m.Button(this.createId("btnInJsFragment"), {
    // use createId() to let the fragment influence the ID
        text: "Hello JS World"
    });
    return oButton ;
}
```

If an ID is given when the fragment is instantiated, `createId()` will add it as prefix. Else, `createId()` will leave the given ID untouched.

It is, however, possible that a containing view may add its prefix. For more information, see IDs of Fragments in Views [page 1013].

IDs of Fragments in Views

For fragments that are used within declarative views, generated IDs are not prefixed.

The following rules apply for given IDs:

- Given IDs are prefixed with only the view ID when no fragment ID was given
- Given IDs are prefixed with both view ID and fragment ID when a fragment ID was given

Related Information

Example: JS Fragments Used in XML Views [page 1014]

Retrieving Control Instances by Their ID

A control instance can be found in a fragment by means of its ID.

Context

Due to the above prefixing that guarantees unique IDs, there are different cases possible which require different calls.

Assuming the control has the ID `myControl`, there are two ways how to retrieve it by its ID.

- Retrieving a control instance when the fragment is not part of a view
  - When no fragment ID was given: `myControl = sap.ui.getCore().byId("myControl")`
  - When a fragment ID `myFrag` was given: `myControl = sap.ui.core.Fragment.byId("myFrag", "myControl")`
- Retrieving a control instance when the fragment is embedded into a view and the code is inside a controller. The controller is called `this` in the following examples.
  - When no fragment ID was given: `myControl = this.byId("myControl")`
  - When a fragment ID `myFrag` was given: `myControl = this.byId(sap.ui.core.Fragment.createId("myFrag", "myControl"))`

Example: JS Fragments Used in XML Views

Example of a JS fragment used in an XML view

The example uses different combinations. Make sure that the `sap-ui-core.js` script location points to an existing SAPUI5 installation.

```
<!DOCTYPE html>
<html>
    <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta content="charset=utf-8">

    <title>JSFragment used in XmlView</title>

    <!-- Load UI5, select theme and the "sap.m" control library -->
    <script id='sap-ui-bootstrap' type='text/javascript'
            src='/sapui5/resources/sap-ui-core.js'
            data-sap-ui-theme='sap_belize'
            data-sap-ui-libs='sap.m,sap.ui.layout'></script>


    <!-- DEFINE RE-USE COMPONENTS - NORMALLY DONE IN SEPARATE FILES -->

    <!-- define a JS Fragment - normally done in a separate file -->
    <script>
        // define a new (simple) View type
        // ...alternatively this can be done in an XML file without JavaScript!
        sap.ui.jsfragment("my.own.frag", {

            // defines the UI of this View
```

```
            createContent: function() {
                // button text is bound to Model, "press" action is bound to
Controller's event handler
                return [
                    new sap.m.Button({text:'my Fragment Button'}),
                    new sap.m.Button(this.createId("btn2"), {text:'my second
Fragment Button'})
                ]
            }
        });
    </script>


    <!-- define an XMLView - normally done in a separate file -->
    <script id="view1" type="sapui5/xmlview">
        <mvc:View xmlns:core="sap.ui.core" xmlns:layout="sap.ui.layout"
xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
                controllerName="my.own.controller" xmlns:html="http://www.w3.org/
1999/xhtml">
            <layout:VerticalLayout id="vl">
                <Button text="Find controls by ID" press="findControls"></Button>
                <Text text="Fragment referenced inline, no Fragment ID:" />
                <core:Fragment fragmentName='my.own.frag' type='JS' />
                <Text text="Fragment referenced inline, with Fragment ID
'myFrag':" />
                <core:Fragment id="myFrag" fragmentName='my.own.frag'
type='JS' />
            </layout:VerticalLayout>
        </mvc:View>
    </script>


    <script>
        // define a new (simple) Controller type
        sap.ui.controller("my.own.controller", {

            // implement an event handler in the Controller
            findControls: function() {
                // Fragment is instantiated within an XMLView => all GIVEN IDs
are prefixed with the
                // View ID and View.byId() needs to be used to find the controls
                var b1 = null; // ID is generated: "__button1"
                var b2 = this.byId("btn2"); // Button ID is given, Fragment has
no ID: "myView--btn2"
                var b3 = null // Fragment has an ID, but Control ID is generated
and hence not prefixed: "__button2"
                var b4 = this.byId(sap.ui.core.Fragment.createId("myFrag",
"btn2")); // Button and Fragment ID are given, let the Fragment construct the
prefixed ID and then let the View search the again prefixed ID
                alert("Controls in Fragment:\nButton 1: has no given ID, cannot
be found\nButton 2: " + b2 + "\nButton 3: has no given ID, cannot be found
\nButton 4: " + b4);
            }
        });



        /*** THIS IS THE "APPLICATION" CODE ***/

        // instantiate the View
        var myView = sap.ui.xmlview("myView",
{viewContent:jQuery('#view1').html()}); // accessing the HTML inside the script
tag above

        // put the View onto the screen
        myView.placeAt('content');
```

```
    </script>

    </head>
    <body class='sapUiBody'>
        <div id='content'></div>
    </body>
</html>
```

Dialogs and other Popups as Fragments

You can use fragments to declaratively define dialogs and other popup controls which are not part of the normal page UI structure.

This is a huge advantage over views that do not support this use. The main reason for this is that the view control is always a wrapper around the view content.

Defining Dialogs as Fragments

You can use fragments for the definition of dialogs.

Context

To use fragments for defining popups, just let the root control of the fragment be a dialog or similar control.

The following shows an XML fragment dialog example:

```
<Dialog xmlns="sap.m" title="XML Fragment Dialog">
    <TextView text="{/dialogText}" />
    <buttons>
        <Button text="Close" press="closeDialog"/>
    </buttons>
</Dialog>
```

Other fragment types are used the same way to define, for instance, a dialog as fragment.

For example, in JS fragments, the `createContent()` method returns a dialog control:

```
sap.ui.jsfragment("testdata.fragments.JSFragmentDialog", {
    createContent: function(oController) {
     var oDialog = new sap.m.Dialog({
            title: "JavaScript Fragment Dialog",
            content: [
                new sap.m.Input({
                    text: "{/dialogText}"
                })
            ],
            buttons: [
                new sap.m.Button({
                    text: "Close",
                    press: function(){
                        oDialog.close();
```

```
                    }
                })
            ]
        return oDialog;
    }
});
```

Using Dialogs Defined as Fragments

The fragment instantiation function always returns the fragment's root control, which is a dialog control that can be used like any dialog.

Context

In the following example, the dialog is opened immediately:

```
// "Fragment" required from module "sap/ui/core/Fragment"
Fragment.load({type: "XML", name:
"testdata.fragments.XMLFragmentDialog"}).then(function(oDialog) {
    oDialog.open();
});
```

Note that any global model is available for data binding within this dialog. Also any model set on the dialog itself. However, if this dialog is opened from a controller, the model of this controller's view is NOT automatically available within the dialog fragment. The reason for this is that the dialog is not part of the view UI. If the above code for opening the fragment dialog is part of a controller, it could set the view's model on the dialog:

```
// "Fragment" required from module "sap/ui/core/Fragment"
Fragment.load({type: "XML", name:
"testdata.fragments.XMLFragmentDialog"}).then(function(oDialog) {
    oDialog.setModel(this.getView().getModel());
    oDialog.open();
}.bind(this));
```

Alternatively, the special aggregation `dependents` of `sap.ui.core.Element` can be used to connect the dialog to the lifecycle management and data binding of the view:

```
// "Fragment" required from module "sap/ui/core/Fragment"
Fragment.load({type: "XML", name:
"testdata.fragments.XMLFragmentDialog"}).then(function(oDialog) {
    this.getView().addDependent(oDialog);
    oDialog.open();
}.bind(this));
```

Fragments with Multiple Root Nodes

XML fragments and JS fragments can have more than one root control.

In JS fragments, the `createContent()` method can optionally return an array of controls:

```
// "Label" required from module "sap/m/Label"
// "Input" required from module "sap/m/Input"
// "Button" required from module "sap/m/Button"
sap.ui.jsfragment("sap.ui.core.fragmenttest.MultiRootFragment", {
    createContent: function(oController) {
        var oLabel = new Label({text:"These controls are within one multi-root
Fragment:"});
        var oInput = new Input();
        var oButton = new Button({text: "Still in the same Fragment"});
        return [ oLabel, oInput, oButton ];
    }
});
```

As XML documents need to have exactly one root node, to achieve XML fragments with multiple root nodes, an additional `<FragmentDefinition>` tag needs to be added as root element containing the actual root controls. This `<FragmentDefinition>` tag will not have any representation in HTML at runtime; the children are added directly to wherever this fragment is placed.

```
<core:FragmentDefinition xmlns="sap.m" xmlns:core="sap.ui.core">
    <Label text="These controls are within one multi-root Fragment:" />
    <Input />
    <Button text="Still in the same Fragment" />
</core:FragmentDefinition>
```

> **i Note**
>
> For HTML fragments this feature is currently not available.

XML Templating

The XML templating concept enables you to use an XML view as a template. This template is transformed by an XML preprocessor on the source level, the XML DOM, at runtime just before an SAPUI5 control tree is created from the XML source.

The label texts and binding paths in the example below come from SAP Annotations for OData Version 2.0 (http://www.sap.com/Protocols/SAPData) such as `sap:semantics`, and from OData Version 4.0 annotations such as `com.sap.vocabularies.UI.v1.Badge`. Much more complex tasks than shown in this simple example are possible.

> **i Note**
>
> HTML templating is no longer supported as of version 1.56.

The transformation happens if a preprocessor for XML is called when the view is created, see lines 4 and 5 in the *Calling the XML Preprocessor* example. This preprocessor can be given one or more models along with a corresponding binding context, see lines 6 and 9; this concept exists for any SAPUI5 control's constructor. Typically, an OData model's meta model is given, along with the meta context corresponding to a data path.

XML templating operates on meta data. If the data changes, the XML templating can **not** be executed again. This is due to the processing time. Only the resulting bindings are evaluated again.

If the view is loaded asynchronously, fragments and required modules are loaded asynchronously, too.

> **!** Restriction
>
> XML templating is not directly supported with routing, that is, there is no way to declare that the XML Preprocessor should run on the target view of a route. Instead, you should define a JavaScript view as the route's target and use that view's `createContent` method to create an XML view with templating. In case you need access to models (which are not yet available in that hook), you should return some dummy content first (for instance sap.m.HBox), register to the view's modelContextChange event and create the inner view in that event's handler, finally adding it to the dummy content.
>
> JavaScript Target View For Routing
>
> ```
> sap.ui.jsview("some.package.RouteTargetView", {
>     createContent : function () {
>         return sap.ui.view({
>             async : true,
>             preprocessors : {
>                 xml : {
>                     // ...
>                 }
>             },
>             type : sap.ui.core.mvc.ViewType.XML,
>             viewName : "some.package.TemplateView"
>         });
>     }
> });
> ```

In the example, `sPath = "/ProductSet('HT-1021')/ToSupplier"` and the corresponding meta context point to `"/dataServices/schema/0/entityType/0"` (the entity type `BusinessPartner`). The resulting view is bound to the data path within the OData model in order to display the supplier of that product.

Example: Calling the XML Preprocessor

```
1   View.create({
2      async : true,
3      models : oModel,
4      preprocessors : {
5         xml : {
6            bindingContexts : {
7               meta : oMetaModel.getMetaContext(sPath)
8            },
9            models : {
10              meta : oMetaModel
11           }
12        }
13     },
14     type : ViewType.XML,
15     viewName : "sap.ui.core.sample.ViewTemplate.tiny.Template"
16  }).then(function (oTemplateView) {
17     oTemplateView.bindElement(sPath);
18     ...
19  }
```

The XML preprocessor traverses the view's XML DOM in a depth-first, parent-before-child manner and does the following:

- All XML attributes which represent an available binding, that is, a binding based only on models available to the preprocessor, are replaced by the result of that binding. Formatters and so on can be used as with any SAPUI5 binding.
- XML fragments are inlined; that is, the reference is replaced by the fragment's XML DOM and preprocessing takes place on that DOM as well.
- The preprocessing instructions `<template:with>`, `<template:if>` and `<template:repeat>` are processed.

Example: Component.js

Template.view.xml

Field.fragment.xml

Resulting XML View

See sample sap.ui.core.sample.ViewTemplate.tiny. This sample is based on OData Version 4.0 annotations. It consists of the following three pieces:

- A component controller that creates an OData model (line 17), waits for the meta model to be loaded (line 28) and then creates a template view (line 29) as its content. A preprocessor for XML is requested (line 31) and settings are passed to it, namely the meta model and the binding context that identifies the starting point within that model. The resulting view is bound to the actual data (model and path).
- A template view that includes a fragment twice (line 20 and 25) to demonstrate how to reuse code.
- An XML fragment that demonstrates a simple test (line 10), using expression binding.

> → Tip
>
> You can find more elaborate XML templating samples here: XMLView.
>
> Take a look at the demo scenario for a complete overview of all OData v4 notations.

> ⚠ Caution
>
> The OData model is based on `GWSAMPLE_BASIC` and will not work unless a suitable proxy for back-end access is used. For simplicity, no mock data is included in this example.
>
> For more information, see the Help topic, Sample Service - Basic.

```
1   /*!
2    * ${copyright}
3    */
4
5   /**
6    * @fileOverview Application component to display supplier of "/
ProductSet('HT-1021')"
7    *   from GWSAMPLE_BASIC via XML Templating.
8    * @version @version@
9    */
10  sap.ui.define([
```

```
    'sap/m/VBox',
    'sap/ui/core/UIComponent',
    'sap/ui/core/mvc/View',
    'sap/ui/core/mvc/ViewType',
    'sap/ui/model/odata/v2/ODataModel'
], function (VBox, UIComponent, View, ViewType, ODataModel) {
    "use strict";

    return
UIComponent.extend("sap.ui.core.sample.ViewTemplate.tiny.Component", {
        metadata : "json",

        createContent : function () {
           var oModel = new ODataModel(
              "proxy/sap/opu/odata/IWBEP/GWSAMPLE_BASIC/", {
                 annotationURI : "proxy/sap/opu/odata/IWFND/CATALOGSERVICE;v=2"
                 + "/
Annotations(TechnicalName='ZANNO4SAMPLE_ANNO_MDL',Version='0001')/$value",
                 json : true,
                 loadMetadataAsync : true
              }),
              oMetaModel = oModel.getMetaModel(),
              sPath = "/ProductSet('HT-1021')/ToSupplier",
              oViewContainer = new VBox();

           oMetaModel.loaded().then(function () {
              View.create({
                 async : true,
                 models : oModel,
                 preprocessors : {
                    xml : {
                       bindingContexts : {
                          meta : oMetaModel.getMetaContext(sPath)
                       },
                       models : {
                          meta : oMetaModel
                       }
                    }
                 },
                 type : ViewType.XML,
                 viewName : "sap.ui.core.sample.ViewTemplate.tiny.Template"
              }).then(function (oTemplateView) {
                 oTemplateView.bindElement(sPath);
                 oViewContainer.addItem(oTemplateView);
              });
           });

           // Note: synchronously return s.th. here and add content to it
later on
           return oViewContainer;
        }
    });
});
```

```
<mvc:View
    xmlns="sap.m"
    xmlns:core="sap.ui.core"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">

    <!-- "meta" model's binding context MUST point to an entity type -->
    <template:with path="meta>com.sap.vocabularies.UI.v1.Badge" var="badge">
        <form:SimpleForm>
            <form:title>
```

```
12                  <core:Title text="{path: 'badge>HeadLine', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
13              </form:title>
14
15              <Label text="{path: 'badge>Title/Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
16              <Text text="{path: 'badge>Title/Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
17
18              <Label text="{path: 'badge>MainInfo/Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
19              <template:with path="badge>MainInfo" var="field">
20                  <core:Fragment
fragmentName="sap.ui.core.sample.ViewTemplate.tiny.Field" type="XML"/>
21              </template:with>
22
23              <Label text="{path: 'badge>SecondaryInfo/Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
24              <template:with path="badge>SecondaryInfo" var="field">
25                  <core:Fragment
fragmentName="sap.ui.core.sample.ViewTemplate.tiny.Field" type="XML"/>
26              </template:with>
27          </form:SimpleForm>
28      </template:with>
29  </mvc:View>
```

```
1   <core:FragmentDefinition
2       xmlns="sap.m"
3       xmlns:core="sap.ui.core"
4       xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
5
6       <!-- "field" MUST point to a
com.sap.vocabularies.Communication.v1.DataField -->
7       <HBox>
8           <template:with path="field>Value"
helper="sap.ui.model.odata.AnnotationHelper.resolvePath" var="target">
9               <!-- go to entity type's property and check SAP Annotations for
OData Version 2.0 -->
10              <template:if test="{= ${target>sap:semantics} === 'tel'}" >
11                  <core:Icon src="sap-icon://phone" width="2em"/>
12              </template:if>
13          </template:with>
14          <Text text="{path: 'field>Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
15      </HBox>
16  </core:FragmentDefinition>
```

The result is equivalent to the following handwritten XML view. Any references to the meta model are gone. Type information has been inserted into the bindings and an `"odata.concat"` expression for `badge>MainInfo/Value` has been processed by `sap.ui.model.odata.AnnotationHelper.format`, concatenating the company name and legal form.

```
<mvc:View xmlns="sap.m" xmlns:core="sap.ui.core" xmlns:form="sap.ui.layout.form"
xmlns:mvc="sap.ui.core.mvc">
  <form:SimpleForm>
    <form:title>
      <core:Title text="{path : 'BusinessPartnerID', type :
'sap.ui.model.odata.type.String', constraints :
{'maxLength':'10','nullable':'false'}}"/>
    </form:title>
    <Label text="Name"/>
    <Text text="{path : 'CompanyName', type : 'sap.ui.model.odata.type.String',
constraints : {'maxLength':'80'}} {path : 'LegalForm', type :
'sap.ui.model.odata.type.String', constraints : {'maxLength':'10'}}"/>
```

```
    <Label text="Phone"/>
    <HBox>
      <core:Icon src="sap-icon://phone" width="2em"/>
      <Text text="{path : 'PhoneNumber', type :
'sap.ui.model.odata.type.String', constraints : {'maxLength':'30'}}"/>
    </HBox>
    <Label text="Web"/>
    <HBox>
      <Text text="{path : 'WebAddress', type : 'sap.ui.model.odata.type.String',
constraints : {}}"/>
    </HBox>
  </form:SimpleForm>
</mvc:View>
```

Summary

Overall, XML templating is based on:

- Preprocessing instructions such as `<template:if>`, which can be used inside XML views
- An OData meta model which offers a unified access to both, OData V2 metadata and OData V4 annotations
- A set of OData type implementations which add knowledge of OData types to SAPUI5
- Expression binding which facilitates the use of expressions instead of custom formatter functions
- The helper class `sap.ui.model.odata.AnnotationHelper` that offers formatter and helper functions to be used inside XML template views. It knows about the OData meta model and helps with standard tasks like accessing a label or providing a runtime binding path. It brings in the OData types, along with their facets. Its output uses expression binding, if needed.

> **i Note**
>
> XML Templating works almost the same for OData V4 as for OData V2; for the differences see the *Annotations* section in Meta Model for OData V4 [page 956].

Related Information

Meta Model for OData V2 [page 899]
Expression Binding [page 845]
SAP Annotations for OData Version 2.0
`sap.ui.model.odata.AnnotationHelper`

Preprocessing Instructions

Preprocessing intructions are processed by the XML preprocessor when it traverses the view's `XML DOM`.

> **i** Note
>
> A model name can be seen as a variable with a value that consists of two parts: a model instance and a binding context path.

You can base conditions on the available variables. XML attributes that represent an available binding are replaced automatically.

with

The `<template:with>` instruction can be used to change a variable's value or to add a variable with a new name.

This changed variable is available only within the scope of the `with` instruction. In the example titled **"with" Template**, `"meta>com.sap.vocabularies.UI.v1.Badge"` refers to `"/dataServices/schema/0/entityType/0/com.sap.vocabularies.UI.v1.Badge"` within `oMetaModel`. `"badge"` then becomes a valid model name while processing the content of the `with` instruction, in addition to the existing ones:

- meta = `oMetaModel, "/dataServices/schema/0/entityType/0"`
- badge = `oMetaModel, "/dataServices/schema/0/entityType/0/com.sap.vocabularies.UI.v1.Badge"`

If you omit the "var" attribute, the same model name will be reused; for example, "meta" in our example, and the `with` instruction changes the binding context locally. A new variable name improves readability.

Example: "with" Template

```
<template:with path="meta>com.sap.vocabularies.UI.v1.Badge" var="badge">
    <!-- ... -->
</template:with>
```

A helper can be called from a `with` instruction. It receives an `sap.ui.model.Context` object identifying the model and path from the instruction's "path" property and may return one of the following:

- A `sap.ui.model.Context` object that is assigned to the variable
- A non-empty string that is used as a path within the same model and assigned to the variable
- Undefined, in which case the helper is ignored and the original path is assigned to the variable
- A thenable (usually a Promise) resolving with any of the above, if the view is loaded asynchronously.

The helper can analyze the object the path points to and derive a "resolved path" from that, such as by normalization or following references. Typically, it only points to a different path, but it can even change the model instance, such as jumping from a data model to its meta model or jumping to a resource model, and so on.

The example titled **"with" Template Including Helper** assumes that "field" is an OData meta model with a current binding context that points to a field inside some annotation, such as `"/dataServices/schema/0/entityType/0/com.sap.vocabularies.UI.v1.Badge/MainInfo"`. The helper function `sap.ui.model.odata.AnnotationHelper.resolvePath` is used to follow the field value's `path` property. For more information, see Annotation Helper [page 1035] and API Reference: `sap.ui.model.odata.AnnotationHelper.resolvePath`.

It returns a path inside the meta model which refers to the corresponding property's meta data, such as `"/dataServices/schema/<i>/entityType/<j>/property/<k>"`. This result is in turn assigned by the `with` instruction to the variable named "target".

Example: "with" Template Including Annotation Helper

```
<template:with path="meta>Value"
helper="sap.ui.model.odata.AnnotationHelper.resolvePath" var="target">
  <template:if test="{= {target>sap:semantics} === 'email'}" >
    <core:Icon src="sap-icon://email" />
  </template:if>
  <template:if test="{= {target>sap:semantics} === 'tel'}" >
    <core:Icon src="sap-icon://phone" />
  </template:if>
</template:with>
```

Related Information

Annotation Helper [page 1035]

repeat

The `<template:repeat` instruction iterates the `sap.ui.model.ListBinding` given by the `list` attribute.

> **i Note**
>
> Sorting and filtering is already supported by the list binding via an extended syntax. For more information, see Sorting, Grouping, and Filtering for List Binding [page 833].

The `var` attribute holds the name of the loop variable which can be used to access the current list element in a child element of `repeat`. In the preprocessing, `repeat` is replaced by multiple clones of its content, one clone per list element, with each clone again preprocessed as if it were contained in a `with` instruction defining the loop variable.

The following example iterates all fields in the identification annotation from the SAP UI vocabulary in the currently referenced element of the model `meta` and displays a label and content for each field. It is completely transparent to the `repeat` implementation whether the list binding refers to data or meta data. The templating engine is replacing "template time" binding expressions which refer to meta data with corresponding runtime

binding expression which refer to data. The formatter `sap.ui.model.odata.AnnotationHelper.format` is used, which encapsulates knowledge about the SAP UI vocabulary and so on.

Example: Template for "repeat" Instruction

```
<template:repeat list="{meta>com.sap.vocabularies.UI.v1.Identification}"
var="field">
  <Label text="{path: 'field>Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
  <Text text="{path: 'field>Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
</template:repeat>
```

Example: Output Template for the "repeat" Instruction (in Memory Only)

```
<Label text="Product ID" />
<Text text="{path:'ProductID',type:'sap.ui.model.odata.type.String',constraints:
{"nullable":false,"maxLength":10}}" />
<Label text="Price" />
<Text text="{path:'Price/
Amount',type:'sap.ui.model.odata.type.Decimal',constraints:{"precision":
13,"scale":3}}" />
<Label text="Category" />
<Text text="{path:'Category',type:'sap.ui.model.odata.type.String',constraints:
{"maxLength":40}}" />
<Label text="Supplier" />
<Text
text="{path:'SupplierName',type:'sap.ui.model.odata.type.String',constraints:
{"maxLength":80}}" />
```

Example: Template for the "repeat" Instruction with `startIndex` and `length`

You can start the iteration at an index other than 0 or limit the length of the iterated list in the usual manner. For this, specify `startIndex` and `length`. Both are optional and the defaults are 0 for `startIndex` and full length for `length`.

```
<template:repeat
list="{path:'entityType>com.sap.vocabularies.UI.v1.Identification',startIndex:
1,length:3}" var="field">
  <!-- ... -->
</template:repeat>
```

As the OData meta model supports filtering by name, you can repeat all `FieldGroup` annotations regardless of their qualifier.

> **i Note**
>
> In JSON content, the annotation can be called `com.sap.vocabularies.UI.v1.FieldGroup` or `com.sap.vocabularies.UI.v1.FieldGroup#Dimension`, and so on, depending on its qualifier. The filter that is used in the following code snippet for the `<template:repeat>` instruction uses the special path name `@sapui.name` which refers back to the name of the object that is inspected for filtering. This name is, for example, `com.sap.vocabularies.UI.v1.FieldGroup#Dimension`.

Example: Filter By Annotation Term

```
<template:repeat list="{path:'entityType>', filters: {path: '@sapui.name',
operator: 'StartsWith', value1: 'com.sap.vocabularies.UI.v1.FieldGroup'}}"
var="fieldGroup">
  <form:SimpleForm>
    <form:title>
```

```
      <core:Title text="{path: 'fieldGroup>Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
    </form:title>
    <template:repeat list="{fieldGroup>Data}" var="field">
      <Label text="{path: 'field>Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
      <core:Fragment
fragmentName="sap.ui.core.sample.ViewTemplate.scenario.Field" type="XML"/>
    </template:repeat>
  </form:SimpleForm>
</template:repeat>
```

if

The `<template:if>` instruction evaluates a condition expressed via existing SAPUI5 data binding features, such as extended syntax; in the preprocessing it is removed or replaced by its child elements based on the value of this condition.

For more information, see Sorting, Grouping, and Filtering for List Binding [page 833].

You set the condition to the `test` attribute of the `if` instruction. It is recommended to use expression binding if you need to write logical expressions or convert values into Booleans. You can also use formatter functions, as with any SAPUI5 binding, such as `sap.ui.model.odata.AnnotationHelper.isMultiple`. For more information, see `sap.ui.model.odata.AnnotationHelper.isMultiple` in the API Reference.

> **i Note**
>
> The test condition is treated as a property binding and the result is converted to the Boolean type according to the usual JavaScript rules, with the exception of the string `"false"`, which is converted to the Boolean `false` for convenience. For more information about the JavaScript rules, see the ECMAScript® Language Specification on the ECMA International website.

Example: "if" Instruction to Include an Image Only if the URL is Set

The output of the template below after preprocessing is as follows: If the test condition does not hold, the `<template:if>` node is dropped and if the test condition holds, the node is replaced by its content.

```
<template:if test="{meta>ImageUrl}">
  <Image src="{path: 'meta>ImageUrl', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
</template:if>
```

> **i Note**
>
> The example above shows a shortcut syntax where `<template:then>` can be omitted in case no `<template:else>` is present.

Example: "if/then/else" Instruction to Include an Image Only if the URL is Set and Display a Title Otherwise

The syntax of this example is more complex due to the additional `<template:then>/<template:else>` elements. The output is the `<template:if>` node replaced by the content of the appropriate child node.

Sample Code

```
<template:if test="{meta>ImageUrl}">
  <template:then>
    <Image src="{path: 'meta>ImageUrl', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
  </template:then>
  <template:else>
    <Text text="{path: 'meta>Title/Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
  </template:else>
</template:if>
```

Example: if/then/else Instruction

It is even possible to check multiple conditions in one `<template:if>` construct using the `<template:elseif>` element as shown in the example below.

```
<template:if test="{meta>ImageUrl}">
  <template:then>
    <m:Image src="{path: 'meta>ImageUrl', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
  </template:then>
  <template:elseif test="{meta>TypeImageUrl}">
    <commons:Image src="{path: 'meta>TypeImageUrl', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
  </template:elseif>
  <template:else>
    <commons:Text text="{path: 'meta>Title/Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
  </template:else>
</template:if>
```

Related Information

Expression Binding [page 845]
XML Templating [page 1018]

alias

The `<template:alias>` instruction can be used to define a shortcut alias name for a JavaScript value, for example a static helper class, a formatter function, or a helper function.

The shortcut alias is only available in the scope of the `alias` instruction. Each nested scope can redefine aliases.

A proper relative name must be used. This proper name must not contain or consist of a dot. The value to which the alias refers to can be any JavaScript object or property including a function that is available at that point. Exisiting alias names can be used in new alias definitions to refer to values, for example, `<template:alias name="format" value="AH.format">`. Aliases are inherited into fragments, but of course a fragment may (re)define its own set of aliases. For compatibility reasons, aliases can also start with a dot, both in definition and in references.

Example: "alias" Template

```
<template:alias name="AH" value="sap.ui.model.odata.AnnotationHelper">
  <template:with path="meta>com.sap.vocabularies.UI.v1.Badge" var="badge">
    <form:SimpleForm>
      <form:title>
        <core:Title text="{path: 'badge>HeadLine', formatter: 'AH.format'}"/>
      </form:title>
      <Label text="{path: 'badge>Title/Label', formatter: 'AH.format'}"/>
      <Text text="{path: 'badge>Title/Value', formatter: 'AH.format'}"/>
      <Label text="{path: 'badge>MainInfo/Label', formatter: 'AH.format'}"/>
      <template:with path="badge>MainInfo" var="field">
        <core:Fragment fragmentName="sap.ui.core.sample.ViewTemplate.tiny.Field"
type="XML"/>
      </template:with>
      <Label text="{path: 'badge>SecondaryInfo/Label', formatter: 'AH.format'}"/>
      <template:with path="badge>SecondaryInfo" var="field">
        <core:Fragment fragmentName="sap.ui.core.sample.ViewTemplate.tiny.Field"
type="XML"/>
      </template:with>
    </form:SimpleForm>
  </template:with>
</template:alias>
```

require

The `template:require` attribute can be used at the root element of an XML template view or fragment. You can specify a list of required modules as Unified Resource Names, similar to `sap.ui.require`, and assign aliases to them using a JSON-like syntax. The aliases can then be used to access the modules in the same way `<template:alias>` works. (This requires that the view is loaded asynchronously.)

> **i** Note
>
> The aliases can be used for formatter references (first `text` element in the code sample) as well as for function calls inside an expression binding (second `text` element in the code sample).

Example: "require" Template

```
<mvc:View
  controllerName="sap.ui.core.sample.ViewTemplate.scenario.Detail"
  template:require="{Helper: 'sap/ui/core/sample/ViewTemplate/scenario/Helper',
    AnnotationHelper: 'sap/ui/model/odata/AnnotationHelper'}"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
<template:alias name="bar" value="Helper.bar">
<Text text="{formatter: 'bar', path: '/foo'}"/>
<Text text="{= bar(${/foo}) }"/>
```

For compatibility reasons, especially if the view is loaded synchronously, the `template:require` attribute may contain a space-separated list of module names for `jQuery.sap.require`. These modules will then be required before processing. You have to access them via their global names.

`sap.ui.model.odata.AnnotationHelper` is automatically available.

Example

```
<mvc:View
  controllerName="sap.ui.core.sample.ViewTemplate.scenario.Detail"
  template:require="sap.ui.core.sample.ViewTemplate.scenario.Helper"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
```

Replacement of Bindings

For attributes, SAPUI5 binding expressions are used instead of preprocessing instructions. If the value of an XML attribute represents a valid SAPUI5 binding which refers to currently available model (= `<variable>`) names only, the binding is evaluated and the result is written back into the XML attribute.

The `sap.ui.model.odata.AnnotationHelper.format` method can be used as a formatter which properly interprets OData Version 4.0 annotations from the OData meta model. You can use the same formatter for labels and values. For more information, see SAP Annotations for OData Version 2.0.

Formatters for labels are usually not needed. Instead, pointing to `'badge>MainInfo/Label/String'` could be used, but this does not take care of escaping and works only for string constants. The `sap.ui.model.odata.AnnotationHelper.format` inserts references to translatable texts in case the preprocessor has been called with `bindTexts : true`. This is important for design-time templating.

The following example shows a template with binding:

```
<Label text="{path: 'badge>MainInfo/Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
<Text text="{path: 'field>Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}" />
```

At the time the binding is resolved, `field>Value` refers to `meta>/dataServices/schema/0/entityType/0/com.sap.vocabularies.UI.v1.Badge/MainInfo/Value`. In the resulting `XML DOM`, the references to the meta model are gone and the type information including the constraints is inserted in the binding. This is shown in the following example:

```
<Label text="Phone"/>
<Text text="{path : 'PhoneNumber', type : 'sap.ui.model.odata.type.String',
constraints : {'maxLength':'30'}}" />
Custom Formatter Functions
```

Custom Formatter Functions

You can also write your own custom formatter functions. For information how you access the model and path related to the current formatter call, see sap.ui.core.util.XMLPreprocessor.IContext.

The following code snippet shows a composite binding example. It assumes that the structure of the meta model that is used corresponds to the structure in the example and that `title>` refers to the header info's title property (line 2 in the code snippet):

```
1   "com.sap.vocabularies.UI.v1.HeaderInfo": {
2       "Title": {
3           "Label": {
4               "String": "Customer"
5           },
6           "Value": {
7               "Path": "CustomerName"
8           }
9       }
10  }
```

The following code snippet shows a formatter function that can be used in composite bindings either at root level (line 1), or for individual parts (Line 2). You can also use it for a simple binding. In this case, it behaves in the same way as a single part.

```
1   <Text text="{path: 'title>Label', formatter: 'formatParts'}: {path:
'title>Value', formatter: 'formatParts'}"/>
2   <Text text="{parts: [{path: 'title>Label', formatter: 'formatParts'}: {path:
'title>Value'}], formatter: 'formatParts'}"/>
```

The following code snippet shows an example formatter function.

```
1   /*
2    * Custom formatter function for complex bindings to demonstrate access to
ith part of binding.
3    * Delegates to {@link sap.ui.model.odata.AnnotationHelper#format} and wraps
label texts in
4    * square brackets. Joins parts together, separated by a space.
5    *
6    * @param {sap.ui.core.util.XMLPreprocessor.IContext} oInterface
7    *   the callback interface related to the current formatter call
8    * @param {...any} [vRawValue]
9    *   the raw value(s) from the meta model
10   * @returns {string}
11   *   the resulting string value to write into the processed XML
12   */
13  window.formatParts = function(oInterface, vRawValue) {
```

```
14      var i, aResult;
15
16      /*
17       * Delegates to {@link sap.ui.model.odata.AnnotationHelper#format} and
wraps label texts
18       * in square brackets.
19       *
20       * @param {sap.ui.core.util.XMLPreprocessor.IContext|
sap.ui.model.Context} oInterface
21       *   the callback interface related to the current formatter call
22       * @param {any} [vRawValue]
23       *   the raw value from the meta model
24       * @returns {string}
25       */
26      function formatLabelValue(oInterface, vRawValue) {
27        var sResult = sap.ui.model.odata.AnnotationHelper.format(oInterface,
vRawValue);
28        return sMyString.endsWith(oInterface.getPath(), "/Label")
29             ? "["+ sResult + "]"
30             : sResult;
31      }
32
33       try {
34          if(oInterface.getModel()) {
35              returnformatLabelValue(oInterface, vRawValue);
36          } else {
37              // root formatter for a composite binding
38              aResult = [];
39              // "probe for the smallest non-negative integer"
40              for (i = 0; oInterface.getModel(i); i += 1) {
41                  aResult.push(
42                      // Note: arguments[i + 1] is the raw value of the ith
part!
43                      formatLabelValue(oInterface.getInterface(i), arguments[i
+ 1])
44                  );
45              }
46              return aResult.join(" ");
47          }
48      } catch (e) {
49          return e.message;
50      }
51  }
52  window.formatParts.requiresIContext = true;
```

This example formatter opts to the extended signature (see line 51), which provides a context interface as the first parameter. It distinguishes between root level calls and others (see line 34), delegates to `sap.ui.model.odata.AnnotationHelper#format` (see line 27), and wraps label texts in square brackets for demo purposes (see line 29). For root level calls, it loops over all available parts (see line 40) and accesses each part (see line 43). The demo code handles each part individually and joins the result, but in practice some more complicated dependency between parts would be realistic.

The delegation to `sap.ui.model.odata.AnnotationHelper#format` provides the raw value we already have at hand, even for root level calls (see comment in line 42).

> **i Note**
>
> The custom formatter function needs to be accessible globally, because XML templating cannot call formatter functions inside the view's controller. This controller does not yet exist at the time of preprocessing. However, it is not sufficient to simply put it into window (see line 8). You must put it into your own namespace.

The example formatter has the following output:

```
<Text text="[Customer]: {CustomerName}"/>
<Text text="[Customer] {CustomerName}"/>
```

> **i Note**
>
> The colon (":") is taken over literally from the first composite binding which consists of two bindings separated by a string literal.

Mixing Runtime Data and Meta Data Within a Single Binding

The following code snippet shows a single binding that mixes runtime data and meta data. This will not work: The binding refers to the runtime and as runtime is not available ar this point, XML templating cannot replace the binding. (In the example, "runtime" stands for the name of the model at runtime. This would typically be the default model and thus has no name.)

```
{= ${runtime>value} > ${meta>threshold} }
```

`meta` will no longer be available anymore, so this binding cannot work as expected, but it will be evaluated every time the value changes and will compare the value to `undefined`.

To solve this, a clear separation is required: One expression binding that refers to meta data only and can be replaced by XML templating, and another expression binding that refers to runtime data only and can be evaluated later on. These two bindings need to be nested as follows:

```
{= '{= ${runtime<value} > ' + ${meta>threshold} + ' }' }
```

XML templating replaces this with a kind of a partial evaluation of the original mixed binding. By carefully putting the pieces into string literals and by taking care of escaping, you have full control over this process of partial evaluation. This is shown in the following examples, where the threshold value is assumed to be a number:

Expression binding with runtime data only:

```
{= ${runtime>value} > 42 }
```

Escaping for string constants:

```
{= '{= \'' + ${meta>A} + '\' + ${/B} } // --> {= 'A' + ${/B} }
```

Using the annotation helper:

```
{= '{= $' + ${path : 'meta>value, formatter :
'sap.ui.model.odata.AnnotationHelper.format'} + ' > ' + ${path :
'meta>threshold',
formatter : 'sap.ui.model.odata.AnnotationHelper.format'} + ' }' }
// --> {= ${path : 'path/to/property/value', type :
'sap.ui.model.odata.type.Int16'} > 42 }
```

Related Information

`sap.ui.model.odata.AnnotationHelper.format`
`sap.ui.core.util.XML.Preprocessor.IContext`
XML Templating [page 1018]
Formatting, Parsing, and Validating Data [page 854]

XML Fragments

SAPUI5 fragments of type XML are used in the context of XML templating to provide reuse parts for templates.

Any reference to an XML fragment is inlined by the preprocessor; that is, the reference is replaced by the fragment's XML DOM and preprocessing takes place on that DOM as well. All currently available variable names are inherited into the fragment.

Example: XML Fragment

```
<core:Fragment fragmentName="sap.ui.core.sample.ViewTemplate.tiny.Field"
type="XML"/>
```

The fragment name can also result from a binding, including an expression binding which evaluates to a constant. As formatter functions return strings, and not booleans, `=== 'true'` has been added in the following example:

Example: Dynamic Fragment Name

```
<core:Fragment fragmentName="{= ${path: 'facet>Target', formatter:
'sap.ui.model.odata.AnnotationHelper.isMultiple'} === 'true'
    ? 'sap.ui.core.sample.ViewTemplate.scenario.TableFacet'
    : 'sap.ui.core.sample.ViewTemplate.scenario.FormFacet' }" type="XML"/>
```

Related Information

XML Templating [page 1018]

Extension Points

Extension points can be used in XML templating to extend the standard with custom content.

The extension point has a default content which is used unless the extension point is replaced via customizing. The extension point name can result from a binding, including an expression binding which evaluates to a constant. If the extension point is to be replaced by an XML fragment, the extension point element is replaced by the fragment's XML DOM and preprocessing takes place on the DOM as well. All currently available variable

names and aliases are inherited into the fragment as usual. You get the same debug output as for fragment instructions, and you see the customized fragment name there.

```
<!-- expression binding just to showcase dynamic names -->
<core:ExtensionPoint name="{:= 'HeaderInfo' }">
    <form:SimpleForm>
        <form:title>
            <core:Title text="HeaderInfo"/>
        </form:title>
        <template:with path="entityType>com.sap.vocabularies.UI.v1.HeaderInfo">
            <!-- ... -->
        </template:with>
    </form:SimpleForm>
</core:ExtensionPoint>
```

Related Information

View Extension [page 2149]

Annotation Helper

A collection of methods which help to consume OData Version 4.0 annotations in XML template views.

The `AnnotationHelper` connects all the pieces related to XML templating: It knows the OData meta model and its structure as well as the OData v4 annotations. The `AnnotationHelper` offers formatter functions and helper functions. You can call these methods directly from the JavaScript code without XML runtime templating. You do not need to require `sap.ui.model.odata.AnnotationHelper` before use.

> → Tip
>
> You can see more information on the expressions, constants and functions used by the `AnnotationHelper`, in the respective chapters of the specification OData Version 4.0 Specification: Part 3: Common Schema Definition Language .

Formatter Functions

The formatter functions can be used in binding expressions and `<template:if>` instructions for test conditions.

> ℹ Note
>
> To use formatter functions, you need to enable the extended binding syntax by setting the configuration option `xx-bindingSyntax` to `complex`. For more information, see Composite Binding [page 843]. If the extended binding syntax is not enabled and expressions are created by means of the annotation helper's format functions, the following warning is issued in the console: *Complex binding syntax not active*.

The following formatter functions exist:

- `createPropertySetting`: Creates a property setting which is either a constant value or a binding info object from the given parts and from the optional root formatter function. Each part can have one of the following types:
  - boolean, number, undefined: The part is a constant value.
  - string: The part is a data binding expression with complex binding syntax (for example, as created by format) and is parsed accordingly to create either a constant value or a binding info object. Proper backslash escaping must be used for constant values with curly braces.
  - object: The part is a binding info object if it has a "path" or "parts" property, otherwise it is a constant value.

  If a binding info object is not the only part and has a `parts` property itself, then it must have no other properties except `formatter`. This applies to expression bindings and data binding expressions that are created by `format`. If all parts are constant values, the resulting property setting is also a constant value computed by applying the root formatter function to the constant parts once. If at least one part is a binding info object, the resulting property setting is also a binding info object and the root formatter function will be applied again and again to the current values of all parts, no matter whether constant or variable.

  > **i Note**
  >
  > The root formatter function should not rely on its `this` value because it depends on how the function is called.
  >
  > A single data binding expression can be given directly to `applySettings`; you do **not** need to call `this` function first.

- `format`: General purpose method that handles proper escaping and formatting of constant values and provides binding expressions with suitable types. `format` supports the following constructs:
  - The *"14.4 Constant Expressions"* for *"edm:Bool"*, *"edm:Date"*, *"edm:DateTimeOffset"*, *"edm:Decimal"*, *"edm:Float"*, *"edm:Guid"*, *"edm:Int"*, *"edm:TimeOfDay"*.
  - Constant *"14.4.11 Expression edm:String"*: This constant is either turned into fixed text, for example *"Width"*, or into a data binding expression, for example `"{/##/dataServices/schema/0/entityType/1/com.sap.vocabularies.UI.v1.FieldGroup#Dimensions/Data/0/Label/String}"`. If XML template processing has been started with the setting `bindTexts : true`, data binding expressions are used. The constant is used to reference translatable texts from OData v4 annotations, especially for XML template processing at design time. The string constants that contain a simple binding `"{@i18n>...}"` to the hard-coded model name `"@i18n"` with an arbitrary path are not turned into a fixed text, but kept as a data binding expression. This enables local annotation files to refer to a resource bundle for internationalization. If you want to avoid this behaviour, add a space at the end of the string constant and it will be turned into a fixed text again.
  - Dynamic *"14.5.1 Comparison and Logical Operators"*: Turned into an expression binding to perform the operations at runtime
  - Dynamic *"14.5.3 Expression edm:Apply"*:
    - *"14.5.3.1.1 Function odata.concat"*: Turned into a data binding expression relative to an entity
    - *"14.5.3.1.2 Function odata.fillUriTemplate"*: Turned into an expression binding to fill the template at runtime
    - *"14.5.3.1.3 Function odata.uriEncode"*: Turned into an expression binding to encode the parameter at runtime

    The *apply* functions can be nested arbitrarily.

- Dynamic *"14.5.12 Expression edm:Path"* and *"14.5.13 Expression edm.PropertyPath*: This dynamic expression is turned into a data binding relative to an entity including type information and constraints as available from metadata, for example `"{path : 'Name', type : 'sap.ui.model.odata.type.String', constraints : {'maxLength':'255'}}"`.
- Dynamic *"14.5.6 Expression edm:If"*: This dynamic expression is turned into an expression binding to be evaluated at runtime. The expression is conditional, for example, `"{=condition ? expression1 : expression2}"`.

> **i Note**
>
> Unsupported values are turned into strings, and indicated as such. To ensure that the data binding syntax is not corrupted, proper escaping is used.

```
<Text text="{path: 'meta>Value', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/>
```

- `getNavigationPath`: Special formatter that extracts a data binding expression for the navigation path from one of the following dynamic expressions: *14.5.2 Expression edm:AnnotationPath*, *"14.5.11 Expression edm:NavigationPropertyPath"*, *"14.5.12 Expression edm:Path"* and *"14.5.13 Expression edm:PropertyPath"*; example:
  - The input value `{AnnotationPath : "ToSupplier/@com.sap.vocabularies.Communication.v1.Address"}` returns `"{ToSupplier}"`
  - The input value `{AnnotationPath : "@com.sap.vocabularies.UI.v1.FieldGroup#Dimensions"}` returns `"{}"`
  - The input value `{}` returns `""`

```
<template:if test="{path: 'facet>Target', formatter:
'sap.ui.model.odata.AnnotationHelper.getNavigationPath'}">
    <form:SimpleForm binding="{path: 'facet>Target', formatter:
'sap.ui.model.odata.AnnotationHelper.getNavigationPath'}" />
</template:if>
```

- `isMultiple`: Special formatter that knows about the one of the following dynamic expressions: *14.5.2 Expression edm:AnnotationPath*, *"14.5.11 Expression edm:NavigationPropertyPath"*, *"14.5.12 Expression edm:Path"* and *"14.5.13 Expression edm:PropertyPath"*. The formatter returns the information whether the navigation path ends with an association end with multiple "*". If the multiple "*" are not the last characters, the formatter returns an error.

```
<template:if test="{path: 'facet>Target', formatter:
'sap.ui.model.odata.AnnotationHelper.isMultiple'}">
```

- `simplePath`: Specialized method useful for design-time templating in connection with smart fields; it can only return simple binding expressions without type information. This has the advantage that the resulting XML view, which is shown at design-time, looks much simpler and nicer without `&quot;` escapes.
  Example:

```
<mvc:View
  xmlns:mvc="sap.ui.core.mvc"
  xmlns:sfi="sap.ui.comp.smartfield"
  xmlns:sfo="sap.ui.comp.smartform"
  xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
  <sfo:SmartForm title="{path: 'meta>com.sap.vocabularies.UI.v1.HeaderInfo/
TypeName', formatter: 'sap.ui.model.odata.AnnotationHelper.format'}">
    <template:repeat list="{path:'meta>', filters: {path: 'RecordType',
operator: 'EQ', value1: 'com.sap.vocabularies.UI.v1.FieldGroupType'}}">
```

```
      <sfo:Group label="{path: 'meta>Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}">
        <template:repeat list="{meta>Data}">
          <sfo:GroupElement label="{path: 'meta>Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}">
            <sfi:SmartField value="{path: 'meta>Value', formatter:
'sap.ui.model.odata.AnnotationHelper.simplePath'}"/>
          </sfo:GroupElement>
        </template:repeat>
      </sfo:Group>
    </template:repeat>
  </sfo:SmartForm>
</mvc:View>
```

Output:

```
<mvc:View xmlns:mvc="sap.ui.core.mvc" xmlns:sfi="sap.ui.comp.smartfield"
xmlns:sfo="sap.ui.comp.smartform">
  <sfo:SmartForm title="Sales Order">
      <sfo:Group label="Order Note">
          <sfo:GroupElement label="Text">
            <sfi:SmartField value="{Note}"/>
          </sfo:GroupElement>
      </sfo:Group>
  </sfo:SmartForm>
</mvc:View>
```

The formatter functions can also be called directly from JavaScript. The following code snippet shows an example for the use of a formatter function outside templating:

```
var oModel = this.getModel(),
    sPath = "##com.sap.vocabularies.UI.v1.HeaderInfo/Description/Label",
    oContext = oModel.getContext(oModel.resolve(sPath,
this.getBindingContext())),
    oLabel = new sap.m.Label({
        text : sap.ui.model.odata.AnnotationHelper.format(oContext)
    }));
...
```

The formatter functions are called with a context object as first parameter. The second parameter (`vRawValue`) is optional. If the value is not provided, it is calculated in the formatter function. If the application has already calculated the value, it passes the raw value as second parameter, thus avoiding unnecessary further calculations of the raw value.

Helper Functions

The following helper functions can be used with `<template:with>`:

- `gotoEntityset`: Helper function for a `with` instruction for the entity set with the given name or the entity set that depending on how it is called has been determined by the last navigation property of one of the following dynamic expressions: *14.5.2 Expression edm:AnnotationPath*, *"14.5.11 Expression edm:NavigationPropertyPath"*, *"14.5.12 Expression edm:Path"* and *"14.5.13 Expression edm:PropertyPath"*.

```
<template:with path="facet>Target"
helper="sap.ui.model.odata.AnnotationHelper.gotoEntitySet" var="entitySet" />
<template:with path="associationSetEnd>entitySet"
helper="sap.ui.model.odata.AnnotationHelper.gotoEntitySet" var="entitySet"/>
```

The binding context passed to the helper function, as determined by the `template:with` instruction's `path` property, must point to a simple string or to an annotation (or annotation property) of type `Edm.AnnotationPath`, embedded within an entity set or type; the context's model must be an `sap.ui.model.odata.ODataMetaModel`.

- `gotoEntityType`: Helper function for a `with` instruction that goes to the entity type with the given qualified name. The binding context passed to the helper function, as determined by the `template: with` instruction's `path` property, must point to the qualified name of an entity type; the context's model must be an `sap.ui.model.odata.ODataMetaModel`.
  Example: Assume that `entitySet` refers to an entity set within an OData meta model; the helper function is then called on the `entityType` property of that entity set (which holds the qualified name of the entity type) and in turn the path of that entity type is assigned to the variable `entityType`.

```
<template:with path="entitySet>entityType"
helper="sap.ui.model.odata.AnnotationHelper.gotoEntityType" var="entityType">
```

- `resolvePath` is a helper function for a `template:with` instruction that resolves one of the following dynamic expressions: *14.5.2 Expression edm:AnnotationPath*, *"14.5.11 Expression edm:NavigationPropertyPath"*, *"14.5.12 Expression edm:Path"* and *"14.5.13 Expression edm:PropertyPath"*. The function supports navigation properties and term casts. The binding context passed to the helper function, as determined by the `template:with` instruction's path property, must point to an annotation or annotation property of type `Edm.AnnotationPath`, embedded within an entity type, The context's model must be `sap.ui.model.odata.ODataMetaModel`.

```
<template:with path="meta>Value"
helper="sap.ui.model.odata.AnnotationHelper.resolvePath" var="target">
```

- `gotoFunctionImport`: Helper function for a `template:with` instruction that goes to the function import with the name which `oContext` points at. Example: Assume that `dataField` refers to a `DataFieldForAction` within an OData meta model; the helper function is then called on the `Action` property of that data field (which holds an object with the qualified name of the function import in the String property) and in turn the path of that function import is assigned to the variable `function`.

```
<template:with path="dataField>Action"
helper="sap.ui.model.odata.AnnotationHelper.gotoEntityType" var="function">
```

Related Information

sap.ui.model.odata.AnnotationHelper
OData Version 4.0

Debugging

For the debug levels `DEBUG` and `ALL`, the XML preprocessor writes a trace for what it exactly does.

The following events are traced (the numbers represent the line numbers in the example below):

- The start including the view being processed (1)

- A list of all binding contexts with the path they are pointing to (2)
- Start of processing of a `<with>` statement including the new variable assignment (3,6)
- Evaluation of an `<if>` or `<elseif>` statement including the test result (4,10)
- Start of processing of a `<repeat>` statement (8)
- Each iteration of the `<repeat>` statement including the current variable assignment (9)
- Start of processing of a `<Fragment>` statement including the resulting fragment name (7)
- Finished processing of any of the following statements: `<with>`, `<if>`, `<repeat>`, `<Fragment>` (12-17)
- If any attribute of any other node has been resolved (5,11)
- The finish (18)

Each debug line looks as follows:

```
[level] message - <node> sap.ui.core.util.XMLPreprocessor
```

`[level]` is the number of currently active preprocessor statements. It is incremented each time when the processing of an `<if>`, `<with>`, `<repeat>` or `<Fragment>` starts. It is decremented when the node is completely processed. `<node>` is the node being processed with all its attributes.

Example:

```
1  [ 0] Start processing Element sap.ui.core.mvc.XMLView#__xmlview5
(sap.ui.core.sample.ViewTemplate.scenario.Detail) -
sap.ui.core.util.XMLPreprocessor
2  [ 0] meta = /dataServices/schema/0/entityContainer/0/entitySet/0 -
sap.ui.core.util.XMLPreprocessor
3  [ 1] entityType = /dataServices/schema/0/entityType/0 - <template:with
path="meta>entityType"
helper="sap.ui.model.odata.AnnotationHelper.gotoEntityType" var="entityType">
sap.ui.core.util.XMLPreprocessor
4  [ 2] test == [object Array] --> true - <template:if
test="{entityType>com.sap.vocabularies.UI.v1.LineItem}">
sap.ui.core.util.XMLPreprocessor
5  [ 2] items = {path:'/BusinessPartnerSet', length: 5} - <Table
includeItemInSelection="true" mode="SingleSelect"
selectionChange="onSelectionChange" items="{= '{path:\'/' + ${meta>name} + '\',
length: 5}' }"> sap.ui.core.util.XMLPreprocessor
6  [ 3] target = /dataServices/schema/0/entityType/0/
com.sap.vocabularies.UI.v1.LineItem - <template:with
path="entityType>com.sap.vocabularies.UI.v1.LineItem" var="target">
sap.ui.core.util.XMLPreprocessor
7  [ 4] fragmentName = sap.ui.core.sample.ViewTemplate.scenario.Table -
<core:Fragment fragmentName="sap.ui.core.sample.ViewTemplate.scenario.Table"
type="XML"/> sap.ui.core.util.XMLPreprocessor
8  [ 5] Starting - <template:repeat list="{target>}" var="field">
sap.ui.core.util.XMLPreprocessor
9  [ 5] field = /dataServices/schema/0/entityType/0/
com.sap.vocabularies.UI.v1.LineItem/0 - <template:repeat list="{target>}"
var="field"> sap.ui.core.util.XMLPreprocessor
10 [ 6] test == [object Object] --> true - <template:if test="{field>Value}">
sap.ui.core.util.XMLPreprocessor
11 [ 6] text = ID - <Text text="{path: 'field>Label', formatter:
'sap.ui.model.odata.AnnotationHelper.format'}"/> sap.ui.core.util.XMLPreprocessor
12 [ 6] Finished - </template:if> sap.ui.core.util.XMLPreprocessor
13 [ 5] Finished - </template:repeat> sap.ui.core.util.XMLPreprocessor
14 [ 4] Finished - </core:Fragment> sap.ui.core.util.XMLPreprocessor
15 [ 3] Finished - </template:with> sap.ui.core.util.XMLPreprocessor
16 [ 2] Finished - </template:if> sap.ui.core.util.XMLPreprocessor
17 [ 1] Finished - </template:with> sap.ui.core.util.XMLPreprocessor
```

```
18 [ 0] Finished processing Element sap.ui.core.mvc.XMLView#__xmlview5
(sap.ui.core.sample.ViewTemplate.scenario.Detail) -
sap.ui.core.util.XMLPreprocessor
```

Working with Controls

Controls are used to define the appearance and behavior of screen areas.

Controls consist of:

- Control name
  The control name is a string that consists of the library name and the control name, separated by a dot. The library name can be omitted if there is no need to assign the control to a library. It is possible, for example, to use `Square` as control name. For controls that are reused by others, we recommend to use a unique library name, for example `sap.byd.Square`.
- Control metadata
  The metadata defines the properties, events, aggregations and associations of a control.
  Control properties, such as text or width, are used to modify the appearance or to relate to data that is displayed by the control. The controls are defined by the control metadata, which is the public API of the control. The API can be used by applications at runtime and also contains information on runtime features such as data binding and type validation checks.
  Controls can aggregate other controls. These controls with aggregations serve as a container or layout control to which the application can add child controls. They can also serve as composite controls if the control itself adds child controls and reuses available components. In an aggregation, child controls are owned by the parent control and are destroyed together with the parent control. A control can only have one aggregation parent. Adding the control to another aggregation removes it from the previous parent control.
  Associated controls are not part or children of an aggregation control. They are connected by ID instead of reference. Destroying a control in an association does not affect the other control. It is possible that an associated control does not yet or no longer exist.
  Controls fire events. Events typically relate to the control's purpose and functionality on a semantically higher level than browser events such as `click`. Examples for control events are `triggerSearch` for a search field or `collapse` in a panel.
- Elements
  Elements are parts of controls or rather configuration packages for parts of controls. Elements **cannot** be used standalone and do **not** have their own renderer. Instead, the control that uses the element does the rendering: The `ComboBox` control, for example, renders the `Item` elements. The information provided for controls also applies to elements but not to the renderer. The `sap.ui.core.Element` class is the base class of `sap.ui.core.Control`.
- Methods
  By convention, methods are public, unless their name starts with an underscore or if it is one of the special method types. When developing control libraries, public methods must be annotated with `@public` in the JSDoc, and private methods with `@private`. The generated getter/setter methods for properties are also public methods.
  Methods are added to a new control by simply providing the implementation. It is not necessary to add the method to the metadata. Other controls and the application must only call public methods and the control ensures that they remain compatible. There are no technical rules that prevent the call of private methods, but it is not allowed.

The base class for all controls in SAPUI5 is `sap.ui.core.Control`. To inherit and extend the functionality, specific controls can either inherit from the base class, or from another control.

UI Control Constructors

A constructor is a special type of function that is called to create an object. The constructor uses values to set control properties, thus preparing the new object for use.

In SAPUI5, control constructors accept the following arguments in the specified order:

1. An optional unique identifier of type `string` which must either be the first argument, or omitted altogether. If you omit the ID, the SAPUI5 framework automatically computes an ID. Specifying your own identifier allows your application to easily find the control and, for example, retrieve the current user input from it. Alternatively, you can keep a reference to the control in a variable.
2. A simple object as `mSettings` parameter that defines values for any property, aggregation, association, or event.

The following code snippet shows an example of a constructor that is called to create a new text control saying "Hello World" with the specified tooltip and width:

```
// required from sap/m/Text
var oText = new Text("testText",
{text : "Hello World", tooltip: "This is an example tooltip", width: "100px"});
```

The above example is an abbreviated version of the following code snippet with a detailed list of statements, which is alternatively supported:

```
// required from sap/m/Text
var oText = new Text("testText");
oText.setText("Hello World");
oText.setTooltip("This is an example tooltip");
oText.setWidth("100px");
```

The supported parameters are documented in the *API Reference* of the respective control.

Related Information

Developing Controls [page 2158]

Custom Data - Attaching Data Objects to Controls

SAPUI5 provides the `data()` method to attach data objects to controls.

The `data()` method is contained in `sap/ui/core/Element`. You can use this method to set and get data. The API is equivalent to `jQuery.data()`.

The following additional options exist for attaching data to SAPUI5 controls:

- Attaching data declaratively in XML views and JSON views, see XML View [page 787]
- Using data binding, see Data Binding [page 815]
- For strings only: Writing data to the HTML DOM as "data-*" attribute, see Writing Data to the HTML DOM as DATA-* Attribute [page 1045]

Setting and Retrieving Data

To set and retrieve data, use the following code:

```
myButton.data("myData", "Hello");  // attach some data to the Button
alert(myButton.data("myData"));     // alerts "Hello"
var dataObject = myButton.data();  // a JS object containing ALL data
alert(dataObject.myData);          // alerts "Hello"
```

Binding Data: Use in a List Binding

For list bindings, use the following code:

```
// "CustomData" required from "sap/ui/core/CustomData"
// "JSONModel" required from module "sap/ui/model/json/JSONModel"
// "List" required from module "sap/m/List"
// "StandardListItem" required from module "sap/m/StandardListItem"
function giveAnswer(oEvent) {
  var oItem = oEvent.getSource();      // the StandardListItem
  var sData = oItem.data("theAnswer"); // access the custom data stored under
the key "theAnswer"
  alert("The answer is: " + sData);
}
// create a JSONModel, fill in the data and bind the ListBox to this model
var oModel = new JSONModel(aData);         // aData.questions is an array of
elements like {question:"Some question?",answer:"Some answer!"}
var oList = new List({select:giveAnswer}); // method giveAnswer() retrieves the
custom data from the selected ListItem
oList.setModel(oModel);
// create an item template and bind the question data to the "text" property
var oItemTemplate = new StandardListItem({title: "{question}", press:
giveAnswer, type: "Active"});
// create a CustomData template, set its key to "answer" and bind its value to
the answer data
var oDataTemplate = new CustomData({key:"theAnswer", value: "{answer}"});
// add the CustomData template to the item template
oItemTemplate.addCustomData(oDataTemplate);
// bind the items to the "questions" (which is the name of the data array)
oList.bindAggregation("items", "/questions", oItemTemplate);
```

You can find a productive example in the SAPUI5 test suite by searching for `CustomData` in `sap.ui.core`.

Use in XML Views

In XML views, `CustomData` objects can be written as normal aggregated objects. However, to reduce the amount of code and improve the readability, a shortcut notation has been introduced: You can directly write the data attributes into the control tags. Simply use the following namespace for the respective attributes:

`myNamespace="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"`.

The difference between this more formal namespace and the existing MVC namespaces is intentional.

Example

Use without Data Binding

The following example shows how you attach the string "just great" to a button:

```
<mvc:View xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
controllerName="my.own.controller"
          xmlns:app="http://schemas.sap.com/sapui5/extension/
sap.ui.core.CustomData/1">
     <Button id="myBtn" text="Click to show stored coordinates data"
app:mySuperExtraData="just great" press="alertCoordinates"></Button>
</mvc:View>
```

The string is returned at runtime by calling `button.data("mySuperExtraData")`.

Example

Use with Data Binding

You can use data binding with the following notation:

```
<mvc:View xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
controllerName="my.own.controller"
          xmlns:app="http://schemas.sap.com/sapui5/extension/
sap.ui.core.CustomData/1">
     <Button id="myBtn" text="Click to show stored coordinates data"
app:coords="{data}" press="alertCoordinates"></Button>
</mvc:View>
```

Use in JSON Views

To add custom data to an element in a JSON view, add the following code to the element properties (examples with data binding):

```
customData: {
  Type:"sap.ui.core.CustomData",
    key:"coords",
    value:"{data}" // bind custom data
  }
```

To add multiple data elements, use an array:

```
customData: [{
  Type:"sap.ui.core.CustomData",
```

```
    key:"coords",
    value:"{data}" // bind custom data
  },
  {
  Type:"sap.ui.core.CustomData",
    key:"coords",
    value:"{data}" // bind custom data
  }]
```

In context, this looks as follows:

```
var json =
  {
    Type: "sap.ui.core.mvc.JSONView",
    controllerName:"my.own.controller",
    content: [{
      Type:"sap.m.Panel",
      content:[{
        Type:"sap.m.Button",
        text:"{actionName}",
        press: "doSomething",
        customData: {
          Type:"sap.ui.core.CustomData",
          key:"coords",
          value:"{data}" // bind custom data
        }
      }]
    }]
  };
```

Use in HTML Views

To add custom data objects to a control or an element in HTML views, use a specific HTML attribute with the following syntax: `data-custom-data:my-key="myValue"`. A custom data attribute starts with `data-custom-data:` followed by the name of the key. The dashes convert the respective following character into an upper case character. The value can be either a string or a binding expression:

```
<div data-sap-ui-type="sap.m.Button" data-text="This button is added
dynamically" data-custom-data:my-key="myValue" data-custom-data:my-bound-key="{/
mypath}"></div>
```

Writing Data to the HTML DOM as DATA-* Attribute

SAPUI5 supports writing custom data to the HTML DOM.

These are two use cases, where this feature can be useful:

- To generate markers in the HTML from data binding which then can then be used for data-dependent styling.
- To create stable anchors in the HTML which can be used for automated tests.

A "data-" prefix is added to the key and the result is then written as an attribute into the root HTML element of the control. The `CustomData` value is written as an attribute value.

For this, the key has to be a valid HTML ID and the value has to be a string; otherwise an error is logged.

> **i Note**
>
> HTML attribute names are case-insensitive and browsers may convert the key to lowercase.
>
> Do not write too much data into the DOM.

In JavaScript, you can set the flag as shown in the following code snippet:

```
myButton.data("mydata", "Hello", true); // attach some data to the Button and
mark it as
          "write to HTML"
```

To set the `writeToDom` flag in XML views, the aggregation has to be written in expanded notation:

```
<Button ... >
  <customData>
  <core:CustomData key="mydata" value="Hello" writeToDom="true" />
  </customData>
</Button>
```

This results in the following HTML:

```
<button ...  data-myData="Hello"  ... >
```

This is done similarly in HTML views:

```
<div data-sap-ui-type="sap.m.Button" data-text="This button has custom data
written to the DOM">
   <div data-sap-ui-aggregation="customData">
      <div data-sap-ui-type="sap.ui.core.CustomData" data-key="mydata" data-
value="Hello" data-write-to-dom="true"/>
   </div>
</div>
```

The CSS can now use attribute selectors to check the presence or the value of the custom data attribute:

```
button[data-mydata="Hello"] { border: 3px solid red !important; }
```

Using Predefined CSS Margin Classes

SAPUI5 gives you the option of adding spacing in between controls by adding a margin. A margin clears an area around its respective control, outside of its border.

Unlike paddings, margins are transparent, are not part of the control's clickable area, and they collapse with adjacent margins, meaning that they do not add to each other. For instance, if you have two 32px margins next to each other, the result is that only one 32px margin is displayed, not 64px of space.

All margins predefined in SAPUI5 support right-to-left (RTL) languages: when you add a margin to the left, we make sure that it's displayed on the right if your user has chosen an RTL language such as Hebrew or Arabic. For our CSS classes, we offer four standard sizes, namely tiny (0.5rem or 8px), small (1rem or 16px), medium (2rem or 32px) and large (3rem or 48px).

There are four types of margins available:

- Full margins, which completely surround your control
- Single-sided margins
- Two-sided margins
- Responsive margins, which adapt to the available screen width

Full Margins

If you would like to clear an area **all around your control**, use one of the following margin classes:

- `sapUiTinyMargin`
- `sapUiSmallMargin`
- `sapUiMediumMargin`
- `sapUiLargeMargin`

Single-Sided Margins

For single-sided margins, choose a size (`Tiny`, `Small`, `Medium`, or `Large`, which stands for 8, 16, 32 or 48px respectively) and a direction (`Begin`, `End`, `Top`, or `Bottom`, where `Begin` is left and `End` is right and vice versa in RTL mode). For example, if you need to clear a 32px space to the left of your control (or to the right in RTL mode), you would add the class `sapUiMediumMarginBegin`. You can also add several classes at once, as long as they point to different directions. For example, you would add classes `sapUiLargeMarginEnd` and `sapUiLargeMarginBottom` to clear a 48px space to the bottom and to the right of a control (or to the left in RTL mode).

Here are the classes we provide for single-sided margins:

`sapUiTinyMarginTop`	`sapUiSmallMarginTop`	`sapUiMediumMarginTop`	`sapUiLargeMarginTop`
`sapUiTinyMarginBottom`	`sapUiSmallMarginBottom`	`sapUiMediumMarginBottom`	`sapUiLargeMarginBottom`
`sapUiTinyMarginBegin`	`sapUiSmallMarginBegin`	`sapUiMediumMarginBegin`	`sapUiLargeMarginBegin`
`sapUiTinyMarginEnd`	`sapUiSmallMarginEnd`	`sapUiMediumMarginEnd`	`sapUiLargeMarginEnd`

Two-Sided Margins

If you'd like to clear the space to the left and right or top and bottom of your control, we've provided several two-sided margin classes for you to use. Again, just choose the size and orientation that you need (`BeginEnd`,

TopBottom). For example, if you need to clear a 32px space both to the left and right of a control, you would add the class sapUiMediumMarginBeginEnd. Here are the classes that are available:

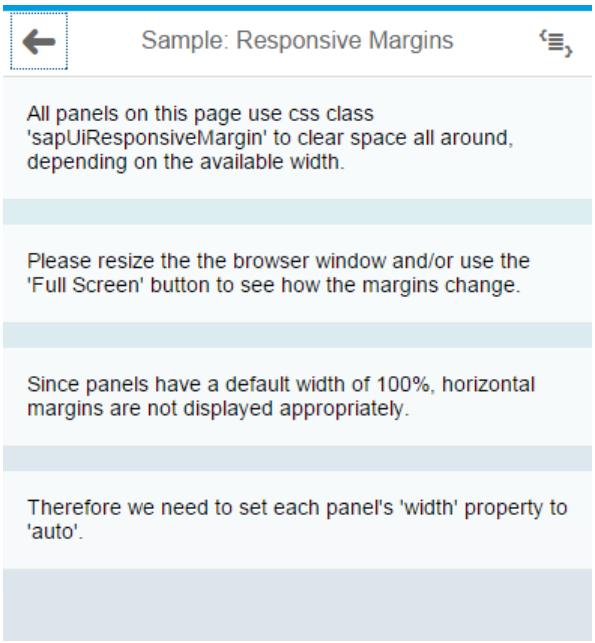
sapUiTinyMarginBeginEnd	sapUiSmallMarginBeginEnd	sapUiMediumMarginBeginEnd	sapUiLargeMarginBeginEnd
sapUiTinyMarginTopBottom	sapUiSmallMarginTopBottom	sapUiMediumMarginTopBottom	sapUiLargeMarginTopBottom

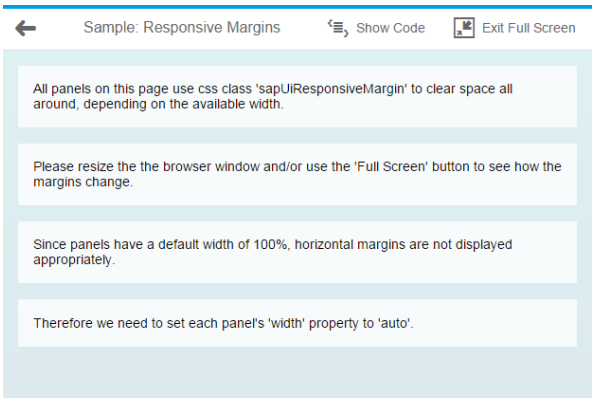
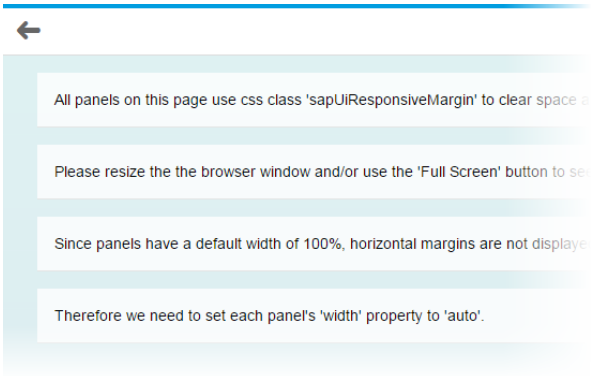
We've also provided a set of negative margin classes that add a two-sided (BeginEnd) negative margin of an element. The negative margins are useful for aligning elements with built-in paddings.

sapUiTinyNegativeMarginBeginEnd	sapUiSmallNegativeMarginBeginEnd	sapUiMediumNegativeMarginBeginEnd	sapUiLargeNegativeMarginBeginEnd

Responsive Margins

If your application is supposed to run on smartphone, tablet and desktop, it can be useful to choose your margins depending on the screen width that is available. SAPUI5 now comes with CSS class sapUiResponsiveMargin, which does just that. It works with media queries to determine the available screen width and adapts its margin as follows:

Screen Width	Example
Screen width less than 600px (smartphones): For devices such as these, sapUiResponsiveMargin provides a 16px (1rem) bottom margin to your control. Each of the panels shown in the screenshot is using sapUiResponsiveMargin. As a result, they're all clearing the same 16px area of space below them.	

Screen Width	Example
Between 600px and 1023px (tablets and small desktops): For these, `sapUiResponsiveMargin` provides a 16px (1rem) margin all around your control, as you can see in the screenshot.	
Larger than 1023px (large desktops): For these, `sapUiResponsiveMargin` provides 16px (1 rem) top and bottom margin as well as a 32px (2 rem) left and right margin, as shown in the screenshot.	

The following exceptions to these rules exist:

- When your `sapUiResponsiveMargin` control is placed within an `sap.m.SplitApp` master view, it's always provided with a 16px bottom margin and nothing else. Here, the thresholds mentioned previously do not affect your control.
- Within a `SplitApp`'s detail view, there's always a 16px margin all around your `sapUiResponsiveMargin` control, regardless of the available screen width. Usually, the `SplitApp` is responsive as well, though. If it hides its master view because the available screen width isn't sufficient, or if it's running in 'HideMode', your control also ignores the fact that it's placed into a `SplitApp`'s detail view and it becomes responsive again.

Controls with 100% Width

When applying classes with horizontal margins to a control, such as `sapUiSmallMargin` or `sapUiSmallMarginBegin`, for example, make sure that your control doesn't have a 100% width. If your control has a `width` property (which most controls have), set the width value to `auto`, for example:

```
<Panel width="auto" class="sapUiLargeMarginBegin
      sapUiLargeMarginBottom">
```

If your control does **not** have a `width` property but still has a default width of 100%, you can add our CSS class `sapUiForceWidthAuto` to your control, which ensures that the control's default width is overwritten with the value `auto`. An example for such a control is `sap.m.IconTabBar`.

Adding Margin Classes to Your Code

To apply the classes described here in your code, simply add a `class` attribute and the margin class to the respective control tag in your declarative xml views. If you need to add several classes at once (which can be the case if you're using single-sided margins), separate them by a space. Here's a sample snippet containing a panel and an `IconTabBar`:

```
<mvc:View
      height="100%"
      controllerName="sap.m.sample.StandardMarginsEnforceWidthAuto.Page"
      xmlns:mvc="sap.ui.core.mvc"
      xmlns="sap.m">
      .
      .
      .
      <Panel width="auto" class="sapUiLargeMarginBegin sapUiLargeMarginBottom">
          <content>

          </content>
      </Panel>
<IconTabBar
  expanded="{device>/isNoPhone}"
  class="sapUiForceWidthAuto sapUiMediumMarginBeginEnd">
  <items>
        .
        .
        .
  </items>
</IconTabBar>
      .
      .
</mvc:View>
```

If you would like to add margins to javascript code, use the control's `addStyleClass` method. For example: `myPanel.addStyleClass('sapUiLargeMargin')`. Again, if you need to add several classes at once, you can do so by separating them by a space. Make sure that you set the panel's width to `auto` as shown here: `myPanel.setWidth('auto')`.

Removing Margins

If a control comes with a margin that you don't want for some particular reason, you can use one of our convenience classes to remove it. You can either use `sapUiNoMargin` to remove the margins all around your control, or you can choose one or more classes from the following list to remove a margin in one or more particular directions:

- `sapUiNoMarginTop`
- `sapUiNoMarginBottom`

- `sapUiNoMarginBegin`
- `sapUiNoMarginEnd`

Sample

For a detailed example of how our predefined margin classes work, see the Samples.

Using Container Content Padding CSS Classes

For many container controls in SAPUI5, such as a Dialog or a Page, you can define whether the container should have a padding within the content area. A padding clears the area between the container layout and the controls that are displayed in the content area.

You can either choose to have no padding, a small padding, or a responsive padding that is based on the user's screen size.

The following CSS classes for content padding are provided by SAPUI5:

```
sapUiNoContentPadding
sapUiContentPadding
sapUiResponsiveContentPadding
```

The effects of these paddings will be different depending on the container that is being used:

- If the container is defining a padding on its content area by default, then setting `sapUiNoContentPadding` in the application code will remove the padding and thus not display any space between the container layout and the content controls.
- If the container is defining no padding on its content by default, then setting `sapUiContentPadding` will add 1rem (16px) of padding around the content area to layout the content controls.
- Setting `sapUiResponsiveContentPadding` on the container control will add a padding based on the user's screen size. On small screen devices such as smartphones, or when the browser window is resized to a small size, no padding is displayed. On medium screen sizes and applications that are using a `SplitApp`, the control will get 1rem (16px) of padding. Finally, on large screen sizes in full-screen mode, the container control will get 2rem (32px) of padding to the left and right and 1rem (16px) of padding at the top and bottom.

The following list shows examples of controls that support container content padding CSS classes:

```
sap.f.DynamicPage
sap.m.Carousel
sap.m.Dialog
sap.m.IconTabBar
sap.m.List
sap.m.Page
sap.m.Panel
sap.m.Popover
sap.m.ScrollContainer
```

```
sap.m.Table
sap.ui.layout.BlockLayoutCell
sap.ui.layout.DynamicSideContent
sap.ui.layout.HorizontalLayout
sap.ui.layout.VerticalLayout
```

Example

```
<Page class="sapUiResponsiveContentPadding">
```

Example

For a detailed example of how our container padding classes work, see the sample sap.ui.core.ContainerPadding.

Enabling Responsive Paddings

Apply responsive paddings over separate parts of the controls.

> **i** Note
>
> You can enable the responsive paddings only for the SAP Quartz themes.

Usage

Application developers can now apply responsive paddings over separate parts of the controls and align the space distribution properly, according to the width of the control (and not the whole screen). This can be done by using a set of classes, which are available for the different controls.

Applied Paddings

Based on the container's size, one of the following classes is added, and the corresponding padding-left and padding-right are applied:

Container Size (pixels)	Class	Padding-Left and Padding-Right Applied
<= 600	sapUi-Std-PaddingS	1rem
>600	sapUi-Std-PaddingM	2rem

Container Size (pixels)	Class	Padding-Left and Padding-Right Applied
>1024	sapUi-Std-PaddingL	2rem
>1440	sapUi-Std-PaddingXL	3rem

Supported Controls

The following table shows the controls that support responsive paddings. To enable this concept and to add responsive paddings to an element of the controls, add the corresponding classes to the controls, depending on your use case:

Supported Controls	sapUiResponsivePadding--header	sapUiResponsivePadding--subHeader	sapUiResponsivePadding--content	sapUiResponsivePadding--footer	sapUiResponsivePadding--floatingFooter
sap.m.Dialog (sap.m.MessageBox, sap.m.SelectDialog, sap.m.TableSelectDialog)					
sap.m.IconTabBar					
sap.m.ObjectHeader					
sap.m.Page					
sap.m.Popover					
sap.m.TabContainer					
sap.m.Wizard					

> **i Note**
>
> If nessecary, you can further align controls by using the available set of predefined CSS margin classes. For example, you can add negative margins to an element on its left and right sides. For more information, see Using Predefined CSS Margin Classes [page 1046].

Example

```
<Page class="sapUiResponsivePadding--header sapUiResponsivePadding--subHeader
sapUiResponsivePadding--content sapUiResponsivePadding--footer
sapUiResponsivePadding--floatingFooter">
```

For a detailed example, see the Samples for `sap.m.Page`.

How to Enable Responsive Paddings

As a control developer, you can enable application developers to apply responsive paddings, by implementing the `sap.ui.core.util.ResponsivePaddingsEnablement` utility.

Here is an example:

```
ResponsivePaddingsEnablement.call(MyCustomControl.prototype, {
                    header: {suffix: "-myCustomControlHeader"},
                    content: {selector: ".myCustomControlContent"}
                    });
```

As the example demonstrates, there are two ways to select an element:

- Using suffix: This enables you to select an element by its ID.
- Using selector: This covers all possible CSS selections.

To call the utility, when initializing the control, use:

```
MyCustomControl._initResponsivePaddingsEnablement()
```

As a result, application developers will be able to use classes, such as `sapUiResponsivePadding—header` and `sapUiResponsivePadding--content`, to enable the paddings on the respective element.

Field Groups

Group of controls that belong together semantically. This group can be used, for example, for validating the data consistency for the field group.

Field groups are built by means of a common `fieldGroupIds` array for a group that can be set for each control. When the user changes the focus to a control with a different `fieldGroupIds` array, the `validateFieldGroup` event is raised and bubbled in the control hierarchy, that is, the event is propagated to the parent control until it reaches the top most control, or the event is handled and `oEvent.bCancelBubble` is set to `true`. The application developer can now validate the fields within the group. The `validateFieldGroup` event is also raised if the user presses Enter in a field without any modifier keys.

Example: Validation of Credit Card Information

Depending on the credit card vendor, different validation rules can be implemented for a field group, for example:

- Vendor 1: First digit must be a 3 and second digit must be a 4 or 7, the valid length is 15 digits
- Vendor 2: First digit must be a 5 and second digit must be in the range from 1 to 5 inclusive, the valid length is 16 digits

Plenty of other validation rules for vendors exist. In addition to this, the expiry date must not be in the past and, in case of Vendor 1, is only allowed to be 4 years in the future.

Defining a Field Group ID

`fieldGroupIds` is a property of all `sap.ui.core.Control` instances and can be set there as an array of strings. The developer has to make sure that the ID is unique. Field group IDs can be added as a comma-separated string as it is usually done in an xml view declaration.

```
var myInput1 = new sap.m.Input({fieldGroupIds:["MyGroup","MyGroup2"]}),
myInput2 = new sap.m.Input({fieldGroupIds:["MyGroup","MyGroup2"]});
```

```
//XMLView declaration of multiple groups
<input fieldGroupIds="MyGroup,MyGroup2" />
```

Validating Field Groups

The `validateFieldGroup` event is raised on the control that lost the focus and at least one field group was left. The event bubbles up the control hierarchy. In the example below, the surrounding `VerticalLayout` is handling the event for its fields.

```
var myVerticalLayout = new sap.ui.layout.VerticalLayout({content:[myInput1,
myInput2], validateFieldGroup: function(oEvent) {
    var aFieldGroup = oEvent.getParameters().fieldGroupIds;
    if (aFieldGroup.indexOf("MyGroup") > -1) {
        //do validation
        oEvent.bCancelBubble = true; //stop bubbling to the parent control
    }
}});
```

Accessing Controls in a Field Group

In some scenarios, it is required to find all controls that belong to a specific field group, or to all controls with a `fieldGroupId`. For this, the control implements the public `getControlsByFieldGroupId` method that gets a list of child controls in the application code.

```
var aAllControlsWithFieldGroupId =
myVerticalLayout.getControlsByFieldGroupId();              //all where
fieldGroupId is not empty
var aMyGroupControls             =
myVerticalLayout.getControlsByFieldGroupId("myGroup");     //exact matches to
myGroup
```

Similar to the above you can use the `byFieldGroupId` method of `sap.ui.Core` to all controls with certain field group IDs.

```
var aAllControlsWithFieldGroupId =
sap.ui.getCore().byFieldGroupId();              //all where fieldGroupId is not
empty
var aMyGroupControls             =
sap.ui.getCore().byFieldGroupId("MyGroup");     //exact matches to myGroup
var aNotGrouped                  =
sap.ui.getCore().byFieldGroupId([]);            //exact empty array (default
value of fieldGroupIds)
```

Using the `fieldGroupId` With Composite Controls

Composite controls that derive from control base class automatically support setting a `FieldGroupId`. Nevertheless, the `FieldGroupId` is not propagated to inner controls of the composite control as they are unknown to the SAPUI5 framework. Therefore, a composite control needs to propagate the `FieldGroupId` on its own. For all internally aggregated controls, the `FieldGroupId` should be propagated as follows:

```
MyCompositeControl.prototype.setFieldGroupIds = function(vValue,
bSuppressInvalidate) {
    this._myAggregatedInnerControl.setFieldGroupIds(vValue, bSuppressInvalidate);
    this.setProperty("fieldGroupIds",vValue, bSuppressInvalidate);
}
```

Note

For a control with a `fieldGroupId` that currently has the focus, the following applies:

- If the control is destroyed before the focus is moved to another control, the `validateFieldGroup` event is not fired.
- If the control changes its `fieldGroupIds`, the `validateFieldGroup` event is fired for the new `fieldGroupIds` and **not** for the old.

Related Information

Sample: `Field Groups`

Declarative Support

Declarative programming allows you to define the UI within the HTML document as elements.

For this, SAPUI5 provides the `sap.ui.core.plugin.DeclarativeSupport` plugin that can be included either as required or marked as a module in the initial bootstrap script tag. The plugin parses the document and converts its tags with special attributes into SAPUI5 controls.

Declarative support is aware of properties, associations, events, and aggregations in a SAPUI5 control manner. This means that you can specify them within the markup of the HTML document either as data attributes or as child elements.

The following sections provide an overview of the declarative support and introduce the use of declarative support in SAPUI5.

Example

The following example shows the concept by combining a `sap.m.input` with a `sap.m.Button` control. When you click the button, the value of the text field is displayed in an alert box:

```
<!Doctype HTML>
<html>
<head>
    <title>Declarative Programming for SAPUI5 - sample01</title>
    <script id="sap-ui-bootstrap"
         type="text/javascript"
         src="resources/sap-ui-core.js"
         data-sap-ui-theme="sap_belize"
         data-sap-ui-libs="sap.m"
         data-sap-ui-modules="sap.ui.core.plugin.DeclarativeSupport"
         >
    </script>
</head>
<body class="sapUiBody">
  <div data-sap-ui-type="sap.m.Input" id="message" class="my-button" data-
value="Hello World"></div>
  <div data-sap-ui-type="sap.m.Button" data-text="Click me!" data-
press="handlePress"></div>
</body>
</html>
```

Summary: Attributes Used by Declarative Support

The table summarizes the attributes used by declarative support and gives examples.

Attribute	Description	Example
`data-sap-ui-type`	Type of control	`<div data-sap-ui-type="sap.m.Button"></div>`
`data-sap-ui-aggregation`	Defines the aggregation that shall be used for the element or child element	`<div data-sap-ui-type="sap.m.Panel"><div data-sap-ui-aggregation="content" data-sap-ui-type="sap.m.Button" data-text="My Button"></div></div>`
`data-sap-ui.default-aggregation`	Sets or overrides the default aggregation of a control	`<div data-sap-ui-type="sap.m.Panel" data-sap-ui-default-aggregation="headerToolbar"><div data-sap-ui-type="sap.m.Toolbar"></div></div>`
`id`	Defines the ID property of a control	`<div data-sap-ui-type="sap.m.Button" id="myButton"></div>`
`class`	Adds a style class to the control	`<div data-sap-ui-type="sap.m.Button" class="myButton"></div>`

Enabling Declarative Support

Declarative support needs to be enabled in the HTML document by adding an attribute to the SAPUI5 bootstrap script tag.

This is done as follows:

```
data-sap-ui-modules="sap.ui.core.plugin.DeclarativeSupport"
```

SAPUI5 then requires (loads) the plugin `sap.ui.core.plugin.DeclarativeSupport`. When started, the plugin parses and enhances special HTML tags in the HTML document. The complete bootstrap script tag for SAPUI5 (based on a CDN version) looks as follows:

```
<script id="sap-ui-bootstrap"
  type="text/javascript"
  src="resources/sap-ui-core.js"
  data-sap-ui-theme="sap_belize"
  data-sap-ui-libs="sap.m"
  data-sap-ui-modules="sap.ui.core.plugin.DeclarativeSupport"
  >
</script>
```

Defining Controls

For declarative support, define the controls in your HTML document as HTML tags.

For this, use the following data attribute that defines the SAPUI5 control that should be rendered in the HTML tag by using the HTML tag as its UI area:

```
data-sap-ui-type="sap.m.Button"
```

Rendering a button in the body of an HTML document without setting any property, association, event, or aggregation looks as follows:

```
<body>
  <div data-sap-ui-type="sap.m.Button"></div>
</body>
```

> **i** Note
>
> Make sure that you close the tags properly. HTML5 does not support self-closing tags.

> **i** Note
>
> All attributes used to define properties, associations, events, or aggregations are data attributes except for attributes that exist in HTML, for example `id` or `class`. Data attributes are prefixed with `data-*`, for example `data-text`.

Declarative Support: Properties

For setting a property, define the property as a data attribute of the corresponding HTML tag.

To add text to the button, add the attribute `data-text` to its HTML tag:

```
<div data-sap-ui-type="sap.m.Button" data-text="HelloWorld"></div>
```

> **i** Note
>
> To define a property with upper case characters, you have to "escape" them with a dash character, similar to CSS attributes. The following code gives an example:
>
> ```
> <div data-sap-ui-type="sap.ui.commons.ApplicationHeader" data-display-
> logoff="false" data-display-welcome="false">
> ```
>
> As the name of the attributes of HTML tags are case-insensitive, the properties `displayLogoff` and `displayWelcome` of the `ApplicationHeader` control have to be "escaped" as `data-display-logoff` and `data-display-welcome` for the name of the attributes of the HTML tag. Keep this in mind when matching properties, associations, or events as an attribute of the HTML tag.
>
> The `id` attribute defines the ID of a control:
>
> ```
> 
> ```

To add a CSS class to the control, use the class attribute:

```
<div data-sap-ui-type="sap.m.Button" class="my-button"></div>
```

Declarative Support: Associations

An association is defined as a data attribute of the HTML tag. Instead of passing the reference to another control you define the ID of another control.

The following code gives an example:

```
<div data-sap-ui-type="sap.m.Label" data-text="Message:" data-label-
for="message"></div>
<div data-sap-ui-type="sap.m.Input" id="message"></div>
```

The code snippet defines the link between `Label` and `Input` by using the ID of `Input` as a value for the `data-label-for` attribute of the `Label`.

Declarative Support: Events

The value of the event data attribute contains the name of a JavaScript function which will be used as callback once the event has been triggered.

The following code snippet gives an example how a change of `Input` results in an alert with its new value when the focus is lost:

```
<script>
  function handleChange (oEvent) {
    alert (oEvent.getSource().getValue());
  }
</script>
<div data-sap-ui-type="sap.m.Input" data-value="Change me!" data-
change="handleChange"></div>
```

Currently, SAPUI5 only supports to specify the name of a callback function. You can define callback functions within any class, see the following code example:

```
<div data-sap-ui-type="sap.m.Input" data-value="Change me!" data-change=
"my.company.MyClass.handleChange"></div>
```

Declarative Support: Aggregations

Aggregation support is required to allow nested controls for layout containers and/or add elements to a control, for example, for `ComboBox`.

SAPUI5 uses the control's default aggregation as default. If, for example, the panel control has the default aggregation content, all child elements of the `data-sap-ui-type="sap.ui.commons.Panel"` element are added to this aggregation:

```
<div data-sap-ui-type="sap.ui.commons.Panel">
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 1"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 2"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 3"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 4"></div>
</div>
```

The markup in the example above generates an instance of the `sap.ui.commons.Panel` control and adds implicit four buttons to the default aggregation content of the control.

You can also explicitly declare an aggregation. In general, an explicit aggregation is expressed with a meta HTML tag between the parent controls HTML tag and the HTML tags of the children. The following code adds four buttons explicitly to the "content" aggregation of the declared panel:

```
<div data-sap-ui-type="sap.ui.commons.Panel">
  <div data-sap-ui-aggregation="content">
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 1"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 2"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 3"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 4"></div>
  </div>
</div>
```

For aggregations with the cardinality "0..1" the `"data-sap-ui-aggregation"` attribute can be written directly to the control tag:

```
<div data-sap-ui-type="sap.ui.commons.Panel">
  <div data-sap-ui-aggregation="title" div data-sap-ui-
type="sap.ui.commons.Title" data-text="My Panel"></div>
</div>
```

The default aggregation of the declarative support is usually also the default aggregation of the control as defined in the control's meta information. However, when no default aggregation is set or another aggregation should be used as a default, for example to avoid unnecessary meta tags, it can be useful to define a so-called default aggregration attribute on the parent controls HTML tag. This is done as follows:

```
data-sap-ui-default-aggregation="title"
```

With this, all children which are not included in the `data-sap-ui-aggregation` meta tag are added to the default aggregation. This is shown in the following example:

```
<div data-sap-ui-type="sap.ui.commons.Panel" data-sap-ui-default-
aggregation="title">
  <div data-sap-ui-type="sap.ui.commons.Title" text="My Panel"></div>
  <div data-sap-ui-default-aggregation="content">
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 1"></div>
    <div data-sap-ui-type="sap.ui.commons.Button" data-text="My Button 2"></div>
  </div>
/div>
```

You can now apply this to the `MatrixLayout` as follows:

```
<div data-sap-ui-type="sap.ui.commons.layout.MatrixLayout" data-layout-
fixed="false">
  <div data-sap-ui-type="sap.ui.commons.layout.MatrixLayoutRow">
    <div data-sap-ui-type="sap.ui.commons.layout.MatrixLayoutCell">
      <div data-sap-ui-type="sap.ui.commons.TextField" data-value="Hello
World"></div>
    </div>
    <div data-sap-ui-type="sap.ui.commons.layout.MatrixLayoutCell">
      <div data-sap-ui-type="sap.ui.commons.Button" data-text="Hello World"></
div>
    </div>
  </div>
</div>
```

Or you can add `ListItems` to a `ComboBox`:

```
<div data-sap-ui-type="sap.ui.commons.ComboBox" data-value="Item 1">
  <div data-sap-ui-type="sap.ui.core.ListItem" data-text="Item 1"></div>
  <div data-sap-ui-type="sap.ui.core.ListItem" data-text="Item 2"></div>
  <div data-sap-ui-type="sap.ui.core.ListItem" data-text="Item 3"></div>
  <div data-sap-ui-type="sap.ui.core.ListItem" data-text="Item 4"></div>
  <div data-sap-ui-type="sap.ui.core.ListItem" data-text="Item 5"></div>
</div>
```

Declarative Support: Data Binding

Declarative support in SAPUI5 also enables data binding.

Just add the model path in curly brackets and bind the model to the control (or parent control):

```
<div data-sap-ui-type="sap.m.Button" data-text="{/stringValue}" data-
enabled="{model2>/booleanValue}"></div>
```

0..n aggregations can define templates to use for the list binding:

```
<div data-sap-ui-type="sap.m.Carousel" data-content="{/buttons}">
    <div data-sap-ui-type="sap.m.Button" data-text="{title}"></div>
</div>
```

In the example above, the button template is used for the carousel content data binding.

Related Information

Data Binding [page 815]

Compiling Declarative HTML

SAPUI5 provides a plugin for controls that are defined as declarative markup on startup time.

To compile the declarative UI markup deferred, for example, when the markup is dynamically loaded and added to the DOM you can call the `sap.ui.core.plugin.DeclarativeSupport`.compile method, see the following code snippet:

```
<div id="button">
  <div data-sap-ui-type="sap.m.Button" data-text="This button is added
dynamically"></div>
</div>
<script>

sap.ui.core.plugin.DeclarativeSupport.compile(document.getElementById("button"));
</script>
```

Error, Warning, and Info Messages

SAPUI5 provides a central place for storing and managing info, warning, and error messages.

Messages can be used to notify the user about specific states of the application and can help the user to correct their incorrect inputs. The central `MessageManager` for storing messages is available globally by calling `sap.ui.getCore().getMessageManager()` and the central `MessageModel` for managing messages is available by calling `sap.ui.getCore().getMessageManager().getMessageModel()`.

Message Object Properties

The following properties of `sap.ui.core.message.Message` instances are important:

- Target: Describes the part of the application to which the message applies. If the target is empty, the message applies to the entire application. The target format depends on the used message processor. Currently,SAPUI5 supports two types of targets:
  - Control IDs with control properties: The `sap.ui.core.message.ControlMessageProcessor` propagates these messages to the affected control.
  - Binding path: The `sap.ui.model.Model` propagates these messages to affected bindings.
- Message processor: The object that handles the message in the application and propagates the message to correct controls, bindings, or other objects, see `sap.ui.core.message.ControlMessageProcessor` in the API reference.
- Type: Defines the severity of the message; possible types are: error, warning, info, and success, see `sap.ui.core.MessageType` in the API reference.
- Message text: The actual message text describing the issue. This text is shown to the user.
- Persistent: This property influences the lifecycle of the message. Non-persistent messages are cleaned up by the framework messaging lifecycle, persistent messages have to be removed manually by the application.

Message Creation

There are several ways to create messages automatically and push them into the central message model:

- Validation messages refer to a control. They are created by the SAPUI5 framework when data is parsed, formatted, and validated according to defined data types, see Formatting, Parsing, and Validating Data [page 854]. Such messages are propagated to one specific control. For more information, see Validation Messages [page 1065].
- OData V2 messages refer to a binding path. They are typically managed by the server and are changed every time the back end responds to a request. Such messages are propagated to all bindings with the specific binding path. For more information, see OData V2 Messages [page 1067].

You can also create messages manually or extend the messaging features provided by the framework:

- You can create custom messages manually via the central `sap.ui.core.message.MessageManager` APIs. For these manually created messages, the application has to ensure a proper message lifecycle.
- For custom target formats, you can use the custom message processor. The own message processor has to inherit from the class `sap.ui.core.message.MessageProcessor`.
- If the used back end serves messages in a special way, you can use your own implementation of `sap.ui.core.message.MessageParser`. For more information, see Implementing Your Own OData V2 Message Parser [page 1071].

Related Information

API Reference: `sap.ui.core.message.Message`

API Reference: `sap.ui.core.MessageType`

API Reference: `sap.ui.core.message.MessageManager`

API Reference: `sap.ui.model.message.MessageModel`

API Reference: `sap.ui.core.message.MessageParser`

API Reference: `sap.ui.core.message.MessageProcessor`

API Reference: `sap.ui.core.message.ControlMessageProcessor`

API Reference: `sap.ui.model.Model`

Validation Messages

Validation messages are either created by the framework and processed by the `sap.ui.core.message.ControlMessageProcessor` or manually by the application.

Target

The target of a validation message can be empty. In this case, the message has no specific target and is relevant for the whole application. If a target is set, the target is a string consisting of a control ID, a slash ("/"), and the name of the property to which the message applies.

Example: `label0/text`

Lifecycle

Validation messages are added with a target referencing a control and its specific property. The messages are kept until a validation message for the property is created and assigned. If new data for the same property is received from the server, the valdation messages are erased unless their `persistent` property is set to `true`.

Automatically Created Messages

Validation messages are generated by the framework type validation when data changes. If a bound property has an assigned type, the validation can trigger the message creation. To activate the automatic message creation, the following options exist:

- Component:
  You can activate the automatic message generation in the component metadata or as a parameter when instantiating the component as follows:

```
// "UIComponent" required from "sap/ui/core/UIComponent"
// "ComponentContainer" required from "sap/ui/core/ComponentContainer"
UIComponent.extend("MyComponent", {
    metadata  : {
        version  : "1.0" ,
        handleValidation  : true
    }
});
```

```
var oComponentContainer = new ComponentContainer("MyComponentContainer", {
    name: "MyComponent",
    id: "myComponentId",
    handleValidation: true
});
```

- Descriptor for Applications

You can activate the automatic message generation in the `"sap.ui5"` section of the `manifest.json` file as follows:

```
"sap.ui5": {
       "handleValidation": true
}
```

- Control
  You can activate automatic message generation for controls by registering the control in the message manager as follows:

```
// "Input" required from "sap/m/Input"
// "TypeFloat" required from "sap/ui/model/type/Float"
var oInput = new Input({
    value: { path: "/Products(1)/Price", type: new TypeFloat() }
    value: { path: "/Products(1)/Price", type: new sap.ui.model.type.Float() }
});
sap.ui.getCore().getMessageManager().registerObject(oInput, true);
```

> **i Note**
>
> If you don't set the second attribute of `sap.ui.core.message.MessageManager.registerObject` to `true`, the event is canceled without any message, see `sap.ui.core.message.MessageManager.registerObject` in the API reference.

Manually Created Messages

You can also create validation messages manually and add them to the message manager. If you add the message to a control property that is bound and validated by a data binding type, your message gets deleted when new validation results from the type comes in. You can override this behavior by setting the `persistent` property of the message to `true`.

```
var oMessageProcessor = new sap.ui.core.message.ControlMessageProcessor();
var oMessageManager  = sap.ui.getCore().getMessageManager();
oMessageManager.registerMessageProcessor(oMessageProcessor);
var oInput = new sap.m.Input({
    id: "myInputId",
    value: { path: "/Products(1)/Price" , type: new sap.ui.model.type.Float() }
});
oMessageManager.addMessages(
    new sap.ui.core.message.Message({
        message: "ZIP codes must have at least 23 digits",
        type: sap.ui.core.MessageType.Error,
        target: "/myInputId/value",
        processor: oMessageProcessor
     })
);
```

Related Information

API Reference: `sap.ui.core.ControlMessageProcessor`

API Reference: `sap.ui.core.message.MessageManager`

OData V2 Messages

OData V2 messages are either created automatically by `sap.ui.model.odata.ODataMessageParser` and processed by the `sap.ui.model.odata.v2.ODataModel` or can be created manually by the application.

Target

The target of these messages can be empty. In this case, the message has no specific target and is relevant for the whole application. If a target is set, it must correspond to a binding path which is then used to propagate the message to the corresponding bindings. If these bindings belong to a control that implements the `refreshDataState` function, the control is able to react to data state changes.

Lifecycle

OData V2 messages are kept until a message from the server for the same path arrives. The server always sends all messages for a specific target which means that all current messages are replaced with the ones sent by the server, except for `persistent` UI messages. Back-end messages with property `transition` set to `true` are parsed to `persistent` UI messages.

Manually Created Messages

To create messages manually that are handled like OData messages, use `model` as message processor as follows:

```
// oMyModel is defined elsewhere...
// "Input" required from module "sap/m/Input"
// "TypeFloat" required from module "sap/ui/model/type/Float"
// "Message" required from module "sap/ui/core/message/Message"
// "coreLibrary" required from module "sap/ui/core/library"
var oMessageManager = sap.ui.getCore().getMessageManager();
oMessageManager.registerMessageProcessor(oMyModel);
var oInput = new Input({
    id: "myInputId",
    value: { path: "/Products(1)/Price", type: new TypeFloat() }
});
oMessageManager.addMessages(
    new Message({
        message: "Price must contain only numbers",
        type: coreLibrary.MessageType.Error,
        target: "/Products(1)/Price",
        processor: oMyModel
    })
);
```

Automatically Created Messages

The `sap.ui.model.odata.v2.ODataModel` supports automatic parsing of OData V2 messages by means of `sap.ui.model.odata.ODataMessageParser`.

For other back-end service types, an application can implement its own parser, see Implementing Your Own OData V2 Message Parser [page 1071].

OData V2 Message Parser

The `ODataMessageParser` is created automatically for all `v2.ODataModel` instances and parses all responses from the server. The `ODataModel` implements the message processor interface and is used to propagate the messages to the message manager. In case of an error response, the response body is parsed for error messages. In case of a successful response, the "sap-message" header is parsed as a JSON-formatted error object. The name of the header field can be changed by calling the `setHeaderField()` method on the `ODataMessageParser`.

Troubleshooting for the OData V2 Message Parser

In this section you find known limitations of the `ODataMessageParser` and how you can resolve issues with unexpected numbers of UI messages.

Duplicate messages with different targets

Mulitple OData changes that are part of the same change set are send as batch request to the back end, for example:

- Change operation (POST Product('id=123'))
- Change operation (POST Product('id=456'))

If one change operation fails, the back end rolls back all operations of the change set, but returns only a single message, for example:

```
{
  "code": "MYCODE/111",
  "message": "Invalid input!",
  "severity": "error",
  "target": ""
}
```

This leads to two UI message objects with the following message targets:

- "/Product('id=123')"
- "/Product('id=456')"

Solution: The message target has to be defined in the back-end error message. By this, only one UI message which represents the failed change is created and pushed into the central message model. Otherwise, the `ODataMessageParser` creates a separate error message for every change included in the change set.

Duplicate messages with the same target

The OData service error response can also contain multiple inner-errors to deliever more than one error message to the front end. The inner-error messages should generally be used to describe the problem in more detail, for example

```
{
  "code": "MYCODE/111",
  "message": "Failed operations!",
  "severity": "error",
    "details": [
    {
      "code": "MYCODE/222",
      "message": "Object 1 already exists!"
      "severity": "error"
    },
    {
      "code": "MYCODE/222",
      "message": "Object 2 already exists!"
      "severity": "error"
    },
    ...
  ]
}
```

Solution: The outer-error will also be parsed into a separate UI error message. These general error messages, such as *Failed operations*, can be confusing for end users. To prevent these general UI messages, the outer-error message has to have the same error code and error message text ("message" property) as an inner-error. This way, the outer message information is ignored, that is, not parsed into a UI message, since there already is an inner-error message with more details that represents the same issue.

Hint: The duplicate detection only works for request body error messages. For request header error messages, the unwanted outer-error must be filtered out in the front end. As an alternative, the outer-error could already represent the first detailed error, see the example error with message *Object 1 already exists*.

Scenario 3: Missing UI messages

This can happen when an OData entity is changed and the same entity is requested again shortly afterwards. The change and the read operation could also be part of the same batch request, for example:

- Change operation ('POST' request with target "Product('id=123')")
- Read operation ('GET' request with target "Products")

If the change operation fails, a UI message is created. But this UI message is deleted directly afterwards via the messaging lifecycle since the read operation of the same entity does not return any message. There are two options to get the expected behavior in this scenario:

Solution 1: Mark the UI message as persistent. By this, the message lifecycle will not delete the UI message, but the application has to take care of cleaning up such messages by using the `sap.ui.core.message.MessageManager` APIs.

Solution 2: Defer the read operation. By this, the UI message is also not deleted, but the application has to make sure the read operation is triggered at an appropriate point in time.

Hint: The read operation is often automatically triggered by the `v2.ODataModel`. To prevent this request from being sent, you can use the model parameter `refreshAfterChange`.

Related Information

API Reference: `sap.ui.model.odata.v2.ODataModel`
API Reference: `sap.ui.core.message.MessageManager`
API Reference: `sap.ui.model.message.MessageModel`
API Reference: `sap.ui.mode.odata.ODataMessageParser`
Server Messages in OData V4 Model [page 982]

Message Model

The message model contains all messages and is used to bind to the messages to display them.

The message model is retrieved from the message manager by calling the `getMessageModel()` method. You can use it directly in the application, or you can use it as a reference implementation.

Using the Message Model

You use the message model like any other model to bind an aggregation to a root path ("/), for example the items in a list, and add filters and sorters. The `MessagePopover` control is used to display the messages to the user:

```
// "Button" required from "sap/m/Button"
// "MessagePopover" required from "sap/m/MessagePopover"
// "MessagePopoverItem" required from "sap/m/MessagePopoverItem"
var oMessagePopoverButton = new Button({
    text: "Show MessagePopover",
    type: "Accept",
    press: function() {
        oMP.openBy(this);
    }
});
var oMP = new MessagePopover({
    items: {
        path:"message>/",
        template: new MessagePopoverItem({ description: "{message>description}",
type: "{message>type}", title: "{message>message}"})
    }
});
oMP.setModel (sap.ui.getCore().getMessageManager().getMessageModel(),"message");
oMessagePopoverButton.placeAt("content");
```

> **i Note**
>
> For an example how to bind to the message model and show the messages to the user, see `sap.m.MessagePopover` in the API reference.

Related Information

API Reference: `sap.ui.model.message.MessageModel`
API Reference: `sap.ui.core.message.MessageManager`
API Reference: `sap.ui.core.message.MessagePopover`

Implementing Your Own OData V2 Message Parser

A message parser is a simple interface that is implemented to allow the propagation of messages from back end services. For messages from OData V2 services, the `sap.ui.model.odata.ODataMessageParser` is used.

If you have your own service implementation, for example, a JSON-based back end that also sends messages, you can implement your own message parser by implementing the `sap.ui.core.message.MessageParser` interface.The interface is very simple: It has only the `parse` and the `setProcessor` method. The `parse` method takes at least one parameter, that is, the `response` object from the server. The method can take more model-specific arguments. The `setProcessor` method takes only one argument, the `processor` object that is used to propagate the messages, this is usually the `model` instance.

The main task of the message parser is to retrieve the messages from the back end response and then calculate the message delta that is handed over to the message processor by means of the two parameters `oldMessages` and `newMessages` of the `messageChange` event. The `oldMessages` parameter specifies the messages that are to be removed, and the `newMessages` parameter specifies the messages that are to be added.

```
this.getProcessor().fireMessageChange({
    oldMessages: aRemovedMessages,
    newMessages: aNewMessages
});
```

The delta calculation must be a back end-specific implementation. In the OData implementation, for example, all messages for the requested resource(s) must be returned from the back end on every request. This means that all messages that were available before with a target that corresponds to the requested resources must be put in the `oldMessages` parameter of the event.

Related Information

sap.ui.model.odata.ODataMessageParser
sap.ui.model.odata.v2.ODataModel
sap.ui.core.message.MessageParser

Routing and Navigation

SAPUI5 offers hash-based navigation, which allows you to build single-page apps where the navigation is done by changing the hash. In this way the browser does not have to reload the page; instead there is a callback to which the app and especially the affected view can react. A hash string is parsed and matched against patterns which will then inform the handlers.

You use routing in the following cases:

- Enable users to navigate back using the browser history, for example, the *Back* button of the browser or a physical back button on mobile devices.
- Enable bookmarks and deep links to pages inside an app; this means that you can start the app and resume the bookmarked state.
- Pass on data via the hash to application logic.

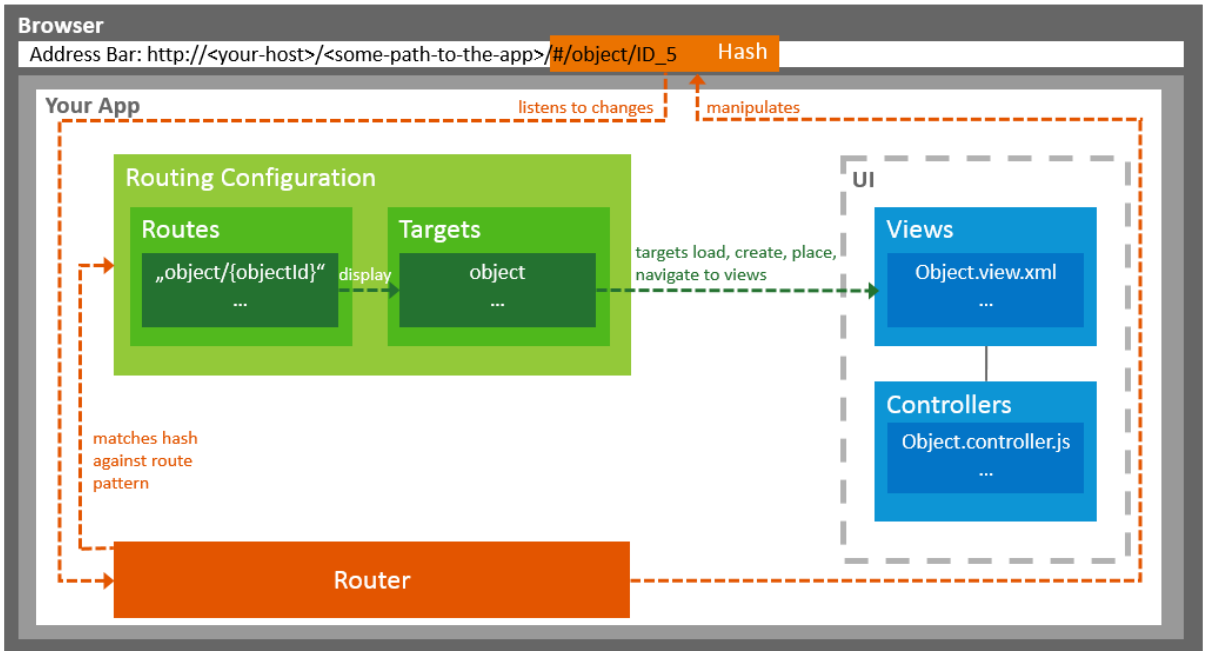


Figure 216: Routing overview

In SAPUI5, navigation and routing is implemented using a “router” to forward the hash change and the data in the hash to one or more views of the app.

You use **routes** to notify your application that the hash has changed to a certain value. For each route, you define the pattern that can be used in the app implementation.

With **targets**, you define where a view or a component is loaded and where the view or component is shown on the UI. By referring to one or multiple targets in a route's definition, you can load and show the views or components once the route's pattern matches the current hash.

You configure routing in SAPUI5 in the descriptor file (`manifest.json`) (see Descriptor for Applications, Components, and Libraries [page 734]) or in the `Component.js` file (see Components [page 720]) to have it available globally throughout your app, but you can also define routes and targets locally by calling the constructors of the classes, for example under the `sap.ui.core.routing` and `sap.m.routing` namespaces.

> **i Note**
>
> You can also define only routes or only targets, but then just have to make sure that you implement the counterpart elsewhere.

Routing Patterns

Whenever a hash is added to a URL, the router checks whether there is a route with a matching pattern. The first matching route is taken and the corresponding target view is called. The data provided with the hash are passed on to the target.

You can use the following kinds of patterns:

- **Hard-coded pattern**:
  The pattern matches the hash exactly. For example, when a pattern is defined as `product/settings`, this pattern matches only if the hash is **`product/settings`** and no data is passed on to the events of the route.
  For more information, see the tutorial Step 6: Navigate to Routes with Hard-Coded Patterns [page 312].
- **Route with mandatory parameter**:
  You can define mandatory parameters for the pattern by placing the parameter in curly brackets (`{`*parameter ID*`}`).
  For example, if you define the pattern `product/{id}`, the hashes **`product/5`** and **`product/3`** (where 3 and 5 are product IDs) match the pattern. The matched event handler gets `5` or `3` passed on with the key `id` in its arguments. But hash **`product/`** does not match the pattern because the mandatory parameter is missing.
  For more information, see the tutorial Step 7: Navigate to Routes with Mandatory Parameters [page 317].
- **Route with optional parameter**:
  You can define optional parameters for the pattern by placing the parameter between colons (`:`*parameter ID*`:`).
  For example, if you define a pattern `product/{id}/detail/:detailId:`, the `detailId` parameter is optional, whereas `id` is mandatory. Both hashes `product/5/detail` and `product/3/detail/2` match the pattern.
- **Route with query parameter**:
  The query parameter allows you to pass on queries with any parameter. A query parameter starts with `?`, and you can either define it as mandatory (`product/{?query}`) or optional (`product/:?query:`).
  The matched value will be converted into an object saved with the parameter name as the key when passed to the event handler.
  For more information, see the tutorial Step 9: Allow Bookmarkable Tabs with Optional Query Parameters [page 334].
- **"rest as string" parameter**:
  A parameter that ends with an asterisk (`*`) is called a "rest as string" parameter. Such a parameter matches as much as possible. It can be combined with the syntax of mandatory or optional parameters.
  For example, a pattern `product/{id}/:detail*:` defines a mandatory parameter with the name `id` and an optional "rest as string" parameter with the name `detail`. It matches `product/5/3` and `product/5/detail/3/foo`. The event handler gets `3` or `detail/3/foo` passed on with the key `detail` in its arguments.

→ Tip

For a better understanding about how patterns work and what matched parameters look like, see the following page in the *Samples* in the Demo Kit: sap.ui.core.sample.PatternMatching/preview.

ℹ Note

SAPUI5 uses Crossroads.js for parsing the hash and the Hasher framework for manipulating the hash.

Related Information

Tutorial: Navigation and Routing [page 291]
API Reference: `sap.ui.core.routing`
API Reference: `sap.m.routing.Router`
API Reference: `sap.ui.core.routing.Route`: Constructor Detail
Crossroads.js
Hasher framework on GitHub

Routing Configuration

Routing configuration consists of `routes`, `targets`, `config`, and `owner`.

Routes

Each route defines a name, a pattern, and optionally one or more targets to which to navigate when the route has been matched. In the `routes` section, you define which patterns are available for navigation.

- The `name` of the route (unique within one router instance)
- The `pattern` as hash part of the URL that matches the route
- The navigation `target` as defined in the `targets` section
  If you want to load multiple views/components at the same time, you can assign multiple targets (see Working with Multiple Targets [page 1082]).
- If a target is configured for loading a component, you can enable the routing in the loaded component, see Enabling Routing in Nested Components [page 1086].
- The `titleTarget` to specify from which target the title is taken when multiple targets are displayed. If no `titleTarget` is defined, the first target that has a `title` is chosen (see Using the title Property in Targets [page 1083]).

The sequence of the routes in the `routes` definition is important. As soon as a pattern is matched, the following patterns are ignored. To prevent this for a specific route, you use the `greedy` parameter. If set to `true`, the route is always taken into account.

For more information, see API Reference: `sap.m.routing.Router`.

Targets

A target defines the view or component that is displayed. It is associated with one or more routes or it can be displayed manually from within the app. Whenever a target is displayed, the corresponding view or component is loaded and added to the aggregation configured with the `controlAggregation` option of the control. The target definition can contain the following parameters:

- The target key
- The `type` to specify whether the target is a view or a component
- The `name` to specify the name of the view or component
- Additional optional parameters
  If you don't specify a parameter, the default value is taken from the `config` section.
  - `viewType` (e.g. `XML`) which is valid only when the `type` is set to "View"
  - `id` of the view or component instance
    A view or component instance is cached in SAPUI5 routing under the combination of its `name` and `id`. If there already is one instance created for a specific view or component with an `id`, this instance is reused if another target with the same `name` and `id` is displayed. If a new instance needs to be created instead of reusing the existing ones, assign the target a different `id`.
  - `viewLevel`
    You can use different levels to define the navigation direction,for example the navigation from a lower view level to a higher view level leads to forward navigation. This is, for example, important for `flip` and `slide` transitions, where the slide animation should go from left to right or vice versa.
  - `controlId` of the control that is used as the parent to insert the view or component (e.g. `app`)
  - `controlAggregation` target aggregation of the control with `controlId` to which the view or component is added
    The `NavContainer` control, for example, has an aggregation called `Pages` and the shell container has `Content`.
  - `parent`: the key of another target which a view is created and added before the target view or component is added
  - `path`: the namespace of the view or component
  - `targetParent` where the control with the `controlId` is located (see Working with Multiple Targets [page 1082]); this option is set automatically for the root view of a component if the router instance is instantiated by the component.
  - `clearAggregation` specifies whether the aggregation should be cleared before adding the new view instance.
    When you use the `sap.m.routing.Router` the default is `false`, for `sap.ui.core.routing.Router` it is `true`.
    When using `sap.ui.ux3.Shell` this value should be set to `true`, for `sap.m.NavContainer` to `false` to ensure that the correct content is shown.
  - `transition` defines how the transition happens; you can choose between `slide` (default), `flip`, `fade`, and `show`.
  - `title` contains either a static text or a valid binding syntax, e.g. to an i18n model, which is resolved under the binding context of the view (see Using the title Property in Targets [page 1083])

> **i Note**
>
> You can also use targets without routes to call a view directly . For more information, see the tutorial Step 5: Display a Target Without Changing the Hash [page 308] and Step 10: Implement "Lazy Loading" [page 338], and the sample Targets Without a Router in the *Samples* in the Demo Kit.

For more information, see API Reference: `sap.m.routing.Router`.

Config

The `config` section contains the global router configuration and default values that apply for all routes and targets. The `config` section contains the following settings.

- `routerClass` defines which router is used.
  You can either use class `sap.ui.core.routing.Router` (default) or `sap.m.routing.Router`. If you use a `sap.m` control (such as `NavContainer` or `SplitApp`) in your app, you can benefit more from using `sap.m.routing.Router` because it not only loads the targets and places them in the corresponding container, but also triggers the animation for navigating to the right target.

  > **i Note**
  >
  > The possible values for `routerClass` are `sap.ui.core.routing.Router`, `sap.m.routing.Router`, or any other subclasses of `sap.ui.core.routing.Router`.
  >
  > Compared to `sap.ui.core.routing.Router`, the `sap.m.routing.Router` is optimized for mobile apps and adds the properties `viewLevel`, `transition`, and `transitionParameters` which can be specified for each route or target created by the `sap.m.routing.Router`. The `transitionParameters` can also be used for custom transitions. See the *API Reference* for more information.

- The `homeRoute` defines the route whose target title is inserted as the first entry in the title history in the `titleChanged` event or in the return value of `sap.ui.core.routing.Router.prototype.getTitleHistory`. For more information, see section *Initial title of the home page* of Using the title Property in Targets [page 1083].
  The property contains the `name` of one of the routes that are defined in the `routes` section as value.
- You can also define default values for all target parameters
- `async` defines whether targets are loaded asynchronously; the default value is `false`. We recommend setting this parameter to `true` to improve performance.

  > **i Note**
  >
  > A target with `type` "Component" is only displayed with asynchronous loading.

  > **i Note**
  >
  > If you use asynchronous loading, you cannot rely on the sequence of events that are fired during the load phase. If you follow our programming model with MVC, this should not be a problem.

- Using the `bypassed` parameter, you specify the navigation target that is used whenever no navigation pattern is matched. If you use this setting, you also have to define a corresponding target in the `targets` section.

For more information, see API Reference: `sap.m.routing.Router`.

Owner

The `owner` parameter defines the owner of all views that are created by the router. This is typically a `UIComponent`. This parameter is set automatically if the router instance is instantiated by a component.

Example

```
{
    metadata: {
        routing: {
            config: {
                async: true
                viewType: "XML",
                path: "view",
                controlId: "splitApp",
                clearTarget: false,
                bypassed: {
                    target: "notFound"
                },
                homeRoute: "home"
            },
            routes: [
                {
                    pattern: "",
                    name : "home",
                    target: "home"
                },
                {
                    pattern: "category/{id}",
                    name: "category",
                    target: "category"
                },
                {
                    pattern: "category/{id}/product/{productId}",
                    name: "product",
                    target: ["category", "product"]
                },
],
            targets: {
                category: {
                    type: "View",
                    name: "Category",
                    controlAggregation: "masterPages"
                },
                product: {
                    type: "View",
                    name: "Product",
                    controlAggregation: "detailPages",
                },
                home: {
                    type: "View",
```

```
                    name: "Home",
                    controlAggregation: "masterPages"
                },
                notFound: {
                    type: "View",
                    name: "NotFound",
                    controlAggregation: "detailPages",
                    parent: "home"
                }
            }
        }
    }
}
```

In this example, the `Home` view is always shown when the hash is empty. The `Category` view is shown when the hash matches the pattern `category/{id}`. Both, the `Category` and the `Product` view are shown when the hash matches the pattern `category/{id}/product/{productId}`, because both of them are added to the `target` property of the `product` route.

Related Information

API Reference: `sap.ui.core.routing`
API Reference: `sap.m.routing.Router`
Sample: Targets Without a Router
Working with Multiple Targets [page 1082]
Tutorial: Navigation and Routing [page 291]
Enabling Routing in Nested Components [page 1086]

Methods and Events for Navigation

SAPUI5 provides a method and events for navigation.

Methods

Navigation can be triggered by method `navTo` on `Router` with changing the hash or method `display` on `Targets` for showing a new view without changing the hash.

`navTo` method

Use this method to navigate to the given route and fill the hash with the corresponding data. It the route contains a target, the target is displayed. The listener callbacks of controllers listening to this route are provided with data. When changing the hash, all listeners to this hash are informed.

The method uses the following parameters:

- `name` of the route parameter

- route `parameters`
- route information for the `Component` target(s), see Navigate with Nested Components [page 1090].
- `replace` (default: `false`) to define whether the hash should be replaced (no new browser history entry) or set (browser history entry)

```
sap.ui.require([
    "sap/ui/core/UIComponent", ...
], function(UIComponent, ...) {
    sap.ui.controller("MyApp.View2", {
        anyEvent: function() {
            var oRouter = this.getOwnerComponent().getRouter();
            oRouter.navTo("product", {
                id: "5",
                productId: "3"
            });
        }
    });
});
```

`display` method

Use this method to navigate to display one or multiple targets. The method uses the target name or an array of target names as only parameter.

Events

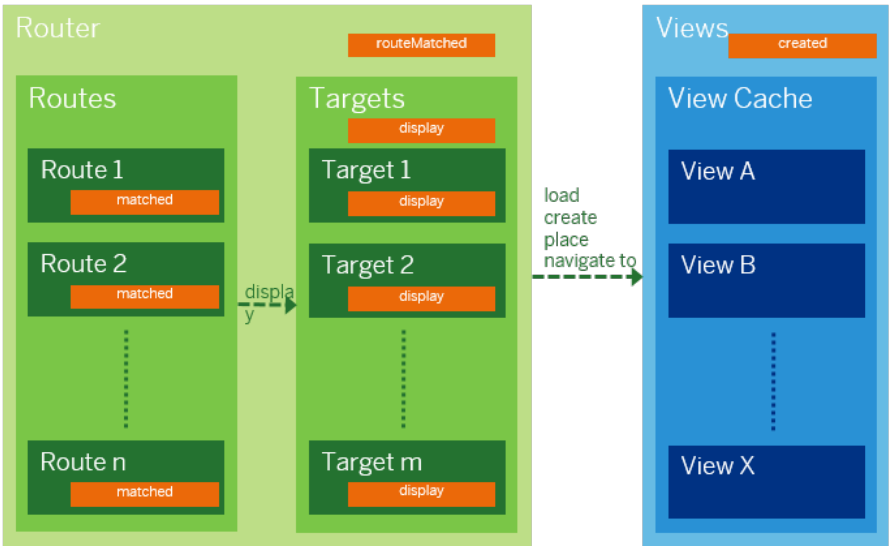


Figure 217: Navigation events

Events `attachRouteMatched` on `Router` and `attachMatched` on `Route`

These events are fired when a hash matches a route or a pattern. The `routeMatched` event is fired if a pattern of any route in the routing configuration is matched. The matched event is fired for a specific route.

If you want to only react to specific routes, check if the `name` parameter matches the route that you want to listen to. The events have the following parameters:

- `name` of the route that has been matched
- `arguments` that are part of the route, mainly the parameters of the hash
- `config` of the route

```
sap.ui.controller("MyApp.View1", {
    onInit: function() {
        var oRouter = this.getOwnerComponent().getRouter();
        oRouter.getRoute("view1").attachMatched(function(oEvent) {
            this._selectItemWithId(oEvent.getParameter("arguments").id);
        }, this);
    },

    _selectItemWithId : function(id) {
        //implementation
    }
}
```

`display` **event on** `Target`

This event is fired on the target instance when this target is added and displayed on the UI. The event has the following parameters:

- `object` for the instance which is displayed; this is either a `View` instance or a `ComponentContainer` instance which wraps the loaded component
- `control` in which the target object is displayed
- `config` of the target
- `data` of the object passed when calling the `display` method

`created` **event on** `Views`

This event is fired on the view/component cache in SAPUI5 routing which can be fetched by calling the `getViews()` method on a router instance every time a new view or component has been created by navigation. The event has the following parameters:

- `object` for the created instance
- `options` containing additional options

Related Information

Tutorial: Navigation and Routing [page 291]

API Reference: `sap.ui.core.routing`

API Reference: `sap.m.routing.Router`

Initializing and Accessing a Routing Instance

This topic describes how to initialize routing in a component and access the routing functions.

Initializing

You initialize the router in the component using the following code:

```
sap.ui.define([
    "sap/ui/core/UIComponent", ...
], function (UIComponent, ...) {
    "use strict";
    return UIComponent.extend("YourComponentClassName", {
        ...,

        init: function () {
            ...
            // call the init function of the parent
            UIComponent.prototype.init.apply(this, arguments);
            // this component should automatically initialize the router
            this.getRouter().initialize();
            ...
        }
    });
});
```

The router instance is automatically destroyed when the component is destroyed.

Accessing

To access the router and to use its functions, use the `getRouter()` function or the static `getRouterFor` function of the UI component. You can pass either a controller, or a view:

```
var oRouter = sap.ui.core.UIComponent.getRouterFor(this);
```

You can also use the `getRouter` function of your UI component.

All views that are generated by the router are automatically created in the context of the component.

Related Information

Components [page 720]

Working with Multiple Targets

If you want to navigate to multiple targets with the same hash, you can either assign multiple targets to a route, or define a parent for the target.

Multiple Targets for the Same Route

In the routing configuration, you can add multiple targets for the same route.

All target view or component instances are created and loaded in the order that they appear in the target option of a route when the pattern is matched.

If you add multiple targets to a route, this builds a connection between the targets on the route level only. The targets can still be displayed separately if needed. Use this approach when the connection between the targets is only needed in some routes, but not always.

Parent Relationship

You can also define a parent for a target in the target definition. For the parent, you also create an entry in the `targets` configuration, but you don't have to create a corresponding route.

Whenever a target has a parent, an instance of the parent is always created before an instance of the target is created.

A parent relationship between targets tightly couples the two targets together. The parent target is always displayed before the child target is displayed. The child target can't be displayed without first displaying the parent target. This approach is mainly used when the view in the child target is added to an aggregation of the view in the parent target.

> **i** Note
>
> There is also a `parent` property for the `route`. This property is only used when its parent exists in a parent component of the current component (see Enabling Routing in Nested Components [page 1086]). For all other types of parent-child relationships, either use multiple targets or a parent relationship in targets.

Example

In the following example, the relationship between the views `employeeOverviewTop` and `employeeOverviewContent` is established by assigning both to the same route.

The relationship between the target `employeeOverview` and `employeeOverviewTop` (`employeeOverviewContent` respectively) is a parent relationship.

```
"routing": {
  "config": {
```

```
    path: "sap.ui.demo.nav.view.employee.overview",
    [...]
  },
  "routes": [{
    "pattern": "employees/overview",
    "name": "employeeOverview",
    "target": ["employeeOverviewTop", "employeeOverviewContent"]
  }],
  "targets": {
    "employeeOverview": {
      "type": "View",
      "name": "EmployeeOverview",
      "viewLevel": 2,
      "controlId": "app",
      "controlAggregation": "content"
    },
    "employeeOverviewTop": {
      "parent": "employeeOverview",
      "type": "View",
      "name": "EmployeeOverviewTop",
      "controlId": "EmployeeOverviewParent",
      "controlAggregation": "content"
    },
    "employeeOverviewContent": {
      "parent": "employeeOverview",
      "type": "View",
      "name": "EmployeeOverviewContent",
      "controlId": "EmployeeOverviewParent",
      "controlAggregation": "content"
    }
  }
```

For more information, see the tutorial Step 11: Assign Multiple Targets [page 343].

Using the title Property in Targets

Routing in SAPUI5 allows you to define titles declaratively in the configuration. The title can be set with valid binding syntax which is then resolved under the scope of the target to which it belongs. This means that the title can be translated when it's bound to the i18n model or resolved dynamically under the current binding context.

When a new target that has the `title` property defined is displayed, or the title of the current target changes, the `titleChanged` event is fired. The event contains the current title and the history of previously displayed titles. You can use this event to update the title of your app.

Examples for setting the title in `Target`

```
{
    ...,
    "routes": [{
        "pattern": "products/overview",
        "name": "ProductsOverview",
        "target": "products"
    }],
    "targets": {
        "products": {
```

```
            "type": "View",
            "path": "shop.products",
            "title": "Products Overview"
        }
    },
    ...
}
```

```
{
    ...,
    "routes": [{
        "pattern": "products/{id}",
        "name": "Product",
        "target": "product"
    }],
    "targets": {
        "product": {
            "type": "View",
            "path": "shop.products",
            "title": "{ parts: ['helperModel>/PRODUCTS_TITLE',
'myModel>productName'], formatter: '.myFormatterFunction' }"
        }
    },
    ...
}
```

The `title` property can also be defined on a "Component" type target. When it is set with a binding syntax, the binding is resolved in the context of the root view of the component that is loaded by this target. The router of the loaded component may also have `title` property defined on its own target(s) and eventually fire its own `titleChanged` event once a target is displayed inside the loaded component. UI5 provides a way to propagate the `titleChanged` event from a "Component" target to its owner router in order to let the event be consumed at one central place (and not at any available router). For detailed information, see Propagate `titleChanged` Event from the Nested Component to the Parent Component [page 1089].

```
{
    ...,
    "routes": [{
        "pattern": "attachment/{id}",
        "name": "Attachment",
        "target": {
          "name": "attachment",
          "prefix": "atch"
        }
    }],
    "targets": {
        "attachment": {
            "type": "Component",
            "usage": "productComponent",
            "title": "Attachment"
        }
    },
    ...
}
```

Defining `titleTarget` in `Route`

A route can display multiple targets and you can use the `titleTarget` option in the `Route` configuration to specify which target the title should be taken from explicitly. By default, the `Route` takes the title of the first target that has the `title` property defined.

```
{
    ...,
    "routes": [{
        "pattern": "product/{id}/parts",
        "name": "ProductParts",
        "target": ["product", "productParts"],
        "titleTarget": "productParts"
    }],
    "targets": {
        "product": {
            "viewPath": "shop.products",
            "viewName": "Product",
            "title": "Product"
        },
        "productParts": {
            "viewPath": "shop.products",
            "viewName": "Product",
            "title": "Product Parts"
        }
    },
    ...
}
```

Listening to the `titleChanged` event

To receive a notification when the title is changed, you can register to the `titleChanged` event on the `Router` instance. The `titleChanged` event is then fired when a target with a set `title` options displayed, or the title of a displayed target is changed (for example, because the binding context changes).

```
oRouter.attachTitleChanged(function(oEvent) {
    var sTitle = oEvent.getParameter("title"),
        aHistory = oEvent.getParameter("history");
    // Example usage: set the browser page title (optional)
    document.title = sTitle;
    aHistory.reverse().forEach(function(oHistory) {
        // show the history in a dropdown
        // oDropdown.addItem(new Item({
        //    text: oHistory.title
        //}).data("hash", oHistory.hash));
    });
});
```

> **i** Note
>
> You don't need this event in the SAP Fiori launchpad. The title is updated automatically.

Initial title of the home page

In the routing configuration, you select one of the routes as a home route that leads to the home page of your app.

If a user navigates to any view of the app using deep link navigation, the home page is also added to the navigation history as the first entry:

```
{
   hash: sHomeRoutePattern,
   isHome: true,
   title: sAppTitle
}
```

This ensures that the user can also navigate to the home page from any other view.

The title of the home page (and also any title of a route) is only defined in the `targets` section of the routing configuration. Since the user did not navigate to the home page yet, this target information is not loaded, and the title is not available. Therefore, the `title` attribute that is defined in the `manifest.json` descriptor file, is taken as placeholder for the home page title until the actual title is loaded.

Enabling Routing in Nested Components

Every SAPUI5 component can define routing configuration in its manifest and a UI5 router instance will be created automatically after the component is instantiated.

Using components as targets in routing presents another challenge: When multiple components with their own routing configuration are used in an application, their router instances listen to the browser's `hashChange` event simultaneously and may do concurrent changes to the hash. This can lead to conflicts, hence, the hash access has to be coordinated. Therefore, some additional configuration has to be made for these nested components to ensure everything is running stable.

Configure a Component as Routing Target

A target in SAPUI5 routing can load either a view, or a component. To load a component, you need to define the component in the `componentUsages` section of the owner component's `manifest.json`, see Using and Nesting Components [page 726].

Loading a child component with a type `Component` target in a router builds up a hierarchy between this router and the router in the child component.

```
{
    "sap.ui5": {
        "componentUsages": {
            "myreuse": {
                "name": "reuse.component",
                "settings": {},
                "componentData": {},
                "lazy": false
            }
```

```
        }
    }
}
```

Use the following configuration to load the component from the target:

- `type`: Set the `type` to `Component`; this loads and instantiates the `Component.js` that is available under `componentUsages`.
- `usage`: Use the key of the component usage as used in the the `componentUsages` section of the parent component's `manifest.json`.
- `options` (optional): Add additional options that are merged with the options defined in the `componentUsage` section, see sap.ui.core.UIComponent.
- `containerOptions` (optional): Add additional options that are passed to the constructor of the component container where the component is rendered, see sap.ui.core.ComponentContainer.

```
{
    "sap.ui5": {
        "componentUsages": {
            "myreuse": {
                "name": "reuse.component",
                "settings": {},
                "componentData": {},
                "lazy": false
            }
        },
        "routing": {
            "config": {
                ...
            },
            "routes": [
                ...
            ],
            "targets": {
                "attachment": {
                    "type": "Component",
                    "usage": "myresue",
                    "options": {
                        // optional
                        // define the additional parameter for
                        // instatiating the component instance
                    },
                    "containerOptions": {
                        // optional
                        // define the additional parameter for
                        // instantiating the component container
                        // which enables the component to be rendered
                        // in the parent control
                    },
                    "controlId": "page",
                    "controlAggregation": "content"
                }
            }
        }
    }
}
```

Configure Hash Prefix for the Nested Component

The hash from every router needs to be persisted in the browser hash. To identify the ownership of the hash segments from the browser hash, a prefix needs to be assigned to the component which is loaded by a `Target`. The prefix can be defined in the `Route` where the `Target` is used.

Instead of assigning the `target` option in a route with the name of a target which is going to be displayed once the route's pattern is matched, an object is assigned which also contains the prefix of the hash for this component besides the name of the target. The loaded component from the target has its own hash segment which begins with the given prefix and can change the hash by using method `navTo` on `Router` in the same way as it is done in the top level component.

```
{
    "sap.ui5": {
        "componentUsages": {
            "myreuse": {
                "name": "reuse.component",
                "settings": {},
                "componentData": {},
                "lazy": false
            }
        },
        "routing": {
            "config": {
                ...
            },
            "routes": [{
                "name": "home",
                "pattern": "",
                "target": {
                    "name": "attachment",
                    "prefix": "atch"
                }
            }],
            "targets": {
                "attachment": {
                    "type": "Component",
                    "usage": "myreuse",
                    "options": {
                        // optional
                        // define the additional parameter for
                        // instatiating the component instance
                    },
                    "containerOptions": {
                        // optional
                        // define the additional parameter for
                        // instantiating the component container
                        // which enables the component to be rendered
                        // in the parent control
                    },
                    "controlId": "page",
                    "controlAggregation": "content"
                }
        }
    }
}
```

Propagate `titleChanged` Event from the Nested Component to the Parent Component

When the nested component `myreuse` has routing enabled, the router instance within the `myreuse` component fires on its own a `titleChanged` event once the displayed target has the `title` property defined. It is easier for an application to react to a `titleChanged` event if any `titleChanged` event(s) fired in the nested component(s) can be propagated to the router in the root component. To enable this, the property `propagateTitle` can be set in two ways:

- in the `target` object of a route to enable the title propagation for this `Component` target.
- in the `config` section of the routing configuration to enable the title propagation for all "Component" targets.

If `propagateTitle` is not set, no `titleChanged` event will be propagated from the nested component.

```
{
    "sap.ui5": {
        "componentUsages": {
            "myreuse": {
                "name": "reuse.component",
                "settings": {},
                "componentData": {},
                "lazy": false
            }
        },
        "routing": {
            "config": {
                ...
            },
            "routes": [{
                "name": "home",
                "pattern": "",
                "target": {
                    "name": "attachment",
                    "prefix": "atch",
            "propagateTitle": true
                }
            }],
            "targets": {
                "attachment": {
                    "type": "Component",
                    "usage": "myreuse",
                    "options": {
                        // optional
                        // define the additional parameter for
                        // instatiating the component instance
                    },
                    "containerOptions": {
                        // optional
                        // define the additional parameter for
                        // instantiating the component container
                        // which enables the component to be rendered
                        // in the parent control
                    },
                    "controlId": "page",
                    "controlAggregation": "content"
                }
        }
    }
}
```

The existing `titleChanged` event is extended with the following properties:

- `propagated`: whether the event is propagated from the router of a nested component
- `nestedHistory`: an array which contains the title and title history information of both the current router and the routers of the nested component(s). An application doesn't need to merge `nestedHistory` with the existing history parameter, because `nestedHistory` also contains the title history of the current router. Each element in the array has the following properties:
  - `ownerComponentId`: the ID of the component whose router fired the event. The router instance of this component can be retrieved by using the property `sap.ui.getCore().getComponent(sOwnerComponentId).getRouter()` which can be used for applying the `hash` of one title history entry to the browser. See the `hash` property for more information.
  - `history`: an array which contains the previous titles fired on the router. If the current event is fired on this router directly, the array contains the current title information as well, so that the application doesn't need to consider the existing `title` property of the event anymore. Each element in the array contains the following properties:
    - `title`: the title
    - `hash`: the browser hash part that belongs to this router when the event was fired. When only one component is created in the application, the entire browser hash can be used by the router of this component. Some applications use the global hash changer:

      ```
      HashChanger.getInstance().setHash(sHash) // HashChanger is required
      from sap/ui/core/routing/HashChanger
      ```

      to apply the hash to the browser. However, with nested components all component instances share the browser hash. The global hash changer then can't be used anymore, because it overwrites the entire browser hash without considering the other components. Instead, the application can parse the hash by using the method `getRouteInfoByHash` and navigate to the route by using the method `navTo`:

      ```
      var oRouter =
      sap.ui.getCore().getComponent(sOwnerComponentId).getRouter();
      var oRouteInfo = oRouter.getRouteInfoByHash(sHash);
      if (oRouteInfo) {
        oRouter.navTo(oRouteInfo.name, oRouteInfo.arguments);
      }
      ```

    - `isHome`: whether the title was changed from the home route

Navigate with Nested Components

The `navTo` method in the `sap.ui.core.routing.Router` class enables you to define a set of parameters to navigate to a specific route.

To use the `navTo` method for navigation with nested components, you need to call the method with the following information:

- Name of the route
- Parameters for the route
- Target information for the route name and the parameters in the nested components (optional)

- Information, whether the current browser hash is replaced or a or a new hash entry is set (optional)

For more information, `sap.ui.core.routing.Router.navTo` in the API Reference.

The call triggers the following actions in the given order:

1. For the new hash, the variable placeholders in the route's pattern are replaced with the given parameters. If the method is called with information for a router in nested components, the targets with type `Component` are loaded to compose the hash parts of these `Component` targets.
2. The new hash is set to the browser.
3. The browser fires a `hashchange` event.
4. The router processes the event and propagates the event along the hierarchy which was built while loading the nested components.
5. Each router checks its own hash part and informs the matched route. The matched route displays the targets which are configured for this route.
6. Each targets loads its `View` or `Component` and adds it to the configured `controlAggregation` of the `controlId` container.
7. The router fires a `routeMatched` event and the route fires a `matched` event to inform the application that the hash change is completed.

Using `navTo` for Passing Information to a Nested Router

For passing information about the route name and parameters for a nested router, you use the `oComponentTargetInfo` parameter of the `navTo` method. By this, the router in nested components can show the targets which are configured to one specific route instead of giving the router an empty hash as default. This `oComponentTargetInfo` parameter contains key-value pairs with the name of a `Component` target as the key, and the value must be an object which has at least the route name in the `route` property. The route name should be matched within the router of this component with the parameters for this route. If this route has again `Component` targets, the property `componentTargetInfo` can be used to specify the route information. The value of the `componentTargetInfo` property has the same structure as the `oComponentTargetInfo` parameter of the `navTo` method.

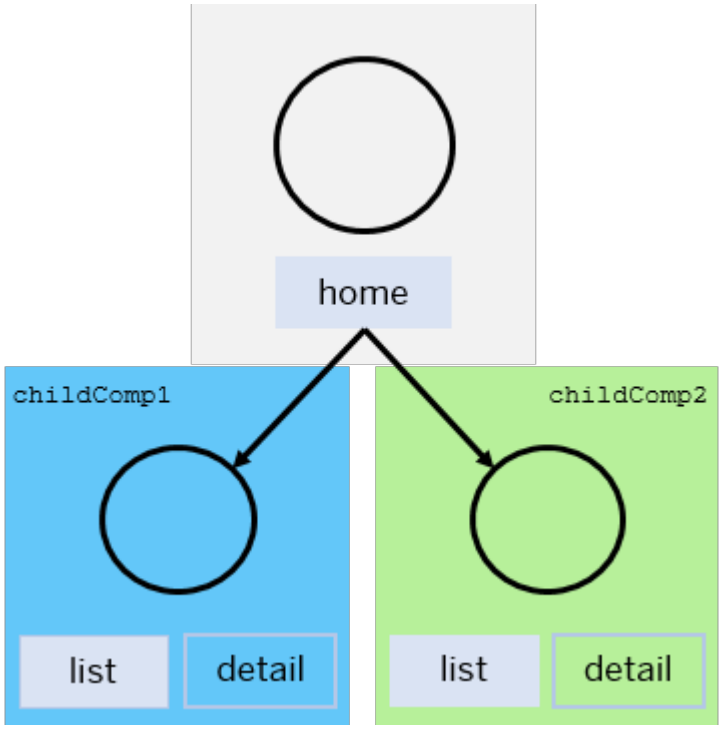
The following example shows a top level router with a "home" route with two `Component` targets:

- `Component` target `childComp1` with the following two defined routes:
  - Route `list`: Has an empty string hash as pattern and shows a list of items
  - Route `detail`: Shows the details for an item
- `Component` target `childComp2` with the following two defined routes:
  - Route `list`: Has an empty string hash as pattern and shows a list of items
  - Route `detail`: Shows the details for an item which displays again a nested `Component` target `grandChildComp1`

The `grandChildComp1` target has the following two routes defined:

- Route `list`: Has an empty string hash as pattern and shows a list of items
- Route `detail`: Shows the details for an item

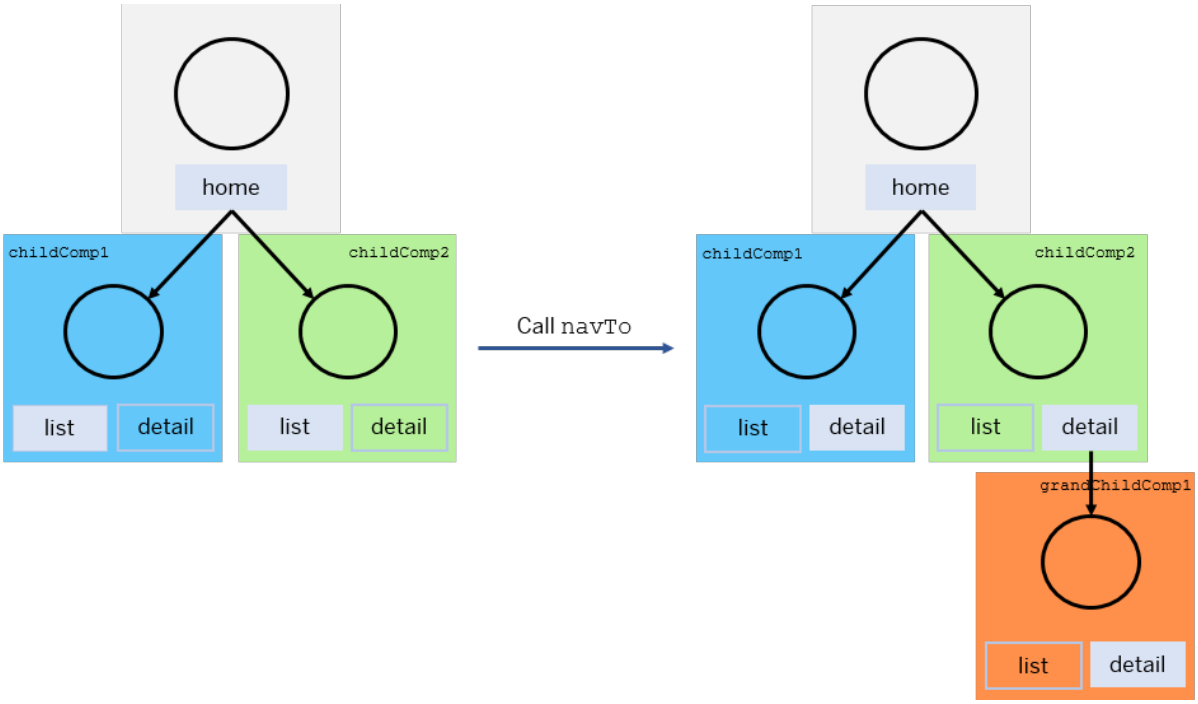
When the `home` route in the top level router is matched, the `Component` targets `childComp1` and `childComp2` are loaded and shown. Each of them receives an empty string hash as default, and so the `list` routes of their routers are matched.



By using the `navTo` method, specific route information can be given to multiple nested components and, if available, their deep nested components. For example, the `detail` routes in both `Component` targets `childComp1` and `childComp2` need to be matched. Since the `detail` route of target `childComp2` loads another nested component (`grandChildComp1`), it is also possible to match the `detail` route in the deep nested component `grandChildComp1` with the same `navTo` call, see the following code snippet.

```
oRouter.navTo("home", {
    // this route doesn't need any parameter
}, {
    childComp1: {
        route: "detail",
        parameters: {
            ...
        }
    },
    childComp2: {
        route: "detail",
        parameters: {
            ...
        },
        componentTargetInfo: {
            grandChildComp1: {
                route: "detail",
                parameters: {
                    ...
                }
            }
        }
    }
});
```

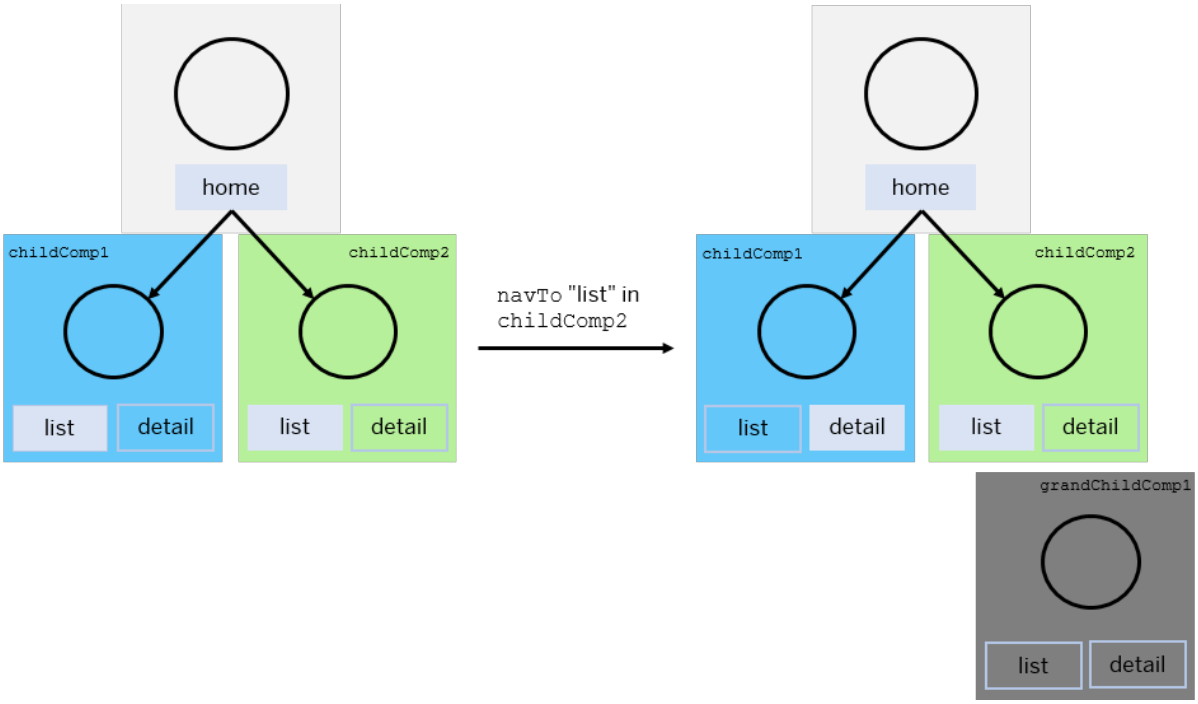
After the `navTo` call, the route state of each router looks as depicted in the following figure:



Navigating Away From a Nested Component

When a new route is matched within a router and a `Component` target was displayed within the old route, it is necessary to avoid that this `Component` target still reacts to unnecessary events such as `hashChanged`. For example, after switching from the `detail` route to the `list` route within the `Component` target `childComp2`,

the deep nested `Component` target `grandChildComp1` is no longer relevant for the UI. This is shown in the following figure:



To avoid this,

- the hash part is removed from the browser hash.
- the router is stopped, so that it no longer reacts to the `hashChanged` event.

Modules and Dependencies

The SAPUI5 framework has built-in support for modularizing comprehensive JavaScript applications. That means, instead of defining and loading one large bundle of JavaScript code, an application can be split into smaller parts which then can be loaded at runtime at the time when they are needed. These smaller individual files are called modules.

A module is a JavaScript file that can be loaded and executed in a browser. There are no rules or definitions what code belongs to a module, and what code does **not**. The content bundled in a module is up to the developer, but typically the content has a common topic, such as forming a JavaScript class or namespace or the contained functions address a specific topic, for example client to server communication or mathematical functions.

Modules have no predefined syntax or structure, but module developers can use the following features:

- **Name**
  The name indentifies the module in connection with the `sap.ui.define` and `sap.ui.require` syntax. A module defined under a certain name can be required using the same name.
- **Definition**
  Modules have a predefined structure based on the function `sap.ui.define`.
  The `sap.ui.define` syntax for defining modules helps to ensure an asynchronous loading of resources.

For more information, see the API Reference: `sap.ui.define`.

- **Dependencies**
  Every module defines a list of dependencies that have to be resolved before the callback function for the module is executed.
  The dependency declarations can be evaluated at runtime, but can also be analyzed at build time or at deploy time on the server.

Example

The following code snippet shows a typical module that uses all of features listed above. The name of the module is `someClass`:

```
  sap.ui.define("SomeClass", ['sap/mylib/Helper', 'sap/m/Bar'], function(Helper,
Bar) {

    // create a new class
    var SomeClass = function () {};

    // add methods to its prototype
    SomeClass.prototype.foo = function () {

        // use a function from the dependency 'Helper' in the same package (e.g.
'sap/mylib/Helper' )
        var mSettings = Helper.foo();

        // create and return an sap.m.Bar (using its local name 'Bar')
        return new Bar(mSettings);
    };
    // return the class as module value
    return SomeClass;

  });


// later requiring the previously defined module
sap.ui.require(['SomeClass'], function(SomeClass) {
    var oInstance = new SomeClass();
});
```

Static and Dynamic Dependencies

Adding each and every dependency to the `sap.ui.define` call can lead to many modules that have to be loaded before your module can be executed. Often, dependencies are not needed initially when the module is started. For rarely or not immediately used references, it might be overhead to load them in advance before executing your module.

Therefore, you have to decide whether you want to use static or dynamic dependencies:

- **Static** dependencies are loaded in the dependency declaration array of the `sap.ui.define` call. These dependencies are always loaded in advance before executing the defined module:

  ```
  sap.ui.define(['sap/m/Input'], function(Input) {
  ```

```
    // callback is executed once all dependencies are loaded
    ...
});
```

- **Dynamic** dependencies are resolved on demand after the initial module execution, as they are not needed for the initialisation of the module and are often tied to either a conditional or a user interaction.
  Dynamic dependencies should always be loaded asynchronously via `sap.ui.require`. The use of `jQuery.sap.require` is synchronous and considered as "bad practice" because `syncXHR` is deprecated by the Web Hypertext Application Technology Working Group (WHATWG).
  When dynamically requiring modules, the callback function will be called once all referenced modules (and their dependencies) are fully loaded:

```
sap.ui.define(['sap/m/Input'], function(Input) {

    var MyControl = ...;

    MyControl.prototype.onSavePress = function () {
        // dynamically load a dialog once it is needed
        sap.ui.require(['sap/m/Dialog'], function(Dialog) {
            var oDialog = new Dialog(...);
            oDialog.open(...);
        });
    };

    return MyControl;
});
```

> **i** Note
>
> Many code samples in the SAPUI5 documentation use the `sap.ui.require` syntax even though we could also have used `sap.ui.define`.

Loading a Module

For loading (requiring) a module, SAPUI5, you use the `sap.ui.require` function, which takes over the dependency resolution for you.

You can either load modules asynchronously or synchonously.

Asynchronous Loading

If the arguments of the `sap.ui.require` call consist of an array of one or more strings (module names) and an optional callback function, the string array is interpreted as a list of dependent modules.

The corresponding modules are loaded and the callback function is called asynchronously once all required modules are loaded.

```
// the callback function will be executed once the JSONModel, and the
UIComponent modules are loaded
sap.ui.require(['sap/ui/model/json/JSONModel', 'sap/ui/core/UIComponent'],
function(JSONModel, UIComponent) {
```

```
    var MyComponent = UIComponent.extend('MyComponent', {
      ...
    });
    ...
});
```

> ⚠ Caution
>
> If necessary, you can load a module synchronously. Be aware, that synchronous requests are already deprecated in some modern browsers and may not be supported in future. It is a better practice to load modules asynchronously.

Synchronous Retrieval of a Single Module Value

When calling `sap.ui.require` with a single string as argument, the respective module has to be loaded already.

If the module is not yet loaded or it is not a SAPUI5 module (third-party module), the return value is `undefined`.

By using `sap.ui.require`, you can synchronously access modules without triggering a loading request in case the module is not present.

```
// If JSONModel class is loaded, it is returned. If the module is not loaded
yet, there will be no additional loading request.
// The variable JSONModel might be undefined after making this call.
var JSONModel = sap.ui.require("sap/ui/model/json/JSONModel");
```

Loading Dependencies

You can load dependencies at different points in time.

Constructor and `init`

If a module is needed during the constructor call or initialization of a class, you declare the dependency as a static dependency in the `sap.ui.define` call.

If the dependency is required in the constructor, the instantiation is of course delayed until the dependency is loaded.

User interaction

Some modules can be required dynamically on user interaction. An example could be a dialog, which is not needed in most cases, but needs to be loaded only in case the user performs a certain interaction.

Other modules might be required dynamically while a data request is running to minimize the overall load time, as the user has to wait on the data anyway.

Checking the Availability of Modules

The `sap.ui.require` function can not only be used to load modules, but also to check the availability of modules.

The return value of the following function call is either a reference on the already loaded module or `undefined`. If `undefined` is returned, the module was not loaded yet and the `sap.ui.require` call without a callback function will not trigger a load.

```
var ModuleInQuestion = sap.ui.require("name/of/module/in/Question");
```

`instanceof` Checks

Since the above `sap.ui.require` call retrieves a module reference, you can use the reference not only to instantiate instances of classes but also to perform JavaScript instanceof checks.

```
sap.ui.define(['sap/ui/core/mvc/View', 'sap/ui/core/Fragment'], function(View,
Fragment) {
    ...
    if (oControl instanceof View) {
        ...
    } else if (oControl instanceof Fragment) {
        ...
    }
});
```

`instanceof` Checks for Dynamically Required Modules

You can use the following approach to make sure your `instanceof` check is valid without the need to actually load the module.

To perform an `instanceof` check, the respective class does not need to be loaded. If the class module is not loaded, there can never be an instance of that class. The `sap.ui.require` call returns `undefined` in case the module is not loaded.

The `lazyInstanceOf` convenience function makes sure that the `instanceof` check is performed against a function and not `undefined`, in case the module or class was not loaded yet.

```
function lazyInstanceof(obj, module) {
    var FNClass = sap.ui.require(module);
    return typeof FNClass === 'function' ? obj instanceof FNClass : false;
}

if (lazyInstanceof(oControl, "sap/ui/core/mvc/View")) {
                …
} else if (lazyInstanceof(oControl, "sap/ui/core/Fragment")) {
                …
}
```

Multiple Module Locations

SAPUI5 supports multiple module locations by means of the `sap.ui.loader.config` function.

In web applications, modules can be located in different locations, such as servers and web apps. A web application can, for example, be deployed as an individual web app and contain modules that have to be loaded at runtime. SAPUI5 and its modules, however, have to be loaded either from a content delivery network (CDN) or from a centrally deployed web app. By default, SAPUI5 loads modules from its resource root URL, that is, from the centrally deployed web application. This would fail for modules that are contained in your web application.

The `sap.ui.loader.config` function associates a module name prefix with a URL prefix. All modules are loaded from the registered URL instead of the standard resource root URL:

```
sap.ui.loader.config({
    paths: {
        'my/module': 'https://example.com/resources/my/module'
    }
});
```

Thus, it is possible to redirect the request for the application-specific modules to the corresponding web application:

```
    <script src="https://openui5.hana.ondemand.com/resources/sap-ui-core.js" ></
script>
    <script>
        // redirect the 'my.webapp' package to the local web app
        sap.ui.loader.config({
            paths:{
                "my/webapp": "my-webapp/resources/my/webapp"
            }
        });

        sap.ui.require([
            'sap/ui/core/Core',
            'my/webapp/MyModule01’    // loads /my-webapp/resources/my/webapp/
MyModule01.js
        ], function ( Core, MyModule01 ) {
            //[…] use modules
        }
    </script>
```

i Note

The registered URL above contains the transformed module name prefix `my/webapp/`. This allows a more flexible packaging of the modules, for example, if you decide to deploy all modules named `my.company.*` to the central URL `http://my.company/shared/` without packaging them into a two level hierarchy of subfolders:

```
sap.ui.loader.config({
     paths:{
           "my/company": "http://my.company/shared/"
      }
});
```

However, when the standard build tools of the SAPUI5 framework are used, the full package name will be part of the runtime file hierarchy and the registration must contain the transformed package hierarchy as above.

Best Practices for Loading Modules

This section provides best practices for SAPUI5 module loading patterns.

These best practices are especially important when you switch from the synchronuous variant of the SAPUI5 module loader to the asynchronous variant. Patterns that may have worked in synchronuous module loading may lead to applications that cannot start in asynchonuous module loading.

> **i Note**
>
> Use the `async` configuration parameter to enable asynchronous module loading via the bootstrap. For more information, see Standard Variant for Bootstrapping [page 694].

How to Define Modules

Every SAPUI5 module file must contain exactly one unnamed module definition on the top level: the `sap.ui.define` call. Also, to avoid side-effects, all module-related functions must be defined within the callback function.

Example: The following two modules are unnamed. They only contain one top-level `sap.ui.define` and can be addressed with the respective unique module name:

`myLib/MyModuleA.js`

```
sap.ui.define(function(){
    ...
});
...
```

For troubleshooting information with regard to loading your module, see Why is my Module Not Loading? [page 1101].

How to Address Modules

A module must always be addressed with the unique module name. The module name is case-sensitive.

`myLib/MyModuleB.js`

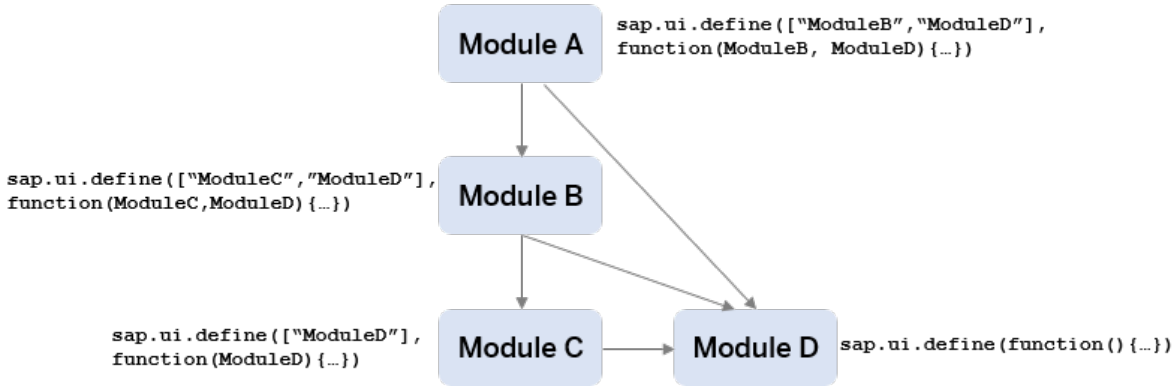
```
sap.ui.define(["myLib/MyModuleA"], function(MyModuleA){
    ...
});
```

For troubleshooting information with regard to addressing modules, see What is wrong with the way I am addressing the modules? [page 1105].

How to Structure a Project

The entry point of an SAPUI5 application is often a module that is used to instatiate a SAPUI5 component. This central module is considered as single node of a graph and all dependent modules as well as their dependencies are nodes which must be connected by directed edges: the graph must fulfill the requirements of a directed acyclic graph (DAG).

Example: All modules are evaluated in a clearly defined order. The evaluation starts with module D, then module C and module B, and ends with module A.



For troubleshooting information with regard to the project structure, see How can I remove project structures with cyclic dependencies? [page 1106].

Troubleshooting for Loading Modules

The following sections give examples that you must avoid because they cause problems when loading your module.

Why was my Module Not Loaded Correctly?

The following list contains possible reasons why your module does not load. To see how it is done correctly, see Best Practices for Loading Modules - How to Define Modules [page 1100].

The module name is given in `sap.ui.define`

If you explicitly give the module name in `sap.ui.define`, you introduce additional complexity to the project structure which may cause inconcistencies and clashing module names. This problem is difficult to detect and can easily and proactively be avoided by omitting the module name in `sap.ui.define`.

The following example shows how it must *not* be done: The library file structure of `myLib` does not fit the module name. If there is another module named `MyModule` in the `myLib` library, the module would be hard to

address. If you use an unnamed module instead, the module names would reflect the library file structure. By this, you reduce the probability of module name conflicts. In general, when addressing UI5 modules, make sure you separate all parts of the module's name with slashes instead of dots, for example `myLib/MyModule` instead of `myLib.MyModule`.

`myLib/myAdditionalPathSegment/MyModule.js`

```
// CAUTION: BAD EXAMPLE - DON'T DO THIS
sap.ui.define("myLib.MyModule", [], function(){
    ...
});
```

Using multiple `sap.ui.define` calls with unnamed modules

If you have more than one `sap.ui.define` call in a JavaScript file, the module loader does not know which definition actually represents the module. As there is no scenario that requires multiple module definitions in one file and in order to comply with the AMD specification (see https://github.com/amdjs/amdjs-api/wiki/AMD), the async variant of the SAPUI5 module loader does *not* tolerate multiple definitions anymore and throws an error.

Example: The `myModule` module is defined twice. This was most probably done by accident. To resolve this, the two module definitions have to be split into two separate modules.

`myModule.js`

```
// BAD EXAMPLE - DON'T DO THIS
sap.ui.define([], function(){
    ...
});
sap.ui.define([], function(){
    ...
});
```

Combining conditional modules with `sap.ui.define`

Conditional module definitions should *not* be used because of the following reasons:

- The modules cannot be required with parameters because the check conditions are related to globals.
- The export value is not consistent. This makes it difficult to consume the module.
- The module dependencies are unclear. This prevents an efficient module bundling.

Example: The export value of `myModule` depends on the global `myProperty` property. In this case, it makes sense to split the two definitions into separate files for example into the two variants `myModuleA` and `myModuleB`. Another module can then make the required `myProperty` check and require the variant of `myModule` via `sap.ui.require`.

`myModule.js`

```
// BAD EXAMPLE - DON'T DO THIS
if (myProperty){
    sap.ui.define([], function(){
        ...
    });
else {
    sap.ui.define([], function(){
        ...
    });
}
```

Mixing old and new loader APIs

Using deprecated APIs is not recommended and mixing old and new loader APIs is even worse: If the synchronicity has changed between older and newer APIs, mixing them will cause timing-related issues as well as general inconsistencies.

Example: The namespace `myLib.myModule` is registered through the `jQuery.sap.declare` call. Besides actually defining the module export value, the subsequent `sap.ui.define` call does the same registration. So, the `jQuery.sap.declare` in this example is unnecessary and must be omitted in this example.

`myLib/MyModule.js`

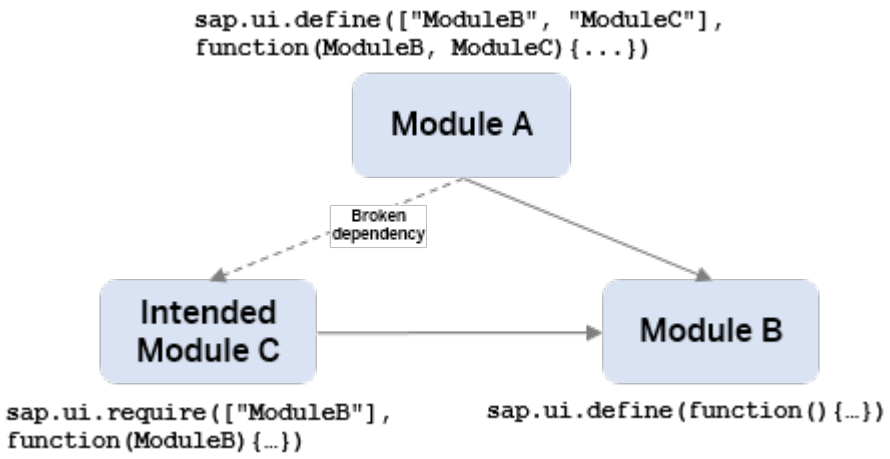
```
// BAD EXAMPLE - DON'T DO THIS
jQuery.sap.declare("myLib.myModule");
sap.ui.define([], function(){
    ...
});
```

Using `sap.ui.require` instead of `sap.ui.define` on the top level

Although the API signature for `sap.ui.define` and `sap.ui.require` looks similar, you must use the `sap.ui.define` API to define a reusable JavaScript object (that is, a module). Note the following differences between `sap.ui.require` and `sap.ui.define`:

Action	sap.ui.require	sap.ui.define
Value export	Not possible	The callback function defines an export to provide functionality to other modules.
Module name registration	Not possible	The module name is registered at the loader registry and can be used to address the module.
Relative dependencies	This is not possible, because no module name is registered and a reference point is missing.	Can be used.
Execution order	Dependent modules can be executed before the `sap.ui.require` callback has been executed. Therefore, using `sap.ui.require` instead of `sap.ui.define` can break the intended dependency graph and module execution order.	The dependent modules are waiting for the module callback execution to be finished.

Example: The file for module C has one top-level `sap.ui.require` instead of a top-level `sap.ui.define` call. The module callback evaluation order starts with module B, because it has no dependencies. Afterwards, the framework can execute module A or module C, because the intended module C is not a module from the module loader perspective. Furthermore, the undefined export value of module C will most probably lead to errors in module A. If module C is defined correctly via a top-level `sap.ui.define` call, the module callback execution order is clear: B - C - A.



Defining (unnamed) modules via inline scripts

It is unclear how modules that are defined via inline scripts can by addresses by other modules. Therefore, the inline scripts must be omitted.

Example: Module A is defined after bootstrapping UI5 and before the actual app is started. As the module is not addressable, the module definition must be moved to a separate file.

`startMyApp.html`

```
<!-- BAD EXAMPLE - DON'T DO THIS -->
<html>
...
    <script>
        //Boot UI5
    </script>
    <script>
        //Definition for Module A
        sap.ui.define(function(){
            ...
        });
    </script>
    <script>
        // Start UI5 Application
    </script>
...
</html>
```

Avoiding synchronous access to a module definition export

Never do a synchronous access to the export of a module definition because the module definition could be done asynchronously. Never rely on the synchronicity of a module definition, even if a module has no dependencies.

Example: The `sap.ui.define` call for the `myModule` module is made and the export value is synchronously used by creating a new object of that export. Although this may work in some scenarios, never do it this way, because it is unclear whether the module definition is already done. Instead, use the export of `myModule` in a separate module with a correctly maintained dependency to the `myModule` module.

`myLib/MyModule.js`

```
// BAD EXAMPLE - DON'T DO THIS
sap.ui.define([], function(){
```

```
    ...
});
...
var oMyModule = new myLib.MyModule();
...
```

For more information, see the API Reference for sap.ui.define - Asynchronuous Contract.

Avoid synchronous probing after module definition

Similar to the synchronous access of a module's export value, you also must omit the synchronous probing for modules defined in the same browser task.

Example: The `sap.ui.define` call for the `myModule` module is made and is synchronously checked by probing through calling `sap.ui.require`. Instead, the probing for `myModule` must be done in a separate module with a correctly maintained dependency to `myModule`.

`myLib/MyModule.js`

```
// BAD EXAMPLE - DON'T DO THIS
sap.ui.define([], function(){
    ...
});
var MyModule = sap.ui.require('myLib/MyModule');
```

What is wrong with the way I am addressing the modules?

The following examples show how you should *not* address a module. To see how it is done correctly, see Best Practices for Loading Modules - How to Address Modules [page 1100].

Case insensitivity when addressing modules

Addressing a module inconsistently can cause various side-effects. If the server is not case sensitive, for example, the same resource can be addressed with URLs that differ only in case sensitivity. Besides that, it is bad from a performance perspective if the same resource is loaded twice and the same module is defined twice. This is similar to the example for multiple definitions above: multiple definitions of the same module can cause several issues, such as failing checks of `instanceof`.

Example: If we assume a server that is *not* case-sensitive, the `sap.m` library's `Button` control is loaded and evaluated twice.

`myView.xml`

```
<!-- BAD EXAMPLE - DON'T DO THIS -->
<mvc:View xmlns:mvc="sap.ui.core.mvc" xmlns:m="sap.m">
    ...
    <m:Button></m:Button>
    <m:button></m:button>
    ...
</mvc:View>
```

`myModule.js`

```
...
    // BAD EXAMPLE - DON'T DO THIS
    sap.ui.require(['sap/m/button'], function(){
```

```
        ...
    });
    sap.ui.require(['sap/m/Button'], function(){
        ...
    });
...
```

Manual loading of UI5 modules via script tags

When you load modules manually, the module loader cannot know how the module shall be named. Therefore, UI5 modules must always be loaded and evaluated via the UI5 module loader APIs.

Example: The `myModule` module is loaded via a script tag. Instead, use a `sap.ui.require` call to loading the module.

`startMyApp.js`

```
<html>
...
    <script src="https://myhost/mypath/myModule.js"></script>
...
</html>
```

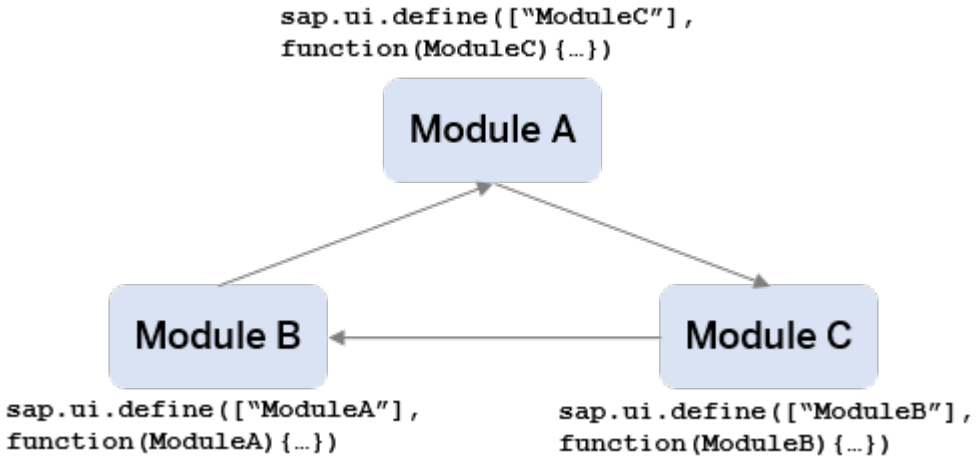
How can I remove project structures with cyclic dependencies?

When you use cyclic dependencies in the project structure, the module dependencies cannot be resolved. The UI5 module loade detects the cycle and returns an undefined value instead of the correct module export.

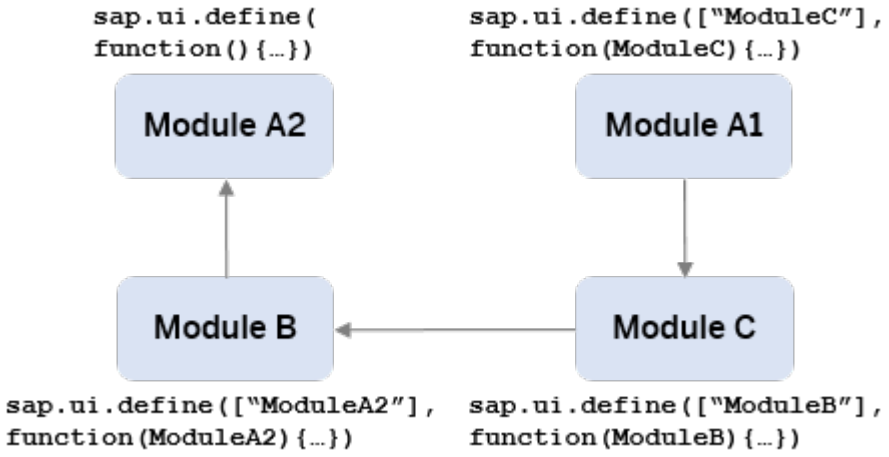
As an exception, in specific scenarios, you may make the involved modules robust enough to handle undefined module exports at module callback execution time and use the export value via probing later. However, if you use the async variant of the loader, all modules that belong to a cycle must be able to handle undefined exports.

To see how to set up a correct project structure, see Best Practices for Loading Modules - How to Structure a Project [page 1101].

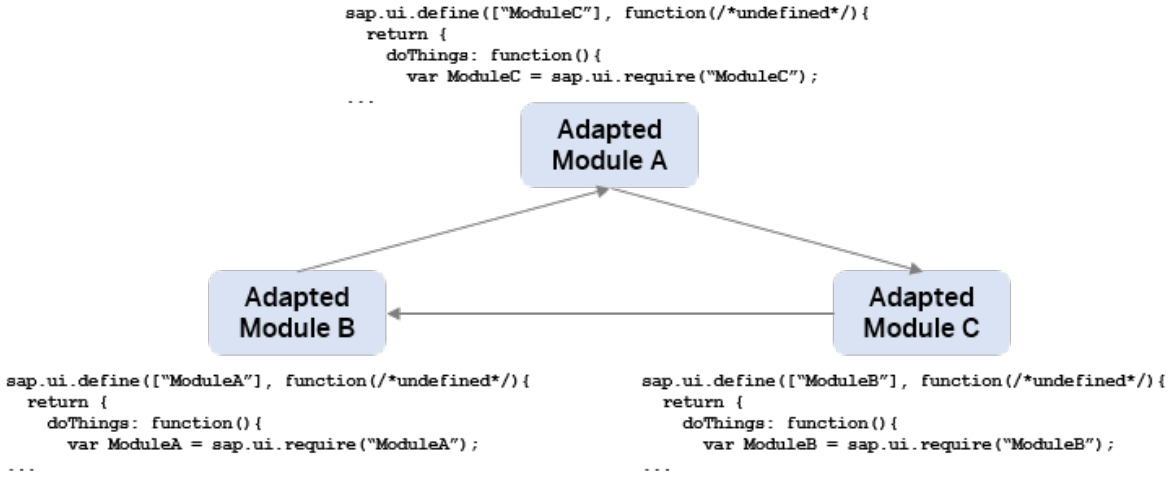
Example: All modules have exactly one dependency, which cannot be resolved correctly.



Solution 1 – Resolved cycle: The following figure shows how the cycle can be resolved by moving the functionality of module A, which is used by module B, to a separate module (module A2). In general, resolving cyclic dependencies can require a larger refactoring of all involved modules, especially when multiple cycles have to be resolved.



Solution 2 – Probing modules: In the example given in the following figure, the cycle is not resolved, but the involved modules do not access the dependent modules directly when the module callback is executed. They access them later via probing.



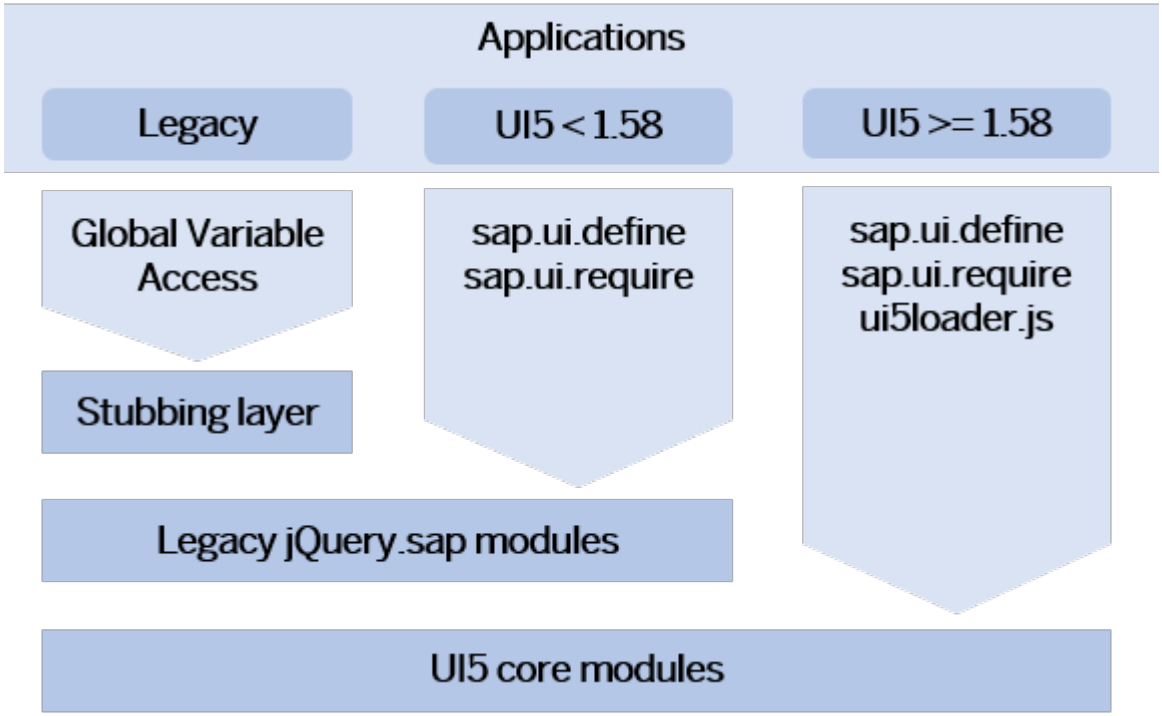
Adapting to the Modularization of the Core

Small, predefined modules for specific purposes, providing standalone functionality can be used any time SAPUI5 is loaded.

The modules are either Browser-dependent (`sap/ui/core`) and use the DOM or any other Browser-native API, or not Browser-dependent (`sap/base`) and could run in `node.js` without DOM access. Note that `node.js` is not an officially supported environment.

Compatibility With Existing Modules

The modules are introduced with SAPUI5 version 1.58 and replace the existing larger core modules to make the code easier to understand and maintain, and to decrease the initial payload of SAPUI5. To avoid that the removal of dependencies caused by the switch to the new modules causes exceptions, a lazy loading of the legacy modules is provided. For compatibility reasons, this lazy loading is done synchronously and it provides just the API namespace without loading the actual implementation.



As it may not be obvious where those calls occur or where a dependency is missing, a rule in the Support Assistant reports the use and provides guidance on how to avoid them. A second rule with lower priority reports the use of an `jQuery.sap` API in general. There are also log warnings in the console of the browser's development tools, including a stack trace which makes it easy to locate the call in your coding.

Migration

To benefit from the improvements provided by the modules, perform the following steps:

- Always declare the full dependencies as described in Loading a Module [page 1096].
- Migrate to the new module API as described in Legacy jQuery.sap Replacement [page 1109]. Do **not** use the global `jQuery.sap` API anymore.
- Do **not** use the global `sap.ui` factory functions. Instead, use their replacements, see Legacy Factories Replacement [page 1124].

Legacy jQuery.sap Replacement

Overview of the mapping of legacy APIs to the new APIs for the migration

The deprecation of the `jQuery.sap` API requires that it is replaced with the new API. The following list provides an overview of the required replacements.

Replacement With New Modules

To migrate the simple replacements, add the new module dependency and replace the call with the added argument name as shown in the following example:

Example

Old API

```
sap.ui.define([], function() {
 jQuery.sap.log.info("My log message");
});
```

New API

```
sap.ui.define(['sap/base/Log'], function(Log) {
 Log.info("My log message");
});
```

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.assert`	`sap/base/assert`	Simple replacement	`assert`
`jQuery.sap.resources`	`sap/base/i18n/ResourceBundle`	Method changed	`ResourceBundle.create`
`jQuery.sap.log`	`sap/base/Log`	Simple replacement	`Log`
`jQuery.sap.log.addLogListener`	`sap/base/Log`	Simple replacement	`Log.addLogListener`
`jQuery.sap.log.debug`	`sap/base/Log`	Simple replacement	`Log.debug`
`jQuery.sap.log.error`	`sap/base/Log`	Simple replacement	`Log.error`
`jQuery.sap.log.fatal`	`sap/base/Log`	Simple replacement	`Log.fatal`
`jQuery.sap.log.getLevel`	`sap/base/Log`	Simple replacement	`Log.getLevel`

Old API Call	New Module	Replacement Type	Replace with
jQuery.sap.log.getLog	sap/base/Log	Method changed	Log.getLogEntries
jQuery.sap.log.getLogEntries	sap/base/Log	Simple replacement	Log.getLogEntries
jQuery.sap.log.getLogger	sap/base/Log	Simple replacement	Log.getLogger
jQuery.sap.log.info	sap/base/Log	Simple replacement	Log.info
jQuery.sap.log.isLoggable	sap/base/Log	Simple replacement	Log.isLoggable
jQuery.sap.log.Level	sap/base/Log	Simple replacement	Log.Level
jQuery.sap.log.logSupportInfo	sap/base/Log	Simple replacement	Log.logSupportInfo
jQuery.sap.log.removeLogListener	sap/base/Log	Simple replacement	Log.removeLogListener
jQuery.sap.log.trace	sap/base/Log	Simple replacement	Log.trace
jQuery.sap.log.warning	sap/base/Log	Simple replacement	Log.warning
jQuery.sap.encodeCSS	sap/base/security/encodeCSS	Simple replacement	encodeCSS
jQuery.sap.encodeJS	sap/base/security/encodeJS	Simple replacement	encodeJS
jQuery.sap.encodeURL	sap/base/security/encodeURL	Simple replacement	encodeURL
jQuery.sap.encodeURLParameters	sap/base/security/encodeURLParameters	Simple replacement	encodeURLParameters
jQuery.sap.encodeHTML	sap/base/security/encodeXML	Simple replacement	encodeXML
jQuery.sap.encodeXML	sap/base/security/encodeXML	Simple replacement	encodeXML

Old API Call	New Module	Replacement Type	Replace with
jQuery.sap.addUrlWhitelist	sap/base/security/URLWhiteList	Method changed	URLWhitelist.add
jQuery.sap.clearUrlWhitelist	sap/base/security/URLWhiteList	Method changed	URLWhitelist.clear
jQuery.sap.getUrlWhitelist	sap/base/security/URLWhiteList	Method changed	URLWhitelist.entries
jQuery.sap.removeUrlWhitelist	sap/base/security/URLWhiteList	Method changed	URLWhitelist.delete
jQuery.sap.validateUrl	sap/base/security/URLWhiteList	Method changed	URLWhitelist.validate
jQuery.sap.camelCase	sap/base/strings/camelize	Simple replacement	camelize
jQuery.sap.charToUpperCase	sap/base/strings/capitalize	Simple replacement	capitalize
jQuery.sap.escapeRegExp	sap/base/strings/escapeRegExp	Simple replacement	escapeRegExp
jQuery.sap.formatMessage	sap/base/strings/formatMessage	Simple replacement	formatMessage
jQuery.sap.hashCode	sap/base/strings/hash	Simple replacement	hash
jQuery.sap.hyphen	sap/base/strings/hyphenate	Simple replacement	hyphenate
jQuery.sap.unicode jQuery.sap.isStringNFC	sap/base/strings/NormalizePolyfill	Simple replacement	NormalizePolyfill
-	sap/base/strings/toHex	-	-
jQuery.sap.arraySymbolDiff	sap/base/util/array/diff	Simple replacement	diff
jQuery.sap.unique	sap/base/util/array/uniqueSort	Simple replacement	uniqueSort

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.equal`	`sap/base/util/deepEqual`	Simple replacement	`deepEqual`
-	`sap/base/util/defineLazyProperty`	-	-
`jQuery.sap.each`	`sap/base/util/each`	Simple replacement	`each`
`jQuery.sap.forIn`	`sap/base/util/each`	Simple replacement	`each`
-	`sap/base/util/includes`	-	-
`jQuery.isPlainObject`	`sap/base/util/isPlainObject`	Simple replacement	`isPlainObject`
`jQuery.sap.parseJS`	`sap/base/util/JSTokenizer`	Simple Replacement	`JSTokenizer.parseJS`

Old API Call	New Module	Replacement Type	Replace with

`jQuery.sap.extend`	`sap/base/util/merge`	Complex Replacement	Old:
			`// Shallow jQuery.sap.extend({}, sContent); // Deep jQuery.sap.extend(true, {}, sContent)`
			New:
			`// Shallow // // No actual replacement for shallow copies available, see the note below for more info. // Deep merge({}, sContent);`
			i Note
			`jQuery.sap.extend` **vs.** `jQuery.extend`
			The use of `jQuery.sap.extend()` is the same as `jQuery.extend()`, but arguments that are `null` or `undefined` are **not** ignored.
			`Object.assign`
			The `Object.assign()` method only copies enumerable and own properties, but does not copy properties on the prototype chain and non-enumerable properties.
			Considering this, `Object.assign()` might be a suitable replacement for

Old API Call	New Module	Replacement Type	Replace with
			`jQuery.sap.extend` for a shallow copy. `null` and `undefined` arguments are **not** ignored.
`jQuery.sap.now`	`sap/base/util/now`	Simple Replacement	`now`

`jQuery.sap.getObject`	`sap/base/util/ObjectPath`	Complex Replacement	`ObjectPath.get("some.object.path", "someProperty");`
			If the object path does not exist, the method doesn't create it anymore. If the path needs to be create it has do be done separately:
			`ObjectPath.create("some.object.path", window.myLib);`

`jQuery.sap.setObject`	`sap/base/util/ObjectPath`	Complex Replacement	`ObjectPath.set("some.object.path", "myValue", window.myLib);` The object path is created if it does not exist.
`jQuery.sap.properties`	`sap/base/util/Properties`	Method changed	`Properties.create`
`jQuery.sap.uid`	`sap/base/util/uid`	Simple Replacement	`uid`
`jQuery.sap.getUriParameters`	`sap/base/util/UriParameters`	Changed to class for instantiation	`var oUriParameters = new UriParameters(window.location.href);` `oUriParameters.get("sap-ui-debug");`
-	`sap/base/util/values`	Simple Replacement	-

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.Version`	`sap/base/util/Version`	Simple Replacement	`Version`
-	`sap/ui/core/support/HotKeys`	-	-
`jQuery.sap.syncStyleClass`	`sap/ui/core/syncStyleClass`	Simple Replacement	`syncStyleClass`
`jQuery.device.is.android_phone`	`sap/ui/Device`	Complex Replacement	`Device.os.android && Device.system.phone`
`jQuery.device.is.android_tablet`	`sap/ui/Device`	Complex Replacement	`Device.os.android && Device.system.tablet`
`jQuery.device.is.desktop`	`sap/ui/Device`	Complex Replacement	`Device.system.desktop`
`jQuery.device.is.ipad`	`sap/ui/Device`	Complex Replacement	`Device.os.ios && Device.system.ipad`
`jQuery.device.is.iphone`	`sap/ui/Device`	Complex Replacement	`Device.os.ios && Device.system.phone`
`jQuery.device.is.landscape`	`sap/ui/Device`	Complex Replacement	`Device.orientation.landscape`
`jQuery.device.is.phone`	`sap/ui/Device`	Complex Replacement	`Device.system.phone`
`jQuery.device.is.portrait`	`sap/ui/Device`	Complex Replacement	`Device.orientation.portrait`
`jQuery.device.is.tablet`	`sap/ui/Device`	Complex Replacement	`Device.system.tablet`
`jQuery.os.Android`	`sap/ui/Device`	Complex Replacement	`Device.os.name === "Android"`

Old API Call	New Module	Replacement Type	Replace with
jQuery.os.bb	sap/ui/Device	Complex Replacement	`Device.os.name === "bb"`
jQuery.os.fVersion	sap/ui/Device	Complex Replacement	`Device.os.version`
jQuery.os.iOS	sap/ui/Device	Complex Replacement	`Device.os.name === "iOS"`
jQuery.os.linux	sap/ui/Device	Complex Replacement	`Device.os.name === "linux"`
jQuery.os.mac	sap/ui/Device	Complex Replacement	`Device.os.name === "mac"`
jQuery.os.os	sap/ui/Device	Complex Replacement	`Device.os.name`
jQuery.os.version	sap/ui/Device	Complex Replacement	`Device.os.versionStr`
jQuery.os.win	sap/ui/Device	Complex Replacement	`Device.os.name === "win"`
jQuery.os.winphone	sap/ui/Device	Complex Replacement	`Device.os.name === "winphone"`
jQuery.sap.containsOrEquals	sap/ui/dom/containsOrEquals	Simple Replacement	containsOrEquals
jQuery.sap.denormalizeScrollBeginRTL	sap/ui/dom/denormalizeScrollBeginRTL	Simple Replacement	denormalizeScrollBeginRTL
jQuery.sap.denormalizeScrollLeftRTL	sap/ui/dom/denormalizeScrollLeftRTL	Simple Replacement	denormalizeScrollLeftRTL
-	sap/ui/dom/getComputedStyleFix	-	-
jQuery.sap.ownerWindow	sap/ui/dom/getOwnerWindow	Simple Replacement	getOwnerWindow

Old API Call	New Module	Replacement Type	Replace with
jQuery.sap.scrollbarSize	sap/ui/dom/getScrollbarSize	Simple Replacement	getScrollbarSize
jQuery.sap.includeScript	sap/ui/dom/includeScript	Simple Replacement	includeScript
jQuery.sap.includeStylesheet	sap/ui/dom/includeStylesheet	Simple Replacement	includeStylesheet
jQuery.sap.replaceDOM	sap/ui/dom/patch	Simple Replacement	patch
jQuery.sap.pxToRem	sap/ui/dom/units/Rem	Simple Replacement	Rem.fromPx
jQuery.sap.remToPx	sap/ui/dom/units/Rem	Simple Replacement	Rem.toPx
jQuery.sap.checkMouseEnterOrLeave	sap/ui/events/checkMouseEnterOrLeave	Simple Replacement	checkMouseEnterOrLeave
jQuery.sap.bindAnyEvent	sap/ui/events/ControlEvents	Simple Replacement	bindAnyEvent
jQuery.sap.ControlEvents	sap/ui/events/ControlEvents	Simple Replacement	events
jQuery.sap.unbindAnyEvent	sap/ui/events/ControlEvents	Simple Replacement	unbindAnyEvent
jQuery.sap.handleF6GroupNavigation	sap/ui/events/F6Navigation	Simple Replacement	handleF6GroupNavigation
jQuery.sap.isMouseEventDelayed	sap/ui/events/isMouseEventDelayed	Simple Replacement	isMouseEventDelayed
jQuery.sap.isSpecialKey	sap/ui/events/isSpecialKey	Simple Replacement	isSpecialKey
jQuery.sap.touchEventMode	sap/ui/events/jquery/EventSimulation	Simple Replacement	touchEventMode
jQuery.sap.keycodes	sap/ui/events/KeyCodes	Simple Replacement	KeyCodes

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.PseudoEvents`	`sap/ui/events/PseudoEvents`	Simple Replacement	`PseudoEvents`
`jQuery.sap.disableTouchToMouseHandling`	`sap/ui/events/TouchToMouseMapping`	Simple Replacement	`disableTouchToMouseHandling`
`jQuery.sap.registerModulePath`	-	Complex Replacement	`sap.ui.loader.config({paths: {"myPath": "some/path"}});`
`jQuery.sap.registerResourcePath`	-	Complex Replacement	`sap.ui.loader.config({paths: {"myPath": "some/path"}});`
`jQuery.sap.measure.add`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.add`
`jQuery.sap.measure.average`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.average`
`jQuery.sap.measure.clear`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.clear`
`jQuery.sap.measure.end`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.end`
`jQuery.sap.measure.filterMeasurements`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.filterMeasurements`
`jQuery.sap.measure.getActive`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.getActive`
`jQuery.sap.measure.getAllMeasurements`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.getAllMeasurements`
`jQuery.sap.measure.getMeasurement`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.getMeasurement`
`jQuery.sap.measure.pause`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.pause`
`jQuery.sap.measure.registerMethod`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.registerMethod`

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.measure.remove`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.remove`
`jQuery.sap.measure.resume`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.resume`
`jQuery.sap.measure.setActive`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.setActive`
`jQuery.sap.measure.start`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.start`
`jQuery.sap.measure.unregisterAllMethods`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.unregisterAllMethods`
`jQuery.sap.measure.unregisterMethod`	`sap/ui/performance/Measurement`	Simple Replacement	`Measurement.unregisterMethod`
`jQuery.sap.fesr.getActive`	`sap/ui/performance/trace/FESR`	Simple Replacement	`FESR.getActive`
`jQuery.sap.fesr.setActive`	`sap/ui/performance/trace/FESR`	Simple Replacement	`FESR.setActive`
`jQuery.sap.fesr.addBusyDuration`	`sap/ui/performance/trace/Interaction`	Simple Replacement	`Interaction.addBusyDuration`
`jQuery.sap.interaction.*`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.*`
`jQuery.sap.measure.clearInteractionMeasurements`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.clear`
`jQuery.sap.measure.endInteraction`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.end`
`jQuery.sap.measure.filterInteractionMeasurements`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.filter`
`jQuery.sap.measure.getAllInteractionMeasurements`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.getAll`

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.measure.getPendingInteractionMeasurement`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.getPending`
`jQuery.sap.measure.startInteraction`	`sap/ui/performance/trace/Interaction`	Method changed	`Interaction.start`
`jQuery.sap.fesr.getCurrentTransactionId`	`sap/ui/performance/trace/Passport`	Method changed	`Passport.getTransactionId`
`jQuery.sap.fesr.getRootId`	`sap/ui/performance/trace/Passport`	Method changed	`Passport.getRootId`
`jQuery.sap.passport.*`	`sap/ui/performance/trace/Passport`	Simple replacement	`Passport.*`
`jQuery.sap.getModulePath`	-	Complex replacement	`sap.ui.require.toUrl("some/path/to/module.js");`
`jQuery.sap.getResourcePath`	-	Complex replacement	`sap.ui.require.toUrl("some/path/to/resource.json");`
`jQuery.sap.FrameOptions`	`sap/ui/security/FrameOptions`	Simple replacement	`FrameOptions`
`jQuery.sap.act`	`sap/ui/util/ActivityDetection`	Simple replacement	`ActivityDetection`
`jQuery.sap.initMobile`	`sap/ui/util/Mobile`	Method changed	`Mobile.init`
`jQuery.sap.setIcons`	`sap/ui/util/Mobile`	Simple replacement	`Mobile.setIcons`
`jQuery.sap.setMobileWebAppCapable`	`sap/ui/util/Mobile`	Simple replacement	`Mobile.setWebAppCable`
`jQuery.sap.storage`	`sap/ui/util/Storage`	Method changed	`Storage`
`jQuery.sap.storage.Type.*`	`sap/ui/util/Storage`	Simple replacement	`Storage.Type`

Old API Call	New Module	Replacement Type	Replace with
`jQuery.sap.getParseError`	`sap/ui/util/XMLHelper`	Simple replacement	`Helper.getParseError`
`jQuery.sap.parseXML`	`sap/ui/util/XMLHelper`	Method changed	`Helper.parse`
`jQuery.sap.serializeXML`	`sap/ui/util/XMLHelper`	Method changed	`Helper.serialize`

Replacement with Native Browser APIs

Old API Call	New Native Replacement
`jQuery.device.is.standalone`	`window.navigator.standalone`
`jQuery.inArray`	`var b = (aElements ? Array.prototype.indexOf.call(aElements, 4) : -1);`
`jQuery.isArray`	`Array.isArray`
`jQuery.sap.clearDelayedCall`	`window.clearTimout`
`jQuery.sap.clearIntervalCall`	`window.clearInterval`
`jQuery.sap.delayedCall`	`window.setTimeout`
`jQuery.sap.domById`	`window.document.getElementById`
`jQuery.sap.endsWith`	`sMyString.endsWith("abc")`
`jQuery.sap.endsWithIgnoreCase`	`sMyString.toLowerCase().endsWith(sMyOtherString.toLowerCase())`
`jQuery.sap.getter`	`function(value) { return function() { return value; }; }(myValue);`
`jQuery.sap.intervalCall`	`window.setInterval`
`jQuery.sap.isEqualNode`	`Node.isEqualNode`

Old API Call	New Native Replacement
`jQuery.sap.newObject`	`Object.create`
`jQuery.sap.padLeft`	`"a".padStart(110, "0");`
`jQuery.sap.padRight`	`"a".padEnd(110, "0");`
`jQuery.sap.resources.isBundle`	`instanceof`
`jQuery.sap.startsWith`	`sMyString.startsWith("abc");`
`jQuery.sap.startsWithIgnoreCase`	`sMyString.toLowerCase().startsWith(sMyOtherString.toLowerCase())`
`jQuery.support.retina`	`window.devicePixelRatio >= 2`

jQuery Extensions Dependencies

jQuery extensions have been extracted into different modules. If the jQuery extension is required, it needs to be added to the module dependencies.

Example

Change from the global dependencies to adding the module dependencies to the jQuery extensions:

Previously the dependency was global

```
sap.ui.define([], function() {
    var oControl = jQuery("#myElement").control(0);
});
```

Dependencies should be added

```
sap.ui.define(["sap/ui/thirdparty/jquery", "sap/ui/dom/jquery/control"], function(jQuery) {
    var oControl = jQuery("#myElement").control(0);
});
```

jQuery Call	Old Module	New Module
`jQuery.*`	`jQuery.sap.global`	`sap/ui/thirdparty/jquery`
`jQuery.position`	`jQuery.sap.global`	`sap/ui/thirdparty/jqueryui/jquery-ui-position`
`jQuery.fn.control`	`jquery.sap.ui`	`sap/ui/dom/jquery/control`
`jQuery.fn.addLabelledBy`	`jquery.sap.dom`	`sap/ui/dom/jquery/Aria`
`jQuery.fn.removeLabelledBy`	`jquery.sap.dom`	`sap/ui/dom/jquery/Aria`

jQuery Call	Old Module	New Module
jQuery.fn.addDescribedBy	jquery.sap.dom	sap/ui/dom/jquery/Aria
jQuery.fn.removeDescribedBy	jquery.sap.dom	sap/ui/dom/jquery/Aria
jQuery.fn.cursorPos	jquery.sap.dom	sap/ui/dom/jquery/cursorPos
jQuery.fn.firstFocusableDomRef	jquery.sap.dom	sap/ui/dom/jquery/Focusable
jQuery.fn.lastFocusableDomRef	jquery.sap.dom	sap/ui/dom/jquery/Focusable
jQuery.fn.getSelectedText	jquery.sap.dom	sap/ui/dom/jquery/getSelectedText
jQuery.fn.hasTabIndex	jquery.sap.dom	sap/ui/dom/jquery/hasTabIndex
jQuery.fn.parentByAttribute	jquery.sap.dom	sap/ui/dom/jquery/parentByAttribute
jQuery.fn.rect	jquery.sap.dom	sap/ui/dom/jquery/rect
jQuery.fn.rectContains	jquery.sap.dom	sap/ui/dom/jquery/rectContains
jQuery.fn.scrollLeftRTL	jquery.sap.dom	sap/ui/dom/jquery/scrollLeftRTL
jQuery.fn.scrollRightRTL	jquery.sap.dom	sap/ui/dom/jquery/scrollRightRTL
jQuery.fn.enableSelection	jquery.sap.dom	sap/ui/dom/jquery/Selection
jQuery.fn.disableSelection	jquery.sap.dom	sap/ui/dom/jquery/Selection
:sapTabbable, :focusable, :sapFocusable	jquery.sap.dom	sap/ui/dom/jquery/Selectors
jQuery.fn.selectText	jquery.sap.dom	sap/ui/dom/jquery/selectText
jQuery.fn.zIndex	jquery.sap.dom	sap/ui/dom/jquery/zIndex

Legacy Factories Replacement

Overview of the replacement of global functions with the factory functions

The AMD module syntax already avoids Globals and enforces the strict dependency declaration. The following table shows how APIs which use synchronous requests to fetch modules or resources internally, can be replaced with asynchronous alternatives. The W3C has already deprecated the use of synchronous requests in the browser main thread, so this replacement prepares your applications for the removal of synchronous requests.

The SAPUI5 framework by default uses synchronous requests internally in several places. Most have already been replaced by asynchronous alternatives, or prepared to exchange the synchronous behaviour shown below. The asynchronous adoption starts from the beginning with the bootstrap script tag, where the `async` configuration parameter should be set to `true`. Applications can register an event callback via `sap.ui.getCore()#attachInit`. . The examples below show only the most frequently used synchronous APIs. There are more of these APIs, and most often the asynchronous alternatives return a `Promise` that can be used to retrieve the former return value.

Legacy, synchronous API	Modern API
Declarative App Description	

`sap.ui.component({`
`  name: "my.comp"`
`});`
	`sap.ui.require(['sap/ui/core/`
	`Component'], function(Component){`
	`  Component.create({`
	`  name: "my.comp"`
	`  // default: manifest: true`
	`  }).then(function(oComp) { ... });`

Components - Some API still experimental	

`var oComponentInstance =`
`sap.ui.component({`
`  name: "my.comp"`
`});`
	`sap.ui.require(['sap/ui/core/`
	`Component'], function(Component){`
	`  Component.create({`
	`  name: "my.comp"`
	`  }).then(function(oComp) { ... });`
	
	Alternatively, migrate to `componentUsages` with an additional adaption in the `manifest.json` file:
	
	`createContent: function() {`
	`  var oReuseComponentPromise =`
	`this.createComponent({`
	`  "usage": "reuse"`
	`  }).then(function(oComp) { ... });`
	`}`

Legacy, synchronous API	Modern API
`var oComponentClass = sap.ui.component.load({ name: "my.comp" });`	`sap.ui.require(['sap/ui/core/Component'], function(Component){ Component.load({ name: "my.comp" }).then(function(oClass) { var oComponentInstance = new oClass({...}); }); });`
`var oComponentInstance = sap.ui.component("my-comp-id");`	`sap.ui.require(['sap/ui/core/Component'], function(Component){ var oComponentInstance = Component.get("my-comp-id"); });`
i18n texts	
`jQuery.sap.resources({ url: "mybundle.properties" });`	`// sap/ui/Resources -> sap/base/i18n/ResourceBundle sap.ui.require(['sap/base/i18n/ResourceBundle'], function(Resource){ ResourceBundle.create({ url: "mybundle.properties", async: true }).then(function(oResource) { ... });`
Views	
`var oView = sap.ui.view({ viewName: "my.View", type: "XML" });`	`sap.ui.require(['sap/ui/core/mvc/View'], function(View){ View.create({ viewName: "my.View", type: "XML" }).then(function(oView) { ... });`
`var oView = sap.ui.xmlview({ viewName: "my.View" });`	`sap.ui.require(['sap/ui/core/mvc/XMLView'], function(XMLView){ XMLView.create({ viewName: "my.View" }).then(function(oView) { ... });`
Controllers	
`var oController = sap.ui.controller({ ... });`	`sap.ui.require(['sap/ui/core/mvc/Controller'], function(Controller){ Controller.create({ ... }).then(function(oController) { ... });`

Legacy, synchronous API	Modern API
Extension Points	
`var oView = sap.ui.extensionpoint( ... );`	`sap.ui.require(['sap/ui/core/ExtensionPoint'], function(ExtensionPoint){ ExtensionPoint.load({ ... }).then(function(aControls) { ... });`
Fragments	
`var oView = sap.ui.fragment({ name: "my.fragment", type: "XML" });`	`sap.ui.require(['sap/ui/core/Fragment'], function(Fragment){ Fragment.load({ name: "my.fragment", type: "XML" }).then(function(aControls) { ... });`
Version Info	
`var oView = sap.ui.getVersionInfo();`	`sap.ui.require(['sap/ui/core/VersionInfo'], function(VersionInfo){ VersionInfo.load({ ... }).then(function(oVersionInfo) { ... });`

Troubleshooting

How do I get the new Logger module on the browser console?

`jQuery.sap.log` is currently still available and the module `sap/base/Log` is not yet globally available. So, if a developer wants to set a log level with the new module on the console, additional code is required.

Old:

```
jQuery.sap.log.setLevel(3);
// OR
jQuery.sap.log.setLevel(jQuery.sap.log.Level.INFO);
```

New:

```
sap.ui.require("sap/base/Log").setLevel(3);
// OR
```

```
var Log = sap.ui.require("sap/base/Log"); Log.setLevel(Log.Level.INFO);
```

How can I mock or spy the new modules?

The new modules are no longer global. Spying or mocking them on `jQuery.sap.*` does not work. The module has to be loaded with `sap.ui.require`:

```
sap.ui.require(["sap/base/Log"],function(Log){
    QUnit.test("My Test", function(assert) {
        var oLogSpy = sinon.spy(Log, "warning");
        ...
        assert.equal(oLogSpy.callCount, 1, "Warning should be logged");
        Log.warning.restore();
    });
});
```

> **i** Note
>
> Some of the new modules were changed from objects to functions, for example `sap/base/util/uid` and `sap/base/strings/hash`. This means that the export of a module is used for import into other modules. Each importing module has its own reference to the original export. The test can change its own reference, but not the reference that other modules have obtained already. The value of the reference (the function), however, is always the same.
>
> Until further testing capabilities are available, you should write the tests in a way that it is not necessary to mock them, because they are used inside the actual API, which should be tested instead.

Optimizing Applications

SAPUI5 supports several means of optimizing the loading time for applications.

Resource Handling: Modularization and Localization

The handling of resources in SAPUI5 is divided in a client-side and a server-side part. The two parts are complementary and don't depend on each other.

The server-side resource handling is an optional component to improve the client-server interaction by providing a server-side locale fallback instead of a client-side fallback with multiple requests. The server-side resource handling is mainly used in Eclipse to support the modularized development of SAPUI5 applications and libraries.

Client-Side Resource Handling

SAPUI5 provides the following mechanism for handling resources on the client:

- Modularization of JavaScript files, see Modules and Dependencies [page 1094]
- Localization of application texts with resource bundles, see Resource Bundles [page 1272]

In both cases, SAPUI5 loads additional resources from a server. This server can be any kind of web server (simple, Java, ABAP, ...). Both do **not** depend on a specific server-side technology.

Server-Side Resource Handling

For the Java server and the integration into Eclipse, SAPUI5 provides a resource handler to improve the interaction between client and server, for example by providing a server-side locale fallback for the language to avoid multiple requests to get the correct language. It's also used to support modularized development of SAPUI5 applications and libraries. The Java resource handler is aligned with the concept of the JavaServer Faces.

- The default implementation must support packaging resources in the web application root under the path `resources/<resourceIdentifier>` relative to the web app root.
- Resources packaged in the classpath must reside under the JAR entry name `META-INF/resources/<resourceIdentifier>`

The SAPUI5 resource handler extends this concept to support standard and test-relevant resources. The resources are therefore packaged into the following paths:

- `resources/**`
  Resources are all kind of JavaScript, CSS, Mimes, Resource Bundles, which are relevant for the runtime.
- `test-resources/**`
  Test resources are resources that are samples and only relevant for testing purposes, for example, the content of the SAPUI5 test suite.

The resource handler in SAPUI5 provides the following additional features:

- Theme fallback:
  If resources aren't available for a theme, the resource handler automatically checks the base theme for such resources and returns them instead of a 404 error message.
- Resource bundle fallback:
  This fallback is similar to the client-side mechanism for loading resource bundles, but it negotiates the request on the server and returns the best found resource bundle instead of issuing a 404 error, for example:
  `messagebundle_en_US.properties > messagebundle_en.properties > messagebundle.properties`

Resource Servlet

For Java Servlet containers, SAPUI5 provides a `ResourceServlet` to manage the access to SAPUI5 resources within the web application and the various UI libraries in the classpath. The following snippet shows how to enable the resource servlet for SAPUI5:

```
<!-- ============================================================ -->
<!-- SAPUI5 resource servlet used to handle application resources -->
<!-- ============================================================ -->

<servlet>
  <display-name>ResourceServlet</display-name>
  <servlet-name>ResourceServlet</servlet-name>
  <servlet-class>com.sap.ui5.resource.ResourceServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ResourceServlet</servlet-name>
  <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ResourceServlet</servlet-name>
  <url-pattern>/test-resources/*</url-pattern>
</servlet-mapping>
```

Before you use it, make sure that the `ResourceServlet` is available in the classpath as JAR file.

Configuration

The resource handler is configured via context parameters, which are defined in the `web.xml`. The following table gives an overview about configuration parameters:

Key	Description
`com.sap.ui5.resource.USE_CACHE`	Flag for resource cache enabling; default: `true`
`com.sap.ui5.resource.MAX_AGE`	Specifies the maximum age of resources in milliseconds; default: `604800000` = 1 week
`com.sap.ui5.resource.ACCEPTED_ORIGINS`	List of accepted origins, for example `*`, `*mycompany.corp`, or `server.mycompany.corp`; default: empty
`com.sap.ui5.resource.DEV_MODE`	Flag development mode enabling; default: `false`
`com.sap.ui5.resource.TEMPLATE_PATH`	Specifies path to template for resource listing; default: `/templates/listing.html`
`com.sap.ui5.resource.VERBOSE`	Specifies verbosity of the resource handler; default: `false`
`com.sap.ui5.resource.REMOTE_LOCATION`	Specifies the location that is used to proxy requests to resources that aren't available locally; default: empty
`com.sap.ui5.resource.PREFER_REMOTE_LOCATION`	Flag to resolve the resource from the remote location before fallback to classpath; default: `false`
`com.sap.ui5.resource.USE_SERVER_CACHE`	Flag to enable caching of any resources in resource servlet; default: `true` (default in dev mode: `false`

Configuration parameters are added as context parameters to the web.xml.

Development Mode

When you're starting to develop SAPUI5 controls and modules being located inside the servlet paths `resources/` or `test-resources/`, it makes the development process easier to disable the caching of such resources as well as to enable the resource browsing. To activate the development mode, add the following context parameter:

```
<!-- BEGIN: DEV MODE -->
<context-param>
  <param-name>com.sap.ui5.resource.DEV_MODE</param-name>
  <param-value>true</param-value>
</context-param>
<!-- END: DEV MODE -->
```

Resource Browsing

If the development mode is active, you can browse resources via the resource browser:

- `%SERVER_URL%!/resources/`
- `%SERVER_URL%/test-resources/`

Tunneling a Remote Location

You can use the `ResourceServlet` to tunnel/proxy requests to another server that provides SAPUI5 resources instead of referring to SAPUI5 from a remote location inside the bootstrap script tag and thus avoid cross domain issues. To activate the remote location tunneling/proxying, add the following context parameter to the web.xml of your application:

```
<context-param>
  <param-name>com.sap.ui5.resource.REMOTE_LOCATION</param-name>
  <param-value>http://%server%:%port%/sapui5</param-value>
</context-param>
```

This dispatches the requests from `resources/sap/m/Button.js` to `http://%server%:%port%/sapui5/resources/sap/m/Button.js`.

If you are located behind a proxy and the remote location is outside your localnetwork, you can configure the proxy settings via the standard Java Networking and Proxy configurations by setting the system properties (for HTTP): `http.proxyHost`, `http.proxyPort`, `http.nonProxyHosts`, or (for HTTPS) `https.proxyHost`, `https.proxyPort`, `https.nonProxyHosts` of your Java runtime environment.

In general, for the resources returned from the proxy the `ResourceServlet` is enabling caching. By default, it uses the configured `com.sap.ui5.resource.MAX_AGE` to avoid too much load on the `ResourceServlet`.

Verify that a Resource was Retrieved from Remote Location

When in development mode, it's possible to verify that a resource was retrieved from the desired remote location by checking the response header of the respective request. In this case, the response header has an entry `x-sap-ResourceUrl = remote resource URL`, for example:

```
x-sap-ResourceUrl = http://%server%:%port%/sap/public/bc/ui5_ui5/resources/sap-ui-core.js
```

Resource Packaging

Resource packaging for web applications and Java modules can be any kind of JAR file, for example SAPUI5 UI library that is available in the classpath of the web application.

Store the resources as follows:

- Web application:

  ```
  WebContent/
    resources/
      **/**
    test-resources/
      **/**
  ```

- SAPUI5 UI libraries:

  ```
  META-INF/
    resources/
      **/**
    test-resources/
      **/**
  ```

  For custom JAR files, you need to apply to this on your own.

OSGi Servlet Container

When you run SAPUI5 as an OSGi web bundle and reference the UI libraries as OSGi bundles, you need to determine the SAPUI5 OSGi bundles:

- Extend the `ResourceServlet` in the OSGi servlet container by using an OSGi fragment that is responsible to add the OSGi flavor for the determination of UI libraries. Now, the `ResourceServlet` is aware of the OSGi bundles and can search within the OSGi servlet container for UI libraries.
- The `OSGiResourceServlet` uses the following entry in the `MANIFEST.MF` of the UI library's JAR files to determine the relevant UI libraries:

  ```
  x-sap-ui5-ContentTypes: UILibrary
  ```

SAPUI5 Library Location Used for Testing

The location of the SAPUI5 library that is used for testing may differ depending on several parameters.

If the SAPUI5 bootstrap tag contains `src="resources/sap-ui-core.js"`, the SAPUI5 runtime libraries from the Eclipse plugin are used.

If you want to test your SAPUI5 application in Eclipse against a different SAPUI5 Library location, for example on the ABAP server when running in the SAP NetWeaver UI AddOn scenario, you can configure the ResourceServlet. For that, open the web.xml file located in the <WebContent folder name>/WEB-INF folder and configure the parameter `com.sap.ui5.resource.REMOTE_LOCATION` and `com.sap.ui5.resource.PREFER_REMOTE_LOCATION` of the ResourceServlet where the placeholders {protocol}, {host name}, {port number}, {path to UI5 library} are to be exchanged by the real protocol, host name, port number and path to the SAPUI5 library, see Resource Handling [page 1127], section *Tunneling a Remote Location*.

```
  <servlet>
    <display-name>ResourceServlet</display-name>
    <servlet-name>ResourceServlet</servlet-name>
    <servlet-class>com.sap.ui5.resource.ResourceServlet</servlet-class>
 </servlet>
  ...
  <!-- force to use the remote location -->
 <context-param>
    <param-name>com.sap.ui5.resource.PREFER_REMOTE_LOCATION</param-name>
    <param-value>true</param-value>
 </context-param>
 <!-- add the remote location for the UI5 libraries -->
 <context-param>
    <param-name>com.sap.ui5.resource.REMOTE_LOCATION</param-name>
    <param-value>{protocol}://{host name}:{port number}/{path to UI5 library}</
param-value>
 </context-param>
```

Cache Buster for SAPUI5

A cache buster allows SAPUI5 to notify the browser to refresh the resources only when the SAPUI5 resources have been changed. As long as they are not changed, the resources can always be fetched from the browser's cache.

> **i Note**
>
> SAPUI5 supports the cache buster concept for Java and ABAP servers and for SAP Cloud Platform. SAP HANA XS does **not** support the cache buster concept.

When you want to cache your resources permanently, you simply need to change the URL in the SAPUI5 bootstrap tag from `resources/sap-ui-core.js` to `resources/sap-ui-cachebuster/sap-ui-core.js`.

The cache buster mechanism allows to always put the SAPUI5 resources into the browsers cache until a UI library or a web application has been changed. The default behavior of the SAPUI5 resource handler is either to cache the resources for a specific amount of time or alternatively in development mode it is using the `304/NOT MODIFIED` mechanism to check the SAPUI5 resources for being up-to-date. Both mechanisms are not optimal

in a final, productive scenario - that is the reason for the implementation of the cache buster mechanism. Applications, which want to use the cache buster mechanism, have to explicitly decide to use it.

The cache buster mechanism is part of the resource servlet. In general, requests to JavaScript resources can be handled via the cache buster mechanism. Typically this is used for the initial request for the bootstrap JavaScript:

```
  <script type="text/javascript"
      id="sap-ui-bootstrap"
      src="resources/sap-ui-cachebuster/sap-ui-core.js"
      data-sap-ui-libs="sap.ui.core,sap.m,sap.ui.table"
    data-sap-ui-theme="sap_belize"></script>
```

The bootstrap JavaScript will be included via the URL `resources/sap-ui-cachebuster/sap-ui-core.js` instead of `resources/sap-ui-core.js`.

Mechanism

The basic mechanism is implemented in the `ResourceServlet`. For the request to the bootstrap JavaScript it now serves a JavaScript file with the following content:

```
(function() {
  var sTimeStamp = '~20120716-0201~';
  var sScriptPath = 'sap\x2dui\x2dcore.js';
  var aScriptTags = document.getElementsByTagName('script');
  for (var i = 0; i < aScriptTags.length; i++) {
    if (aScriptTags[i].src) {
      var iIdxCb = aScriptTags[i].src.indexOf('/sap-ui-cachebuster/');
      if (iIdxCb >= 0 && aScriptTags[i].src.substring(iIdxCb + '/sap-ui-
cachebuster/'.length) == sScriptPath) {
        var sBasePath = aScriptTags[i].src.substring(0, iIdxCb);
        sBasePath += '/' + sTimeStamp + '/';
        window["sap-ui-config"] = window["sap-ui-config"] || {};
        window["sap-ui-config"].resourceRoots = window["sap-ui-
config"].resourceRoots || {};
        window["sap-ui-config"].resourceRoots[''] = sBasePath;
        document.write('<script type="text/javascript" src="' + sBasePath +
sScriptPath + '"></script>')
        break;
      }
    }
  }
})();
```

This script basically ensures that the global SAPUI5 configuration variable (`window["sap-ui-config"]`) exists, without modifying any existing values. It defines the resource root of SAPUI5 (the location where SAPUI5 loads all JavaScript modules, controls and control related resources from). Finally, another script tag is added to the page that points to the real boostrap JavaScript. The new resource root and the request path to the bootstrap JavaScript now contain a timestamp. Additionally the cache headers of the responses now look like the following:

```
Date: Mon, 16 Jul 2012 05:17:54 GMT
Expires: Thu, 14 Jul 2022 05:17:54 GMT
Cache-Control: max-age=315360000, public
```

By default all cache buster resources will be cached for one year.

Request Flow

When using the cache buster mechanism, the first request must never be cached because it is being used to determine the timestamp / and to finally redirect to the correct script. The following list explains the flow:

- `resources/sap-ui-cachebuster/sap-ui-core.js` => NO_CACHE
- `resources/~201106210204~/sap-ui-core.js` =>CACHE

Timestamp

If you are interested in the timestamp of the cache buster, you can grab it with the following request:

`resources/sap-ui-cachebuster`

The response is `text/plain` with such value: `~20120716-0201~`

Application Cache Buster

The application cache buster (short `AppCacheBuster`) is similar to the cache buster but is used for application resources.

> **i** Note
>
> SAPUI5 supports the application cache buster on SAP NetWeaver AS for ABAP only.
>
> For Java apps on SAP NetWeaver AS for Java and SAP HANA XS the application cache buster concept is **not** supported.

Applications provide an index file named `sap-ui-cachebuster-info.json` (created on the fly) containing the last modified timestamps of all included files (like scripts, properties, or any other file that we load via XHR programmatically). Technically this file is a mapping between the request path (below the context path of the application) and the last modified time stamp.

The server instructs the client to cache all the above resources (not using the 304/not modified mechanism). For the index file we are using the 304/not modified mechanism to avoid to load when it has not been changed.

On the client side, we initially load this file of the application when enabled via configuration option `sap-ui-appcachebuster` and use this for the XHR requests. If the request path is contained in the above mentioned index file we simply add the time stamp as leading path segment to this request. If the time stamp doesn't change the URL is unique and therefore it will be taken from cache. Once the file is modified the URL parameter will be changed and therefore loaded again from the back end.

The server has to delete the time stamp from this URL to look up the file properly. For SAP NetWeaver AS for ABAP, the logic is implemented in the ICF handler. Both back end implementations, SAP NetWeaver AS for Java and SAP NetWeaver AS for ABAP, also generate the index file on-the-fly.

> **i** Note
>
> The application cache buster does **not** work across application borders. If you require resources from another application they are not loaded via this mechanism.

Application Cache Buster: Index File

The index file includes all files that should use the cache buster.

Unlike the cache buster mechanism for runtime resources, the application files have an own timestamp for each file. Thus, the application provides the index file `sap-ui-cachebuster-info.json`. The index file looks as follows:

```
{
  "mvc/MyMVC.view.js": "20120907134005",
  "mvc/MyMVC.controller.js": "20120907134005",
  "mvc/MyMVC.view2.js": "20120906113301",
  "mvc/MyMVC.controller2.js": "20120906113023"
}
```

Application Cache Buster: Configuration

The configuration `data-sap-ui-appCacheBuster="./"` must be added to the bootstrap script of the application page.

The following code shows an example how the configuration is added to activate the application cache buster:

```
<script id="sap-ui-bootstrap"
  src="resources/sap-ui-cachebuster/sap-ui-core.js"
  data-sap-ui-libs="sap.ui.core,sap.m,sap.ui.table"
  data-sap-ui-theme="sap_belize"
  data-sap-ui-appCacheBuster="./"></script>
```

The parameter `data-sap-ui-appCacheBuster` is a `string[]` which means you can pass a comma-separated list of base URLs pointing to other applications which should be considered by the Application Cache Buster. By default it should contain the base path of your local application.

These base URLs are used to load the index files.

Application Cache Buster: Request Flow

When using the application cache buster, a request order must be observed.

When using the Application Cache Buster mechanism, the first request must never be cached because it is being used to fetch the index file. The following list explains the flow:

1. http://myserver/myapp/sap-ui-cachebuster-info.json ⇒ *NO_CACHE*

2. http://myserver/myapp/~201106210204~/mvc/MyMVC.view.js ⇒ **CACHE**
    - http://myserver/myapp/mvc/MyMVC.view.js ⇒ *internally resolve to this URL*

Application Cache Buster: Enhanced Concept

The enhanced concept for application cache buster takes care about most of the URLs in a general way.

The first iteration of the Application Cache Buster only supports files which have been loaded via `jQuery.ajax`. The enhanced concept supports the transformation of URLs for `jQuery.sap.includeScript`, `jQuery.sap.includeStyleSheet`, and properties of the type `sap.ui.core/URI`. Additionally the enhanced concept allows to register components or base URLs which are considered by the Application Cache Buster. This base URL is used to load the index file with the timestamp information.

Registration of external URLs

If you do not specify all the applications in the bootstrap configuration, you can also register them during runtime. To register additional locations, use the following API:

```
sap.ui.core.AppCacheBuster.register("/sap/bc/my/other/component");
```

Avoid handling of specific URLs

To avoid handling of specific URLs, you can override the default behavior as follows:

```
sap.ui.core.AppCacheBuster.handleURL = function(sURL) {
  return sURL !== "my/specific/url";
};
```

Adapting to Operating Systems And Devices

No need to worry about device specifics! SAPUI5 apps run on smartphones, tablets, and desktops. The UI controls automatically adapt themselves to each device's capabilities and make the most of the available real estate. supports several functions to adapt to operating systems and devices.

The Device API

The device API (`sap.ui.Device`) is an API which provides information about device specifics, like the operating system along with its version, the browser and browser version, screen size, current orientation and support for specific features like touch event support, orientation change and so on.

For example, the `sap.ui.Device.orientation` object holds the current orientation information:

- `landscape`: Flag indicating whether the current orientation is landscape
- `portrait`: Flag indicating whether the current orientation is portrait

If you want to register a handler for a particular event like a resize or an orientation change, for example, you can do so easily by attaching to such an event:

```
sap.ui.Device.orientationChange.attachHandler(function(){
    alert("orientation changed");
});
```

Both work across platforms, even in cases where the orientation change event is not natively supported by the device, for instance.

If you want to check for a certain system category (phone, tablet, desktop) in your code, you can ask for the value of `sap.ui.Device.system.phone`, which would be set to `true` if you are accessing the page from a phone.

> **i Note**
>
> Categorization is based on various factors like screen size, touch enablement, operating system and user agent. Depending on the combination of features on a device, it could happen that more than one flag is set to `true`.

In the same way, you can also check for different browsers, different operating systems and available features for the current device or browser.

An API for screen width change events is available under `sap.ui.Device.media`. It allows you to attach handlers to screen width changes between certain intervals. Whenever such an interval is met, there is a certain CSS class added to the HTML root tag on the page. There are predefined range sets for these intervals for typical widths. The standard sets which are automatically initialized are as follows:

- SAP_3STEPS: A 3-step range set (S-L)
- SAP_STANDARD_EXTENDED: A 4-step range set (Phone, Tablet, Desktop, LargeDesktop)

For more information about the exact values and corresponding css classes, see sap.ui.Device.media.RANGESETS in the *API Reference* in the Demo Kit.

The range sets described above will add a CSS class to the body element of the HTML according to the following pattern: `sapUiMedia-Std-NAME_OF_THE_INTERVAL` and `sapUiMedia-StdExt-NAME_OF_THE_INTERVAL`.

The following ranges are available by default:

- `"Phone"`: For screens smaller than 600 pixels.
- `"Tablet"`: For screens greater than or equal to 600 pixels and smaller than 1024 pixels.
- `"Desktop"`: For screens greater than or equal to 1024 pixels and smaller than 1440 pixels.
- `"LargeDesktop"`: For screens greater than or equal to 1440 pixels (coming from the Extended range set)

You can use any of the available range sets to attach to a particular width interval change, like in this example here:

```
// attach to event
sap.ui.Device.media.attachHandler(fnSizeChanged, null,
sap.ui.Device.media.RANGESETS.SAP_STANDARD);
// eventHandler:
function fnSizeChanged(mParams) {
    switch(mParams.name) {
        case "Phone":
            // Do what is needed for a little screen
            break;
        case "Tablet":
            // Do what is needed for a medium sized screen
            break;
        case "Desktop":
            // Do what is needed for a large screen
    }
}
```

Your event handler will be called with a single argument, a map of parameters you can then access which contain the following information about the current interval after the width change:

- `mParams.from`: The start value (inclusive) of the entered interval as a number
- `mParams.to`: The end value (exclusive) range of the entered interval as a number or undefined for the last interval (infinity)

- `mParams.unit`: The unit used for the values above, for example "px"
- `mParams.name`: The name of the entered interval, if available

In your event handler, you can then easily check for the current interval or interval name and implement your code to update the UI for the new range set accordingly. If you want to work with a different range set, you need to make sure that you initialize it explicitly by using the `initRangeSet` method available with `sap.ui.Device.media`.

If your use case requires it, you can also define your own range set, which would again require explicit initialization from your side.

For more information, see sap.ui.Device.media.initRangeSet in the *API Reference* in the Demo Kit.

Controls with Built-In Device Adaptation

SAPUI5 comes with several controls which are already able to react to the available screen real estate and resolution by themselves. Some require particular properties to be set, and with some, everything just works out of the box.

> **i** Note
>
> This list does not claim to be comprehensive, but shows some widely used examples outlining the steps you can take to make use of this functionality. For more details, browse through the Samples in the Demo Kit and see what SAPUI5 has in store for your app.

sap.m.SplitApp

The `SplitApp` is a control that can act as a root element of an application for mobile and desktop devices. It is designed to be used as the base for applications following the master-detail pattern.

It maintains two `NavContainers` if running on tablet or desktop, and one if running on a phone. In default mode, the master `NavContainer` will always be displayed on desktop screens and on tablets in landscape mode. It will be hidden on tablet-sized screens and can either be swiped in and out (if the device is touch-enabled) or the visibility can be toggled by clicking a button. On phone devices, either the master `NavContainer` or the detail `NavContainer` will be shown, and a true forward-backward navigation is established between the two.

For more information, see the sap.m.SplitApp samples and sap.m.SplitApp in the *API Reference* in the Demo Kit.

As the `SplitApp` control inherits from `sap.m.SplitContainer`, you can alter this behavior by setting the corresponding mode property. You can set this property to `StretchCompressMode`, for instance, if you want the master to always be displayed on tablet-sized screens, irrespective of the current orientation. The different modes that are available are described under sap.m.SplitAppMode in the *API Reference* in the Demo Kit.

For more information about the `SplitContainer`, see the sap.m.SplitContainer samples and sap.m.SplitContainer in the *API Reference* in the Demo Kit.

sap.m.Table (also called the “Responsive Table”)

One control that is widely used across all kinds of different applications is `sap.m.Table`, which has several features you can use for device adaptation. On smaller devices, for example, you can set certain properties that will make particular columns pop in instead of being displayed as a normal column, or show and hide columns completely.

For example, you can set a `minScreenWidth` for the columns. This will cause columns to only show up if a certain screen width is matched. You can define this `minScreenWidth` in px or rem, but here you can also use the standard categories that come from the device API (Phone, Tablet, or Desktop). For more information about the device API, see The Device API [page 1137].

Setting the additional property `demandPopin` to `true` for a column will also react to the `minScreenWidth` you specify. In such a case, the column will be shown as a popin on smaller screens, instead of being completely hidden. For more information about responsive tables and their columns, see the sap.m.Table and sap.m.Column samples and sap.m.Table in the *API Reference* in the Demo Kit.

sap.m.ResponsivePopover

The `ResponsivePopover` is actually a combination of the functionality offered by a `sap.m.Dialog` and a `sap.m.Popover`. Depending on the device it is displayed on, it will either act as the former (on a phone) or as the latter (on tablets and desktop). For reuse, it is best placed within a fragment, which will be instantiated and displayed on demand, like when a user clicks a certain button, for example.

The `ResponsivePopover` does not need any additional properties to achieve proper device adaptation, it can simply be used "as is". There are, however, several properties that only take effect on one particular type of device, as some properties will not make any sense at all for a particular screen size.

For more details on these properties, see the documentation and sample for sap.m.ResponsivePopover in the *Samples* and sap.m.ResponsivePopover in the *API Reference* in the Demo Kit.

sap.m.OverflowToolbar

The `OverflowToolbar` is a special type of toolbar that allows you to decide if elements within should go into an overflow area when space on the screen is limited. All elements going into the overflow area can then be reached by clicking the overflow button.

As a developer, you can decide if there are elements that will never or always go into the overflow area by adding overflow layout data to particular content.

For detailed examples and more information about the `OverflowToolbar`, see the sap.m.OverflowToolbar samples and sap.m.OverflowToolbar in the *API Reference* in the Demo Kit.

sap.m.PullToRefresh

The `PullToRefresh` control allows users to trigger an update operation with touch or mouse interaction. On touch-enabled devices, the control will automatically be hidden, and users can swipe down the page to trigger it. On mouse-operated devices, the control will always be shown and can be clicked in order to trigger the functionality.

To use this control, it first needs to be placed within the content of any scroll container. In the corresponding controller, the refresh handler can be implemented, which in this use case would update the list of products in the view.

To see a full example and read more information about `PullToRefresh`, see the sap.m.PullToRefresh samples and sap.m.PullToRefresh in the *API Reference* in the Demo Kit.

sap.ui.layout.form.Form

`sap.ui.layout.form.Form` is a form that can also adapt to the available screen size, particularly when used with the corresponding `ResponsiveGridLayout`. A form consists of `FormContainers`, which in turn contain the fields and their labels. You can define general layout data for a form. For instance, you can decide how many columns you want to display depending on the available screen size, as shown here:

```
<f:layout>
    <f:ResponsiveGridLayout
        columnsL="4"
        columnsM="2"/>
</f:layout>
```

(assuming that `f` is declared as the `sap.ui.layout.form` namespace under which the `ResponsiveGridLayout` is also available.)

On a small screen, there will always be only one column for form containers.

Furthermore, you can define how much space labels will take when one or more columns are displayed, and you can also specify `sap.ui.layout.GridData` as layout data on the `FormContainers`, labels and content fields, as the `sap.ui.layout.Grid` is used internally in the form.

For more information about `sap.ui.layout.form.Form`, see the sap.ui.layout.form.Form samples and sap.ui.layout.form.Form in the *API Reference* in the Demo Kit.

For more information about `ResponsiveGridLayout` and `GridData`, see sap.ui.layout.form.ResponsiveGridLayout and sap.ui.layout.GridData in the *API Reference* in the Demo Kit.

Checking the Operating System your Application is Running on

A platform attribute is added to the HTML tag when running on mobile devices.

This attribute provides information about the current platform and version.

> **i Note**
>
> The list of supported devices when using `sap.m` library can be found here: Browser and Platform Support [page 20]

In addition to that, SAPUI5 adds a platform-dependent CSS class to the HTML tag of the page. This enables control or application developers to create platform-dependent styling for their controls or applications.

Technical Details

When the SAPUI5 bootstrap script file is loaded, a check is performed to see if the application is running on a mobile platform. If this is the case, the attribute and CSS classes are added to the HTML tag. The platform attribute value has the following connotation: *Operating system* + *version*, for example `iOS6.0`, `Android4.1.1` or `bb10.0.9.2372`. *Operating system* can have the following values:

- `iOS` (Apple devices)
- `Android` (Android devices)
- `bb` (BlackBerry)
- `winphone` (Windows phone)

The version numbers are separated by dots. The possible values for the CSS class are:

- `sap-ios` (Apple devices)
- `sap-android` (Android devices)
- `sap-bb` (BlackBerry)
- `winphone` (Windows phone)

The platform attribute or CSS class is used as follows:

- To provide a different font on Android devices, you specify your font by directly using the CSS class `sap-android`.

```
.sap-android{
         font-family: Roboto;
}
```

- Example for providing a different font when running on `Android 2.x`:

```
html[data-sap-ui-os^='Android2'] .sap-android{
    font-family: "Droid Sans";
}
```

Content Densities

The devices used to run apps that are developed with SAPUI5 run on various different operating systems and have very different screen sizes. SAPUI5 contains different content densities for certain controls that allow your app to adapt to the device in question, allowing you to display larger controls for touch-enabled devices and a smaller, more compact design for devices that are operated by mouse.

Available Content Densities

The table below shows the content densities that are available for the Quartz Light, Belize, Blue Crystal, and high-contrast themes:

Content Density	CSS Class	Explanation
Cozy	`sapUiSizeCozy`	'Large' design: Dimensions of the controls are optimized for touch-enabled devices, such as smartphones, to allow users to interact with controls more easily. This is the default density for most controls, particularly those in the `sap.m` library.
Compact	`sapUiSizeCompact`	Reduced-size design: The font size is the same as for the cozy density, but the dimensions of the controls and the spacing between them are reduced. This density is more suitable for mouse-operated devices, such as desktops. For some controls, this is the default density.
Condensed	`sapUiSizeCondensed`	Size even further reduced compared to *Compact* (in particular, row heights smaller). This density can be used for all tables of the `sap.ui.table` library.

The following two screenshots show the difference between the *Cozy* and *Compact* densities, using a simple `sap.ui.table.Table` example:

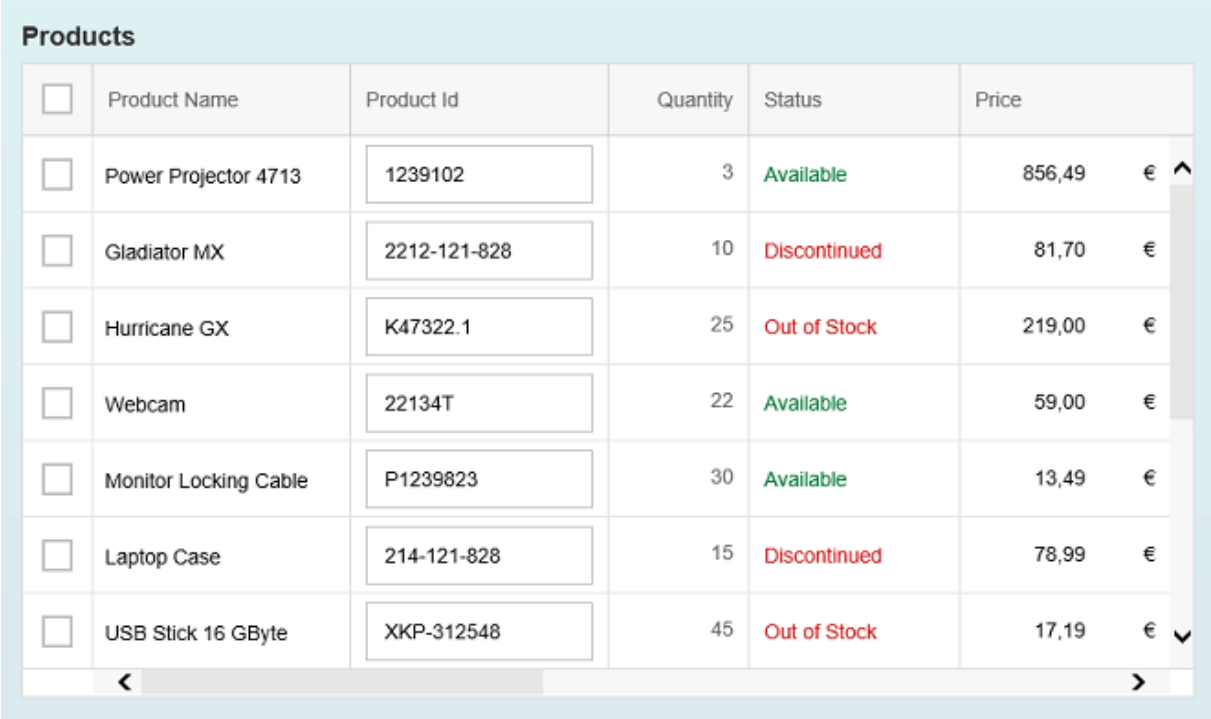


Figure 218: Cozy Density: Mainly for Touch Devices (such as Smartphones)

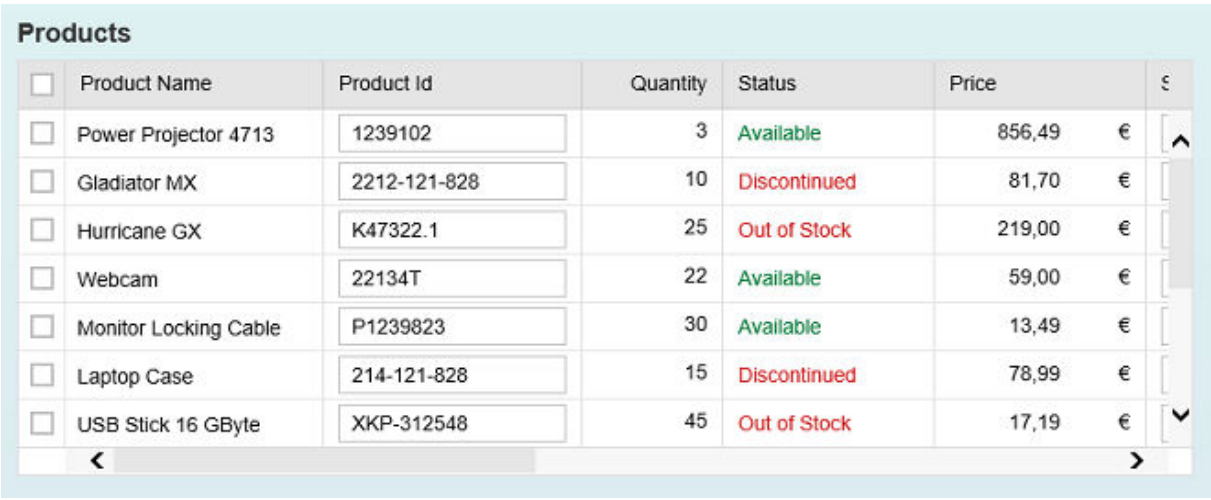


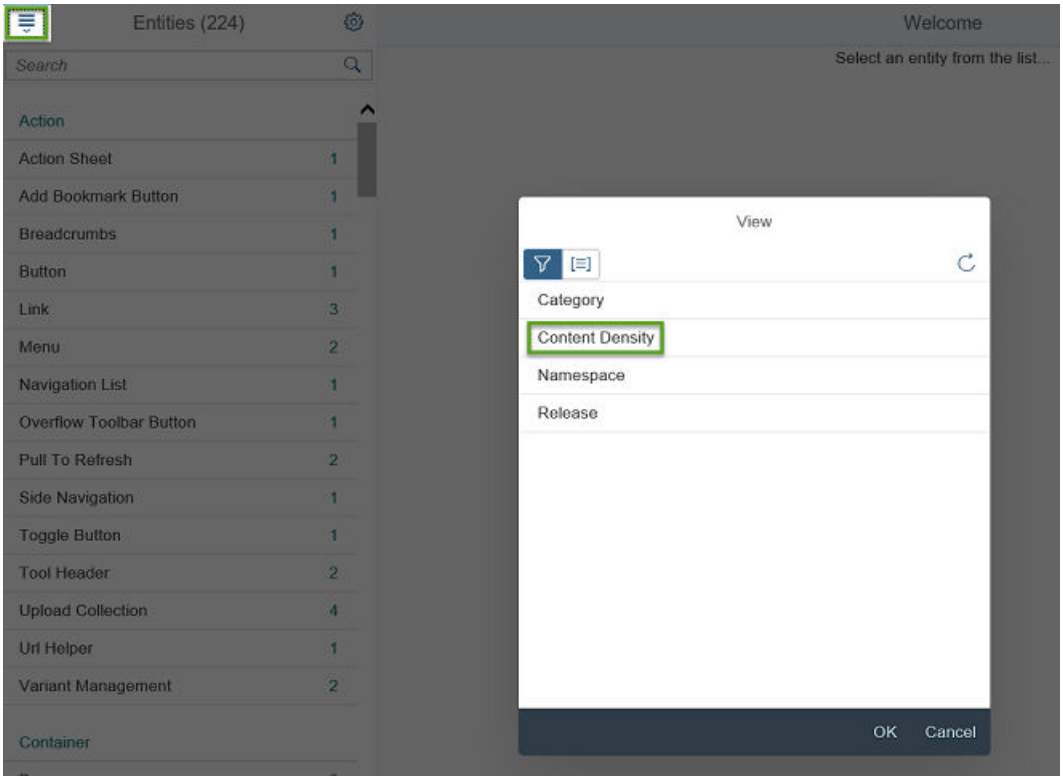
Figure 219: Compact Density: Mainly for Mouse-Operated Devices (such as Desktops)

Checking Which Content Densities Are Supported for a Control

If you need to know which content densities are supported for a particular control, the best place to look is the Samples section in the Demo Kit. After choosing a control from the list, look at the details in the Object Header

area to see which density is supported. In the example shown below, the control supports both the *Compact* and *Cozy* content densities:

Alternatively, you can also use the filter function in the *Samples* in the Demo Kit to filter the controls according to their content densities. Simply choose the filter selection icon in the upper left corner of the screen and then select *Content Density*, as shown below:



Setting Densities

You set the corresponding content density CSS class on the **container** for the part that you need to switch to the content density in question, not on the control itself. This is usually done within the administration settings of the SAP Fiori launchpad, but for a standalone scenario we recommend that you set it at a high level, such as `<body>`, as in most cases you will want to set it for the whole app.

> **i Note**
>
> Be aware that you can only set **one** density within a hierarchy: Once you have set a CSS class at a high level, such as the one described above, it cascades all the way down, meaning you cannot revoke or overwrite it in the lower levels of your coding.

Thus, when using densities, you cannot mix them: You must not combine *Cozy* and *Compact* or *Cozy* and *Condensed* within the same hierarchy.

You can use densities in the following way:

- *Cozy*
  Make sure you always only use the *Cozy* density within a hierarchy.
- *Compact*
  Make sure you always only use the *Compact* density within a hierarchy.
- *Condensed*
  Condensed is a special case and can only be used in combination with the *Compact* density.
  Also, keep in mind that the *Condensed* density has an effect on controls in the `sap.ui.table` library and their content only. If the density is set for controls outside of these tables, it will not have any effect on them.

How to Use Densities for Controls

How content densities are set and how they can be used in the SAP Fiori launchpad is explained and shown in the following code samples (using the *Compact* density as an example).

> **i Note**
>
> The default design for all controls belonging to the `sap.m` library is the *Cozy* density (larger dimensions and spacings). If your application only uses the `sap.m` library, you can skip setting a CSS class if the *Cozy* density is exactly what you require. However, controls belonging to other libraries may also support a cozy design (such as `sap.ui.table.Table`) but the default might be different (such as *Compact* density). For this reason, if your application uses controls belonging to different libraries, we strongly recommend that you set the CSS class `sapUiSizeCozy` if you want to use the *Cozy* density (and similarly, CSS class `sapUiSizeCompact` for the *Compact* density).

Using Densities

A density is triggered by the related CSS class, for example, `sapUiSizeCompact` for the *Compact* density, set on a parent element of the UI region for which you want to use the controls. This means that some parts of the UI or different apps inside a `sap.m.Shell` can use the standard density of the `sap.m` controls, while other parts can use a different density at the same time. However, sub-parts of the UI part that is set to *Compact* density **cannot** use the *Cozy* density because the CSS class affects the entire HTML subtree.

As dialogs and other popups are located at the root of the HTML document, you also have to set the CSS class for those elements to the respective density. The CSS class only affects child controls. You **cannot** make a

control itself compact or cozy by adding the CSS class to it. Instead, set the CSS class on the parent container, for example a view or a component.

XML view definition - Example:

```
<mvc:View class="sapUiSizeCompact" xmlns=....>
   ...
</mvc:View>
```

JS view definition - Example:

```
createContent: function(oController) {
   ...
   this.addStyleClass("sapUiSizeCompact"); // make everything inside this View
appear in Compact density
   ...
}
```

JavaScript opening a dialog - Example:

```
// "Dialog" required from module "sap/m/Dialog"
var myDialog = new Dialog({.....}).addStyleClass("sapUiSizeCompact");
myDialog.open();
```

JavaScript instantiating a view - Example:

```
// "View" required from module "sap/ui/core/mvc/View"
View.create({ ... }).then(function(oView) {
    oView.addStyleClass("sapUiSizeCompact");
});
```

> **i** Note
>
> It is also possible to apply the relevant density only under certain circumstances, for example, for devices that do **not** support touch interaction. In this case, add the class dynamically to the UI instead of statically. You can do this, for example, in the view controller:
>
> ```
> sap.ui.define(['sap/ui/core/mvc/Controller', 'sap/ui/Device'],
> function(Controller, Device) {
>     return Controller.extend("sap.my.controller", {
>             onInit: function() {
>                 // apply compact density if touch is not supported, the
> standard cozy design otherwise
>                 this.getView().addStyleClass(Device.support.touch ?
> "sapUiSizeCozy" : "sapUiSizeCompact");
>             }
>         }
>     );
> });
> ```

As the check depends on several factors, you may not want to repeat the same logic again and again. A dialog opened from a compact or cozy view should, for example, also be in *Compact* or *Cozy* density.

Synchronizing a Density for a Dialog

As dialogs are rendered in a different part of the HTML tree, they do **not** automatically inherit the density. To decide if you set the relevant density for a dialog, either perform the same check as for the view or use the

convenience function `syncStyleClass` from `sap/ui/core/syncStyleClass`. This convenience function synchronizes a style class between elements. The function accepts the following parameters: Name of the style class, source element, and destination element. The following code snippet shows an example:

```
<mvc:View
    controllerName="mycontroller"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns="sap.m">
    <Button text="Show Dialog" press="onOpenDialog" />
</mvc:View>
```

```
<core:FragmentDefinition xmlns="sap.m" xmlns:core="sap.ui.core">
    <Dialog title="Alert" type="Message">
        <Text text="Lorem ipsum dolor sit amet" />
        <beginButton>
            <Button text="Close" press="onDialogClose" />
        </beginButton>
    </Dialog>
</core:FragmentDefinition>
```

```
sap.ui.define(["sap/ui/core/mvc/Controller", "sap/ui/core/Fragment", "sap/ui/
core/syncStyleClass"], function(Controller, Fragment, syncStyleClass) {
    return Controller.extend("mycontroller", {
        onOpenDialog: function (oEvent) {
            var fnSync = function() {
                // sync compact style
                syncStyleClass("sapUiSizeCompact", this.getView(),
this._oDialog);
                this._oDialog.open();
            }.bind(this);
            if (!this._oDialog) {
                Fragment.load({
                    name: "mydialog",
                    controller: this
                }).then(function(oDialog) {
                    this._oDialog = oDialog;
                    this.getView().addDependent(this._oDialog);
                    fnSync();
                }.bind(this));
            } else {
                fnSync();
            }
        }
    });
});
```

When calling `syncStyleClass` from `sap/ui/core/syncStyleClass`, the source element can be a jQuery object, a SAPUI5 control, or the ID of an HTML element. The destination object can either be a jQuery object or a SAPUI5 control.

Checking for the Density Style Class

To determine if the relevant style class is set anywhere above a certain HTML element, you can use the `closest` function from jQuery as shown in the following example:

```
// "Button" required from module "sap/m/Button"
// "Dialog" required from module "sap/m/Dialog"
var btn = new Button({
    text: "Hello World",
```

```
    press: function(){
        var dialog = new Dialog({
            title: "Hello World",
            content: new Button({text:"Test Me"})
        });


        // add the 'sapUiSizeCompact' class if the Button is in an area using
Compact density
        if (this.$().closest(".sapUiSizeCompact").length > 0) { // "this" in the
event handler is the control that triggered the event
            dialog.addStyleClass("sapUiSizeCompact");
        }


        dialog.open();
    }
});
```

Using Density Classes in the SAP Fiori launchpad

The SAP Fiori launchpad (FLP) optionally reads the supported content densities from the app descriptor (`manifest.json`) and - if available - sets the appropriate content density class on the `<body>` tag. On devices with mouse and touch support, the FLP also allows the desired content density to be configured by the user. To avoid situations where an application and the FLP write different content density classes, we recommend using the following logic within all applications that are intended to be used inside the FLP:

```
getContentDensityClass : function() {
      if (this._sContentDensityClass === undefined) {
            // check whether FLP has already set the content density class; do
nothing in this case
            if (jQuery(document.body).hasClass("sapUiSizeCozy") ||
jQuery(document.body).hasClass("sapUiSizeCompact")) {
                  this._sContentDensityClass = "";
            } else {
                  // Store "sapUiSizeCompact" or "sapUiSizeCozy" in
this._sContentDensityClass, depending on which modes are supported by the app.
                  // E.g. the “cozy” class in case sap.ui.Device.support.touch
is “true” and “compact” otherwise.
            }
      }
      return this._sContentDensityClass;
}
```

This function returns an empty string if the FLP has already set a content density CSS class, or the proper CSS class to be set. The result of this function should then be set as a style class on the root view of the application and all dialogs and popups.

Providing Density Support for a Control

If you want to apply content densities to your own controls, provide the default CSS styling for the *Cozy* density regardless of any size density classes and provide additional CSS styling to shrink the size, if an ancestor

element has the `sapUiSizeCompact` class, for example, for the *Compact* density. The following code snippet shows you an example:

```
.myOwnControl { /* the standard (big) style */
   ...
   height: 3rem;
   ...
}
.sapUiSizeCompact .myOwnControl { /* reduce the height in compact density */
   height: 2rem;
}
```

Options for Further Adaptation

In addition to those performed automatically by SAPUI5, the application can apply further platform adaptations.

You can use the `sap.ui.Device` API to check for touch enablement, a particular screen size, orientation, browser or operating system, for example. For more information about this API, see The Device API [page 1137].

Besides using this API, there are also several options available for you to use by using CSS, as outlined below.

Hiding/Displaying Controls Depending on the Device

To determine a control's visibility in a device-dependent way, you can use the following CSS classes:

- `sapUiVisibleOnlyOnDesktop`
- `sapUiHideOnDesktop`
- `sapUiVisibleOnlyOnTablet`
- `sapUiHideOnTablet`
- `sapUiVisibleOnlyOnPhone`
- `sapUiHideOnPhone`

The names are actually self-explanatory; for each device, you have a corresponding class that you can use to either explicitly hide or show the particular control.

> **i Note**
>
> The control will still be part of the app but hidden by CSS only. For managing visibility on a generic level, consider controlling the visible property with the device API instead, as this means the controls will not be added to the DOM at all but just treated as invisible by SAPUI5.

Responsive Margin and Padding Classes

In order to make it possible for app developers to adjust margins and paddings in their apps without needing to write their own CSS, SAPUI5 provides them with convenience classes. For responsiveness, the classes `sapUiResponsiveMargin` and `sapUiResponsiveContentPadding` are particularly useful.

To read more detailed documentation about margins and padding classes, see Using Predefined CSS Margin Classes [page 1046] and Using Container Content Padding CSS Classes [page 1051] respectively.

Writing Device-Dependent Custom CSS

It is easy to add your own device-dependent or screen-size-dependent custom CSS by prefixing your selectors with the classes that come from the device API (for more information about the device API, see the link below under *Related Information*). Whenever you need to set a particular style on, say, a small screen like a phone only, you can do so by picking the `sapUiMedia-Std-Phone` CSS class as part of your selector. For example, a particular style for phone only could look like this:

```
.sapUiMedia-Std-Phone .yourSelector{
    Style-applied-to-phone-only: someValue;
}
```

Additionally, the information regarding which device you are currently on is available on the html root tag as one of these three CSS classes:

- `sap-desktop`
- `sap-tablet`
- `sap-phone`

This means you can provide style for the phone use case using CSS cascades as follows:

```
.sap-phone .myControl {
      font-size: small;
}
```

The main difference between the two options is that the first one makes assumptions based on the current range interval (so the screen size), whereas the latter is indeed set depending on which device is present.

You might also consider checking the screen size using media queries in CSS or the browser/jQuery APIS in JavaScript.

For more information about writing custom CSS in general, please also read CSS Styling Issues [page 1464].

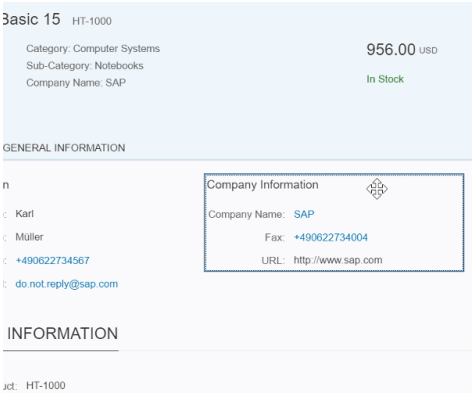
Related Information

The Device API [page 1137]

SAPUI5 Flexibility: Adapting UIs Made Easy

Modification-free, cost-saving, easy to use, and performant: Discover the new flexibility when adapting SAP Fiori UIs using SAPUI5 flexibility.

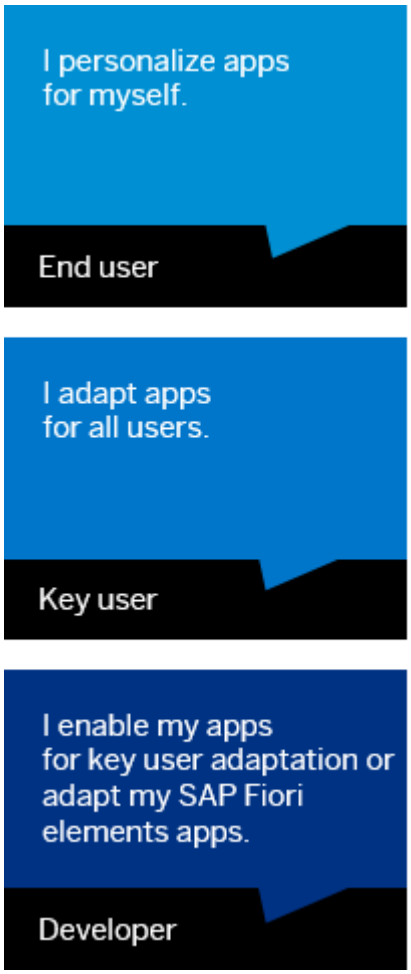
Flexibility is key! Enterprise software must adapt to rapidly changing environments. For example, customers need their apps to fit their processes without long IT projects to adapt them, and cloud providers want to run the same software for everyone to reduce TCO. You think adapting the user interface of SAP Fiori apps (for example, by adding, hiding or rearranging fields) is a complex process? Think again! SAPUI5 flexibility features allow upgrade-safe and modification-free UI changes on different levels (for example, at customer side) that can be performed by different users (end users, key users, and developers).



Here are four reasons why you want to use SAPUI5 flexibility:

- **It allows modification-free UI changes.**
  In contrast to extension points, UI changes made with SAPUI5 flexibility are modification-free. This means better lifecycle stability over release cycles, as the original app stays untouched and the changes are applied to the views only at runtime. This is achieved by storing the changes by the different users in separate layers. For more information, see Layering Concept [page 1156].
- **It saves time and money.**
  In the past, changing the UI was a complex, time and money consuming process. SAPUI5 flexibility changes all that! For example, with its *key user adaptation* feature, even users without technical knowledge can easily make UI changes themselves.
- **It's easy to use.**
  Using SAPUI5 flexibility makes adapting the UIs of apps simple and intuitive, with WYSIWYG features and tools that are available right in the context the user is working in: end users personalize object pages of their apps and key users adapt apps for their teams directly in the SAP Fiori launchpad; developers can use the SAPUI5 Visual Editor in SAP Web IDE .
- **It's performant.**
  Last but not least: This flexibility doesn't come at the expense of performance! By caching the UI changes, SAPUI5 flexibility guarantees smooth working with your adapted apps.

How SAPUI5 flexibility supports its users



- #unique_536/unique_536_Connect_42_subsection-im1 [page 1153]
- #unique_536/unique_536_Connect_42_subsection-im2 [page 1154]
- #unique_536/unique_536_Connect_42_subsection-im3 [page 1155]

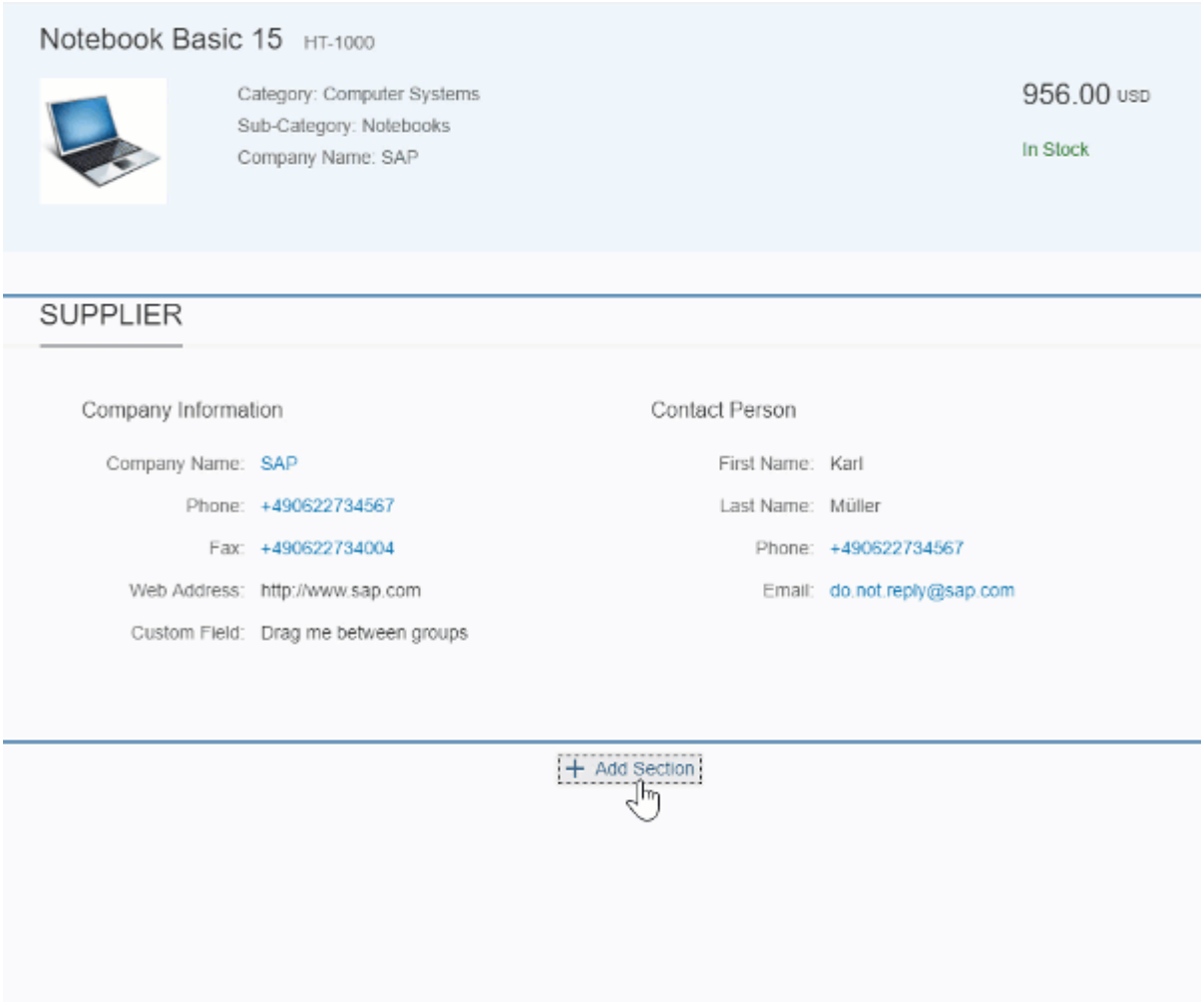
Hover over each quote for a brief description and click for more information.

End users can personalize object pages directly in the SAP Fiori launchpad

They just start the personalization mode and use intuitive WYSIWYG functions to adapt the UI to meet their unique, day-to-day needs.

For the personalization feature to be available, in SAP Fiori launchpad on ABAP or on SAP Cloud Platform Portal on Neo Environment, the app needs to be assigned to one of the user's roles.

> **i Note**
>
> For more information, open the documentation for *SAP Fiori Launchpad in SAP NetWeaver AS for ABAP 7.52 with SAP_UI 752* on the SAP Fiori Launchpad overview page, and search for **`Enabling Personalization of Object Pages (Experimental)`**



For more information about personalizing object pages in SAP Fiori apps using SAPUI5 flexibility, open the documentation for **SAP Fiori Launchpad in SAP NetWeaver AS for ABAP 7.52 with SAP_UI 752** on the SAP Fiori Launchpad overview page, and search for **`Personalizing Apps (Experimental)`**.

Key users can adapt apps, which can then be used by all users, directly in the SAP Fiori launchpad

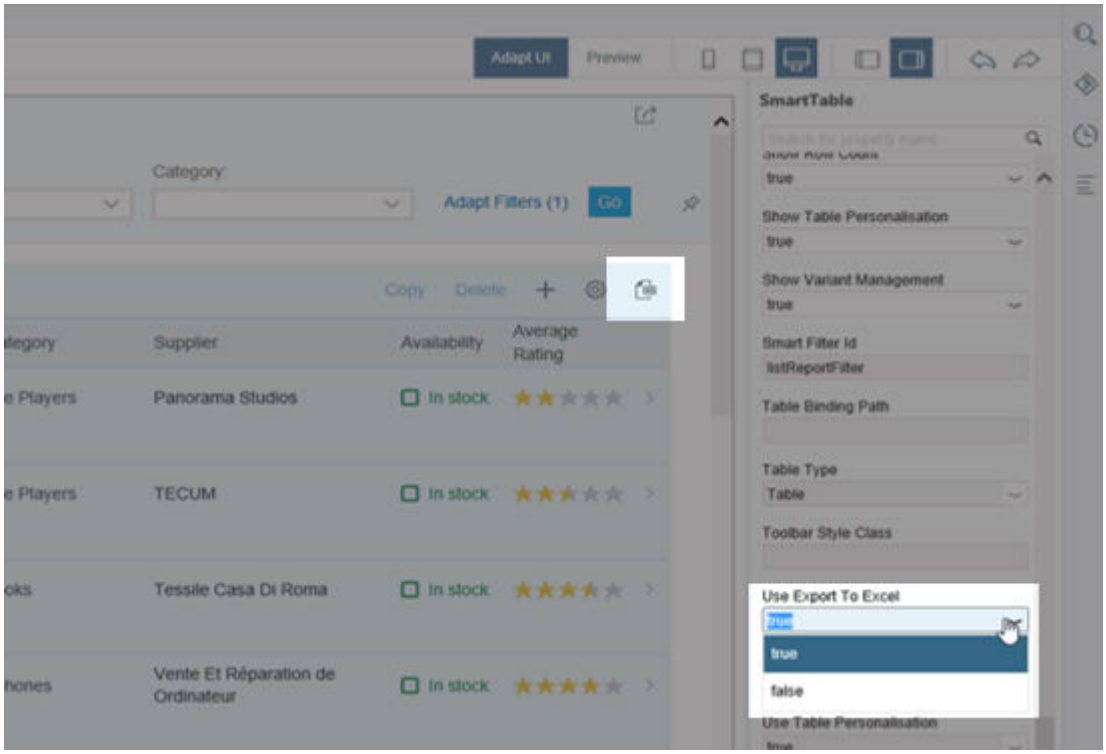
Here's a typical scenario: A team lead who has business knowledge, but probably no technical or development skills, wants to adapt an app for all users of the app. In the context of SAPUI5 flexibility, we call this team lead a *key user*. Let's assume, the users would need to see the supplier number in addition. Using the *key user adaptation* feature of SAPUI5 flexibility, the key user just starts the adaptation mode and changes the user interface using intuitive WYSIWYG functions. So, for example, it's very easy to rearrange UI elements using drag and drop or to add fields to the user interface. The supplier number would be added in no time. After releasing the adapted version of the app, it's available for all users working with the app.

For more information about adapting the user interface of SAP Fiori apps using the key user adaptation feature of SAPUI5 flexibility, open the documentation for **SAP Fiori Launchpad in SAP NetWeaver AS for ABAP 7.52 with SAP_UI 752** on the SAP Fiori Launchpad overview page, and search for **`Adapting SAP Fiori UIs at Runtime`**. If you use SAP Cloud Platform Portal, see Adapting the UI at Runtime in the *SAP Cloud Platform Portal* documentation.

Administrators have to enable key users to be able to use key user adaptation. For more information, search for **`Enabling UI Adaptation at Runtime`** in the documentation for SAP Fiori launchpad in SAP NetWeaver AS for ABAP 7.52 with SAP_UI 752 in the SAP Fiori Launchpad overview page on the SAP Help Portal.

Key user adaptation - minimal effort for developers, maximum benefit for customers

What has to be considered when developing apps that support key user adaptation? In a nutshell: It's all about using the supported controls and stable IDs. For more information, see SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]. If you've developed your app based on SAP Fiori elements, you can make changes to the user interfaces using the intuitive SAPUI5 Visual Editor in SAP Web IDE, for example hiding the *Export to Excel* button.



For more information, see SAPUI Visual Editor in the *SAP Web IDE Full-Stack* guide.

> → Tip
>
> Not all SAP Fiori apps support key user adaptation. How to find out whether an app does? Either contact the developers of the app or check whether it uses the controls supported by key user adaptation as well as stable IDs. To do so, access the diagnosis window and choose *Control Tree*. For more information, see Diagnostics [page 1326].

Related Information

Layering Concept [page 1156]
Example: Layering of UI Changes [page 1157]
Extending Apps [page 2143]
Stable IDs: All You Need to Know [page 1442]
SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]

Layering Concept

SAPUI5 flexibility uses a consistent layering concept to store the UI changes as semantic delta information. This layering concept applies consistently to all users of SAPUI5 flexibility (end users, key users, and developers).

The delta changes are stored in a repository, called *layered repository*, as it contains different layers where the UI changes of the different users are stored in respective layers. Here's an overview:

Layer	Used by	Stores changes by	Type of changes
`USER`	Customer	End users	User-specific personalization settings (for object pages)
`CUSTOMER`	Customer	Key users	UI changes to adapt apps for all users made using key user adaptation
		End users	Views that the end user saves as *Public*
`CUSTOMER_BASE`	Customer	Developers	UI changes made using the SAPUI5 Visual Editor editor in SAP Web IDE
`VENDOR`	SAP	SAP	Example: Update of an app

The semantic changes are attached to stable IDs. This makes them upgrade-safe, for example, if the controls of the app get exchanged.

How are the layered changes stored?

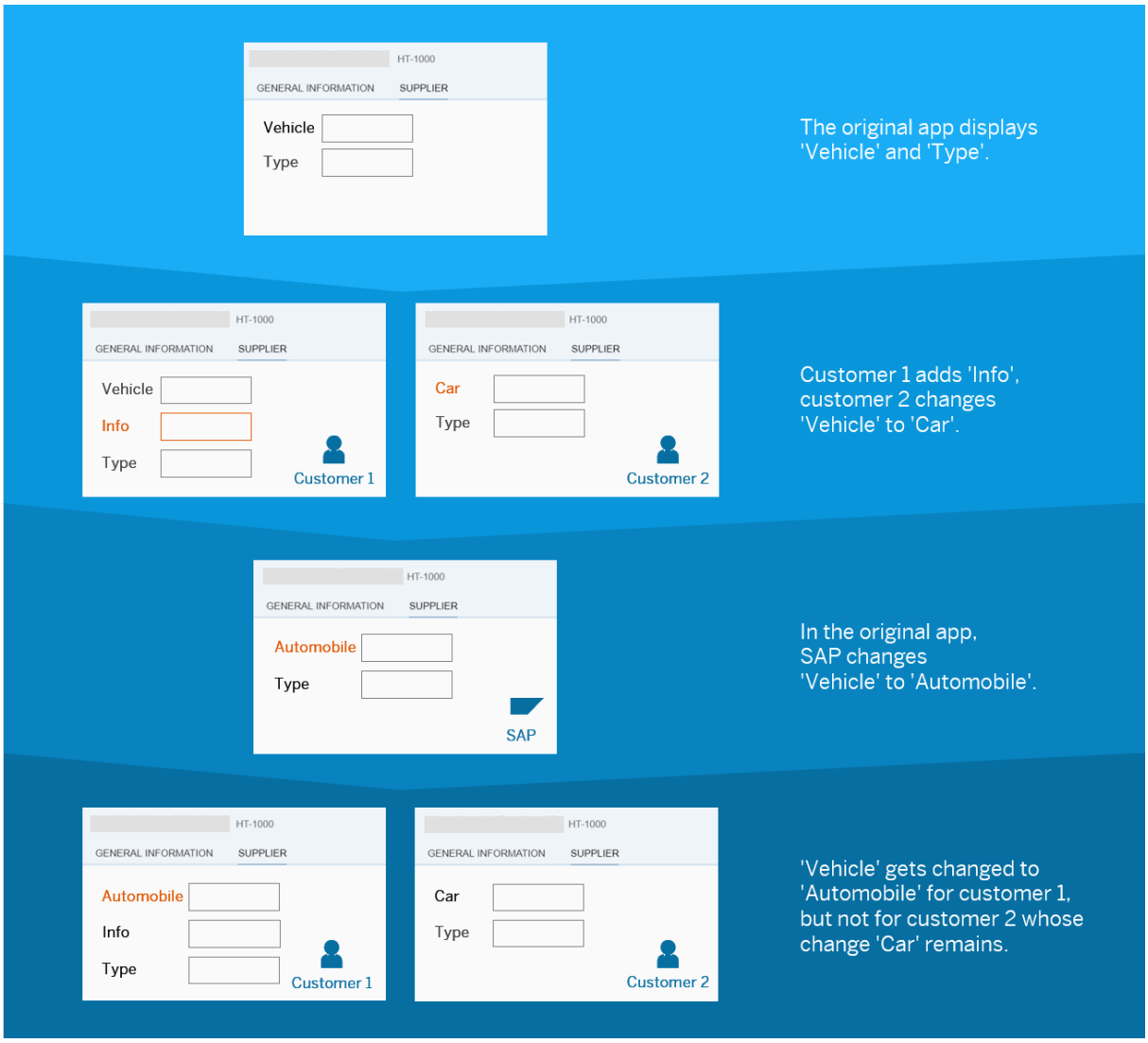
The changes are stored in the respective layers separately from the original content that remains unchanged. The repository stores the logical information for the changes that are to be applied to the original entity in a JSON-based file. The repository calls the client API to create, update, and delete these changes and calls the REST services to update the back-end system.

Related Information

SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152]
Example: Layering of UI Changes [page 1157]
SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]
Stable IDs: All You Need to Know [page 1442]

Example: Layering of UI Changes

Here's an example of how the layering of UI changes based on SAPUI5 flexibility works.



The original app displays the fields *Vehicle* and *Type*.

Using SAPUI5 flexibility, customer 1 adds the additional field *Info* and customer 2 renames the existing field *Vehicle* to *Car*. After the app was shipped to the customers, SAP changes *Vehicle* to *Automobile* in the original app.

Applying the changes based on the layering concept, the customers would now get the following:

- In the app of customer 1, *Vehicle* would be replaced by *Automobile*. The *Info* field added by the customer would also be applied.
- In the app of customer 2, the change made by SAP (*Vehicle* renamed to *Automobile*) would not be applied. Reason: customer 2 renamed this field to *Car* and changes made by customers overrule changes made by SAP.

Bootstrapping SAPUI5 Flexibility

You can define an alternative route from where to load SAPUI5 flexibility.

If you want SAPUI5 flexibility to be loaded from the default location as part of the SAPUI5 bootstrap, you don't need to do anything.

If you'd like SAPUI5 flexibility to be loaded from an alternative location, use the configuration parameter `flexibilityServices="</path/to/alternative/location>"`.

If you'd like to disable SAPUI5 flexibility during SAPUI5 bootstrap, set the parameter value to an empty string.

You can get the value of the parameter using the method `getFlexibilityServices` of class `sap.ui.core.Configuration`.

For more information on how to set a configuration parameter as well as how to retrieve its current value, see sap.ui.core.Configuration.

Testing

SAPUI5 provides several testing options, like to unit and integration tests and the mock server.

Before you start implementing your first test, you should think about how to test the different aspects of your application. The image below shows some examples of testing tools along the agile testing pyramid.

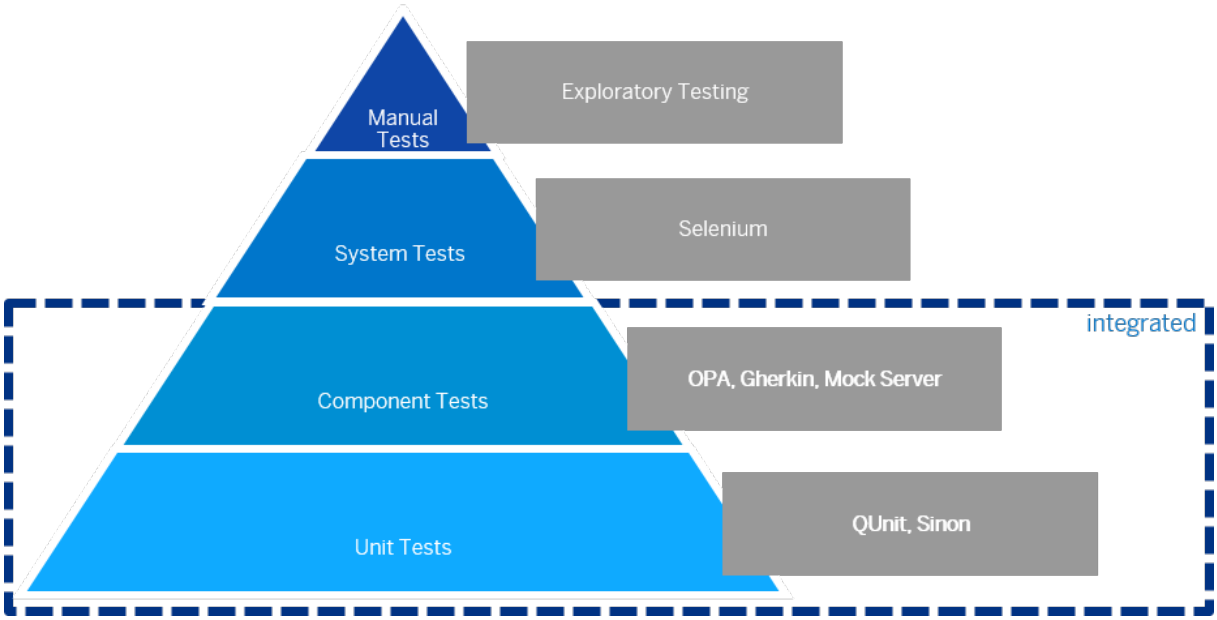


Figure 220: Testing pyramid

You can use a local test runner, such as Selenium or Karma, that automatically executes all tests whenever a file in the app project has been changed.

Related Information

Tutorial: Testing [page 368]
Continuous Integration: Ensure Code Quality [page 1398]
Selenium Home Page
Karma Home Page

Unit Testing with QUnit

QUnit is a powerful, easy-to-use JavaScript unit testing framework. It is used by the jQuery, jQuery UI and jQuery Mobile projects and is capable of testing any generic JavaScript code. It supports asynchronous tests out-of-the-box.

> **i Note**
>
> Before you begin setting up a QUnit test environment, read the background information and introduction to the QUnit test API itself, which is available on the external web site `http://api.qunitjs.com/`. This official QUnit documentation features a complete description of the QUnit test API and contains many examples.

Why Does SAPUI5 Use QUnit Tests?

QUnit tests provide good support for asynchronous testing. These types of tests are often needed for UI functional tests, for example if you have to wait until rendering is done, animations are complete, or a backend call returns. In addition, a QUnit test page can be executed standalone in the browser without the need of an additional "tool". This makes the creation and execution of single QUnit tests much easier. Finally, QUnit is closely related to jQuery, which is also a fundamental part of SAPUI5.

Creating a QUnit Test Page

Prerequisites

As a prerequisite for creating a test, you need to have created a SAPUI5 application (such as `myapp`). Once you have done this, continue with the steps described below.

Creating a Test Page

Create a QUnit test module `myqunittest.qunit.js` in the folder `test-resources/`.

> **i** Note
>
> The file name `XYZ.qunit.js` is a recommendation to clearly indicate that this is a QUnit test. Technically, the `.qunit` name extension is not required.

You can use the file template shown below. This code snippet shows a basic QUnit test template which is used for SAPUI5 control tests.

Each test file represents a UI5 module.

```
/*global QUnit */
sap.ui.define([], function() {
       "use strict";
        QUnit.module("Module A");
        QUnit.test("1. a basic test example", 2, function (assert) {
                assert.ok(true, "this test is fine");
                var value = "hello1";
                assert.equal(value, "hello1", "We expect value to be 'hello1'");
        });
});
```

This QUnit test file does not include the SAPUI5 bootstrap (`sap-ui-core.js`). The test starter ensures that the QUnit tests are loaded within an HTML page.

Writing Test Functions

Write your test code (like in the following example) into the template introduced in the previous section:

```
/*global QUnit */
sap.ui.define(["sap/m/Button", "sap/ui/qunit/QUnitUtils", "sap/ui/qunit/utils/
createAndAppendDiv"], function(Button, QUnitUtils, createAndAppendDiv) {
    "use strict";
    // create content div
    createAndAppendDiv("myContent");
    /* Create e.g. an SAPUI5 control which you need for your tests
       Alternatively you can do this also in the `beforeEach` method of a module
    */
    var oButton = new Button("myButton", {text: "Click me"});
    //...
    oButton.placeAt("myContent");
    /* The QUnit processing starts automatically when the page is
       loaded. If you want to delay the start because of some
       additional preparation work you can use the following utility
       function:
    */
    QUnitUtils.delayTestStart(5000);
    /* The module call can be used to categorize your test functions.
       In addition it is possible to define actions which are processed
       during `beforeEach` and `afterEach`.
    */
    QUnit.module("Module A");
    /* Example for a non-asynchronous test function:
       The first parameter is the name of the test,
       the second (optional) parameter is the number of expected assertions in
the test,
       the third parameter is the test function to call when the tests runs.
    */
    QUnit.test("Test 1", 3, function(assert) {
        assert.ok( true, "this test is fine" );
        var value = "hello1";
        assert.equal( value, "hello1", "We expect value to be 'hello1'" );
        /* You can also do some actions between the assertions,
           like triggering a keydown event with Enter key on the
           Dom element with ID 'myButton' using the utilities.
           Note: The utility function simulates a keyboard event
                 using 'jQuery.trigger'. This is not a 'real'
                 event which comes from the browser and there might
                 be differences you must be aware of: When the
                 user presses the Enter key on a button several
                 events are fired by the browser like keydown, keyup,
                 click, .... The function below ONLY simulates a
                 keydown!
        */
        QUnitUtils.triggerKeydown("myButton", "ENTER");
        assert.ok( true, "another test after the action" );
    });
    /* Modules have a second, optional "lifecycle" parameter. The life cycle
object can
       have two methods - `beforeEach` and `afterEach`. Both methods are called
for each test
       of the module. It is best practice to use those life cycle methods to
have standelone
       tests that do not have dependencies on other tests.
    */
    QUnit.module("Module B", {
        beforeEach: function() {
            // Code needed for the tests of this module
            // this.foo = new Bar();
        },
        afterEach: function() {
```

```
            // Cleanup here
            // this.foo = null;
        }
    });
    /* Example for an asynchronous test function: */
    QUnit.test("Test 2", 3, function(assert) {
        var done = assert.async();
        /* Instead of using the second parameter in the test definition you can
define the number expected assertions in the function body. This is handy, when
you write tests with different outcome. */
        // assert.expect(3);
        /* First you start with tests in the normal flow */
        assert.ok(true, "this test is fine");
        setTimeout(function() {
            assert.ok(true, "this test is executed asynchronously");
            /* Do the asynchrounos tests and give QUnit the sign to go on with
the next test function via 'done' when the processing of the current one is
completed */
            done();
        }, 1000);
        /* Do the things which needs a test delay, e.g. press a button which
starts a backend call */
        QUnitUtils.triggerKeydown("myButton", "ENTER");
        assert.ok(true, "this test is not executed asynchronously");
    });
```

Executing a QUnit Test

Creating a QUnit TestSuite

For running QUnit tests, you need a QUnit TestSuite which configures the environment for the test. You create the QUnit TestSuite as follows:

1. Create a file named `testsuite.qunit.html`:

```
    <!DOCTYPE html>
    <html>
        <head>
            <meta http-equiv="X-UA-Compatible" content="IE=edge">
            <meta charset="utf-8">
            <base href="../../">
            <title>TestSuite myapp</title>
            <script src="resources/sap/ui/test/starter/createSuite.js"
                    data-sap-ui-testsuite="test-resources/testsuite.qunit"></
script>
        </head>
        <body>
        </body>
    </html>
```

2. Create an additional configuration file that contains references to all tests, for example, `testsuite.qunit.js`.

The QUnit test starter is configured with one configuration file per QUnit TestSuite. The file is a standard SAPUI5 AMD module (using `sap.ui.define`) which returns an object with the configuration.

The configuration object is an object with the following top level properties:

```
sap.ui.define(function() {
    "use strict";
    return {
        /*
         * Name of the test suite.
         *
         * This name will be used in the title of the index page / testsuite
page.
         */
        name: "TestSuite for myapp",
        /*
         * An Object with default settings for all tests.
         *
         * The defaults and the test configuration will be merged recursively in
a way
         * that the merge contains properties from both, defaults and test
config;
         * if a property is defined by both config objects, the value from the
test config will be used.
         * There's no special handling for other types of values, e.g an array
value in the defaults
         * will be replaced by an array value in the test config.
         */
        defaults: {
            qunit: {
                version: 2
            }
        },

        /*
         * A map with the individual test configurations, keyed by a unique test
name.
         *
         * There's no technical limitation for the length or the characters of
the test names.
         * The will be used only in the overview page showing all tests of your
suite.
         *
         * But by default, the name is also used to derive the ID of the module
that contains the test cases.
         * It is therefore suggested to use module ID like names (no blanks, no
special chars other than / or dot)
         * If you have multiple tests that execute the same module but with
different configurations
         * (e.g. different QUnit versions or different URL parameters), you have
to make up unique names
         * and manually configure the module IDs for them.
         */
        tests: {
            /*
             * A test named 'myqunittest'.
             * By default, it will require the module 'myqunittest.qunit'
             * assuming that your testsuite configuration is stored in
testsuite.qunit.js.
             */
            myqunittest: {
                  title: "My QUnit test for myapp"
            }
        }
    };
});
```

Starting the QUnit TestSuite

After creating the QUnit TestModule, you can easily run this test without any tool in any browser by just using the URL of the QUnit TestSuite page, for example `http://localhost:8080/myapp/test-resources/testsuite.qunit.html`. This executes the test and informs you about its success or shows you any errors.

Migrating Existing HTML-based Testing Suites

The migrating of existing HTML-based QUnit tests to the new QUnit test starter framework brings several benefits:

- Code reduction by removing duplicated HTML environment code
- Separation of concerns: test js code, html environment code, configuration code
- CSP compliance: No inline JavaScript execution

Extract Configuration from QUnit TestSuite and Tests

The list of configured test pages as defined in the existing `testsuite.qunit.html` has to be transformed to the new configuration format described in the Configuration section.

For each individual test page, the necessary configuration has to be extracted from the page itself: the QUnit version and Sinon version that is used, if a Bridge (Sandbox) is used and the options that are defined for the UI5 Core or for Code coverage etc.+

The configuration has to be stored in a new AMD module that has the same name as the QUnit TestSuite but ending with '.js' instead of '.html', for example, `testsuite.qunit.js`.

Make `testsuite.qunit.html` Use the Externalized Configuration

The `testsuite.qunit.html` must be re-written to use the above mentioned `createSuite.js` script and to read the new configuration:

`testsuite.qunit.html`

```
<!DOCTYPE html>
<html>
    <head>
        <!-- the usual headers -->
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta charset="utf-8">
        <!-- it is suggested to use a base tag pointing to the application root,
             as it normalizes URLs across test suites, but this is not mandatory
-->
        <base href="../../../../../../../../">
        <!-- include the generic script that creates the TestRunner suite from
the configuration
             The name of the suite is specified in a separate attribute data-sap-
ui-testsuite.
             Note that this attribute value is a UI5 resource name (no .js at
the end, no relative name).
             The configuration will be read using sap.ui.require. The test
starter will take care to
             register a path for prefix 'test-resources/'.
             -->
        <script src="resources/sap/ui/test/starter/createSuite.js"
```

```
                data-sap-ui-testsuite="test-resources/sap/ui/core/qunit/test/
starter/testsuite.starter.qunit"></script>
    </head>
    <body>
    </body>
</html>
```

(Semi-)Automation

The application `migrate.html` can automate parts of the above work. After starting it, it collects the names of existing QUnit TestSuites. This may take about 60 sec and provides a list of the existing QUnit TestSuites. Enter the name of one QUnit TestSuite. The application reads it, collect the tests, peeks into the individual test files to gather some of the configuration settings, and shows the resulting `testsuite.qunit.html` and `testsuite.qunit.js` in two code editors. You can use the shown code as a starting point for your migration.

Convert Test Pages to AMD Modules

This step is potentially the biggest effort and it has to be done manually. In order that it can be used with the new testing / testsuite approach, note the following minimal requirements:

- The HTML page is converted to an AMD JS module, using `sap.ui.define`.
- The test configuration (used testing framework components, UI5 Core setup ...) has been added to the external configuration described above
- The new module creates all QUnit tests on execution, not later. It does **not** call `QUnit.start()`. If, for some reason, the module cannot fulfill this task on execution, but has to wait for some asynchronous task, the test option `autostart` can be set to `false` and the test module can call `QUnit.start()` at an appropriate point in time. Note: Do not mix the test option `autostart` with the QUnit option `autostart`.

The following requirements that are optional:

- The test code is fully cleaned up and no longer uses globals, only AMD references.
- The test code no longer uses sync APIs.

> **i** Note
>
> This will become a mandatory requirement for CSP level 2 policy as sync code loading always requires a kind of `eval`.

Test Suite Configuration Options

Both, the defaults and the individual test configurations have the same structure, which is documented in the following code snippet. The snippet also shows the internal defaults of the test starter. They are used as a fallback for options that are not defined in the configuration file (neither defaults, nor individual tests).

```
{
    /*
     * ID(s) of the module(s) to load.
     *
     * Can either be a single string or an array of strings.
     * Each string can use the following placeholders
     *  leading "./" - package name of the testsuite configuration
     *        {name} - name of the current test
     */
    module: "./{name}.qunit",
```

```
    /*
     * URL of the test page to start for this test.
     *
     * By default, all tests use the generic starter page which reads the suite
     * configuration, finds the tests and starts the configured test components
     * before it requires and executes the configured test module(s).
     *
     * The URL must be relative to the application root and can use the following
     * placeholders, enclosed in curly braces:
     *       {suite} - name of the testsuite (configuration)
     *        {name} - name of the current test
     */
    page: "resources/sap/ui/test/starter/Test.qunit.html?
testsuite={suite}&test={name}",


    /*
     * Title of the test.
     * The URL must be relative to the application root and can use the following
     * placeholders, enclosed in curly braces:
     *       {suite} - name of the testsuite (configuration)
     *        {name} - name of the current test
     */
    title: "QUnit tests '{name}' of suite '{suite}'",


    /*
     * QUnit configuration.
     *
     * Either can be a null or false or an object with the properties documented
below.
     * The values null and false are equivalent to the object { version: null }
     */
    qunit: {
        /*
         * Version of QUnit that should be loaded.
         * If set to a null, QUnit won't be loaded.
         * If set to "edge", the newest available version of QUnit will be used.
         * If set to a number, the corresponding version of QUnit will be used
if supported.
         * Currently supported versions are 1 and 2, an error will be thrown for
unsupported versions.
         */
        version: "edge",

        /*
         * Most statically configurable options from QUnit.config can be
configured,
         * e.g. reorder, blocking etc.
         * Note that 'autostart' is an exception. To avoid timing issues with
asynchronous test
         * loading, 'autostart' will always be set to false. Only after all
tests have been loaded,
         * QUnit.start() will be called, either by the generic test starter or
by the test module itself,
         * see the general test option 'autostart' below.
         */
        // reorder: true // only serves as an example, not part of the internal
defaults of the starter
    },

    /*
     * Sinon configuration.
     *
     * Either can be a null or false or an object with the properties documented
below.
     * The values null and false are equivalent to the object { version: null }
```

```
     */
    sinon: {

        /*
         * Version of Sinon that should be loaded.
         * If set to null, Sinon won't be loaded.
         * If set to "edge", the newest available version of Sinon will be used.
         * If set to a number, the corresponding version of Sinon will be used
if supported.
         * Currently supported are versions 1 and 4, an error will be thrown for
unsupported versions.
         */
        version: "edge",

        /*
         * Whether one of the sinon-qunit bridges will be loaded.
         * When set to true, the sap/ui/thirdparty/sinon-qunit bridge will be
loaded for Sinon 1
         * and the sap/ui/qunit/sinon-qunit-bridge will be loaded for newer
versions of Sinon.
         *
         * The bridge will only be loaded after both, QUnit and Sinon have been
loaded.
         * If either QUnit or Sinon are not loaded, no bridge will be loaded.
         *
         * If Sinon is not loaded, but QUnit, the bridge will not be loaded, but
a shim
         * with dependencies will be configured. This allows tests to load
Sinon / the bridge on
         * their own without taking care of the bridge dependencies.
         */
        qunitBridge: true,


        /*
         * Any other statically configurable Sinon option can be specified as
well.
         * Note that they only play a role when a sandbox is used.
         */
        useFakeTimers: false,
        useFakeServer: false
    },


    /*
     * Code coverage options.
     * The qunit-coverage module will always be loaded after QUnit has been
loaded to enable the coverage
     * option. When the 'coverage' parameter is set in the URL (e.g. because the
coverage checkbox has been
     * clicked), then blanket will be loaded before qunit-coverage to avoid
synchronous loading of it.
     */
    coverage: {
        only: null,
        never: null,
        branchTracking: false
    },
    /*
     * UI5 runtime configuration options.
     *
     * All properties will be copied to window["sap-ui-config"].
     * If window["sap-ui-config"] doesn't support it or if the value is of a type
     * not supported for window["sap-ui-config"], executing the UI5 Core might
fail.
     *
     * Only exception for now: the libs property can be an array of library
names,
```

```
     * not only a comma separated string.
     *
     * To ease test development, the following defaults are defined by the test
starter:
     */
    ui5: {
        bindingSyntax: 'complex',
        noConflict: true,
        libs: [],
        theme: "sap_belize"
    },


    /*
     * Whether the UI5 Core (sap/ui/core/Core.js) should be required and booted.
     *
     * When this option is true, the Core is not only loaded and started, but
loading and execution
     * of the test module(s) is also delayed until a listener registered with
sap.ui.getCore().attachInit()
     * has been executed.
     */
    bootCore: true,


    /*
     * Whether the test starter should call QUnit.start() after all
prerequisites have been fulfilled
     * (e.g. QUnit, Sinon, a bridge, have been loaded, coverage tooling has been
loaded and configured,
     * the Core has been booted, the test modules have been loaded and executed).
     */
    autostart: true,


    /*
     * Whether the test starter should skip a test file. Such tests will remain
in the overview list,
     * but won't be executed in the test suite.
     */
    skip: false
};
```

Code Coverage Measurement

You can measure the code coverage for your test inside the `Control.qunit.html` page either via HTML or JavaScript code using `Blanket.js`.

HTML

With the following line you enable `Blanket.js` to measure the code coverage:

```
<script type="text/javascript" src="../../../../../resources/sap/ui/qunit/qunit-
coverage.js"></script>
```

With this argument, all files that are executed during the test run are added to the result.

If you want to limit the test run, you can use the following code:

- Limit test to a single file:

```
<script type="text/javascript" src="../../../../../resources/sap/ui/qunit/
qunit-coverage.js"
    data-sap-ui-cover-only="sap/ui/core/Popup.js"
></script>
```

- Limit test to multiple files (provide an array with comma-separated sources that should occur in the result):

```
<script type="text/javascript" src="../../../../../resources/sap/ui/qunit/
qunit-coverage.js"
    data-sap-ui-cover-only="[sap/ui/core/Popup.js, sap/ui/core/EventProvider]"
></script>
```

- Limit test to a specific library:

```
<script type="text/javascript" src="../../../../../resources/sap/ui/qunit/
qunit-coverage.js"
    data-sap-ui-cover-only="sap/ui/core/"
></script>
```

- Exclude specific objects:

```
<script type="text/javascript" src="../../../../../resources/sap/ui/qunit/
qunit-coverage.js"
    data-sap-ui-cover-never="sap/m/"
></script>
```

JavaScript

Inside your test page, you can add these lines before running the tests:

```
sap.ui.require(["sap/ui/qunit/qunit-coverage"], function(/*coverage*/){
    // code
});
```

If you want to limit the test run, you can use the following code:

- Limit test to a single file:

```
if (window.blanket) {
    blanket.options("sap-ui-cover-only", "sap/ui/core/Popup.js");
}
```

- Limit test to multiple files (provide an array with comma-separated sources that should occur in the result):

```
if (window.blanket) {
    blanket.options("sap-ui-cover-only", "[sap/ui/core/Popup.js, sap/ui/core/
EventProvide]");
}
```

- Limit test to a specific library:

```
if (window.blanket) {
    blanket.options("sap-ui-cover-only", "sap/ui/core/");
```

```
}
```

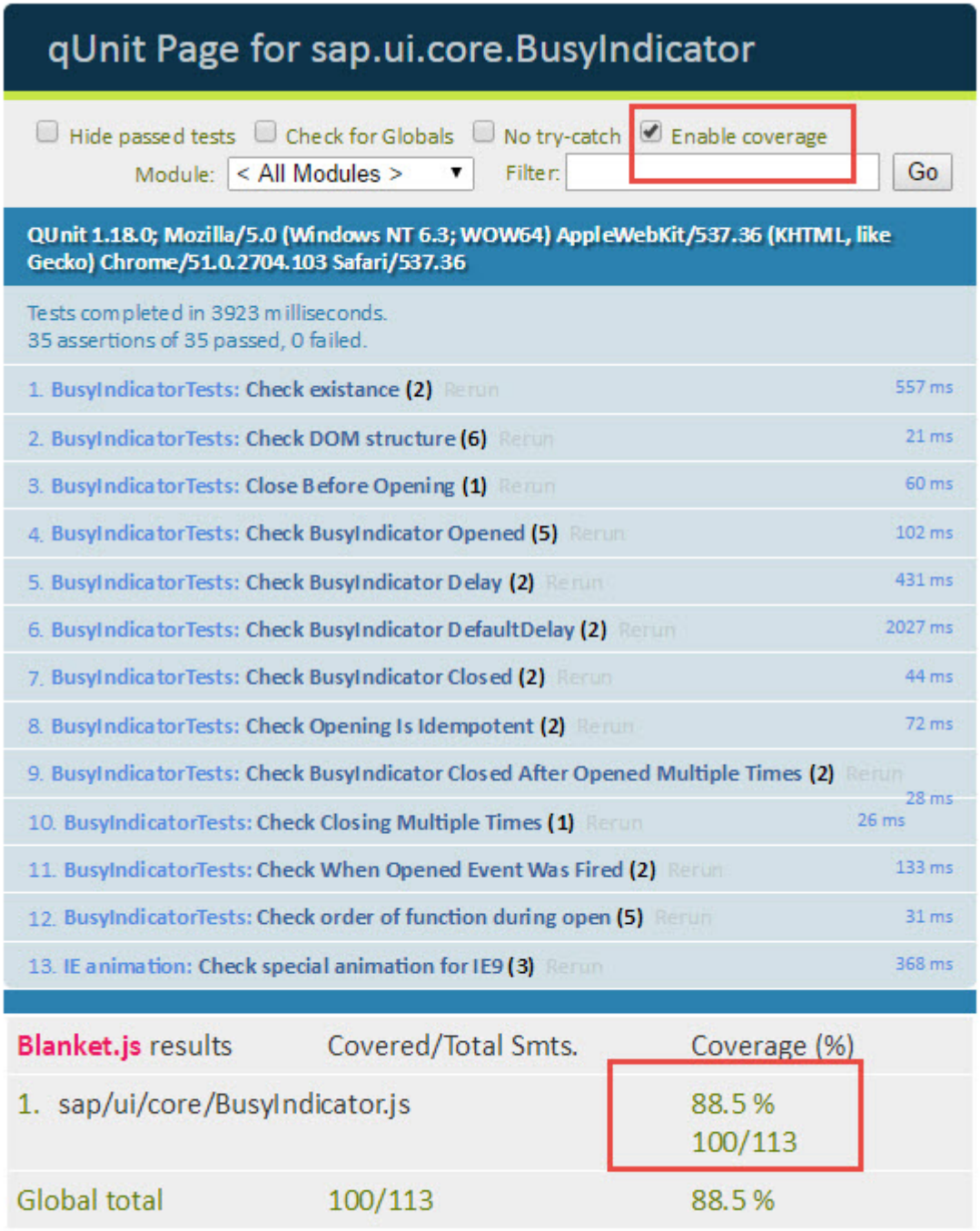
- Exclude specific objects:

```
if (window.blanket) {
    blanket.options("sap-ui-cover-never", "sap/ui/example/thirdparty/");
}
```

Results

To view the results of the measurement, select the *Enable coverage* checkbox on the test page. This will trigger a new test run.

In this example the coverage is limited to one specific file - the only one that is important for this test.



Related Information

More information about `Blanket.js`

Sinon.JS: Spies, Stubs, Mocks, Faked Timers, and XHR

By integrating Sinon.JS for QUnit, you can use spies, stubs, mocks, faked timers or faked XHR. For more information about using sinon.js, see the official documentation at `http://sinonjs.org/docs/`.

All you have to do is add a sinon section to the test suite configuration as shown below:

```
sap.ui.define(function() {
    "use strict";

    return {
        name: "TestSuite for myapp",
        defaults: {
            qunit: {
                version: 2
            },
            sinon: {
                version: 4,
                qunitBridge: true,
                useFakeTimers: false
            }
        },
        // ...
    };
});
```

The variable `sinon` is now globally available in your test.

The following examples show you the basic way in which Sinon.JS can be used. These examples are adapted from the official Sinon.JS documentation available at `http://sinonjs.org/docs/`:

A simple spy test:

```
/*global QUnit sinon */
sap.ui.define(["sap/m/Button"], function(Button) {
     "use strict";
     QUnit.test("Spy", 2, function(assert) {
         var callback = sinon.spy();
         var oButton = new Button();
         oButton.attachPress(callback);
         assert.ok(!callback.called, "Callback Spy not called yet");
         oButton.firePress();
         assert.ok(callback.called, "Callback Spy called");
         oButton.destroy();
     });
});
```

A simple stub test:

```
/*global QUnit sinon */
sap.ui.define([], function() {
     "use strict";

     QUnit.test("Stub", 1, function(assert) {
         sinon.stub(jQuery, "ajax").yieldsTo("success", [1, 2, 3]);

         jQuery.ajax({
             success: function (data) {
                 assert.deepEqual(data, [1, 2, 3], "Right data set");
             }
         });
         jQuery.ajax.restore();
     });
```

```
})
```

A simple mock test:

```
/*global QUnit sinon */
sap.ui.define([], function() {
     "use strict";
     QUnit.test("Mock", 2, function(assert) {
         var myAPI = { method: function () {} };

         var mock = sinon.mock(myAPI);
        mock.expects("method").once().throws();

         try {
             myAPI.method();
         } catch (exc) {
             assert.ok(mock.verify(), "Mock function called and all expectations
are fullfilled");
         }
         mock.restore();
     });
});
```

A simple faked timer test:

```
/*global QUnit sinon */
sap.ui.define([], function() {
     "use strict";

     QUnit.test("Basic", 1, function(assert) {
         var oClock = sinon.useFakeTimers();
         setTimeout(function() {
             assert.ok(true, "Called without need of async test");
         }, 800);
         oClock.tick(800);
         oClock.restore();
     });
});
```

A simple faked XHR test:

```
/*global QUnit sinon */
sap.ui.define([], function() {
    "use strict";
    QUnit.module("Faked XHR", {
        beforeEach: function() {
            this.xhr = sinon.useFakeXMLHttpRequest();
            var requests = this.requests = [];
            this.xhr.onCreate = function (xhr) {
                requests.push(xhr);
            };
        },
        afterEach: function() {
            this.xhr.restore();
        }
    });
    QUnit.test("Basic", 2, function(assert) {
        var callback = sinon.spy();
        jQuery.ajax("test", {
            success: callback
        });
        assert.equal(1, this.requests.length, "Right number of requests");
        this.requests[0].respond(200, {
            "Content-Type": "application/json"
        }, '[{ "foo": "bar", "bar" : "foo" }]');
```

```
        assert.ok(callback.calledWith([{ "foo": "bar", "bar" : "foo" }]), "Data
is called right");
    });
```

How to Test SAPUI5 Controls with QUnit

Comprehensive overview of QUnit testing for controls.

Dos and Don'ts

- When writing QUnits, always keep your tests atomic.
- Don't rely on the execution of previous tests.
- Don't introduce globals, destroy controls after creating them.
- Only test one single thing.
- When writing a test, always make sure you break it first: don't rely on tests that have never failed!
- Write human readable tests - use descriptive names for variables. Readability is more important than performance. You don't have to write a reuse for everything. It's ok to repeat yourself in unit tests if it helps readability.
- Don't test too many internal aspects: try to test the control like an application or user will use it.
- You have to find a balance between not stubbing / expecting too much of the internal aspects and not doing it at all. If you tightly couple your test to the implementation, maintenance will be a pain.
- If your test is too long, you're squeezing too much stuff into one of your control's functions. Divide the complexity, then your test will be simpler and your productive code will be better.
- Don't test general SAPUI5 functionality. Only test things actually done by your control (see "What Should You Test?" below).
- Never write an `if` in a test. It is a sign that you're either not stubbing correctly or you're testing multiple things in one test.
- Never use the `expect` QUnit statement. You should always write your test in a way that every assertion you set up will be hit 100%.
- Whenever you encounter a Bug/Ticket, start by writing a QUnit that fails first, and *then* fix the code.
- Write your tests as small as possible: don't add a statement that is not needed for the test, such as an ID in the control's constructor properties.
- Use fake timers to avoid as many async tests as possible.
- Don't test the exact same thing multiple times.
- Use modules for grouping your tests: this will give you a better organizational test setup.
- You may use modules for `beforeEach/afterEach`, but don't overuse this feature. If you have a longer module, you might not see what the test does because you don't know its setup.
- It's sometimes better to write code multiple times.
- Don't set up your system being tested in the `beforeEach/afterEach`. It is very rare that all tests in a module have the same constructor. Furthermore, using a global constructor object is dangerous.

If you stick to these rules, you will find it much easier to refactor/maintain your tests. Keeping the tests atomic will make debugging much easier, because you will hit your breakpoints for the code being tested only. If you

write QUnits without keeping to these rules, you may well not notice anything bad to begin with, but you **will** eventually end up in the middle of a maintenance nightmare!

Arrange Act Assert Pattern

Internally, we use three templates for testing. The one shown below is the general control template.

Use the following pattern to structure your tests. If everyone sticks to this same pattern, you will be able to read your colleagues' tests very quickly:

```
QUnit.test("Should do Something", function (assert) {
    // Arrange

    // System under Test
    var oMyControl = new nameSpace.myControl({
    });

    // Act

    // Assert
    // Cleanup
    oMyControl.destroy();
});
```

Arrange

In `Arrange`, you should set up the dependencies and options you need for your `System under Test`.

Examples:

- The constructor object of your control
- Sinon spies/stubs and mocks (dependencies of your System under Test)
- Model

System under test

In `System under Test`, you should create your control and you should also render it if you want to test the rendering.

Act

Ideally, this part is only one single line of code executing the function you want to test.

Assert

This part may contain multiple statements of QUnit assertions, but ideally not too many in total.

Make sure that you also test negative paths, not only the expected ones.

Optional: Cleanup

Here you should destroy all the controls/models you created.

If you don't use Sinon sandboxes, revert all the spies/stubs/mocks.

What Should You Test?

- Test all the public functions you introduced.
- Test all the overwritten getters and setters.
- Test your control's events and how often they are called.
- Test all possible user interactions (tap, keyboard, focus).
- You could test how often your control gets rerendered when interacting with it, but only if you are worried that it might be rerenderd too often or not at all.
- Test RTL if you have special things done in javascript.
- Write some integration tests if you have a composite control (don't cover 100% of your child controls - that's overkill and child controls will be hard to maintain).
- You may test default values of properties, since we cannot change them backwards afterwards and a test will recognize this.
- Test how your control interacts with models (OData + Json).
- Test the destruction of your control when working with composites, test if all dependencies/events are unbound on destruction.

What Should You NOT Test?

- Never test non-overwritten getters and setters (these are tested in the core of the framework).
- Never test your complete CSS with computed styles: just check if the classes are set correctly. Focus on testing JavaScript.
- Never test other generic framework functionality. Focus on your control.

Rendering Tests

In the rendering tests part, you have to place your control in the DOM. The best place to put it is the `qunit-fixture` div, since its content gets deleted after every test.

Make sure you destroy your control, since SAPUI5 will keep a reference to it and may also rerender it.

It's crucial that you call `sap.ui.getCore().applyChanges()` after each time you have caused a rerendering.

The call to this function synchronizes the changes of your control with the DOM. If you do not make this call, the DOM will not be updated.

You can use the following template to make sure that you don't forget to destroy your control:

```
QUnit.test("Should do Something", function(assert) {
    // Arrange
        var oContructor = {

    };

    // System under Test
    var oMyControl = new nameSpace.myControl(oContructor);
```

```
    oMyControl.placeAt("qunit-fixture");
    sap.ui.getCore().applyChanges();
    // Act

    // Assert
    // Cleanup
    oMyControl.destroy();
});
```

Pitfalls

Sinon fake timers

If you are using `sinon.qunit`, it will automatically use fake timers by itself. Fake timers will prevent any `setTimeout/setIntervall` function from being executed, unless you call `this.clock.tick(milliseconds)` in your test. This means that a Mock Server with auto-respond will not respond and OPA will not be able to wait for controls.

In addition, control events might be fired inside of a `setTimeout(, 0)`, so the event might not be triggered at all.

Testing SAPUI5 control events with Sinon

If you want to test SAPUI5 events, you can use spies to test how often they are called. If you try to test the parameters, however, you cannot do this with spies as SAPUI5 uses an eventPool that reuses the same object again. This means that after an event is set, all of the parameters will be deleted, Sinon will keep a reference to the object without properties.

The effect of this is that you cannot assert on them anymore. The workaround is to use a stub with a custom implementation that saves a copy of the parameters to your test function scope.

An example of this is shown in the cookbook below (events).

I've set a property on my control: Why aren't the changes in the DOM?

The most likely reason for this is that `sap.ui.getCore().applyChanges()` was not called. SAPUI5 does not render synchronously, but calling this function will render immediately.

Cookbook for Testing Controls with QUnit

Test Cases

You can use a factory function. To keep this pointer and have a descriptive message, you should use the test inside of the function and pass a test name to it.

Internally, we prefer to pass an object to the test for retrieving the values - it makes the test cases readable.

```
// "Bar" required from module "sap/m/Bar"
```

```
// "Core" required from module "sap/ui/core/Core"
function renderBarInPageTestCase(sTestName, oOptions) {
    QUnit.test(sTestName, function (assert) {
        // System under Test
        var oBar = new Bar();
        oBar.placeAt("qunit-fixture");
        // Act
        oBar.applyTagAndContextClassFor(oOptions.context);
        Core.applyChanges();
        // Assert
       assert.strictEqual(oBar.getDomRef().nodeName,
oOptions.expectedTag.toUpperCase());
       assert.ok(oBar.$().hasClass(oOptions.expectedClass), "The bar has the
context class: " + oOptions.expectedClass);
        // Cleanup
        oBar.destroy();
    });
};
renderBarInPageTestCase("Should render the header context", {
    context : "header",
    expectedTag : "header",
    expectedClass : "sapMHeader-CTX"
});
renderBarInPageTestCase("Should render the header context", {
    context : "subheader",
    expectedTag : "header",
    expectedClass : "sapMSubHeader-CTX"
});
renderBarInPageTestCase("Should render the header context", {
    context : "footer",
    expectedTag : "footer",
    expectedClass : "sapMFooter-CTX"
});
```

Testing Control Events

You cannot test for event parameters in SAPUI5 so you have to record them. Nevertheless, you can still use Sinon to retain the spy's call counting capabilities. Here is a working example for this:

```
// "HashChanger" required from module "sap/ui/core/routing/HashChanger"
QUnit.test("Should set the Hash", function(assert) {
    //Arrange
    var aCalls = [],
        fnHashChanged = function(oEvt) {
                aCalls.push({ newHash : oEvt.getParameter("newHash"), oldHash :
oEvt.getParameter("oldHash") });
        },
        oSpy = this.spy(fnHashChanged);

    //System under Test
    var oHashChanger = new HashChanger();
    oHashChanger.init();
    oHashChanger.attachEvent("hashChanged", oSpy);
    //Act
    oHashChanger.setHash("one", true);
    oHashChanger.setHash("two");
    //Assert
    assert.strictEqual(oSpy.callCount, 2, "did change the Hash two times");
    assert.strictEqual(aCalls[0].newHash, "one", "first event was correct");
    assert.strictEqual(aCalls[1].newHash, "two", "second event was correct");

    //Cleanup
    oHashChanger.destroy();
```

```
});
```

Testing User Interactions

When testing user interactions, you can use `sap.ui.test.qunit` to trigger events.

Here is an example for when a user presses `Esc` on the select:

```
// "Item" required from module "sap/ui/core/Item"
// "Select" required from module "sap/m/Select"
// "KeyCodes" required from module "sap/ui/events/KeyCodes"
// "Core" required from module "sap/ui/core/Core"
// "QUnitUtils" required from module "sap/ui/qunit/QUnitUtils"
QUnit.test("Should close the popup menu if it is open and you press escape",
function(assert) {
    // Arrange
    var oContstructor = {
        items: [
        new Item({
            key: "0",
            text: "item 0"
        }),
        new Item({
            key: "1",
            text: "item 1"
        })
       ]
    };
    // System under test
    var oSelect = new Select(oContstructor);
    oSelect.placeAt("select-content");
    Core.applyChanges();
    // Arrange after rendering
    oSelect.focus();
    var fnEscapeSpy = this.spy(oSelect, "onsapescape");
    var fnCloseSpy = this.spy(oSelect, "close");
    // Act
    QUnitUtils.triggerKeydown(oSelect.getDomRef(), KeyCodes.ESCAPE);
    // Assertion
    assert.strictEqual(fnEscapeSpy.callCount, 1, "onsapescape() method was
called exactly once");
    assert.strictEqual(fnCloseSpy.callCount, 0, "close() method is not called");
    // Cleanup
    oSelect.destroy();
});
```

Testing the Re-rendering

In this example, you will test to see whether the control fails to rerender. The control has overwritten the setter of the tooltip property to avoid triggering a re-rendering.

To test this, we add an `eventDelegate` to see how often the rendering function is called. We need to make sure that we apply the changes after setting the property because we want SAPUI5 to render synchronously:

```
// "Label" required from module "sap/m/Label"
// "Core" required from module "sap/ui/core/Core"
```

```
QUnit.test("Should suppress rerendering when tooltip is set", function(assert) {
    // Arrange
    var oContructor = {
        tooltip : "foo"
        };
    var oRerenderingSpy = this.spy();
    // System under Test
    var oLabel = new Label(oContructor);
    oLabel.placeAt("qunit-fixture");
    Core.applyChanges();
    oLabel.addEventDelegate({
        onBeforeRendering : oRerenderingSpy
    });
    // Act
    oLabel.setTooltip("bar");
    Core.applyChanges();
    // Assert
    assert.strictEqual(oRerenderingSpy.callCount, 0, "Did not rerender");
    assert.strictEqual(oLabel.getTooltip(), "bar", "Tooltip property got set");
    assert.strictEqual(oLabel.$().attr("title"), "bar", "Tooltip got updated");
    // Cleanup
    oLabel.destroy();
});
```

Testing with Models

When testing with models, you need to make sure that you also setup/destroy your model inside a test itself. OData tests will always be integration tests, since you will require multiple files in order to use the mock server. You may use a factory method to do this.

An example for setting up the mock server is shown below:

```
// "MockServer" required from module "sap/ui/app/MockServer"
function startMockServer(iRespondAfter) {
    // configure respond to requests delay
    MockServer.config({
        autoRespond : true,
        autoRespondAfter : iRespondAfter || 10
    });
    // create mockserver
    var oMockServer = new MockServer({
        rootUri : "http://sap.com/service/"
    });
    // start and return
    oMockServer.simulate("data/metadata.xml", "data");
    oMockServer.start();
    return oMockServer;
}
//Your test:
QUnit.test("Should do something with the model", function (assert) {
    //Arrange
    var oMockServer = startMockServer(0),

    // System under Test + Act
    //Cleanup
    oMockServer.stop();
});
```

When using the mock server, you can use async tests since calling respond each time on the mock server does not help the readability of the test.

After setting up the mock server, we set up the model as follows:

```
// "ODataModel" required from module "sap/ui/model/v2/ODataModel"
// "jQuery" required from module "sap/ui/thirdparty/jquery"
function createODataModel(sURL, mSettings) {
    sURL = sURL || "http://sap.com/service/";
    var oModel = new ODataModel(sURL);

    mSettings = mSettings || {};
    jQuery.each(mSettings, function(sProperty, vValue) {
        sProperty = sProperty[0].toUpperCase() + sProperty.substring(1);
        oModel["set" + sProperty](vValue);
    });

    return oModel;
}
//Your test:
QUnit.test("Should do something with the model", function(assert) {
    // Arrange
    var oModel = createODataModel(),
    oMockServer = startMockServer(0),
    done = assert.async();
    // System under Test + Act + call done();
    // Cleanup
    oModel.destroy();
    oMockServer.stop();
});
```

You can now bind your model against your control and test whatever you want.

We use `clock.tick` to trigger the server response. If you didn't do this, the text of the label would still be empty:

```
// "Label" required from module "sap/m/Label"
// "Core" required from module "sap/ui/core/Core"
//Your test:
QUnit.test("Should do something with the model", function(assert) {
    // Arrange
    var oModel = createODataModel(),
        oMockServer = startMockServer(50);
    // System under Test
    var oLabel = new Label({
        text : "{/myProperty}"
    });
    oLabel.placeAt("qunit-fixture");
    Core.applyChanges();
    // Act - trigger the request
    sinon.clock.tick(50);
    // Assert
    assert.strictEqual("myExpected", oLabel.getText(), "The expected text was
present");
    // Cleanup
    oModel.destroy();
    oMockServer.stop();
    sinon.clock.reset()
});
```

Integration Testing with One Page Acceptance Tests (OPA5)

OPA5 is an API for SAPUI5 controls. It hides asynchronicity and eases access to SAPUI5 elements. This makes OPA especially helpful for testing user interactions, integration with SAPUI5, navigation, and data binding.

The OPA5 library is JavaScript-based. This means that you can write your tests in the same language in which your app is written. This has the following advantages:

- Quick and easy access to JavaScript functions
- Easy ramp-up as it can be used with any JavaScript unit test framework, such as QUnit or Jasmine
- Using the same runtime enables debugging
- Good SAPUI5 integration
- Feedback within seconds makes it possible to execute tests directly after a change
- Asynchronicity is handled with polling instead of timeouts, which makes it faster
- Enables test-driven development (TDD)

Developers write OPA tests during app development. The test-driven development (TDD) results in less fragile tests, because the app is better isolated and supports less fragile APIs for testing:

- It follows the arrange act assert pattern (corresponds to given when then), which improves readability and understanding of the test cases.
- It is easy to run on mobile devices as no plugins/apps are needed; you can as easily just run it in the browser.
- Saves time for the developer as regressions decrease

In short: Writing acceptance tests with OPA5 is very easy – Give it a try!

Limitations of OPA5

Note the following limitations of OPA:

- Screen capturing
- Testing across more than one page
- Remote test execution
- End-to-end tests are not recommended with OPA due to authentication issues and fragility of test data

Getting Started with OPA5

The following section explains step-by-step how to easily write tests for SAPUI5 apps.

We assume a simple app that displays a button on the page after a random time between 0 and 10 seconds. After pressing the button, the text on the button changes. Again, this may take 0 to 10 seconds.

This simulates the behaviour of many SAPUI5 apps: Depending on user actions and model changes, controls change after some time. How can we easily test these SAPUI5 apps without having to write complicated tests that know a lot about the implementation of the app?

Creating an Asynchronous App

First, we create a very simple view with an invisible button with *Press me* as the button text:

```
<mvc:View controllerName="view.Main"
  xmlns="sap.m"
  xmlns:mvc="sap.ui.core.mvc">
  <App>
    <Page>
      <headerContent>
        <Button id="pressMeButton" text="Press me" press="onPress"
visible="false"/>
      </headerContent>
    </Page>
  </App>
</mvc:View>
```

We display the button in the controller after 0 to 10 seconds. On press, we change the text.

```
sap.ui.define(["sap/ui/core/mvc/Controller"], function(Controller) {
    "use strict";
    return Controller.extend("view.Main", {
        onInit: function() {

            var that = this;
            window.setTimeout(function() {
                that.byId("pressMeButton").setVisible(true);
            }, Math.random() * 10000);
        },
      onPress: function() {
            this.byId("pressMeButton").setText("I got pressed");
      }
    });
   }, true);
    // "Controller" required from module "sap/ui/core/mvc/Controller"
   Controller.create({
    name: "view.Main"
});
```

Now how can we test this app without having to do a lot of mocking or writing of cryptic code?

Creating an OPA Test

When we write tests, we try to write it in a way that everyone can immediately understand what is done and tested with this test:

```
sap.ui.require([
  "sap/ui/test/Opa5",
    "sap/ui/test/opaQUnit",
    "sap/ui/test/actions/Press",
    "sap/ui/test/matchers/PropertyStrictEquals"
], function (Opa5, opaQUnit, Press, PropertyStrictEquals) {
  opaQUnit("Should press a Button", function (Given, When, Then) {
        // Arrangements
        Given.iStartMyApp();
        //Actions
        When.iPressOnTheButton();
        // Assertions
        Then.theButtonShouldHaveADifferentText();
```

```
    });
});
```

If you use `opaQunit`, OPA gives you the following three objects in your QUnit:

- Given = arrangements
- When = actions
- Then = assertions

Given: Defining Arrangements

Let's start by defining arrangements. In the following example, we assume that the app runs in a page called `index.html`. Our OPA test is located in the `test/opa5.html` folder.

We define a relative path pointing to the `index.html` of our application under `test ../index.html - ../`. This means that you go up one directory relative to the current directory:

```
// "Opa5" required from "sap/ui/test/Opa5"
var arrangements = new Opa5({
    iStartMyApp : function () {
        return this.iStartMyAppInAFrame("../index.html");
    }
});
```

This is simple because we already programmed our app and just need to start it. The `return this` is needed for chaining the statements.

When: Defining Actions

We now give OPA the ID and the `viewName` of the control we are looking for. OPA waits until the element is present in the respective view. OPA checks whether it is visible. After OPA has found the button, it invokes the `Press` action. If no button is found, we specify an error message so we know which `waitFor` went wrong.

```
var actions = new Opa5({
    iPressOnTheButton : function () {
        return this.waitFor({
            viewName : "Main",
            id : "pressMeButton",
            actions : new Press(),
            errorMessage : "did not find the Button"
        });
    }
});
```

Then: Defining Assertions

After clicking the button, we want to check if the text has changed. For this, we can use matchers to check if the button we are searching for matches our conditions. We want to be sure that the text property of the button is equal to "I got pressed".

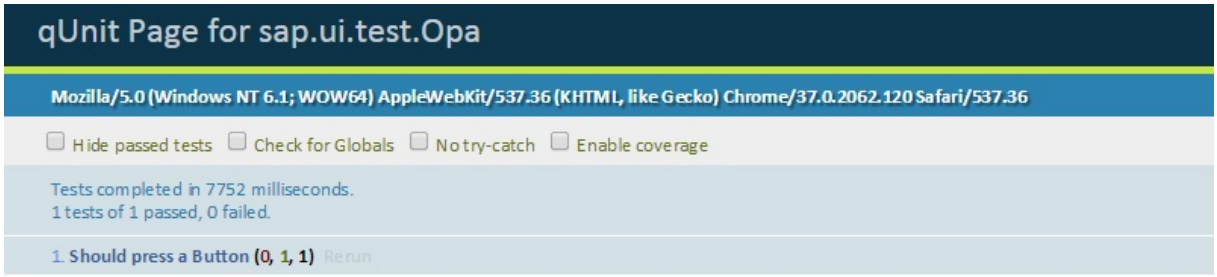
```
var assertions = new Opa5({
    theButtonShouldHaveADifferentText : function () {
        return this.waitFor({
            viewName : "Main",
            id : "pressMeButton",
            matchers : new PropertyStrictEquals({
                name : "text",
                value : "I got pressed"
            }),
            success : function (oButton) {
                Opa5.assert.ok(true, "The button's text changed to: " +
oButton.getText());
            },
            errorMessage : "did not change the Button's text"
        });
    }
});
```

Running the Test

We have now defined all statements and must now add them to the `OpaConfig` as follows:

```
// "Opa5" required from "sap/ui/test/Opa5"
Opa5.extendConfig({
    arrangements : arrangements,
    actions : actions,
    assertions : assertions,
    viewNamespace : "view."
});
```

The `viewNamespace` is very important for finding the correct view. As you probably do not want to set this in every single `waitFor`, a default is provided. You can now launch the test page and the OPA test should run. If everything worked, you get the following result:



For more information, see the API Reference and the Samples.

OPA Startup

Starting a UIComponent

You can use a UIComponent to run your OPA5 tests. To do this, you have to call the `iStartMyUIComponent` function on the OPA5 instance with an object that contains at least the name of your UIComponent (see API documentation about `sap/ui/component` for all possible parameters), for example:

```
// "Opa5" required from "sap/ui/test/Opa5"
new Opa5().iStartMyUIComponent({
    componentConfig: {
        name: "samples.components.button"
    }
});
```

Your UIComponent will now run in the same window as your OPA5 Tests. In addition, you can append a new hash value to the browser URL, for example:

```
// "Opa5" required from "sap/ui/test/Opa5"
new Opa5().iStartMyUIComponent({
    componentConfig: {
        name: "samples.components.button"
    } ,
    hash: "newHashValue"
});
```

This is very helpful if you want to start your tests with a specific target.

> **i Note**
>
> Use the `iStartMyUIComponent` approach instead of an iFrame if you want your tests to run faster (thanks to all resources being loaded at once), make debugging easier (by not having to switch between different frames), and if you want to have full control over the mock server (e.g. Start and Stop time).

> **i Note**
>
> Please note that OPA5 tests can only run for a single UIComponent. You first have to tear down the current UIComponent before starting an OPA5 test for another UIComponent, for example:
>
> ```
> // "Opa5" required from "sap/ui/test/Opa5"
> new Opa5().iTeardownMyApp();
> // or
> new Opa5().iTeardownMyUIComponent();
> ```

Starting an App in an iFrame

You can run the app being tested in an iFrame. You can start only one iFrame at a time. An error will be thrown if you try to start an iFrame when one is already launched or if you try to teardown the iFrame before it is started. If an iFrame element is already present on the page, it will be used. The iFrame and test window must be in the same domain. For example, if you have the `test.html` file next to the `index.html` file, you can start your app with the following code:

```
// "Opa5" required from "sap/ui/test/Opa5"
Opa5().iStartMyAppInAFrame("index.html?responderOn=true");
```

The OPA iFrame launcher overwrites the iFrame's history API so we can later change the iFrame's hash, and pass parameters to the app. In Internet Explorer, the history behaves differently if an iFrame was added with JavaScript, this is why you should add the frame directly to the HTML of your test page:

```
<iframe id="OpaFrame" src="index.html?responderOn=true" style="width:100%;
height:100%"></iframe>
```

You can remove the iFrame using one of the following methods:

```
// "Opa5" required from "sap/ui/test/Opa5"
new Opa5().iTeardownMyApp();
// or
new Opa5().iTeardownMyAppFrame();
```

For more information, see the API Reference: `Opa5`.

Starting the app can be a slow operation so it is not recommended to do this for every test. However, it is good practice to group tests in modules and restart the app in every module to enable faster debugging of larger suites.

Loading an iFrame is significantly slower than loading a component. It requires a separate page, in which the mocked app is started in an SAP Fiori Launchpad sandbox. This is useful as it allows debugging of unmocked data requests and mock app issues in isolation from the OPA test. It is easy to migrate to the component launcher once the test suite grows and the app is proven to be correctly mocked.

SAPUI5 and OPA code (for example, autoWaiter, UI5 plugin, QUnitUtils) is injected asynchronously in the iFrame on launch. The iFrame will be considered launched when all of the scripts are loaded. These scripts will communicate the app state to the test code. Errors in the iFrame will also be logged in the test. If OPA code is already loaded by the app, the newly injected code will be used instead to ensure version compatibility.

OPA provides several getters that give access to certain properties of the context in which the app is loaded. By default, the getters return the test window's objects but if an iFrame is used, they will return the iFrame's objects. You need to keep the context in mind if you want to manipulate app data in your test:

```
// "Opa5" required from "sap/ui/test/Opa5"
// returns the body of the app window wrapped in a jQuery object
Opa5.getJQuery()("body");
// returns the SAPUI5 OPA plugin object of the app window
Opa5.getPlugin();
// returns the SAPUI5 core interface of the app window
Opa5.getWindow().sap.ui.getCore();
// returns the Date in the app context
Opa5.getWindow().Date();
// the following test code will return false if the app is started in an iFrame
new Opa5.getWindow().Date() instanceof Date
```

Cookbook for OPA5

Advanced topics and best practices for OPA tests.

Executing a Single Statement After Other `waitFors` are Done

If you skip all parameters except for the `success` parameter, you can execute your code after the other `waitFors` are done. Since there is no `check` function, OPA runs directly to `success`.

```
iChangeTheHashToTheThirdProduct : function () {
        return this.waitFor({
            success : function () {
                sap.ui.test.Opa5.getWindow().location.hash = "#/Products(2)";
            }
        });
    },
```

Passing a Parameter from One `waitFor` to Another

To check special conditions, for example, how one control relates to another control, you have to pass a control found in one `waitFor` statement as input for another `waitFor` statement. The following two options exist:

- Storing the control in a variable in the outer scope: Use this option if you have a common outer scope, like the same functions, or the same page object file.
- Storing the control in the OPA context: Use this option if you have to pass the parameter, for example, across some page objects.

```
iDoSomething: function () {
        var oControl;
        this.waitFor({
            id : "myControlId",
            success : function (oCtrl) {
                //store control in outer scope
                oControl = oCtrl;

                //as alternative you could store the control in the Opa context
                sap.ui.test.Opa.getContext().control = oCtrl;
            }
        });
        return this.waitFor({
            controlType : "some.other.control"
            check: function (aControlsFromThisWaitFor) {
                //now you can compare oControl with aControlsFromThisWaitFor
                //or you can compare sap.ui.test.Opa.getContext().control with
aControlsFromThisWaitFor
            }
        });
    },
```

Application Parameters

As of version 1.48, you can easily specify URL parameters that are relevant for the application being tested. Simply place them in the `appParams` object under `Opa5.extendConfig()`. Only primitive types are supported. The provided object is serialized to URL search string and all parameters are available to the application being tested.

```
Opa5.extendConfig({
    appParams: {
        "key": "value"
    }
});
```

When the application is started with `Opa5.iStartMyAppInAFrame()`, its parameters are appended to the application URL as provided in the first parameter. Application parameters overwrite any duplicated string in the URL that is given as first parameter of `iStartMyAppInAFrame()`. Alternatively, when `Opa5.iStartMyUIComponent()` is used, the URL parameters are appended to the current URL as the component is started in the same browser window. On `Opa5.iTeardownMyApp()`, the application parameters are cleared from the current URL.

For more details, see the *API Reference* for `Opa5`

URL Parameters

As of version 1.48, OPA supports overwriting global configuration parameters for a single execution from URL. On startup, OPA parses `window.location.href` and extracts all search parameters starting with 'opa'. The prefix is removed and the resulting string has its first character changed to lower case. For example, the `?opaExecutionDelay=600` string in a URL sets the value of `executionDelay` to 600 ms. All OPA config parameters of primitive string and number types that are documented in `Opa.resetConfig()` could be overwritten.

All URL parameters that do not start with 'opa' are considered relevant for the application being tested and are passed to it. Application parameters from a URL always overwrite the application parameters provided in `Opa5.extendConfig()`.

For more details, see the *API Reference*: `Opa5.extendConfig()` and `Opa.resetConfig()`

Working with Message Toasts

A message toast is a small, non-disruptive popup for displaying information or success messages. Message toasts automatically disappear after a timeout unless the user moves the mouse over the message or taps on it.

To ensure stable execution of OPA5 tests which manipulate `messageToast` elements, it is recommended to set explicitly `autoWait` parameter to `false` only for the affected `waitFor` methods, as shown by the following example:

```
this.waitFor({
```

```
        ...
        autoWait: false,
        ...
        }
    });
```

To retrieve a message toast control and manipulate it accordingly, use standard jQuery selectors with the help of the `check` parameter of OPA5 `waitFor` method, as `messageToast` elements cannot be retrieved by interaction with the SAPUI5 API.

Example:

```
iShouldSeeMessageToastAppearance: function () {
                    return this.waitFor({
                        // Turn off autoWait
                        autoWait: false,
                        check: function () {
                            // Locate the message toast using its class name in
a jQuery function
                            return Opa5.getJQuery()(".sapMMessageToast").length
> 0;
                        },
                        success: function () {
                            Opa5.assert.ok(true, "The message toast was shown");
                        },
                        errorMessage: "The message toast did not show up"
                    });
                }
```

Working with Busy Controls

There are OPA5 rules that limit the ways you can use busy controls. Some OPA5 features prevent you from locating controls while they are busy. For example, actions require that the control is interactable and therefore not busy and `autoWait` ensures that all controls on the page are interactable. You can't test a control in its busy state when these features are enabled. You can always work with controls that are not busy as OPA5 either waits for them to become interactable (and not busy) or enforces no limitations.

Table 36: The following table is a cheatsheet with the values for each OPA5 rule and the outcome for busy control testing:

`OPA5.config` `autoWait`	`waitFor` **actions**	`waitFor` `autoWait`	**verify busy control**
√	√	any	X
√	X	true / not modified	X
√	X	false	√
X	√	any	X
X	X	false / not modified	√
X	X	true	X

A common scenario is asserting the busy state of a control. Testing whether a control is not busy is meaningless when `autoWait` is globally enabled. An example of testing for busyness with enabled `autoWait` can be found in the OPA5 Samples.

Working with Responsive Toolbars

A responsive toolbar can have overflowing content depending on the screen size. This content is moved to a popover, which can be opened by pressing a toggle button in the toolbar. A toggle button is displayed only when there's overflowing content. This is a problem for tests because they must only try to press the button when it's visible and interactable. One way to solve this is to always start the application being tested with a fixed screen size. Another way is to first look for toggle button with no visibility restrictions and then press on it only if it exists:

```
this.waitFor({
    id: sToolbarId, // find the toolbar
    success: function (oToolbar) {
        this.waitFor({
        controlType: "sap.m.ToggleButton",
        visible: false, // look for ANY toggle button in the toolbar
        matchers: new Ancestor(oToolbar),
        success: function (aToggleButton) {
            if (aToggleButton[0].$().length) {
                // if the button exists, press on it
                this.waitFor({
                    controlType: "sap.m.ToggleButton",
                    matchers: new Ancestor(oToolbar),
                    actions: new Press()
                });
            } else {
            Opa5.assert.ok(true, "The toggle button is not present");
        }
    }
});
```

Deactivating Tests in Need of Adaptation

As of version 1.61, you can use the `opaTodo` and `opaSkip` methods in addition to `opaTest`. They are similar to `Qunit.todo` and `QUnit.skip` and have the same signatures as their QUnit counterparts.

In QUnit1, `opaTodo` is equivalent to `opaTest` as `QUnit.todo` is not yet available. In QUnit2, `opaTodo` will succeed if the test has at least one failing assertion or if it timeouts with either OPA5 or QUnit timeout.

If a test has to be adapted after recent changes, you have to disable it temporarily in order to have a successful build. A test which is commented out can easily be forgotten and its coverage value lost. `opaTodo` prompts you to uncomment the test once an adaptation is provided.

`opaTodo` and `opaSkip` are readily available to your test as globals.

Example:

```
oOpa.waitFor({
    success: function () {
        Opa5.assert.ok(false, "Should not report test that needs adaptation");
    }
});
```

Retrieving Controls

Common use cases for retrieving controls

Retrieving a Control by Its ID

```
new sap.ui.test.Opa5().waitFor({
    id : "page-title",
    viewName : "Category",
    viewNamespace : "my.Application.",
    success : function (oTitle) {
        Opa5.assert.ok(oTitle.getVisible(), "the title was visible");
    }
});
```

In this example, we search for a control with the ID `page-title`. The control is located in the `my.Application.Category` view.

By default, OPA5 tries to find the element until the default timeout of 15 seconds is reached. You can override this behavior by passing it as a parameter to the `waitFor` function. Zero means infinite timeout.

```
new sap.ui.test.Opa5().waitFor({
    id : "productList",
    viewName : "Category",
    success : function (oList) {
        Opa5.assert.ok(oList.getItems().length, "The list did contain products");
    },
    timeout: 10
});
```

In this example, the `check` function is omitted. In this case, OPA5 creates its own `check` function that waits until the control is found or the specified timeout is reached.

Retrieving a Control That Doesn’t Have an ID

Sometimes you need to test for a control that has no explicit ID set and maybe you can’t or don’t want to provide one for your test. To get around this issue, use a custom check function to filter for this control. Let's assume we have a view called `Detail` and there are multiple `sap.m.ObjectHeaders` on this page. We want to wait until there’s an object header with the title `myTitle`.

To do this, use the following code:

```
return new Opa5().waitFor({
    controlType : "sap.m.ObjectHeader",
    viewName : "Detail",
    matchers : new sap.ui.test.matchers.PropertyStrictEquals({
                                name : "title",
                                value: "myTitle"
                            }),
    success : function (aObjectHeaders) {
        Opa5.assert.strictEqual(aObjectHeaders.length, 1, "was there was only
one Object header with this title on the page");
```

```
        Opa5.assert.strictEqual(aObjectHeaders[0].getTitle(), "myTitle", "was on
the correct Title");
    }
});
```

Since no ID is specified, OPA passes an array of controls to the `check` function. The array contains all visible object header instances in the `Detail` view. However, a built-in support for comparing properties doesn't exist, so we implement a custom check.

More About Matchers

For more information about all matchers, see the API Reference and the Samples.

`sap.ui.test.matchers.Properties`: This matcher checks if the controls have properties with given values. The values can also be defined as regular expressions (`RegExp`) for the string type properties.

```
return new Opa5().waitFor({
            controlType : "sap.ui.commons.TreeNode",
            matchers : new sap.ui.test.matchers.Properties({
                text: new RegExp("root", "i"),
                isSelected: true
            }),
            success : function (aNodes) {
                Opa5.assert.ok(aNodes[0], "Root node is selected")
            },
            errorMessage: "No selected root node found"
});
```

> **Note**
>
> `sap.ui.test.matchers.Properties` and `sap.ui.test.matchers.PropertyStrictEquals` serve the same purpose but it's easier to pass parameters to `sap.ui.test.matchers.Properties`.

`sap.ui.test.matchers.Ancestor`: This matcher checks if the control has the specified ancestor (ancestor is of a control type).

```
var oRootNode = getRootNode();
return new Opa5().waitFor({
            controlType : "sap.ui.commons.TreeNode",
            matchers : new sap.ui.test.matchers.Ancestor(oRootNode),
            success : function (aNodes) {
                Opa5.assert.notStrictEqual(aNodes.length, 0, "Found nodes in a
root node")
            },
            errorMessage: "No nodes in a root node found"
});
```

`sap.ui.test.matchers.Descendant`: This matcher checks if the control has the specified descendant. In this example, we search for a table row, which has a text control with a certain value.

```
this.waitFor({
    controlType: "sap.m.Text",
    matchers: new Properties({
        text: "special text"
    }),
    success: function (aText) {
        return this.waitFor({
```

```
            controlType: "sap.m.ColumnListItem",
            matchers: new Descendant(aText[0]),
            success: function (aRows) {
                Opa5.assert.strictEqual(aRows.length, 1, "Found row with special
text");
            },
            errorMessage: "Did not find row or special text is not inside table
row"
        });
    },
    errorMessage: "Did not find special text"
});
```

`sap.ui.test.matchers.BindingPath`: This matcher checks if the controls have specified data binding paths. The `path` property matches controls by their binding context. Controls with a binding context are usually inside an aggregation or have a parent control with data binding. The `propertyPath` property matches controls by the data binding paths of their own properties. Binding property paths can be part of an expression binding. You can set the `path` and `propertyPath` properties separately or in combination. For a practical example of the various types of data binding, see the Tutorial Samples.

```
// Match a CheckBox located inside a ListItem:
// the CheckBox has a property binding with relative path "Selected"
// the ListItem has a binding context path "/products/0"
return new Opa5().waitFor({
    controlType : "sap.m.CheckBox",
    matchers : new sap.ui.test.matchers.BindingPath({
        path: "/products/0",
        propertyPath: "Selected"
    }),
    success : function (aResult) {
        Opa5.assert.ok(aResult[0], "CheckBox is matched")
    }
});
```

You can also define a matcher as an inline function: The first parameter of the function is a control to match. If the control matches, return `true` to pass the control on to the next matcher and/or to check and success functions.

```
return new Opa5().waitFor({
            controlType : "sap.ui.commons.TreeNode",
            matchers : function(oNode) {
                return oNode.$().hasClass("specialNode");
            },
            success : function (aNodes) {
                Opa5.assert.notStrictEqual(aNodes.length, 0, "Found special
nodes")
            },
            errorMessage: "No special nodes found"
});
```

If you return a 'truthy' value from the matcher, but not a Boolean, it will be used as an input parameter for the next matchers and/or check and success. This allows you to build a matchers pipeline.

```
return new Opa5().waitFor({
    controlType : "sap.ui.commons.TreeNode",
    matchers : [
        function(oNode) {
            // returns truthy value - jQuery instance of control
            return oNode.$();
        },
        function($node) {
            // $node is a previously returned value
```

```
            return $node.hasClass("specialNode");
        }
    ],
    actions : function (oNode) {
        // oNode is a matching control's jQuery instance
        oNode.trigger("click");
    },
    errorMessage: "No special nodes found"
});
```

`sap.ui.test.matchers.LabelFor`: This matcher checks if a given control is associated with an `sap.m.Label` control. This means that there should be a label on the page with a `labelFor` association to the control. The label can be filtered by text value or by the `i18n` key of a given property value. Note that some controls, such as `sap.ui.comp.navpopover.SmartLink`, `sap.m.Link`, `sap.m.Label`, and `sap.m.Text` can't be matched by `sap.ui.test.matchers.LabelFor` as they can't have an associated label.

Using the `i18n` key:

```
return new Opa5().waitFor({
    controlType: "sap.m.Input",
    // Get sap.m.Input which is associated with Label which have i18n text with
key "CART_ORDER_NAME_LABEL"
    matchers: new sap.ui.test.matchers.LabelFor({ key: "CART_ORDER_NAME_LABEL",
modelName: "i18n" }),
    // It will enter the given text in the matched sap.m.Input
    actions: new sap.ui.test.actions.EnterText({ text: "MyName" })
});
```

Using the `text` property:

```
return new Opa5().waitFor({
    controlType: "sap.m.Input",
    // Get sap.m.Input which is associated with Label which have i18n text with
text "Name"
    matchers: new sap.ui.test.matchers.LabelFor({ text: "Name" }),
    // It will enter the given text in the matched sap.m.Input
    actions: new sap.ui.test.actions.EnterText({ text: "MyName" }),
    success: function (oInput) {
        Opa5.assert.ok(oInput.getValue() === "MyName", "Input value is correct");
    }
});
```

For more information, see the API Reference and the Sample.

Searching for Controls Inside a Dialog

Use the option `searchOpenDialogs` to restrict control search to open dialogs only. You can combine `searchOpenDialogs` with `controlType` or any predefined or custom matcher. As of version 1.62, the ID option is also effective in combination with `searchOpenDialogs`. If the dialog is inside a view, the `viewName` option ensures that the given ID is relative to the view. Otherwise, the search is done by global ID.

This is an example of matching a control with ID `mainView--testButton` located inside a dialog. The dialog itself is part of a view with name `main.view` and ID `mainView`:

```
this.waitFor({
    searchOpenDialogs: true,
    id: "testButton",
```

```
    viewName: "main.view"
    actions: new sap.ui.test.actions.Press(),
    errorMessage : "Did not find the dialog button"
});
```

The next example shows the use case where we want to press a button with 'Order Now' text on it inside a dialog.

To do this, we set the `searchOpenDialogs` option to true and then restrict the `controlType` we want to search for to `sap.m.Button`. We use the check function to search for a button with the text 'Order Now' and save it to the outer scope. After we find it, we trigger a `tap` event:

```
iPressOrderNow : function () {
    var oOrderNowButton = null;
    this.waitFor({
        searchOpenDialogs : true,
        controlType : "sap.m.Button",
        check : function (aButtons) {
            return aButtons.filter(function (oButton) {
                if(oButton.getText() !== "Order Now") {
                    return false;
                }
                oOrderNowButton = oButton;
                return true;
            });
        },
        actions: new sap.ui.test.actions.Press(),
        errorMessage : "Did not find the Order Now button"
    });
    return this;
}
```

Searching for Controls Inside a Fragment

As of version 1.63, you can limit control search to a fragment with the option `fragmentId`.

`fragmentId` is effective only when searching by control ID inside a view. Whether a control belongs to a fragment is only relevant when the fragment has a user-assigned ID, which is different from the ID of its parent view. In this case, the fragment ID takes part in the formation of control ID and you have to use the `fragmentId` option to simplify test maintenance.

The next example shows the use case where we want to press a button with ID `theMainView--greeting--helloWorld`, located inside a fragment with ID `greeting` and view with ID `theMainView`:

```
this.waitFor({
    viewId: "theMainView",
    fragmentId: "greeting",
    id: "hello",
    actions: new Press(),
    errorMessage : "Did not find the Hello button"
});
```

Searching for Missing Controls

In OPA5, you can look for controls that are invisible, disabled, or noninteractable by using the respective `waitFor` boolean properties: `visible`, `enabled`, and `interactable`.

You need a more creative approach to verify that no controls on the page match a certain criteria. One idea is to verify that a parent doesn't have a given child. Locate the parent using OPA5 standard matchers and then use a custom `check` function to iterate over the children of the parent. The result of `check` should be truthy if no children match a given condition.

The following example shows a custom `check` function that returns `true` if a popover doesn't contain a button with a certain text.

```
this.waitFor({
   controlType: "sap.m.Popover",
    success: function (aPopovers) {
        return this.waitFor({
            check: function () {
                var aPopoverContent = aPopovers[0].getContent();
                var aButtons = aPopoverContent.forEach(function (oChild) {
                    return oChild.getMetadata().getName() === "sap.m.Button" &&
                        oChild.getText() === "Another text";
                });
                return !aButtons || !aButtons.length;
            },
            success: function () {
                Opa5.assert.ok(true, "The popover button is missing");
            },
            errorMessage: "The popover button is present"
        });
    }
});
```

Searching for Disabled Controls

As of version 1.65, you can search for controls by their enabled state using the `enabled` property. When `enabled` is set to `true`, only enabled controls will match. When `enabled` is set to `false`, both enabled and disabled controls will match.

By default, only enabled controls are matched when:

- `autoWait` is set to `true`, or
- there are actions defined in the `waitFor`

If `autoWait` is disabled and there are no actions, the search matches disabled controls as well.

The next example shows that the `enabled` property has priority over `autoWait`:

```
this.waitFor({
    controlType: "sap.m.Button",
    enabled: false,
    autoWait: true,
    success: function () {...}
})
```

Writing Nested Arrangements and Actions

UI elements can be recursive, for example in a tree. Instead of triggering the action for each known element, you can also define it recursively (see the code snippet below). OPA ensures that the `waitFor` statements triggered in a success handler are executed before the next arrangement, action, or assertion. That also allows you to work with an unknown number of entries, for example in a list. First, you wait for the list, and then trigger actions on each list item.

```
iExpandRecursively : function() {
    return this.waitFor({
        controlType : "sap.ui.commons.TreeNode",
        matchers : new sap.ui.test.matchers.PropertyStrictEquals({
            name : "expanded",
            value : false
        }),
        actions : function (oTreeNode) {
            if (oTreeNode.getNodes().length){
                oTreeNode.expand();
                that.iExpandRecursively()
            }
        },
        errorMessage : "Didn't find collapsed tree nodes"
    });
}
```

Declarative Syntax

Overview

As of version 1.72, OPA5 supports the declarative matcher syntax that allows you to declare built-in matchers in a literal object. The syntax is inspired by control locators in UIVeri5 and promotes reuse between the two testing tools. A matcher declaration is a JSON object. The OPA5 `waitFor` statement is simplified by using a single JSON object, instead of the more verbose matcher instances. Only built-in matchers are allowed. Inline matcher functions and custom matcher instances are only allowed in the matchers `waitFor` parameter:

```
return this.waitFor({
    controlType : "sap.m.Text",
    matchers : function () {
        // ...
    }
});
```

There are two places you can add a matcher declaration in a `waitFor` object:

- On the top level
  In this case, if you use an unknown matcher, an exception is thrown, stating that the parameter isn't defined in OPA5 API.

  ```
  this.waitFor({
      controlType : "sap.m.Text",
      propertyStrictEquals: {
          name : "text",
          value : "foo"
      }
  });
  ```

- In the `matchers` parameter
  In this case, if you use an unknown matcher, an exception is thrown, stating that the matcher isn't supported.

```
this.waitFor({
    controlType : "sap.m.Text",
    matchers: {
        propertyStrictEquals: {
            name : "text",
            value : "foo"
        }
    }
});
```

If there are matchers declared in both places, they're combined.

For more information, see https://github.com/SAP/ui5-uiveri5 .

Matcher Properties

A matcher is declared by its name and properties. The name is a key in the matchers object literal and has to start with a lower-case letter. For example, to declare an `sap.ui.test.matchers.Properties` matcher, use the name `properties`. Every matcher accepts a specific set of properties, which has to be declared as a value in the matchers object. This value has to be an object literal. Behind the scenes, every matcher declaration is transformed into a matcher instance. Every value in the declaration represents the properties that are fed to one matcher instance. There's an example for every built-in matcher in the API documentation.

The following two `waitFor` statements produce the same set of matchers:

```
// declaration
this.waitFor({
    controlType : "sap.m.Text",
    matchers: {
        propertyStrictEquals: {
            name : "text",
            value : "foo"
        }
    }
});
// instantiation
this.waitFor({
    controlType : "sap.m.Text",
    matchers: new PropertyStrictEquals({
        name : "text",
        value : "foo"
    })
});
```

If you have to use one matcher twice, the value for the matcher must be an array. Each element of the array has to be an object literal that is used by one matcher instance. This is useful when you have to match a control by two or more of its properties.

The following two `waitFor` statements produce the same set of matchers:

```
// declaration
this.waitFor({
    matchers: {
        properties: [{
            text: "hello"
        }, {
            number: 0
        }]
```

```
    }
});
// instantiation
this.waitFor({
    matchers: [
        new PropertyStrictEquals({
            name : "text",
            value : "foo"
        }),
        new PropertyStrictEquals({
            name : "number",
            value : 0
        })
    ]
});
```

Ancestors and Descendants

When declaring an `sap.ui.test.matchers.Ancestor` or `sap.ui.test.matchers.Descendant`, you have to declare which control is ancestor or descendant. With matcher instances, you simply pass the control instance that you have already located in a previous `waitFor` statement. Keep in mind that with matcher declarations you can't use object instances or functions as values. The solution is to use a nested declarative matcher for the ancestor or descendant. It's assumed that it will match exactly one control and if it doesn't, any one of the matches is used. This is a special matcher, which supports a superset of matchers, such as, `controlType`, `ID`, and any other JSON-compatible properties available in a typical `waitFor` statement.

The following two `waitFor` statements produce the same result:

```
// declaration
this.waitFor({
    controlType: "sap.m.Text",
    matchers: {
        ancestor: {
            controlType : "sap.m.Bar",
            properties: {
                text: "hello"
            }
        }
    }
});
// instantiation
this.waitFor({
    controlType : "sap.m.Bar",
    matchers: new Properties({
        text: "hello"
    }),
    success: function (aAncestors) {
        var oAncestor = aAncestors[0]; // order not guaranteed
        return this.waitFor({
            controlType: "sap.m.Text",
            matchers: new Ancestor(oAncestor)
        });
    }
});
```

Structuring OPA Tests With Page Objects

The page object design pattern supports UI-based tests with improved readability, fostering the *don't repeat yourself* (DRY) principle of software development that is aimed at reducing repetition of any kind of information.

A page object wraps an HTML page or fragment with an application-specific API, which makes it easy to find a control and provide reuse across multiple tests. If you have multiple pages or UI areas that have several operations, you can place them as reuse functionality in page object. The page object groups all OPA arrangements, actions, and assertions that logically belong to some part of the screen. Since only the test will know if an action is used to set up the test case or to act on the application under test, the page object will combine actions and arrangements into actions. In contrast to the general guidance of Selenium and Martin Fowler, OPA page objects also provide assertions, as the corresponding testing via `waitFor` statements better fit into the page objects. When you define actions or assertions in your page object, have in mind how the test would spell them and if that would be similar to the way you would explain a scenario to your colleagues.

Page objects accept parameters, so you can parametrize your tests either by writing multiple tests, or by repeating your test being on a set of parameters defined in the code. It is also possible to put test fragments into a separate file and refer to this file in the test. This enables you to reuse the same test fragments in different test pages with different setups.

You can also share utility functionality between page objects. Simulating clicks, for example, is useful for most page objects and should be placed in a base class that you can create. As the page objects extend the base class, the functions provided in the base class are available for the page objects. If, for example, you want to share tree handling functions in all tree-based page objects, create a `TreeBase` class by extending the base class. Tree-based page objects such as repository browser and outline then specify `TreeBase` as `baseClass` instead of the generic base class.

OPA5 provides a static method to create page objects, see the OPA Samples in the Demo Kit.

```
Opa5.createPageObjects({
       //give a meaningful name for the test code
       inThe<Page Object> : {
            //Optional: a class extending Opa5, with utility functionality
            baseClass : fnSomeClassExtendingOpa5,

            actions : {
               //place all arrangements and actions here
               <iDoSomething> : function(){
                   //always return this or a waitFor to allow chaining
                    return this.waitFor({
                        //see documentation for possibilities
                    });
                }
           },
           assertions : {
               //place all assertions here
               <iCheckSomething> : function(){
                   //always return this or a waitFor to allow chaining
                    return this.waitFor({
                        //see documentation for possibilities
                    });
                }
           }
       }
   });
```

The method in your test finds all actions at the `Given` and `When` object, the assertions will be at the `Then` object. Everything is prefixed with the page object name.

```
When.inThe<Page Object>.<iDoSomething>();
Then.inThe<Page Object>.<iCheckSomething>();
```

Be careful with `Opa5.extendConfig()` if you give arrangements, actions, or assertions, all previously loaded page objects will be overwritten. So if you mix them, call `extendConfig` before loading the page objects. See the samples in the Demo Kit.

Using the `autoWait` Parameter

Configuring OPA to use `autoWait` parameter for all statements improves test stability and reduces the number of `waitFor` statements.

Overview

OPA `autoWait` parameter is available as of version 1.48. It is a good practice to enable it in your tests. By default, it is not enabled in order to keep old tests running.

`autoWait` synchronizes test execution with the app. No interactions are attempted while the app is performing asynchronous work. This increases the probability that OPA statements succeed as they are only executed when the app is ready to respond.

`autoWait` is used:

- When you retrieve a control with the intent to perform an action on it
- For every control search, when you explicitly set `autoWait` to `true`

`autoWait` is applied before searching for a control, which means before OPA check functions and matchers. If there is no work to await, the controls are retrieved, then actions are executed on them and lastly the success function is called. If there is still pending work, matchers, actions, and success function are skipped and OPA retries the check until it succeeds or a timeout is reached.

Success functions are only called when controls are found, their state is valid and the app is responsive. We recommend that you use actions rather than the success function when interacting with a control. This ensures that the interaction is performed properly and the app is in a state that allows the interaction to be executed.

`autoWait` covers several types of asynchronous work:

- Delayed work set with timeout and immediate
- XHR requests created using XMLHttpRequests and sinon.FakeXMLHttpRequests
- Native promises created with `resolve`, `all`, `race`, and `reject` functions
- UI navigation of parent containers
- UIArea rerendering

Enabling `autoWait` ensures that the controls and their parents are visible, enabled and not busy and also that the controls are not hidden behind static elements, such as dialogs.

If your app has ongoing asynchronous work when the OPA timeout is reached, the test fails. The test failure message includes details of the last detected work before the timeout. This type of OPA timeouts is usually caused by test instability. When writing a huge set of tests and executing them frequently, you might notice some tests that fail sporadically. Setting `autoWait` to `true` should stabilize most of these tests.

If you decide to follow the best practices and to enable `autoWait`, we recommend that you do it only once in your code, near the starting point of your tests. You can then disable it per `waitFor` statement where needed. This will help you to avoid confusion when debugging test failures.

Example:

```
// in QUnit start page, before all OPA tests
Opa5.extendConfig({
    autoWait: true
});
// in an OPA test
oOpa.waitFor({
    id: "myControlID",
    success: function (oControl) {
        Opa5.assert.ok(!oControl.getBusy(), "My control was not busy");
    }
});
// and then in a special waitFor case which requires a control to be non-
interactable
oOpa.waitFor({
    autoWait: false,
    id: "myControlID",
    success: function (oControl) {
        // now you can explicitly check for some blocking condition
        Opa5.assert.ok(oControl.getBusy(), "My control was busy");
    }
})
```

If you decide to start using `autoWait` in your existing tests, the easiest way to migrate is to extend OPA config by enabling `autoWait`, run the tests to see if any `waitFor` statements timeout and then disable `autoWait` specifically for them.

`autoWait` and App Startup

Usually, there is a lot of time-consuming work done on app startup which can make the entire app noninteractive for a long time.

To ensure that OPA doesn't timeout before the app is fully loaded, the timeout for `iStartMyAppInAFrame` and `iStartMyUIComponent` is increased to the default of 80 seconds.

Despite the increase, there are still some tests that timeout. The timeout usually occurs during the first test step, which can be misleading regarding the actual cause of failure. `autoWait` is recommended in such cases but it is disabled during startup to prevent issues with module loading during app launcher initialization.

As of version 1.54, the optional use of `autoWait` after launcher initialization is allowed to make sure that the app is loaded before the first test step. It is disabled by default for backward compatibility as some tests check for busy indicators on app start. You can use the option with both app launchers, for example:

```
Given.iStartMyAppInAFrame({
    source: "applicationUnderTest/index.html",
    autoWait: true
```

```
});
```

`AutoWait` and Overflow Toolbars

Under some specific circumstances, the `autoWait` is not waiting enough time and the next interaction happens before the awaited controls are fully rendered. This problem is particularly visible with overflow toolbars as the interaction with buttons in the toolbar happens before it is completely open and the included buttons are not yet ready, meaning that the interactions are lost.

The root cause is a specific behavior in OPA polling when a control is found on first check. In this case, the next check() is synchronous, for example, it is executed immediately and not on the next poll interval. The problem with this implementation is that the synchronous check prevents the detection of subsequent flows started by the previous interaction. As a result, the synchronization is premature as it happens before the interaction is fully processed and before the UI is completely rendered.

As of version 1.54, there is an `asyncPolling` parameter that overcomes this problem. It causes a postponement of the check() in the next polling and gives a chance for the execution flows caused by the interaction to complete. Unfortunately, it is not possible to make this behavior as default as there are many tests that are coded against the old behavior.

The suggested approach is to set `asyncPolling` as default for all `waitFor` statements:

```
// in QUnit start page, before all OPA tests
Opa5.extendConfig({
    autoWait: true,
    asyncPolling: true
});
```

Setting `asyncPolling` on existing tests may cause a failure because of the more strict synchronization. The most common uncovered problem is a test that is dependent on premature synchronization, such as an assertion for table rows that is executed before the table is fully loaded.

Same parameter can be set for individual `waitFor` statements:

```
// in an OPA test
oOpa.waitFor({
    id: "controlId",
    asyncPolling: true,
    success: function (oControl) {
        // TODO assert status
    }
});
```

Related Information

Pitfalls and Troubleshooting [page 1218]

API Reference: `sap.ui.test.Opa5`

Samples: `sap.ui.test.Opa5`

Extensions for OPA5

Extend OPA capabilities with custom extensions.

You can provide application-aware assertions that are called from the test but operate in the context of the application being tested.

Interface

The extension API is defined in the `sap.ui.test.OpaExtension` class. A custom extension should extend this class and implement the necessary methods. The extension class should be available in the application and loaded in the application frame.

- `onAfterInit()` - called after the application is fully loaded. The test will wait for the returned promise to resolve before starting. This is a good place for extension initialization.
- `onBeforeExit()` - called after the test is finished but before the application is discarded. The application shutdown will wait for the returned promise to resolve. This is the place to clean up the extension.
- `getAssertions()` - called after extension initialization but before the test has started. It should return a map of assertion names and assertion functions. This map is merged in the default QUnit assertion object. The assertion function is called in the context of the application being tested and should return a promise that resolves with `QUnit.pushResult` object. The promise should be resolved for both passing and failing assertion and rejected only if the assertion evaluation fails. The assertion function could interact with the application under test and the test will wait for the returned promise to resolve before continuing. From the point of the view of the test, this assertion is consistent with the classical synchronous QUnit assertions. For more information, see https://api.qunitjs.com/assert/pushResult .

Lifecycle

To load an extension, the test should enable it by specifying extension class name as string in the key 'extensions' in the options object given to `Opa5.extendConfig()`. An array of extension names could be specified or the extension name `?opaExtensions=[my/custom/Extension]` could be given in the test URL. If the extension needs some application parameters, they could be provided in the `appParams`.

For more information, see the *API Reference*: `Opa5.extendConfig()`

Example

Custom extension class:

```
sap.ui.define([
  'sap/ui/test/OpaExtension'
], function(OpaExtension) {
  "use strict";
  var customExtension =
OpaExtension.extend("sap.ui.test.sample.CustomOpaExtension", {
```

```
    metadata : {
      publicMethods : [
        "getAssertions"
      ]
    },
    getAssertions : function() {
      return {
        myCustomAssertion: function() {
          return new Promise(function(resolve, reject) {
            // start custom assertion logic, resolve the promise when ready
            setTimeout(function() {
              // Assertion passes
              resolve({
                result: true,
                message: "Custom assertion passes"
              });
              // OR Assertion fails
              resolve({
                result: false,
                message: "Custom assertion fails"
              });
              // OR Propagate an error while evaluating assertion
              reject(new Error("Error while evaluating assertion, details: " +
details));
            },0);
          });
        }
      }
    }
  });

  return customExtension;
});
```

Activate this extension and provide some URI parameters to the application:

```
Opa5.extendConfig({
  extensions: ["sap/ui/test/sample/CustomOpaExtension"],
  appParams: {
    "key": "value"
  }
});
```

Call the custom extension from the test:

```
Opa5.createPageObjects({
  onMyView : {
    viewName : "MyView",
    assertions : {
      iShouldUseMyCustomAssertion : function () {
        return this.waitFor({
          id: "MyControlId",
          actions: new Press(),
          success : function () {
            Opa5.assert.myCustomAssertion();
          }
        });
      }
    }
  }
});
```

Test Libraries for OPA5

Test libraries are a means of collaboration between app developers and reusable content providers.

As of version 1.48, you can declare OPA5 test libraries to be used within your integration tests.

The main benefit is reduced test maintenance efforts and avoidance of code repetition. You can isolate generic actions and validations in a test library and reuse them across apps, for example, clicking search and back buttons, and validating table content. As a result, app tests are simplified and have compact page objects and short journeys. The test library provider is responsible for keeping it up to date with component changes, which significantly lowers maintenance costs.

Consuming a Test Library

There are 3 simple steps to start using a test library:

1. Define the test library resource root in the QUnit start page.
   For example, the app `my.application` has a dependency on the test library `my.awesome.testlibrary` and its test resources are built into the directory `test-resources`.

```
<script id="sap-ui-bootstrap"
    src="../../resources/sap-ui-core.js"
    data-sap-ui-resourceroots='{
        "my.application.test.integration": "./",
        "my.awesome.testlibrary.integration.testLibrary" : "../../../test-
resources/my/awesome/testlibrary/integration/testLibrary"
    }'>
</script>
```

2. Add the name of the library and its configuration object to the `testLibs` OPA5 configuration property:

```
Opa5.extendConfig({
    testLibs: {
        myAwesomeTestLibrary: {
            appId: "my.application.appId",
            entitySet: "MyExampleEntitySet",
            viewNamespace: "my.application.mainView"
        }
    }
```

3. Require the test library modules in your test files:

```
sap.ui.require([
    "sap/ui/test/Opa5",
    "my/awesome/testlibrary/integration/testLibrary/ExampleList/pages/
ExampleList"
], function (Opa5, ExampleList) {
    // you can now use ExampleList's actions and assertions
    When.onTheTestLibraryPage.iDoThings();
    Then.onTheTestLibraryPage.iCheckTheResult();
});
```

Reusing Functionality

Page Objects

You can directly consume page objects defined by the test library. We recommend you follow the pattern described in Structuring OPA Tests With Page Objects [page 1201].

Here is an example, assuming that the page object `onTheListPage` is defined by a test library:

```
Then.onTheListPage.iSearchForItem();
```

Page Object Utilities

If a test library has exposed utilities, you can use them in your own page objects to simplify interaction with complex controls. There are two steps to start reusing utility functions:

1. Configure the test library:

   ```
   Opa5.extendConfig({
       testLibs: {
           myAwesomeTestLibrary: {...}
       }
   });
   ```

2. Load the test library before the page objects that will use it.
   The utilities will be available on the page object instance under a property matching the name of the test library:

   ```
   Opa5.createPageObjects({
       onTheListPage: {
           viewName: "myTestView",
           actions: {
               iSetTheFilter: function () {
                   this.myAwesomeTestLibrary.iSelectItem();
                   // trigger other interactions
               }
           }
       }
   });
   Then.onTheListPage.iSetTheFilter();
   ```

Global Configuration

Global statements set by a test library are defined and used in the same way as global statements set by a consumer.

Here is an example, assuming that the action `iSetupTheApp` is added by a test library:

```
Given.iSetupTheApp();
```

Creating a Test Library

The test library consists of OPA5 statements written the same way as in a regular test. Users should be able to provide app-specific parameters, such as app ID, view names, control IDs, control labels and texts.

The test library can access the configuration provided by the consumer test in the following manner:

```
var oConfiguration = Opa5.getTestLibConfig("myAwesomeTestLibrary");
oConfiguration.appId === "my.application.appId" // true
```

Exposing Functionality

There are several ways to expose functionality from a test library.

Page Objects

We recommend you use the page objects pattern described in Structuring OPA Tests With Page Objects [page 1201]. Page objects created by this pattern are automatically available for the app tests.

Use this pattern for interactions that always involve a single page:

```
Opa5.createPageObjects({
    onTheListPage: {
        viewName: "myTestView",
        actions: {
            iSearchForItem: function () {
                // find a search field and enter some text
            }
        }
    }
});
```

Page Object Utilities

Define utility functions when you need to expose functionality that will be used as a building block for user page objects. A utility function can be used by multiple page objects.

A common use case is the interaction with a single control:

```
Opa5.extendConfig({
    testLibBase: {
        myAwesomeTestLibrary: {
            actions: {
                iSelectItem: function: () {
                    // choose item of a Select
                }
            }
        }
    }
});
```

Global Configuration

Extending OPA5 configuration from within the test library has an effect on the app test as well. This means that you can also set global OPA5 test statements.

Use this pattern when you need to expose functionality relevant to the the entire app, such as setup and teardown:

```
var Common = Opa5.extend("testLibrary.pageObjects.Common", {
    iSetupTheApp: function () {
        // do some setup actions
```

```
    }
});
Opa5.extendConfig({
    actions: new Common()
});
```

Related Information

Samples: `sap.ui.test.Opa5`

Simulating User Interactions on Controls

OPA5 has a built-in actions parameter that can be used for simulating events. If you use an action, OPA5 will make sure that the UI is in a state that allows the action to be executed.

We recommend that you use actions and not success functions for user interactions as using success functions will not execute the checks on the UI. You can use multiple actions on one control and you can mix built-in actions with custom actions.

Simulating a `press` Event

In this example we trigger a `press` event on a button, using the `waitFor` function of OPA5, and the `Press` action. Note that the action has access to the located button implicitly.

```
oOpa.waitFor({
    id: "myButton",
    actions: new Press()
});
```

Choosing an Item from `sap.m.Select`

Here's an example showing how to choose an item from `sap.m.Select`, using the `waitFor` function of OPA5, and the `Press` action:

```
sap.ui.require([
    "sap/ui/test/opaQUnit",
    "sap/ui/test/actions/Press",
    "sap/ui/test/matchers/Properties",
    "sap/ui/test/matchers/Ancestor"
],  function (opaTest, Press, Properties, Ancestor) {
    opaTest("Should trigger a press event", function (Given, When, Then) {
        // Startup the application using Given
        When.waitFor({
            id: "mySelect",
            actions: new Press(),
```

```
            success: function(oSelect) {
                this.waitFor({
                    controlType: "sap.ui.core.Item",
                    matchers: [
                        new Ancestor(oSelect),
                        new Properties({ key: "Germany"})
                    ],
                    actions: new Press(),
                    errorMessage: "Cannot select Germany from mySelect"
                });
            },
            errorMessage: "Could not find mySelect"
        });
        // Assert what happened after pressing using Then
    });
});
```

For `sap.m.Combobox`, use `controlType: "sap.m.StandardListItem"`.

Entering Text into Input Fields

Use the `EnterText` action when you want to enter text in a form control.

In this example, the text of an `sap.m.Input` is changed twice. First, "Hello " is entered as value. Then, with the second action, "World" is added. As a result, the value of the input is "Hello World".

```
oOpa.waitFor({
    id: "myInput",
    actions: [
        new EnterText({ text: "Hello " }),
        new EnterText({ text: "World" })
    ]
});
```

There are a couple of modifiers to the `EnterText` action:

- Use the `clearTextFirst` property to empty the existing value before entering new text. This example changes a control value to "Hello" and then to "World" with two consecutive actions:

  ```
  actions: [
      new EnterText({ text: "Hello" }), // changes Input value to "Hello"
      new EnterText({ text: "World", clearTextFirst: true }) // changes Input
  value to "World"
  ]
  ```

- Use the `keepFocus` property to preserve the focus on the input after the action completes. This is useful if the control has enabled suggestions that have to remain open after the text is entered. After the text is entered, you can perform another OPA5 search for the suggestion control and select it using a `Press` action.

  ```
  // Show the suggestion list with filter "Jo"
  oOpa.waitFor({
      id: "formInput",
      actions: new EnterText({
          text: "Jo",
          keepFocus: true
      }),
      success: function (oInput) {
          // Select a suggestion by pressing an item with text "John".
  ```

```
        // After the press action, the value of the input should be changed
to "John".
        // Note that the focus will remain in the input field.
        this.waitFor({
            controlType: "sap.m.StandardListItem",
            matchers: [
                new Ancestor(oInput),
                new Properties({
                    title: "John"
                })
            ],
            actions: new Press()
        });
    }
});
```

Table Interaction

A Table consists of columns (`sap.m.Column`) and rows. The rows, defined as `sap.m.ColumnListItems`, consist of cells. In order to utilize a stable locator which is not expected to change frequently, you can use a field/value combination to retrieve and interact with table items.

The following example simulates a click on an item in a table. The name of the field can be found in the $metadata file of your OData service.

```
iClickOnTableItemByFieldValue: function () {
                    return this.waitFor({
                        controlType: "sap.m.ColumnListItem",
                        // Retrieve all list items in the table
                        matchers: [function(oCandidateListItem) {
                            var oTableLine = {};
                            oTableLine =
oCandidateListItem.getBindingContext().getObject();
                            var sFound = false;
                            // Iterate through the list items until the
specified cell is found
                            for (var sName in oTableLine) {
                                if ((sName === "Field Name") &&
(oTableLine[sName].toString() === "Cell Value")) {
                                    QUnit.ok(true, "Cell has been found");
                                    sFound = true;
                                    break;
                                }
                            }
                            return sFound;
                        }],
                        // Click on the specified item
                        actions: new Press(),
                        errorMessage: "Cell could not be found in the table"
                    });
                }
```

Writing Your Own Action

Since OPA5 uses JavaScript for its execution, you cannot use native browser events to simulate user events. Sometimes it's also hard to know the exact position where to click or enter your keystrokes since SAPUI5

controls don't have a common interface for that. If you find you're missing a certain built-in action, you can create your own actions very easily. Just provide an inline function as shown here:

```
sap.ui.require(["sap/ui/test/opaQUnit", "sap/ui/test/matchers/Properties"],
function (opaTest, Properties) {
    opaTest("Should simulate press on the delete button", function (Given, When,
Then) {
        // Startup the application using Given
        When.waitFor({
            id : "entryList",
            matchers : new Properties({ mode : "Delete"}),
            actions: function (oList) {
                oList.fireDelete({listItem : oList.getItems()[0]});
            },
            errorMessage : "The delete button could not be pressed"
        });
        // Assert what happened after selecting the item using Then
    });
});
```

Related Information

API Reference: `sap.ui.test.actions`

API Reference: `sap.ui.test.actions.EnterText`

API Reference: `sap.ui.test.actions.Press`

API Reference: `sap.ui.test.matchers.Interactable`

Using `OpaBuilder`

Write tests by leveraging the builder pattern to create OPA5 descriptors.

`sap.ui.test.OpaBuilder` is available as of version 1.74.

The main benefit for developers is having a function-driven API at hand, which supports and promotes a clean test definition and execution.

In Simulating User Interactions on Controls [page 1210], we provided some examples on how to interact with controls. Let's have a look at some by implementing them using `OpaBuilder`.

Simulating a `press` Event

The `waitFor` options for this straightforward example are as follows:

```
return oOpa.waitFor({
    id: "myButton",
    actions: new Press()
});
```

When you use `OpaBuilder`, it looks like this:

```
return oOpa.waitFor(
    new OpaBuilder()
        .hasId("myButton")
        .doPress()
        .build()
);
```

The result of the `OpaBuilder.build` method is the configuration object for the `Opa5.waitFor` method. Because it's commonly used just as such, `OpaBuilder` comes with a convenient `OpaBuilder.execute()` method. The required `Opa5` instance can be provided as a parameter to the `execute` function, or you can use the `constructor` or `create` method. Taking this into account, the previous example can also be written like this:

```
return OpaBuilder.create(oOpa)
    .hasId("myButton")
    .doPress()
    .execute();
```

For more information, see `OpaBuilder.build`, `OpaBuilder.execute`, and `OpaBuilder.create`.

Complex Interaction with Child Elements

Let's assume we want to show the suggestion list with a filter for "Jo". The `waitFor` definition could look like this:

```
oOpa.waitFor({
    id: "formInput",
    actions: [
        new EnterText({
            text: "Jo",
            keepFocus: true
        }),
        function (oInput) {
            this.waitFor({
                controlType: "sap.m.StandardListItem",
                matchers: [
                    new Ancestor(oInput),
                    new Properties({ title: "John" })
                ],
                actions: new Press()
            });
        }
    ]
});
```

`OpaBuilder` comes with convenient functions to operate on aggregations and child elements: `OpaBuilder.doOnAggregation` and `OpaBuilder.doOnChildren`.

While `doOnAggregation` requires the aggregation name of the defined control and only operates on those SAPUI5 aggregation items, `doOnChildren` addresses any control that is a child within the control hierarchy. Internally, the `sap.ui.test.Matchers.Ancestor` matcher is used as well, but the definition is simplified:

```
OpaBuilder.create(oOpa)
    .hasId("formInput")
    .doEnterText("Jo", false, true),
```

```
    .doOnChildren(
        OpaBuilder.create(oOpa)
            .hasType("sap.m.StandardListItem")
            .hasProperties({ title: "John" })
            .doPress()
    )
    .execute();
```

For more information, see `OpaBuilder.doOnAggregation` and `OpaBuilder.doOnChildren`.

Custom Functions and Chaining

Let's have a look at an example including a custom matcher and an action:

```
When.waitFor({
    id: "entryList",
    matchers: [
        new Properties({ mode: "MultiSelect" }),
        function (oList) {
            return oList.getItems().length > 0;
        }
    ],
    actions: function (oList) {
        for (var i = 0; i < oList.getItems().length; ++i) {
            oList.setSelectedItem(oList.getItems()[i], true);
        }
    },
    errorMessage: "Could not select all items"
});
```

> **i Note**
>
> This example showcases the usage of a custom action. The best practice that we recommend is to use only `Press` and `EnterText` actions when simulating user interactions.

Besides user-defined functions, the example also contains two matchers. As the parameter of the `has` method accepts the same types as the `matchers` property, this part could directly be rewritten as:

```
...
    .has([
        new Properties({ mode: "MultiSelect" }),
        function (oList) {
            return oList.getItems().length > 0;
        }
    ])
...
```

However, by leveraging the builder pattern, the `.has` methods can easily be chained. The resulting `matchers` options are an array consisting of all defined single matchers in the order of definition. This is similar to the `.do` method and the `actions` property.

```
OpaBuilder.create(When)
    .hasId("entryList")
    .hasProperties({ mode: "MultiSelect" })
    .has(function (oList) {
        return oList.getItems().length > 0;
    })
    .do(function (oList) {
```

```
        for (var i = 0; i < oList.getItems().length; ++i) {
            oList.setSelectedItem(oList.getItems()[i], true);
        }
    })
    .error("Could not select all items")
    .execute();
```

While `matchers` and `actions` can be an array of functions, the more seldom used `check` and `success` properties must be a single function. Nevertheless, due to the builder pattern, those functions can be chained as well:

```
OpaBuilder.create()
    .check(fnCheck1)
    .check(fnCheck2)
    .check(fnCheck3)
    .success(fnAssert1)
    .success(fnAssert2)
    .build();
```

`OpaBuilder` chains those functions, which results in the following `waitFor` options:

```
{
    check: function (vInput) {
        return function(vInput) {
            return fnCheck1(vInput) && fnCheck2(vInput);
       ) && fnCheck3(vInput);
    },
    success: function (vInput) {
        fnAssert1(vInput);
        fnAssert2(vInput);
    }
}
```

Additional Features

While `OpaBuilder` itself cannot extend the features provided by `Opa5.waitFor`, it comes with some convenient methods to support test definition. Besides the already mentioned child element support, method chaining, and most commonly used matchers and actions as predefined functions, there are some less obvious features.

Generated Error Message

If no error message is explicitly defined, `OpaBuilder` generates an error message when calling `build()`. The message consists of the `controlType` and `id` properties as well as the number of any additional matchers. A generated `errorMessage` can look like this:

```
sap.m.Button#myButton with 1 additional matcher(s) not found
```

Success Message and Description

When defining an OPA5 test without an assertion, there's no output on success. Most often, such an output is useful for longer journeys, so the `OpaBuilder.success` method also accepts a string argument. This generates a simple truthy assertion with the provided message as a success function:

```
success: function (vControls) {
```

```
    Opa5.assert.ok(true, sSuccessMessage);
}
```

The `OpaBuilder.description` function can be used for even better logging. The provided message is set as `errorMessage` and assertion on success:

```
OpaBuilder.description("Pressing 'Cancel' button")
// Output message...
// ...in case of success
Pressing 'Cancel' button - OK
// ...in case of failure
Pressing 'Cancel' button - FAILURE
```

Aggregation Matcher

A common use case of tests is finding and operating on a control with one or more aggregation items that fulfill certain conditions. While there are already some predefined matchers for aggregations in place, `OpaBuilder` comes with the generic `hasAggregation` and the most commonly used `hasAggregationProperties` methods. The `vMatchers` parameter of `hasAggregation` can be any matcher method (or matcher chain) that is executed against the items of the defined aggregation of the matching control.

```
OpaBuilder.create(oOpa)
    .hasType("sap.m.CustomListItem")
    .hasAggregation("content", [
        function(oContentItem) {
            return oContentItem instanceof sap.m.Title;
        },
        {
            properties: {
                text: sMatchingTitle
            }
        }
    ])
    .press()
    .description("Pressing list item with title: " + sMatchingTitle)
    .execute();
```

> **i** Note
>
> Defining two `hasAggregation` matchers can also match two different aggregation items. To ensure that one item fulfills all criteria, an all-criteria-matcher should be defined in the same `hasAggregation` call.

Conditional Actions

When defining journeys, reusable functions in the page can speed up writing tests and their quality. Sometimes, the generic approach of those functions has its limitations. One limitation is that a test fails if no control is found that matches the conditions. This could be a challenge if the control being tested is not guaranteed to exist.

Example:

Let's have an interaction that selects all items of a list that aren't selected yet.

```
OpaBuilder.create(oOpa)
    .hasType("sap.m.CustomListItem")
    .hasProperties({ selected: false })
    .doPress()
    .description("Selecting unselected items")
    .execute();
```

This is fine as long as there is at least one unselected list item. When all items are already selected, the test fails, which is not what we want. Here, the `doConditional` function comes in handy:

```
OpaBuilder.create(oOpa)
    .hasType("sap.m.CustomListItem")
    .doConditional(
        OpaBuilder.Matchers.properties({ selected: false }),
        OpaBuilder.Actions.press()
    )
    .description("Selecting unselected items")
    .execute();
```

Commonly Used Matchers and Actions

As already seen in the last example, `OpaBuilder` has two static members: `OpaBuilder.Matchers` and `OpaBuilder.Actions`. While there's no issue in using any matchers from `sap.ui.test.Matchers` in the `OpaBuilder` definition, the goal of the two members is to provide the most commonly used matchers and actions to be directly accessed when working with `OpaBuilder` without explicitly requiring them in the test class.

`OpaBuilder.Actions` contains both `sap.ui.test.Actions.Press` and `sap.ui.test.Actions.EnterText`, while `OpaBuilder.Matchers` does **not** contain every predefined matcher in `sap.ui.test.Matchers`, but still provides some additional ones as described in the API.

Related Information

API Reference: `sap.ui.test.OpaBuilder`
API Reference: `sap.ui.test.OpaBuilder.Matchers`
API Reference: `sap.ui.test.OpaBuilder.Actions`

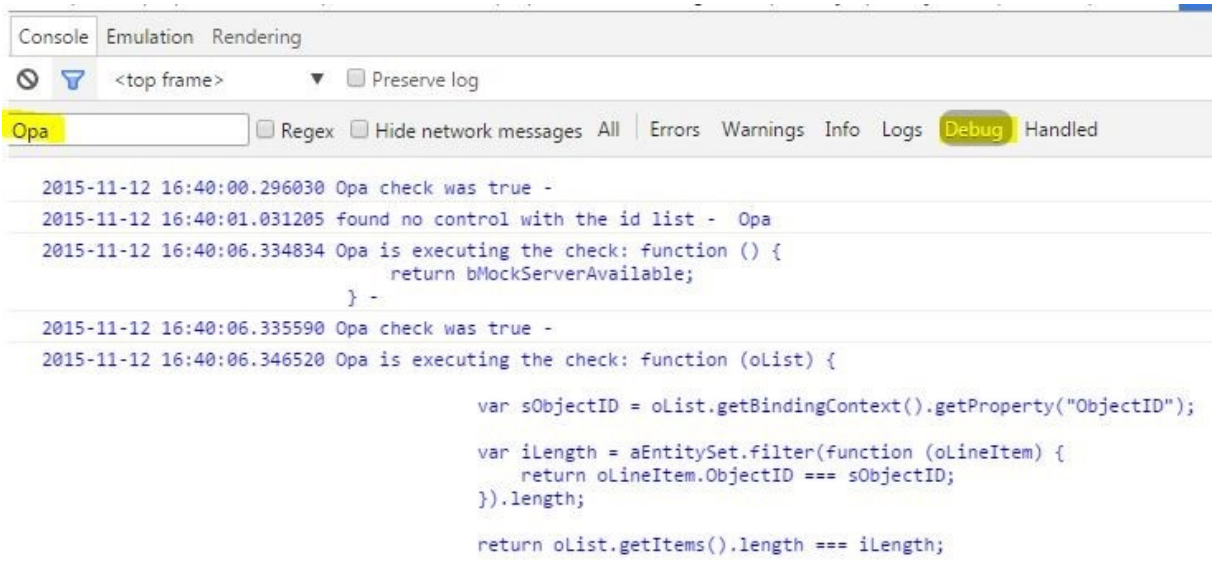
Pitfalls and Troubleshooting

Tips and tricks if OPA isn't behaving or reacting the way you expect it to.

Why Can't OPA Find the Control I'm Looking For?

OPA checks many conditions before it passes a control to your matchers/actions/success functions. If your control doesn't match these conditions, you're not able to set a breakpoint. For such instances, OPA logs lots of information into the browser's console if you turn on the SAPUI5 debug mode. You can either use the `sap-ui-debug=true` URL parameter or the SAPUI5 Diagnostics [page 1326]. The diagnostics can also be helpful to see the state of your UI.

After turning on the debug mode, you can have a look at the log and also filter it by looking for **`opa`** or **`matchers`**.



A frequent cause of error is typos in the view name or control IDs. These are easily found by looking through the logs.

Multiple Views with the Same `viewName`

If there are multiple views with the same `viewName`, OPA5 may not find the exact control you're looking for.

As of version 1.62, there are a couple of ways to ensure a correct match:

- `viewId` parameter is introduced. You can set it in `Opa5.extendConfig()`, `Opa5.waitFor()` and in page object definitions. `viewId` can be used standalone or in combination with `viewName`. If OPA5 finds multiple views with the same name, it prompts you to add a view ID with the test failure message "Please provide `viewId` to locate the exact view.".
- Only views that are rendered are used in OPA5 control search.

Control Isn't Found When Running the Test on a Different Machine or in a Suite

The size of the iFrame in which the app is loaded is as large as the browser window. It's scaled down to leave space for the QUnit info but the content is preserved the same as when run in full size. This means that regardless of the small iFrame, you shouldn't see any responsive change in the app's appearance.

If the test runs fine locally but control isn't found on another machine, there's a chance that the other machine's screen is too small and triggers the responsive behaviour of some controls. For example, CI executors with smaller screens or when the test is part of a suite and the iFrame is placed inside a suite wrapper much smaller than the screen.

One way is to test for the responsive behavior and add conditional `waitFors` and test cases. Tests for different screens, such as phone and desktop, are better separated in different test files.

If you want to work around the sizing issue and don't want to test responsive behavior, you can set a fixed size for the iFrame. The idea is to write the test for the small size which most probably results in the central environment. You can use the `width` and `height` parameters of `iStartMyAppInAFrame` or the `opaFrameWidth` and `opaFrameHeight` URL parameters.

If either width or height isn't defined, a default value is assigned. The default screen size is 1280x1024 px. The iFrame takes 60% of the screen size, which makes the default iFrame size to be 768x614.4 px.

Sometimes My Test Fails, Sometimes It Doesn't

Is It the Startup That's Failing?

Maybe the app is loading too slowly for the OPA tests. If there's a local index file that doesn't contain the library dependencies your app needs, the SAPUI5 bootstrap is very slow. To fix this, add the dependencies you need in your application descriptor's `sap.ui.dependencies` namespace. If you don't have a descriptor, use the bootstrap option libs. For more information, see Descriptor for Applications, Components, and Libraries [page 734] and Configuration Options and URL Parameters [page 703].

It's Failing During the Execution

If this happens, your test is probably executing actions faster than it should. If you encounter a failure, look at the current state of the UI - in almost all cases an action couldn't be triggered or a JavaScript error occurred. This error should be included in the console logs. If an action couldn't be executed, make sure that you use the action parameter of OPA5's `waitFor` function. When using the success function for triggering actions, OPA5 doesn't check many things.

Here are some examples that have occurred in known apps:

- An app was using the `bindingContext` of a control in a press handler. OPA5 was way faster than a human user, so the HTTP-Request that was sometimes finished by the time OPA5 was executing the check, was sometimes still pending and so an exception was thrown. The test failed because OPA was trying to reach a page that couldn't be shown because of this error. This had to be fixed in the app.
- When there was no action parameter available, a `ListItem` got rerendered while a press action was executed on it. Due to the rerendering, the `List` wasn't able to perform the click, meaning it wasn't executed and the test failed. This only happened on certain occasions, depending on the execution speed of the machine executing the test. This is now detected automatically when using actions.

OPA5 Is Failing on a Specific Browser: What Should I Do?

Am I Comparing Language-Dependent Texts and the Browser Has a Different Language?

Check the logs to see if your matcher is failing because it's checking a text against a different language. If you want to always execute your tests with the same language, use the `sap-ui-language=` URL or bootstrap parameter.

Is It Only Internet Explorer That's Affected?

If you're using an IFrame to launch your app, Internet Explorer is more strict when it comes to objects from removed IFrames. If you're using the OPA context to remember objects, destroy your frame, and then execute a function on the object, you get a JavaScript exception. How do you avoid it? By only remembering values like strings or integers when destroying the frame, or by using the component startup in OPA5.

OPA Isn't Even Starting and There's No Logging Either

If you require `sinon-qunit.js`, it overwrites the browser functions `setTimeout` and `setInterval`. OPA needs these functions and without them the tests don't start. You can either set the `fakeTimers` to `false` in your test setup, or maybe consider not using `sinon-qunit.js` together with OPA.

```
module("Opatests", {
    beforeEach : function () {
        sinon.config.useFakeTimers = false;
    },
    afterEach : function () {
        sinon.config.useFakeTimers = true;
    }
});
```

OPA Tests Aren't Stable

For example, the tests run fine most of the time, but they fail:

- in automated test runs
- when run with different OPA speeds
- sporadically on various steps

One way to stabilize your tests is to use OPA `autoWait` and `actions`.

Working with Controls Which Set Timeouts

Examples of such controls are busy indicators, notification popups, and message toasts. These controls set a timeout after which the control is supposed to disappear. In some apps, it can be important to ensure that such a control is displayed. Note that if you enable `autoWait` in your tests globally, then you have to disable `autoWait` specifically in the `waitFor` statements related to these special controls. For example, if you want to test that a busy indicator is displayed during the sending of a request, you don't want to wait for controls to be interactable:

```
oOpa.waitFor({
    autoWait: false,
    id: "myBusyList", // a control that is expected be covered by a busy
indicator
    matchers: new PropertyStrictEquals({
        name: "busy",
        value: true
    }),
    success: function (oList) {
        Opa5.assert.ok(true, "My list is busy");
    }
});
```

Related Information

Cookbook for OPA5 [page 1188]
Diagnostics [page 1326]

Mock Server

Mock server is a mocking framework for HTTP and HTTPS that is used to simplify integration testing and to decouple development teams by allowing to develop against a service that is not complete or unstable.

In SAPUI5, the mock server mimics OData backend calls. It simulates the OData provider and is completely client-based, meaning that no network connectivity to a remote host is required. It intercepts HTTP calls made to the server and provides a fake output to the client. All this is transparent to data binding and use of OData model and feels like a real server. No changes are required on the OData model.

The mock server provides mock services and also mock data. It supports randomly generated data based on the service metadata, as well as mock data provided in JSON files.

> **i** Note
>
> Mock Server only supports the JSON format for representing the resources it exposes, such as collections, entries, links, and so on.

Related Information

sap.ui.core.util.MockServer

OData Features Supported by Mock Server

List of OData version 2.0 features supported or not supported by the mock server

The mock server only supports the JSON format for representing the resources it exposes, such as collections, entries, and links.

> **!** Restriction
>
> The mock server doesn't support more than one draft service in an application.

The following table lists the OData features that are supported by the mock server.

Table 37: Supported OData Features

Feature	Status
CRUD calls	Supported
Navigations	Supported
Query String options:	
- Orderby System Query Option (`$orderby`)	Supported
- Top System Query Option (`$top`)	Supported
- Skip System Query Option (`$skip`)	Supported
- Filter System Query Option (`$filter`)	Supported; `eq`, `ne`, `gt`, `lt`, `ge`, `le`, `substringof`, `startswith`, `endswith`, `and`, `or`
- Expand System Query Option (`$expand`)	Supported
- Select System Query Option (`$select`)	Supported
- Inlinecount System Query Option (`$inlinecount`)	Supported
- Format System Query Option (`$format`)	Supported; only JSON format is allowed
Batch processing	Supported
Multiple services	Supported; we recommend to create one Mock Server instance per service
Update via `MERGE/PATCH`	Supported
ETag handling	Supported

The following table lists the OData features that are **not** supported by the mock server.

Table 38: Unsupported OData Features

Feature	Status
Service operations (function imports)	Unsupported, but can be extended
Annotations	Unsupported
Getting individual properties of an entity (`$value`)	Unsupported
`Edm.DateTime` keys	Unsupported
`Edm.DateTime` values	All dates have to be either before 2017 or after; you cannot use mixed values

Feature	Status
`metadata.xml` with properties from `datetimeoffset`	Unsupported
Key of type `Edm.Boolean`	Unsupported
Key of type `Edm.Int64`	Unsupported
`$links` and `#AddressingLinksBetweenEntries`	Unsupported For more information see Addressing Links between Entries and Referencing Requests in a Change Set on http://www.odata.org/.
Multiple navigation properties	Unsupported
Response in byte array format	Unsupported
Combination of the system queries `$select` and `$expand` in draft OData services	Unsupported
Filter query string operator (`$filter`) with one of the functions `substringof(string po, string p1)`, `endswith(string p0, string p1)` or `startswith(string p0, string p1)`	Does not support input strings (`p0` or `p1`) containing a comma `,`.
Key values containing a comma	Unsupported
System query option `$expand` with multiple multi-level navigation properties with the same root navigation property	Unsupported
Filter query `$filter` on fields from type `edm.datetime`	Unsupported

Mock Server: Frequently Asked Questions

Is the mock server a real server?

No. The mock server runs on the client and only uses the server terminology of 'start' and 'stop'. It does **not** require a network connection since there is no actual server involved.

What module is needed?

The mock server is contained in module `sap/ui/core/util/MockServer`. The module can either be added to the list of dependencies in a `sap.ui.define` call or it can be required with a call to `sap.ui.require`:

```
sap.ui.define([…, 'sap/ui/core/util/MockServer', ...], function(…, MockServer,
…) {
    var oMyMockServer = new MockServer(...);
});
sap.ui.require(['sap/ui/core/util/MockServer'], function(MockServer) {
    var oMyMockServer = new MockServer(…);
});
```

Can we use one mock server instance to mock multiple OData services?

No. Each OData service needs its own mock server. Create one `MockServer` instance per service.

How to obtain metadata xml?

Call the metadata of the service in a browser and save it into a file.

How to obtain mock data? What options do I have for mock data?

You can let the mock server generate random mock data automatically based on services metadata. For this, provide only the path to the metadata file and omit the second parameter of the simulate function as follows:

```
// url to the service metadata document
        var sMetadataUrl = "testdata/rmtsampleflight/metadata.xml";
        oMockServer.simulate(sMetadataUrl);
```

You can provide your own mock data in .json files, which can either be created manually or saved from an OData service response. Mock data in JSON format can be generated from an OData service by adding the `$format=json` parameter to the URL. Save the browser response which is called `<entity set name>.json`, for example `FlightCollection.json` and put it into the model folder. Add the path to the simulate function:

```
// url to the service metadata document
        var sMetadataUrl = "testdata/rmtsampleflight/metadata.xml";
        // base url which contains the mockdata
        var sMockdataBaseUrl = "testdata/rmtsampleflight/";
        oMockServer.simulate(sMetadataUrl, sMockdataBaseUrl);
```

You can specify a path to .json mock data and let the mock server generate data for the rest of the service entities:

```
var sMetadataUrl = "testdata/rmtsampleflight/metadata.xml"// url to the service
metadata document
```

```
        var sMockdataBaseUrl = "testdata/rmtsampleflight/"// base url which
contains the mockdata
            oMockServer.simulate(sMetadataUrl, {
                'sMockdataBaseUrl' : sMockdataBaseUrl,
                'bGenerateMissingMockData' : true
            });
```

You can specify the names of the entity sets that are needed, and the mock server will load data only for the specified service entities:

```
var sMetadataUrl = "testdata/rmtsampleflight/metadata.xml"// url to the service
metadata document
        var sMockdataBaseUrl = "testdata/rmtsampleflight/"// base url which
contains the mockdata
            oMockServer.simulate(sMetadataUrl, {
                'sMockdataBaseUrl' : sMockdataBaseUrl,
                'bGenerateMissingMockData' : true,
                'aEntitySetsNames' : ["EntitySetName1", " EntitySetName2"]
            });
```

I'm using the OData model and I get the following error in the console: *The following problem occurred: no handler for data*

The OData model uses JSON to fetch the data:

```
var oModel = new sap.ui.model.odata.ODataModel(sUri, true);
```

What do I put in the `rootUri`?

Verify that you use the exact same URI prefix in the request as in the `rootUri` you define for the mock server. If a root URI is set, all request path URIs are prefixed with this root URI. The root URI has to be relative and requires a trailing '/'. It also needs to match the URI set in OData/JSON models or simple XHR calls in order for the mock server to intercept them.

The code snippet shows an example:

```
sap.ui.define([…,
    'sap/ui/core/util/MockServer',
    'sap/ui/model/odata/v2/ODataModel',
     ...
], function(…, MockServer, ODataModel, …) {
    var sUri = "/mock/";
    var oMockServer = new MockServer({
        rootUri : sUri
    });
    var oModel = new ODataModel(sUri, true);
    ...
});
```

Can the mock server be used for more than for OData service simulation?

Yes. The mock server can be used to help you fake server response on any given API and stub all AJAX access to resources such as OData service, metadata, annotation files (XML), other JSON or *.properties files.

Is OData navigation supported?

The mock server supports navigation via association also if no referential constraint is defined. However, the result of the navigation is the entire collection of the navigation, or the first entry of the collection according to the association multiplicity. So, if you want the navigation to return "correct" results according to keys, define a respective referential constraint.

> **i** Note
>
> Due to a limitation of the mock server, you **cannot** use the same association to describe a two-way navigation. If the navigation shall work for both directions, you need to define an appropriate association for each direction.

How can I use the mock server in a QUnit?

You can set up the mock server in the setup function. For example:

```
sap.ui.require(['sap/ui/core/util/MockServer'], function(MockServer) {
    ...
        QUnit.module("OData data provider", {
            beforeEach : function() {
                this._oMockServer = new MockServer({ rootUri: "/model/"});
                this._oMockServer.simulate("../../../../qunit/service/
metadata.xml");
                this._oMockServer.start();
            },
            afterEach : function() {
                this._oMockServer.stop();
            }
        });
    ...
});
```

How can I provide exit functions as pre/post functions of requests?

Mock Server has APIs to provide more flexibility and control over its current request processing. During request processing, the callbacks are called before or after native handling of the Mock Server using the SAPUI5 eventing mechanism. You can add a callback in all requests of a specific HTTP method, for example in all `get` requests, but additionally also on a specific entity set name, for example, `POST` to `SaleOrders`).

```
// add a callback in all requests of a specific http method
```

```
oMockServer.attachAfter(sap.ui.core.util.MockServer.HTTPMETHOD.GET, fnCbPost);
```

```
// on a specific entityset name
oMockServer.attachAfter(sap.ui.core.util.MockServer.HTTPMETHOD.GET, fnCbPost,
"CarrierCollection");
```

```
// remove the callback
oMockServer.detachAfter(sap.ui.core.util.MockServer.HTTPMETHOD.GET, fnCbPost);
```

If you add additional request handlers and want to use this hooks mechanism inside your response function, just call:

```
this.fireEvent(sap.ui.core.util.MockServer.HTTPMETHOD.GET + 'sEntityset' +
':before' , {oXhr: oXhr, sUrlParameters: sUrlParameters});
```

What do I need to do to run an OPA test with mock server

Start your app in mock mode. It is not possible to declare a mock server outside the app context.

Using Mock Data

Mock Data can be used when you start the development of an app as well as for testing and problem solving when the data service is not available or it requires effort to set up data services.

To switch to mock mode, set the URL parameter `responderOn` to `true`. We recommend to provide one check for this parameter in the app in a central place, for example in the `model.Config` object in the `model` folder.

```
jQuery.sap.declare("model.Config");
model.Config = {};
(function () {

    // The "reponder" URL parameter defines if the app shall run with mock data
    var responderOn = jQuery.sap.getUriParameters().get("responderOn");

    // set the flag for later usage
    model.Config.isMock = ("true" === responderOn);
}
)();
```

To run your app with mock data, you can use the mock server. The mock server intercepts HTTP calls to the server and produces a faked output to the client. This is transparent to your data binding and the use of OData model and feels like a real server. You start the mock server when you intialize your app as follows:

```
sap.ui.app.Application.extend("Application", {

    init : function () {

        ...

        // start mock server
        if (model.Config.isMock) {
            jQuery.sap.require("sap.ui.core.util.MockServer");
            var oMockServer = new sap.ui.core.util.MockServer({
```

```
                rootUri: model.Config.getServiceUrl();
            });
            oMockServer.simulate("model/metadata.xml", "model/");
            oMockServer.start();
        }
```

The mock server needs a metadata XML file that describes the data structure of your service. You can obtain this by opening the OData service root URL in a browser with the suffix "$metadata" appended. Copy the resulting XML file into the model folder of your application.

Remove any kind of link that points to internal servers.

The following two options for providing mock data exist:

- Provide your own mock data
  You can provide JSON files as test data for the mock server to produce the output. Put all files into the model folder. To avoid a "not found" error messages of the mock server, provide JSON files for each entity of the service. Otherwise, the mock server will log those error messages to the console and create empty data sets for the entities lacking a respective JSON data file. This is all right, in case you do not want to load mock data for those entities. The mock server can also generate mock data for those entities by passing a parameter to the simulate function.
- Mock server generates the mock data
  The mock server can produce random mock data based on the service metadata it simulates. This can be done easily by providing the path to the metadata fil and omitting the second parameter of the simulate function. However, this option does not provide data that matches your business scenario.

Test Automation

To make sure that the code is always tested thoroughly before it is included in a productive app, you should use a test runner that automates tests. The test runner can be included in your project setup so that it is called whenever code changes are submitted.

In the following section, we describe the setup with *Karma*, but you can of course choose a different solution.

Karma uses plugins to add support for various framworks. For SAPUI5, you can use the `karma-ui5` plugin that helps with testing SAPUI5 projects.

Related Information

Continuous Integration: Ensure Code Quality [page 1398]

Karma Home Page

`karma-ui5` on GitHub

Installing Karma for Automated Testing

Initial setup of the application testing environment with Karma.

Prerequisites

You have installed *Node.js* in version 8.5 or higher and *npm*.

Procedure

1. Install *Karma* globally via *npm* so that you can run *Karma* by typing the `karma` command in your command-line interface (CLI).

   For more information, see the Karma Installation Guide on GitHub .

   Use the following command:

   ```
   npm install --global karma-cli
   ```

2. Create a `package.json` file.
   - If you already have an *npm* project, you can skip this step.
   - If not, use the following command:

     ```
     npm init --yes
     ```

3. Install *Karma* locally in your working directory.

   Use the following command:

   ```
   npm install --save-dev karma
   ```

4. Install the *Karma UI5 plugin* locally on your working directory.

   Use the following command:

   ```
   npm install --save-dev karma-ui5
   ```

5. Install the *Karma Chrome Launcher* locally in your working directory.

   Use the following command:

   ```
   npm install --save-dev karma-chrome-launcher
   ```

   In this example, we use Google Chrome as browser. You can find an overview of availabler browser launchers by searching for packages with the keywords `karma-launcher` on the *npm* home page.

6. Create a `karma.conf.js` file in your working directory with the following content:

   ```
   module.exports = function(config) {
     config.set({
       frameworks: ["ui5"],
   ```

```
    ui5: {
      url: "https://<<server\>\>:<<port\>\>"
    },

    browsers: ["Chrome"]

    });
  });
```

Adapt the URL (`<server\>:<port\>` to the SAPUI5 resources according to your installation. You can also use SAPUI5 from a content delivery network, see Variant for Bootstrapping from Content Delivery Network [page 696].

> **i Note**
>
> The SAPUI5 plugin uses sensible defaults to detect your type of project and the relevant folders. If you have a project with a different structure, you need to add some more configuration options. For more information, see the Karma UI5 documentation on GitHub.

7. You can now run the tests with the following command:

```
karma start
```

Related Information

Karma Home Page
Node.js Home Page
npm Home Page
karma-ui5 on GitHub

Continuous Integration With Headless Chrome

For running tests in CI scenarios, such as Travis CI in GitHub, Headless Chrome needs to be used. Headless Chrome is a Chromium Browser without GUI (in a headless environment).

You can use Headless Chrome standalone, but also with Karma. To launch Karma with Headless Chrome, you need to add the following changes to the karma config.

1. Add `karma-chrome-launcher` to the project dev dependencies.

```
npm install -D karma-chrome-launcher
```

> **i Note**
>
> This is only required, if Chrome is **not** used as browser, that means, it was not defined during karma initialization. In some cases, for example for running tests in a docker container, we recommend to add puppeteer as dependency which includes the latest Chrome version. For more details, see the karma-chrome-launcher repository on GitHub.

2. Define the browser environment for running karma. For this, update the `config` object in `karma.conf.js` as follows:

```
module.exports = function(config) {
  config.set({
    [...]
    browsers: ['ChromeHeadless'],
    singleRun: true
    [...]
  })
}
```

> **i Note**
>
> Make sure to set the `singleRun` flag to `true`. Otherwise, the script ends up in watch mode, listening for file changes.

You can launch karma also with a specific config file. It might make sense, for example, to define a config file (default: `karma.conf.js`) for the local environment, and one config file, for example `karma-ci.conf.js` for the CI scenario. To launch karma with a different config file, add the file name as third parameter as follows:

```
karma start karma-ci.conf.js
```

Code Coverage with Istanbul and OPA5

To measure the code coverage, you can use the *Istanbul Code Coverage* plugin for *Karma*. Since the *Istanbul* plugin cannot retrieve results from within iFrames, you may run into problems if you use OPA5 tests.

To get correct code coverage results for OPA5 tests, you need to execute them inside the component containers instead of iFrames. This will also speed up the execution time of your OPA5 tests.

> **i Note**
>
> With component containers, you lose the isolation of your single tests. Also, the `index.html` file of your app is no longer executed (only the `Component.js` file is needed).

To execute the tests inside a component container, replace `iStartMyAppInAFrame()` with `iStartMyUIComponent()` and `iTeardownMyAppFrame()` with `iTeardownMyApp()` in all your OPA5 tests.

> **! Restriction**
>
> Currently, this does not work for apps that are created for the SAP Fiori launchpad (FLP).

For more information, see the API Reference: `sap.ui.test.Opa5`.

Related Information

Istanbul Code Coverage Home Page

Behavior-driven Development with Gherkin

With behavior-driven development (BDD), you as a developer start with a user story that defines the business value that the developed app should have. Next, you write a test that verifies the new functionality (this test initially fails). Finally, you write the needed functionality and your test passes. Gherkin is a test framework that supports this approach.

At first, as you are learning BDD, it will take a long time to implement new tests. Resist the temptation to abandon automated testing. The most important software development phase for successful software is the maintenance phase. Automated tests are the best way to ensure an effective maintenance phase, and help ensure that the code quality remains high over time.

It's true that when you first try automated testing it might take a long time, but even this first attempt will be worthwhile and pay dividends later. In your future projects, when you implement your tests much faster, your initial investment in learning how to do integration testing will really pay off.

One good way to ensure that you get the most value for your investment of time into automated testing is to ensure that you test the right things. Integration testing is best for testing the main path of the major business scenarios. These are what are called "face-saving tests", in the sense that you will lose face if you try to deliver the software when these major business scenarios are failing. Hence, integration tests are a great way to do a quick and painless smoke test every time you commit a change to your software, to ensure that you haven't broken anything important.

Since writing integration tests can be time-consuming, it's better to use unit testing to test all of the nuances and failure cases of your software. Unit tests are cheap and easy to write, and are better suited to achieving full test coverage for the software.

Gherkin

Gherkin is written in JavaScript and is fully compatible with SAPUI5, OPA, and QUnit. It is based on the “cucumber” tool.

The advantages of using Gherkin are:

- You write executable specifications that are easy to understand and that allow the easy generation of integration tests.
- Product specification and documentation are **always up to date**; they evolve during the development project.
- Single source of truth: Reduce communication errors across your development team, because the product owner, developers, and testers are all working from the same specification.
- Maximize the business value you get out of the time spent writing tests, and keep your focus on the customer and their requirements.

The Gherkin library contains the following parts:

- Feature file
  Software specification written in Gherkin syntax. Feature files are human-readable specifications that are also machine-readable. Features are composed of **test scenarios**, which are themselves composed of **test steps**.

  ```
  #!featureFeature: Wearing sunscreen stops skin cancer
  ```

```
  Scenario: Apply sunscreen
    Given the sun is dangerous
    When I apply sunscreen
    Then I protect my skin
```

- Steps file
  Translates the feature file into something a computer can understand and execute. The steps file also contains the tests to be executed to ensure that the software behaves according to its specification. The main elements of a steps file are called **step definitions**.

```
this.register(/^I protect my skin$/i, function() {
  this.assert.assertEqual(this.mySkin, 'protected');
});
```

- Test harness
  Stitches together the feature file and steps file and executes runtime tests on the result using a test framework such as QUnit.
- `DataTableUtils`
  Convenience library for handling data tables and string normalization.

Related Information

cucumber Home Page
Gherkin documentation on Github

Feature Files

Feature files are human-readable specifications that are also machine-readable.

The Gherkin syntax is simple. Each major software feature is written in a separate file. You need to decide how to split your software up into features. For example, if you are testing a coffee machine, features might include: serving coffee, accepting money, dispensing change, setting the cost of each beverage, serving hot chocolate, serving hot water, etc. Each one of these features could have its own feature file.

Each feature file contains exactly one feature, and this feature contains one or more test scenarios. Each test scenario contains one or more test steps. Test steps describe the practical steps that the user needs to perform to execute the overall test scenario.

For example, for the "accepting money" feature, a test scenario might include steps such as (1) the user must insert enough money into the machine before (2) the machine serves coffee. You could create a second test scenario, where (1) the user doesn't insert enough money and (2) the machine does not serve coffee. In this example, each scenario is composed of two steps.

Conventions for Feature Files

- Use the file extension `.feature`

- Can include comments by adding the hash (`#`) symbol at the beginning of a line
- Feature files are composed of one or more test scenarios, which walk the user through using the software; what the user does, and what the expected results are.
- These scenarios are themselves composed of lines starting with the keywords `Given`, `When`, `Then`, `And`, `But` and `*`.
- New lines begin with a keyword
  - Features start with `Feature`
  - Test scenarios start with `Scenario`

  Indentation is purely for readability and is not parsed. Similarly, blank lines are ignored by the parser.
- You can also just create a bulleted list of steps instead of using keywords

Example

The following example shows the structure of a sample feature file :

```
#!featureFeature: this is the name of the feature
    Here you can describe the feature. Indentation is purely to make
    this more readable for you. This section will not be used for
    testing, it is solely for human consumption.
  Scenario: this is the scenario's name
      This is a comment about the scenario
    Given you make a certain assumption here
    And you make another assumption
    When some action is taken
    Then there is an expected response that you write here
    But there is an exception you should test for
  # comment lines must start with #, and will be skipped by the parser
  Scenario: another scenario's name
    * you can also just create a bulleted list of steps
    * instead of using keywords
```

Additional Options for Feature Files

We recommend that you familiarize yourself with the following advanced concepts in behavior-driven development with Gherkin.

Tags

Gherkin supports the concept of tags. A tag is metadata that can augment a feature or scenario with contextual information. Tags begin with an @ symbol, appear on the line above a feature or scenario, and are separated by spaces. Tags can be added before a feature, a scenario, a scenario outline, or an example.

```
#!feature@lemmings
Feature: Clicking Buttons is a Life Saving Activity
  @saved @button
  Scenario: Click a button, save a life!
    Then  I save a lemming's life
```

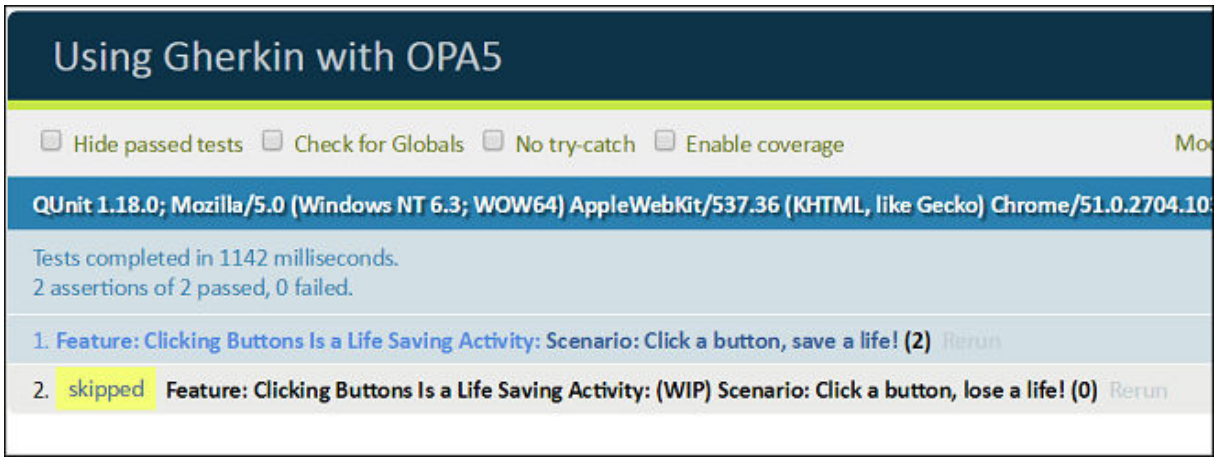
Tags on a feature level are inherited by all of the scenarios and scenario outlines in the feature. In addition, tags on a scenario outline are inherited by its examples. In the example above, the scenario has three tags: `@lemmings`, `@saved`, and `@button`.

Tags generally help to understand the feature file, and have a number of interesting uses:

- Tags can function like a category to create collections of features or scenarios, for example: `@sales` or `@human-resources`.
- Tags can be used to refer to numbered documents, for example: `@BCP-1234567890`.
- Tags can refer to the stage of the development process for that feature, for example: `@requirements`, `@development`, or `@testing`.

There is one special tag : the `@wip` tag. This tag indicates to the Gherkin test harness that it should skip that test during test execution. A skipped test is not executed and passes automatically. Use the `@wip` tag when you're in the middle of implementing the tests for a feature file. You can also use it for scenarios or features that you have no intention of testing.

Here's an example of a test execution with a skipped test:



Background scenarios

When writing a feature file, some test steps might need to be executed for every scenario. For example, the test step that loads the app is often repeated for each test scenario.

```
#!featureFeature: Clicking Buttons is a Life Saving Activity
  Scenario: Click a button, save a life!
    Given I have started the app
    Then I save a lemming's life
  Scenario: The saved lemming has a name
    Given I have started the app
    Then I see Alice at the end of the list
```

You can consolidate all of the repeated steps into a single "background scenario", which uses the keyword `Background`. The test steps in the background scenario get executed at the beginning of each scenario in the feature file. The following feature file is equivalent to the feature file shown above:

```
#!featureFeature: Clicking Buttons is a Life Saving Activity
```

```
  Background:
    Given I have started the app
  Scenario: Click a button, save a life!
    Then I save a lemming's life
  Scenario: The saved lemming has a name
    Then I see Alice at the end of the list
```

Step arguments and regular expressions

When writing test steps in feature files, test steps are sometimes repeated, but with a slight variation in each step.

```
#!featureScenario: Save one lemming
  When I click on the life saving button 1 time
  Then I see Alice at the end of the list of saved lemmings
Scenario: Save two lemmings
  When I click on the life saving button 2 times
  Then I see Bob at the end of the list of saved lemmings
```

To write a steps file for the this feature file you might have to write four separate step definitions. The problem would only get worse if you needed to write more scenarios. However, using step arguments you can consolidate the four step definitions into two step definitions (written here in pseudo-code):

```
I click on the life saving button <X> time(s)

I see <NAME> at the end of the list of saved lemmings
```

How does this work in real JavaScript code? When you write the regular expression for the step definition, you can use a regular expression concept called "capturing groups" to specify arguments to extract from the natural language of the test step. If you've never worked with regular expressions before, it can take some getting used to, but it's a really powerful tool that is worth learning. The capturing groups are passed to the test function as parameters (of type `string`) that you can name whatever you want. Continuing the example above, here are the step definitions that you could write:

```
this.register(
  /^I click on the life saving button (\d+) times?$/i,
  function(sNumTimes) {}
);
this.register(
  /^I see (.*?) at the end of the list of saved lemmings$/i,
  function(sName) {}
);
```

> ⚠ Caution
>
> All parameters extracted from capturing groups are of the JavaScript type `string`. You will need to use `parseInt` to convert numbers into type `int` before you do a numerical comparison.

Here are a few regular expression concepts that are especially useful in Gherkin:

```
(.*?) - captures any text into a parameter
(\d+) - captures any number into a parameter
\s* - matches 0 or more spaces
s? - matches the character "s" if it's there (replace "s" with any character)
(text)? - captures "text" into a parameter if it's there
(?:text)? - matches "text" if it's there, without capturing into a parameter
```

⚠ Caution

A common problem in regular expressions is that many characters are reserved and have a special meaning, in particular backslash (`\`), period (`.`), asterisk (`*`), plus (`+`), dash (`-`) and braces (`[]`, `()` and `{}`). Put the backslash character in front of a special character to treat it as plain text, for example: `\-` or `\+`.

If your regular expression contains multiple parameters, then they will be passed to the test function in the same order as they appear in the regular expression.

```
this.register(
  /^I click (\d+) times and see (.*?) at the end of the list$/i,
  function(sNumTimes, sName) {}
);
```

Context

Look at the following feature file scenario:

```
#!featureScenario: some steps depend on each other
  Given I have a Latte Cappuccino in front of me
  When I drink the coffee
  Then I feel less sleepy
```

Trying to implement the step definitions might be a bit challenging because in the second step, `I drink the coffee`, there is no mention of which coffee. Sometimes, to make a feature file sound more natural, or just to reduce repetition, it can be beneficial to retain the context from one test step to the next.

In Gherkin, the JavaScript `this` variable is unique for each scenario. Any variables assigned to one step definition can be used in subsequent step definitions within the same scenario. Each new scenario in the feature starts with a new `this` object. As a result, we could implement the previous feature file's step definitions in the following manner:

```
this.register(/^I have a (.*?) in front of me$/i, function(coffeeType) {
  this.coffeeType = coffeeType;
});
this.register(/^I drink the coffee$/i, function() {
  this.sleepinessBefore = user.getSleepiness();
  user.drink(this.coffeeType);
});
this.register(/^I feel less sleepy$/i, function() {
  Opa5.assert.ok(user.getSleepiness() < this.sleepinessBefore, "Verified...");
});
```

QUnit Assert Object

To use QUnit for automated testing, it is necessary to use QUnit's built-in assertion methods. QUnit defines these assertion methods in the `QUnit.assert` object. QUnit makes this object globally available to your test code, but it's a good practice to refer to the local `assert` object (particularly when you're doing asynchronous testing).

Gherkin makes the `assert` object available to you in two different ways, depending on whether you are using OPA5 or not. If you are using pure QUnit (no OPA5), then you can access the QUnit assert object inside of a step definition with `this.assert`.

```
this.register(/^I have launched my wombat$/i, function() {
  this.assert.strictEqual(this.myWombat.state, "launched");
});
```

If you are using OPA5, then OPA5 makes the QUnit `assert` object available to you inside a step definition via `Opa5.assert`:

```
this.register(/^My wombat is currently in orbit$/i, function() {
  Opa5.assert.strictEqual(this.myWombat.state, "orbit");
});
```

Data tables

If you want to use a large amount of structured data in your test, you can use a data table. In a feature file, a data table is placed underneath a test step and is composed of rows and columns, with rows separated by line breaks, and columns surrounded with the pipe (`|`) character.

```
#!featureScenario: lots of data
  Given I see the following lemmings:
    | Name    | Age in Months | Role         |
    | Alice   | 24            | Support      |
    | Bob     | 70            |              |
    | Charlie | 120           | Stories      |
```

In the steps file, if a data table is included in the test scenario then an extra parameter is passed at the end of the step definition function (after any step arguments that appear in the regular expression).

```
this.register(
  /^I see the following (.*?):$/i,
  function(sAnimalType, aDataTable) {}
);
```

Data tables are usually passed to the test function as a two-dimensional array (an array of arrays). For example, the above feature file data table would produce the following array in a variable `aDataTable` at runtime:

```
#!feature[
  ["Name", "Age in Months", "Role"],
  ["Alice", "24", "Support"],
  ["Bob", "70", ""],
  ["Charlie", "120", "Stories"]
]
```

If the feature file data is a single row or a single column, then the test function receives a simple array instead of a two-dimensional array.

```
#!featureScenario: lots of data
  Given I see the following lemmings:
    | Alice   |
    | Bob     |
    | Charlie |
  And I see the following lemmings:
```

```
    | Alice | Bob | Charlie |
```

Both test steps will provide the following runtime value for `aDataTable`:

```
["Alice", "Bob", "Charlie"]
```

Data table utilities

The contents of the data table in the feature file are sent to the step definition function with no modifications. Although this raw format is often useful, sometimes a different format would be more helpful. There is a Gherkin namespace called `dataTableUtils` that makes this reformatting task easy. This namespace provides several utilities including the function `toTable`, which transforms the two-dimensional array into a simple array of objects. In the array of objects, each object's attribute names are derived from the header line in the table. For example, consider the following feature file:

```
#!featureScenario: lots of data
  Given I see the following lemmings:
    | Name    | Age in Months | Role         |
    | Alice   | 24            | Support      |
    | Bob     | 70            |              |
    | Charlie | 120           | Stories      |
```

And the following steps file:

```
this.register(
  /^I see the following lemmings:$/i, function(aRawData) {
    var aData = dataTableUtils.toTable(aRawData, "camelCase");
  }
);
```

In the above steps code, we ask the `dataTableUtils` to use camel case when setting the names of the object attributes. (For those unfamiliar with coding conventions, camel case transforms the string "Hello World" into `helloWorld`.) At runtime, the variable `aData` is assigned the following value:

```
 [
  {
    ageInMonths: "24"
    name: "Alice"
    role: "Support"
  },
  {
    ageInMonths: "70"
    name: "Bob"
    role: ""
  },
  {
    ageInMonths: "120"
    name: "Charlie"
    role: "Stories"
  }
]
```

In this `toTable` format, the data is now easier to work with. There are other transformation functions and normalization functions available. For more information, see API Reference: `sap.ui.test.gherkin`.

If you have specialized normalization needs, you can also create your own normalization function. This is a function that accepts a single string parameter and returns a string. You could, for example, pass your custom normalization function into a `toTable` call like this:

```
var aData = dataTableUtils.toTable(aRawData, function(s) {
  return dataTableUtils.normalization.camelCase(s).replace("role", "job");
});
```

Scenario outlines

Sometimes you need to test a repeating pattern of steps. For example:

```
#!featureScenario: Save 1 Lemming
  When I click on the life saving button 1 time
  Then I see Alice at the end of the list of saved lemmings
Scenario: Save 2 Lemmings
  When I click on the life saving button 2 times
  Then I see Bob at the end of the list of saved lemmings
Scenario: Save 3 Lemmings
  When I click on the life saving button 3 times
  Then I see Charlie at the end of the list of saved lemmings
```

Step arguments make it easier to implement this in the steps file, but the repetition looks bad and is difficult to maintain. The solution is to use a scenario outline. With a scenario outline, you can write the test scenarios once, Gherkin will execute the test as many times as you specify, for whichever input examples you have given. Here's how it looks in the feature file:

```
#!featureScenario Outline: Using a scenario outline to Save Lemmings
  When I click on the life saving button <NUM CLICKS> times
  Then I see <NAME> at the end of the list of saved lemmings
Examples: list of lemmings
  | NUM CLICKS | NAME     |
  |  1         |  Alice   |
  |  2         |  Bob     |
  |  3         |  Charlie |
```

The above scenario outline is equivalent to writing out the three scenarios separately. In the above feature file, `NUM CLICKS` and `NAME` are called "placeholders". At test execution, these placeholders get replaced automatically with the values in the examples table. A new test scenario is generated for each row in the `Examples` table.

Pay attention to the following details:

- Placeholders are case-sensitive, and can use spaces or punctuation.
- Placeholders are surrounded by angle brackets (`< >`) in the scenario outline steps, and without angle brackets in the `Examples` table.
- Each placeholder found in the scenario outline requires a column in the `Examples` table, with the header row holding the placeholders themselves.
- The examples section must be immediately after the scenario outline in the feature file.
- You can specify multiple sets of examples for a single scenario outline.

Basic Example How to Use Gherkin

Like test-driven development (TDD), behavior-driven development (BDD) with Gherkin encourages us to write more tests because you do it right from the beginning. Having more tests makes it cheaper and easier to maintain the code over time. Let's dive into the specifics using following an example.

The ideal pattern for a BDD iteration goes like this:

1. Write a scenario in the feature file.
2. Execute the test to verify that the step definition is not found.
   This might seem strange since we haven't actually written a test yet, but this way we can check to see that our feature file works.
3. Write the step definition in the steps file.
4. Execute the test, see the test fail.
5. Develop the missing code in the app.
6. Execute the test and watch it pass.
7. Return to step 1.

> **i** Note
>
> You can find the code for this example in the *Samples* in the Demo Kit at Using Gherkin with OPA5.

Write the first feature file

Do you like lemmings? According to legend, they occasionally throw themselves into the sea in a mass suicide attempt. Imagine that you are writing an app that allows you to save lemmings' lives by clicking a button. In the BDD style, the first thing you do is write a feature file to document what your app is supposed to do.

Requirements1.feature

Create the following feature file (make sure that you use file extension `.feature`):

```
#!featureFeature: Clicking Buttons is a Life Saving Activity
    Let's save some lemmings' lives
  Scenario: Click a button, save a life!
    Given I have started the app
      And I can see the life saving button
      And I check how many lemmings have been saved already
     When I click on the life saving button
     Then I save a lemming's life
```

Execute a test

Next we should execute the test. This might seem strange since we haven't actually written a test yet, but this way we can check to see that our feature file works. Also, if you are working in a large project, some of the tests might have already been written by a colleague. Gherkin notifies us of all of the missing tests, and then we can proceed to write them one by one.

To actually execute the test, we need to create an HTML bootstrap test runner file, and a `Steps.js` stub file.

Steps.js

Here is a stub `Steps.js` file, without any step definitions, to get you started. You need to adjust the path and file name in the call to `extend` to match your scenario:

```
sap.ui.define([
  "jquery.sap.global",
  "sap/ui/test/gherkin/StepDefinitions",
  "sap/ui/test/Opa5",
  "sap/ui/test/gherkin/dataTableUtils"
], function($, StepDefinitions, Opa5, dataTableUtils) {
  "use strict";
  return StepDefinitions.extend("GherkinWithOPA5.Steps", {
    init: function() {
    }
  });
});
```

To execute Gherkin tests, you need to find a version of SAPUI5 or OpenUI5 that works for you. Here are some possibilities, sorted in order from most stable to least stable:

- Stable: `https://sapui5.hana.ondemand.com/resources/sap-ui-core.js` or `https://openui5.hana.ondemand.com/resources/sap-ui-core.js`
- Beta: `https://openui5beta.hana.ondemand.com/resources/sap-ui-core.js`
- Nightly: `https://openui5nightly.hana.ondemand.com/resources/sap-ui-core.js`

Our examples all use the nightly OpenUI5 build.

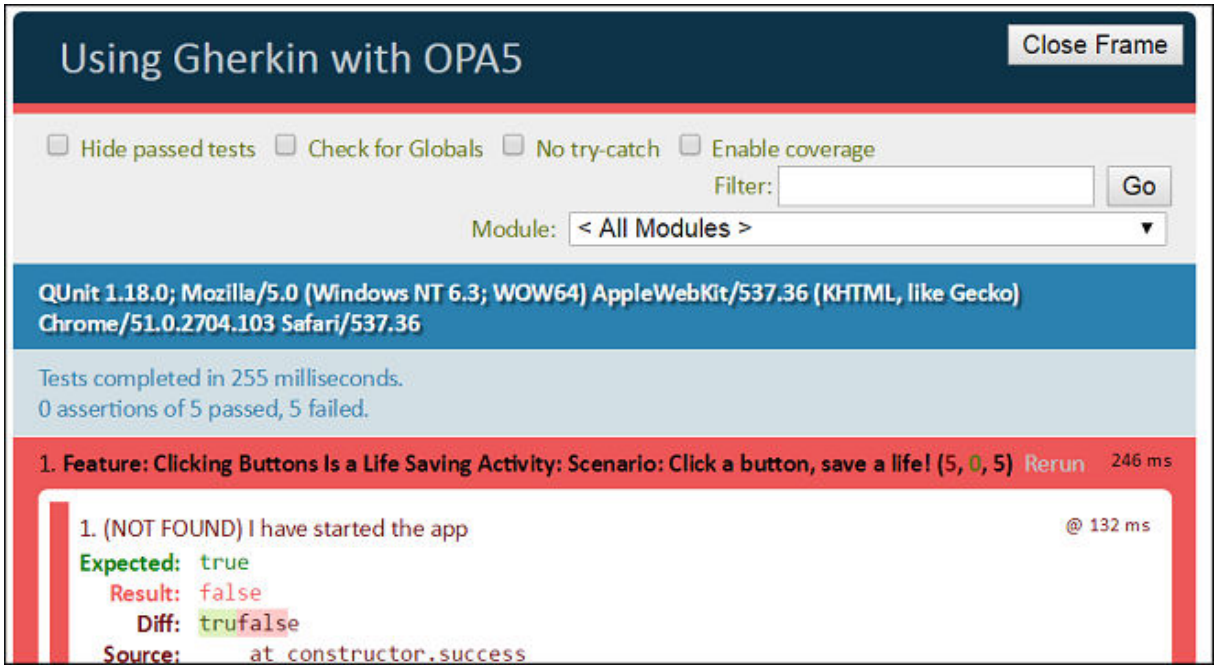
Website.html

Here is a sample HTML bootstrap file for Gherkin. In this example, the feature file is named `Requirements.feature` and the steps file is named `Steps.js`. Both are located in the same directory as your HTML bootstrap. You will need to adjust the SAPUI5 `src` (if you don't want to use the suggested build), SAPUI5 resourceroots, and the feature and steps file names to match your scenario and your app:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Using Gherkin with OPA5</title>
    <script
      id="sap-ui-bootstrap"
      src="https://openui5nightly.hana.ondemand.com/resources/sap-ui-core.js"
      data-sap-ui-resourceroots='{"GherkinWithOPA5": "./"}'
      data-sap-ui-loglevel="INFO"
    ></script>
    <script>
      sap.ui.require([
        "jquery.sap.global",
        "sap/ui/test/gherkin/opa5TestHarness",
        "GherkinWithOPA5/Steps"
      ], function($, opa5TestHarness, Steps) {
        "use strict";
        opa5TestHarness.test({
          featurePath: "GherkinWithOPA5/Requirements",
          steps: Steps
        });
      });
    </script>
  </head>
```

```
  <body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
  </body>
</html>
```

When you load the HTML file in your browser, the Gherkin tests are executed automatically . If you are using Google Chrome, you may need to start it with the command line flags `--allow-file-access-from-files --disable-web-security`. When everything is working correctly, you should see something like the following in your browser:



We expect the test to fail because we haven't written any tests yet. You'll notice that Gherkin has explained that it was not able to find a matching step definition for the first test step, "I have started the app", in the steps file. Scrolling down, you can see that none of the test steps have been found. We will need to write these step definitions.

Looking back at the feature file that we wrote, "I have started the app" is the first test step in the test scenario. Hence, it makes sense that we would see this test step first in the test results. You can also see the exact wording of the `Feature` and `Scenario` text that you entered: `Feature: Clicking Buttons Is a Life Saving Activity: Scenario: Click a button, save a life!`. This should make it easier for you to find your way around in the test results.

Write the first failing test

To verify the feature file, we will implement a steps file, which to recap is both the translation that allows the computer to understand the human-readable feature file, and also the verification steps (tests) to be run. Once you have a working feature file and can execute the test suite, then you are ready to write your first test. We will start by writing a simple test that we expect to fail.

In the `Steps` file, inside the `init` method, add the following code:

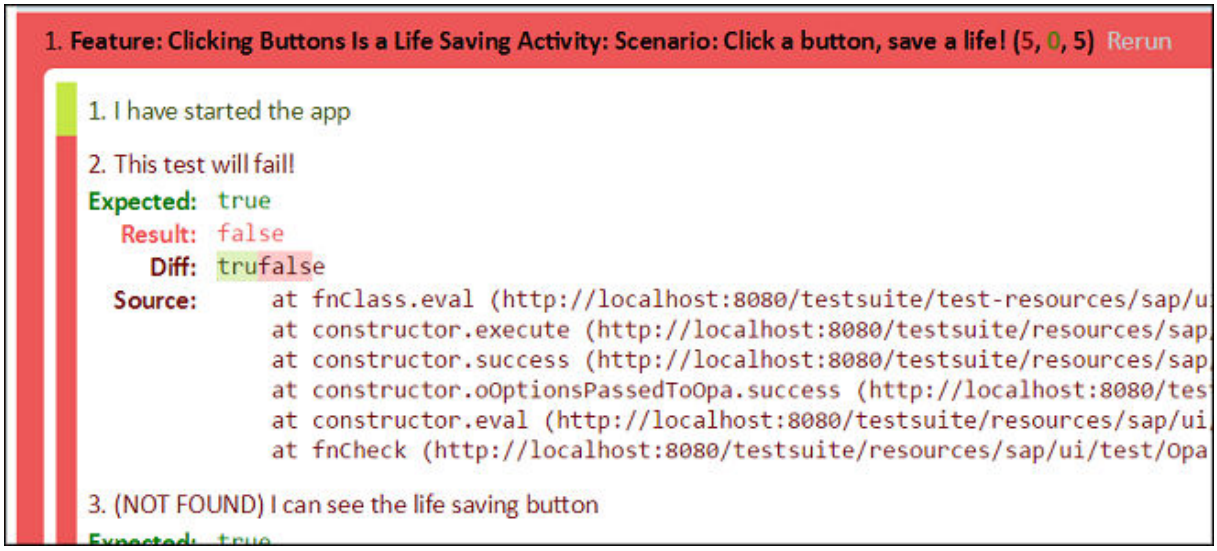
```
this.register(/^I have started the app$/i, function() {
  Opa5.assert.ok(false, 'This test will fail!');
});
```

The `register` method defines a new step definition and takes two arguments:

- a regular expression to match against the test steps in the feature file
- a function to execute when there is a match

At test execution time, the Gherkin test harness tries to find a step definition with a matching regular expression, and execute the step definition's test function.

Try executing the test now. You should see something like this:



Step 1 is green because a matching step definition was found in the steps file. In Gherkin, the test harness always checks for the existence of the step definition first before executing the step definition's function. After Gherkin finds a step definition, it executes the step definition's function, and thus executes any QUnit assertions inside the function.

In step 2, notice how the text "This test will fail!" is copied from the steps file. You can use this functionality to make it easier to debug your test. We recommend that you start any QUnit assertion comment with the word “Verified” to make it easier to read your test executions.

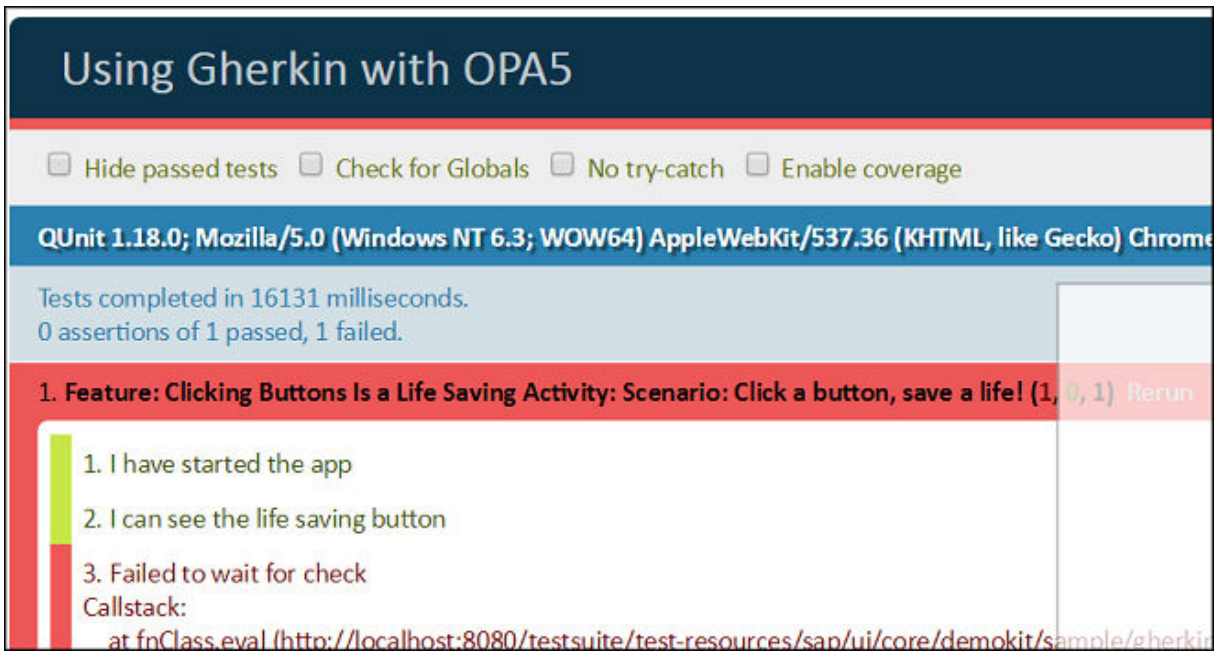
Write the second failing test

Let's write a bit more test code. To make a test useful, it will need to load your app and verify its properties. We will use OPA5 for this purpose. Replace the code inside your steps file's `init` method with the following code:

```
var oOpa5 = new Opa5();
this.register(/^I have started the app$/i, function() {
  oOpa5.iStartMyAppInAFrame("Website.html");
});
this.register(/^I can see the life saving button$/i, function() {
  oOpa5.waitFor({
    id: "life-saving-button",
```

```
    success: function(oButton) {
      Opa5.assert.strictEqual(oButton.getText(), "Save a Lemming",
      "Verified that we can see the life saving button");
    }
  });
});
```

You may need to adapt the above code to fit your situation. When you execute this code, you should see something like this:



There are several important things to note here:

For one, now that you are actually testing the app, you will see a popup overlay of the application under test appear in the bottom right corner of the window. This overlay is interactive, although you should wait until the test is complete before trying to interact with it. The overlay is extremely helpful for debugging your tests since at any given point in time you can see what state the app is in, particularly when the debugger is running and execution is paused. If the overlay is getting in the way, then after the tests have finished executing you can get rid of it by selecting the *Close Frame* checkbox at the top left.

In the above screenshot, steps 1 and 2 are passing because Gherkin was able to match the feature file test step to a step definition in the steps file. The test step "I have started the app" does not actually execute any verifications (that is, it does not call any QUnit assertion functions) and hence there is no verification occurring between "I have started the app" and "I can see the life saving button". Step 3 is the actual verification of the app executed inside the step definition "I can see the life saving button" function, and since in this example the app is an empty Web page, the test is failing. The error message `Failed to wait for check` is an OPA5 error that happens when the `waitFor` function fails to find the SAPUI5 control that's being searched for.

Write the first passing test

To make the "I can see the life saving button" test pass, you need to implement the app that is under test.

Website.html

Here is a simple stub for a test Web site (you may need to update the bootstrap source):

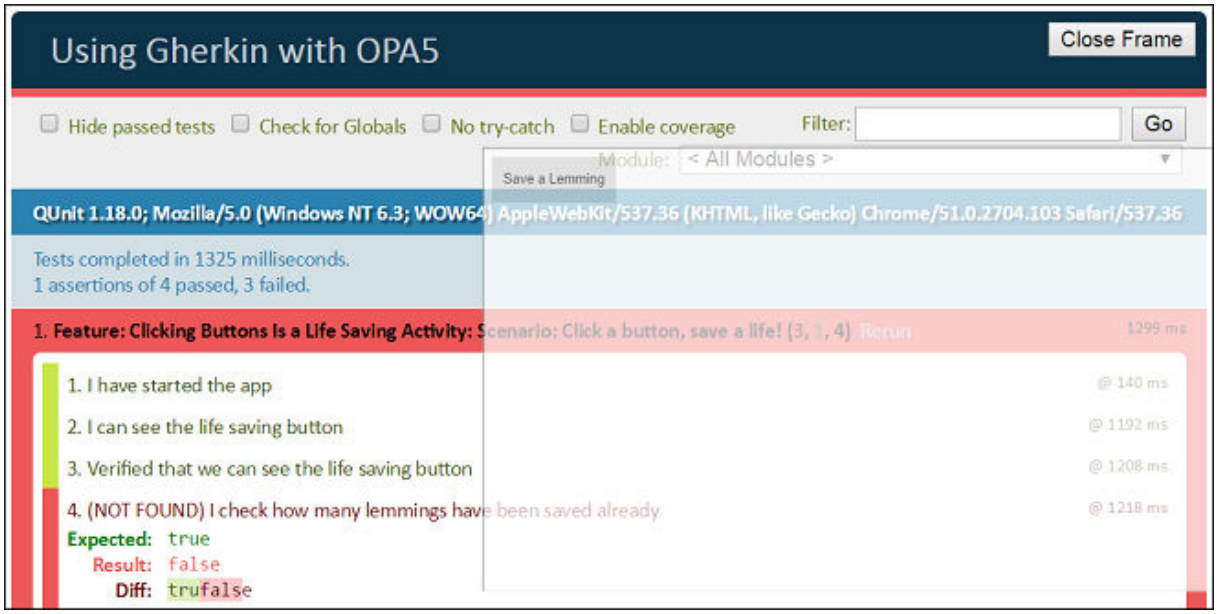
```
<html>
  <head>
    <title>Using Gherkin with OPA5 Website</title>
    <script
      id="sap-ui-bootstrap"
      src="https://openui5nightly.hana.ondemand.com/resources/sap-ui-core.js"
      data-sap-ui-libs="sap.m,sap.ui.layout"
    ></script>
    <script src="WebsiteCode.js"></script>
  </head>
  <body class="sapUiBody">
    <div id="uiArea"></div>
  </body>
</html>
```

WebsiteCode.js

Here's some simple code for an app:

```
sap.ui.getCore().attachInit(function() {
  "use strict";
  var oLayout = new sap.ui.layout.VerticalLayout({id: "layout"});
  var oButton = new sap.m.Button({
    id: "life-saving-button",
    text: "Save a Lemming",
    press: function() {}
  });
  oLayout.addContent(oButton);
  oLayout.placeAt("uiArea");
});
```

Now when you execute the test, you should see a passed verification step:



Steps 1 and 2 passed because the corresponding step definitions were found in the steps file. Here Gherkin is confirming that it was able to find the step definitions.

Step 3 was an actual verification step that executed a QUnit assertion to verify a property of the Web page.

Step 4 is failing because you haven't written that step definition yet.

Your next activity would be to write a step definition for step 4, execute the test and see it fail, then write the new code in the app, execute the test and see it pass, and so on.

Gherkin and OPA Page Objects

Gherkin is compatible with the concept of OPA5 page objects.

OPA5 page objects are a method for architecting integration testing to make test components more intuitive and reusable. For more information about OPA page objects, see Structuring OPA Tests With Page Objects [page 1201].

> **i Note**
>
> You can find a sample implementation in the *Samples* in the Demo Kit at Using Gherkin with OPA5 Page Objects.

To make Gherkin work with page objects, you should load your OPA5 page objects in the HTML bootstrap file, as shown in the sample. The only adaptation you need to make when starting the Gherkin testing is to add the parameter `generateMissingSteps` when calling `opa5TestHarness.test`:

```
opa5TestHarness.test({
  featurePath: "GherkinWithPageObjects/Requirements1",
  generateMissingSteps: true
});
```

This signals to Gherkin that if it cannot find a matching step definition in the steps file then it should try to use an OPA5 page object call instead. In the example above, no steps file is specified, which means that Gherkin will expect to make a page object call for each test step. You could also take a hybrid approach where each test step in the feature file either matches a Gherkin step definition or executes an OPA5 page object call. In addition, you can combine OPA5 page object calls with a Gherkin data table or scenario outline to achieve powerful results (you can see both options in the sample). Here is a sample feature file scenario that takes advantage of page objects:

```
#!featureScenario: Page 1 journey
  When on the overview: I press on "Go to Page 1"
  Then on page 1: I should see the page 1 text
```

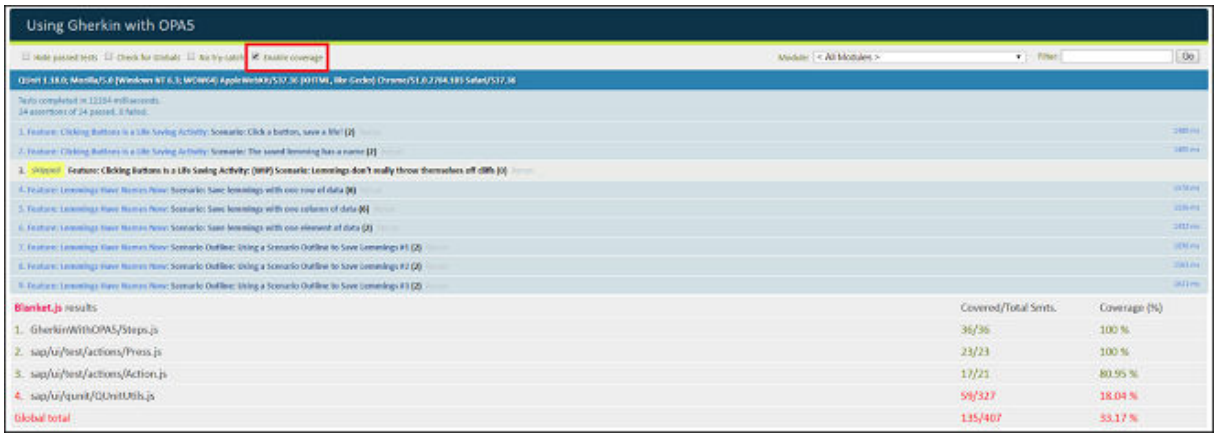
Use the Gherkin console logs to help you debug your OPA5 page object calls.

> **⚠ Caution**
>
> Chaining OPA5 page objects, for example, `When.onTheOverview.iPressOnGoToPage1().and.onPage1.iShouldSeeThePage1Text()` is currently **not** supported in Gherkin feature files.

Code Coverage

It can be handy to calculate the code coverage of your integration tests, for example, to figure out whether you forgot to test something or to provide statistics on your test quality.

At test execution time, Gherkin offers the option *Enable coverage* at the top left of the test results. Enabling the option reruns the tests and then lists the files that were tested at the bottom of the page.



Gherkin calculates code coverage for any JavaScript file that is loaded after the test harness. This may cause some system libraries to appear in the results. You can specify which files to calculate code coverage for by adding code to your HTML bootstrap file (after loading SAPUI5, but before running your tests), as follows.

```
<script
  src="path/to/resources/sap/ui/qunit/qunit-coverage.js"
  data-sap-ui-cover-only="GherkinWithOPA5/"
  data-sap-ui-cover-never="sap/ui/">
</script>
```

For more information, see the documentation for Blanket.js on GitHub. Keep in mind that the attribute name is slightly different in the SAPUI5 implementation (`data-sap-ui-cover-only` instead of `data-cover-only`).

Related Information

`Blanket.js` Documentation on GitHub
Code Coverage Measurement [page 1168]

Logging

During tesing with Gherkin, errors are logged to the test execution Web page.

Most error messages are sufficient to figure out what has gone wrong, for example, if an OPA5 `waitFor` call is failing. Gherkin also logs information to the JavaScript console with the prefix `[GHERKIN]` at priority `INFO`.

> **i Note**
>
> If at test execution time you don't see the logs, then SAPUI5 might not be logging down to this level.
>
> Check the bootstrap in the HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Using Gherkin with OPA5</title>
    <script
      id="sap-ui-bootstrap"
      src="https://openui5nightly.hana.ondemand.com/resources/sap-ui-core.js"
      data-sap-ui-resourceroots='{"GherkinWithOPA5": "./"}'
      data-sap-ui-loglevel="INFO">
    </script>
```

Example

Here are some examples of Gherkin console logs:

- `[GHERKIN] Running feature: 'Feature: Clicking Buttons Is a Life Saving Activity'`
- `[GHERKIN] Running scenario: 'Scenario: Click a button, save a life!'`
- `[GHERKIN] Running step: text='I see Alice' regex='/^I see (.*?)$/i'`

These logs are particularly helpful for telling you which regular expression (the `regex` attribute from step definition in a steps file) was matched with a particular feature file test step (the `text` attribute). This can help you search for the relevant regular expression among your step definitions (when your test code is large), and could also help with troubleshooting if an unexpected regular expression is being matched.

Frequently Asked Questions

How can I avoid timeouts?

There are two situations where you might have trouble with timeouts:

- When loading your app
- When trying to find SAPUI5 controls in the app

When loading the app, the OPA5 command can accept a parameterdefining the number of seconds to wait for the application to load.

```
var opa5 = new Opa5();
opa5.iStartMyAppInAFrame("path/to/your/app.html", 30); // wait time in seconds
```

When trying to find SAPUI5 controls in the app, you can add the following settings to cause OPA5 to wait a different amount of time for a control to become available on the screen. You also need to set the QUnit timeout (to a time equal to or greater than the OPA5 setting), otherwise QUnit might give up early:

```
sap.ui.test.Opa.config.timeout = 20; // wait time in seconds
QUnit.config.testTimeout = 20000; // wait time in milliseconds
```

Who should write feature files?

Ideally, the Product Owner develops feature files together with developers. Depending on how familiar Product Owners are with the tool, they can upload the feature files directly to source control, or let the developers do the upload.

How can I avoid inadequacies when writing the step definitions?

You may find that there is a gap between what the Product Owner has written, and the information that the developer requires to implement an integration test. Developers can modify the feature file to enable testing, as long as they check back with the Product Owner to ensure that the feature file is still correct.

Every time the Product Owner modifies the feature file, it breaks the tests How can I avoid this?

Actually, this is the point of Gherkin. When modifications to a feature file are uploaded to source control, we expect this to break existing tests, since the application's expected behavior has changed — but the application itself has not changed yet. Gherkin is forcing the application to stay in sync with the feature file. Consider that this also encourages you to be more honest about accepting new feature changes into your product — and the amount of extra work that this entails. We recommend that you use your formal code review process to allow developers to change the application and fix the tests, and then submit all the new code together with the feature file changes at the same time into the master branch.

Test Recorder

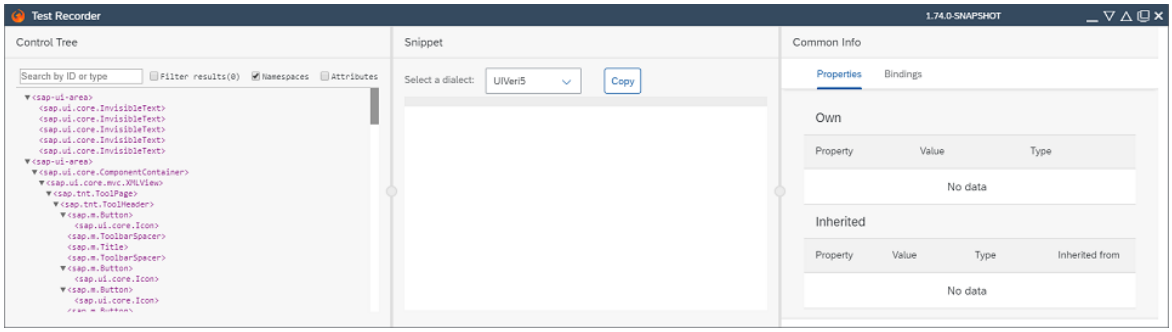
The Test Recorder tool supports app developers who write integration and system tests.

The Test Recorder is part of the SAPUI5 framework and is available in all browsers. As of version 1.74, you can use the tool in any SAPUI5 app to inspect the rendered user interface (UI), view the control properties, and gain hints on writing tests. The Test Recorder is aligned with the two official SAPUI5 testing tools – OPA5 and UIVeri5.

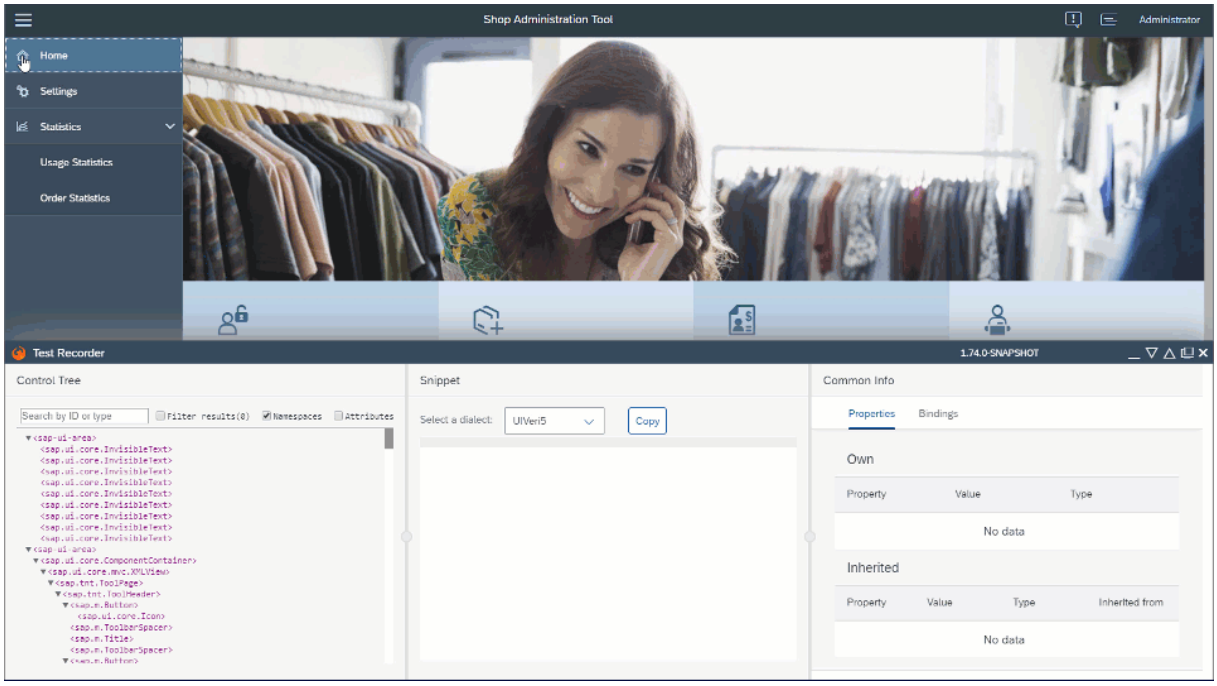
Getting Started

There are two ways to open the Test Recorder:

- In a separate window: Press CTRL + SHIFT + ALT + T
- In an IFrame: Press CTRL + SHIFT + ALT + P to display the *Technical Information Dialog* and then choose *Activate Test Recorder*



The main sections of the tool are *Control Tree*, *Snippet*, and *Common Info*.



From the navigation actions at the top bar of the Test Recorder, you can minimize, resize, open it in a new window, or close the tool.

Control Tree and Common Information

In the *Control Tree* section, you can see the DOM structure of the current app page. When navigating to another page or view, the tree is automatically updated.

You can display more information in the *Control Tree* by selecting the *Namespaces* and *Attributes* checkboxes. Entering text in the *Search* field highlights all elements that (partially) match by namespace, control name, or attribute values.

> **i Note**
>
> Elements in the *Control Tree* get highlighted if there's a match by namespace or attribute value even when the *Namespaces* and *Attributes* checkboxes aren't selected and the information isn't visible.

There are three general types of testing-relevant information that you can gather for any control:

- Position in the rendered control tree, type and ID – displayed in the *Control Tree* section. To see the control type, select the *Namespace* checkbox. To see the ID, select the *Attributes* checkbox.
- Properties (either own or inherited) – displayed in the *Common Info* section on the *Properties* tab.
- Bindings (binding context, properties, and aggregations) – displayed in the *Common Info* section on the *Bindings* tab.

> **i Note**
>
> - If an ID is not stable (because it was generated automatically), it's not suitable for tests. Unstable IDs start with a double underscore.
> - Many controls can have the same property or binding values. Therefore, when you use them in a control locator, the test finds multiple controls. This is a valid scenario, but it's always more reliable to locate only one control with a highly specific locator.

Actions

You can perform the following actions on controls, either from the *Control Tree* or from the rendered UI of the app:

- **Highlight:** Generates a code snippet for finding the control, which can be used to assert the control state.
- **Press:** Generates a code snippet for pressing on the control.
- **Enter Text:** Generates a code snippet for entering text into the control.

To perform an action from the *Control Tree*, right-click and choose *Press* or *Enter Text* in the context menu. If you want to *highlight* the respective control in the rendered UI, simply select the desired element in the tree.

To perform an action from the app page, right-click on any control and select the desired action from the context menu (the respective control is highlighted in the *Control Tree*).

> **i Note**
>
> A *Press* or *Enter Text* action snippet is generated irrespective of whether the control accepts such interactions. Keep in mind that such a snippet is not suitable for tests.

Snippets

The code snippets generated by the Test Recorder usually contain a function invocation that locates one control on the app page. The function receives one argument – a control locator. The control location is a JSON object containing a specific combination of conditions and matchers.

The code snippet can be directly copied and pasted into your test code and is already aligned with the supported tools for testing – *OPA5* and *UIVeri5*. To choose the tool for which to generate a code snippet, select an option from the *Dialect* dropdown menu. The *raw selector* option gives you just the control locator with no function invocations.

Related Information

Integration Testing with One Page Acceptance Tests (OPA5) [page 1182]
Stable IDs: All You Need to Know [page 1442]
API Reference: `sap.ui.test.Opa5`
GitHub: `ui5-ui5veri5`

Theming

SAPUI5 is an HTML UI library, therefore styling is done using Cascading Style Sheets (CSS). This allows for creating an impressive visual experience using a widely known standard technology which is well-accepted on the market.

SAPUI5 supports you when creating and using different visual designs - called **themes** - that can be used alternatively and switched on the fly. This way, the same application can look very different, depending on the user's design preference or accessibility requirements. Existing themes can serve as a basis for new themes and, in case of new design trends, it is possible to create a matching theme for all existing applications without modifying the applications. The theme handling is decoupled from application development and done in a separate layer. The SAPUI5 library loads the required CSS files and offers ways of switching themes.

On top of pure CSS, SAPUI5 offers a variety of optional features that add value regarding modularization, modification, compatibility, and performance:

- CSS variables, mixins, color calculations and other functions, provided by the Open Source library LESS
- In particular, CSS variables are used for centrally defined and centrally modifiable colors
- Compilation of one CSS file per control library from modular per-control CSS files
- Optimization/compression of CSS size
- Clean browser switch and mobile platform detection available (inside CSS code)
- Base theme (as a basis for a style that is always required to reduce the amount of CSS required for specific themes)
- Generic right-to-left support

To ensure these functions, SAPUI5 uses the following components:

- A CSS generator with several functions: LESS processing (CSS variables substitution etc.), merge of CSS files created for different themes and controls for optimal runtime consumption, as well as compression or right-to-left substitution if required.
- The SAPUI5 runtime handles the loading of the appropriate CSS file for the control libraries used in the application page by adding `<style>` tags to the document head. There is also an API available for switching themes, which replaces the CSS URLs and therefore does not modify the application state.

How to Theme Your SAPUI5 Application

To theme your application, you can choose among a number of options:

- Adapt an existing theme by using the UI theme designer, which basically modifies the color scheme, but in a very easy, non-technical manner with instant live preview. Adaptation parameters are limited, but the UI theme designer also lets you add custom CSS, which gives you the freedom to adapt basically everything.
- Create a new theme from scratch, writing every piece of CSS which will then be loaded later. The only requirement is to have `library.css` files within a certain folder structure (which also defines the theme name).
- Adapt an existing theme by adding CSS on application level. This is the easiest option and still sufficient for many use cases. You can technically adapt and change everything. The adaptation is rather done on top of an existing theme and only available within the specific application.

All options except the last one result in a new stand-alone theme which needs to be deployed and referenced by its name in the application and which can be used by any application.

For all these options, the CSS developer might reduce the styling effort and focus on those controls which are actually used in the application (which in turn decreases the reuse value of the theme in other applications).

Developing Custom HTML or Your Own Control – What to Bear in Mind

- To ensure that your SAPUI5 application's theme can be adapted easily, you should follow some recommendations.
  For more information, see Creating Themable User Interfaces [page 1261].
- To ensure that your custom content fits the colors of the SAPUI5 theme used, you can use specific CSS classes.
  For more information, see CSS Classes for Theme Parameters [page 1262].

More Information

UI Theme Designer

Setting Themes

You define which theme is used by your app either in the bootstrap, by using a URL parameter, or by using method `sap.ui.getCore.applyTheme`.

- The initial theme can be hardcoded in the application (in the script tag of the bootstrap loading SAPUI5) or in a JS configuration object defined before SAPUI5 is loaded, for example:

  ```
  <script id="sap-ui-bootstrap"
      type="text/javascript"
      src="resources/sap-ui-core.js"
      data-sap-ui-theme="sap_belize">
  </script>
  ```

  This setting has the lowest priority.
- A URL parameter (for example: `html?sap-ui-theme=sap_belize`) can be used when starting a SAPUI5 application to set or override the initial theme.
  If you use the UI theme designer to define your own custom theme, you can append the location of the custom theme as a server-relative path to the `sap-ui-theme` parameter, separated by an `@` sign:

  ```
  http://myserver.com/sap/myapp/?sap-ui-theme=my-theme@/sap/public/bc/themes/
  ~client-111
  ```

  Although a full URL can be specified, the framework will only use the path information of the URL to prevent CSS-based attacks that would otherwise be possible by referencing CSS from a malicious server. In a more complex landscape, for example, if the infrastructure of the UI theme designer is running on a separate server, either a Web dispatcher can be used to combine both servers in one namespace, or you should set a full URL using method `sap.ui.getCore.applyTheme` for custom apps as described below.

  > **i Note**
  >
  > The UI theme designer infrastructure stores themes for multiple technologies in the same location, each in its own subdirectory (`UI5/` for SAPUI5). Other SAP products (like SAP Enterprise Portal) append only the common root URL to the `sap-theme` parameter. SAPUI5 therefore appends folder `UI5/` to any given path that is defined in the `sap-theme` parameter.
- You can use method `sap.ui.getCore.applyTheme` to switch themes on the fly. The application state is not lost, and there is no server roundtrip (except for loading the CSS, if not cached). Only the style sheets are exchanged.
  You can specify a second parameter containing the root URL of the theme. The URL is not restricted in any way, therefore the caller has to make sure that the URL is valid and safe. If the URL points to the theme infrastructure, it must contain the folder suffix `UI5/`.
  For more information, see the API Reference.

  > **i Note**
  >
  > This option allows you to switch themes in your app during runtime.

Using Custom Themes

To load an external custom theme, you set this theme either by static declaring in the page or by using the `Core.setThemeRoot()` method. This is very much like using `registerModulePath()` for libraries that are in a different location. You can do this as follows:

1. Define the path to the theme with the following code: `sap.ui.getCore().setThemeRoot("my_theme", "http://url.to/the/root/dir");`. SAPUI5 then loads all theme resources from this URL. For example, the `library.css` file of the `sap.ui.core` library is loaded from: `http://url.to/the/root/dir/sap/ui/core/themes/my_theme/library.css`.
   This base directory can also be given as second argument to method `core.applyTheme(...)`.
   If some parts of the theme are in different locations, you can use the above call to set the default, but override the theme location for specific libraries by specifying them in an array as second parameter: `sap.ui.getCore().setThemeRoot("my_theme", ["my.lib.one","my.lib.two"], "http://url.to/the/other/root/dir");`
2. Configure the theme by using one of the following options:
   - Use the same object structure as JSON string in an attribute of the SAPUI5 bootstrap `script` tag, for example:

     ```
     <script id="sap-ui-bootstrap"
         type="text/javascript"
         src="resources/sap-ui-core.js"
         data-sap-ui-theme-roots='{"my_theme" : "http://themes.org/ui5"}'>
     </script>
     ```

   - Specify the location of a theme with a **URL parameter**:

     ```
     http://myserver.com/sap/myapp/?sap-ui-theme=my-theme@/sap/public/bc/themes/~client-111
     ```

   - Use the global configuration object. Insert the following before the bootstrap `script` tag:

     ```
     <script type="text/javascript">
     window["sap-ui-config"] = {
         themeRoots : {
             "my_preconfigured_theme" : "http://preconfig.com/ui5-themes",

             "my_second_preconfigured_theme" : {
                 "" : "http://preconfig.com/ui5-themes",
                 "sap.ui.core" : "http://core.com/ui5"
             }
         }
     }
     </script>
     ```

     The first theme is loaded for all libraries from the location specified. The second theme is loaded for the `sap.ui.core` library from the location specified. For all other libraries, the theme is loaded from the default location.

Theme Origin Whitelist

When configuring a theme with a `themeRoot` URL via the `sap-ui-theme/sap-theme` URL parameter, security restrictions apply. Absolute URLs to a different origin than the current page are stripped off by default. The path segment will be resolved relative to the current page origin.

In order to allow certain origins, according to RFC 6454, to be used via the URL parameter, a `<meta>` tag can be added to the `<head>` of the page:

```
<meta name="sap-allowedThemeOrigins" content="https://example.com">
```

This allows to load a theme from `https://example.com`, that is provided via the URL parameter:

```
https://myserver.com/sap/myapp/?sap-theme=my_theme@https://example.com/custom-themes/
```

Origins provided in the `<meta>` tag must contain the same protocol, host and port as the origin provided in the URL parameter. Multiple allowed origins can be separated with a comma.

A general wildcard (*) can also be used to allow all origins. However this should only be used in combination with additional security mechanisms such as CSP style-src directives. Wildcards to allow sub-domains are not supported.

Listening to the `ThemeChanged` Event

Whenever the theme is switched, an event is fired indicating that a theme switch has been triggered. If you want your application to react on this event, you can attach a handler to it:

```
sap.ui.getCore().attachThemeChanged(function(){
    myFunction();
});
```

You can use the `sap.ui.getCore.applyTheme` method to switch themes.

Enhanced Theming Concepts

On top of pure CSS, SAPUI5 offers advanced theming concepts and functions which can be used optionally. These concepts are outlined in detail below.

CSS Variables, Functions and More

SAPUI5 uses the popular CSS preprocessor LESS . This tool introduces several features, including CSS variables, a concept which has also been heavily demanded by the CSS community: In any UI5-controlled CSS file, variables can be defined and can then be referenced anywhere in the CSS code of the same library. These variables are mainly used for colors. All CSS variables are global. The CSS variable concept contributes to a consistent way of implementing and changing the styles.

LESS adds more features like color calculations, mixins, and CSS selector nesting. The color calculations are used in SAPUI5 to derive many different color shades from just a few variables.

Here is the syntax:

```
@sapUiText: #000000; /* define the text color as 'black' */
button {
   color: @sapUiText; /* buttons will automatically have the current text color,
which is '#000000' right now */
    [...]
}
```

LESS then takes care of substituting all references to a CSS variable by the current value of this variable. This happens during the build of the control library.

> **i** Note
>
> In development scenarios this LESS processing might even happen at runtime in the browser to shorten the build time for SAPUI5 libraries. This is indicated by a *less mode* rectangle when a page is launched.

Additionally, a specific theme can modify the CSS parameter values given by the base theme. So a control can just define its text color to use `sapUiText` by default which will automatically take care of applying the correct color for every theme or user modification: The theme generation will create one CSS file per theme, and the substitution of the CSS parameter references will always take the theme-dependent value into account. So a visually very different theme can easily be created by simply changing a number of colors.

While every library and control can introduce their own CSS variables, those defined in `global.css` are most important. Ideally, there should only be a few of them, and they should be simple enough to be understood by most end users (similar to what the Windows operating system offers end users), but still cover as many aspects of the visual appearance as necessary to make them sufficient for most customer-required theme modifications.

Additional benefits of CSS variables are, for example:

- They can be used to generically build simple styling tools that allow for a limited degree of freedom (=changing the CSS parameter values). The UI theme designer is an example of such a tool.
- They can also be linked to metadata, for example, to which group of colors they belong, to which colors they need to have some visual contrast.

Compilation of One CSS File

SAPUI5 CSS generation does not only substitute the CSS variable values but also merges all CSS files of a control library into **one** file that is loaded at runtime, thus increasing the performance.

Here are some of the reasons why you would not want to have all styles defined within one file during development:

- Less collisions and merging when different developers edit the styles of their controls
- Clear separation between the styles for different controls, which helps to estimate and test the impact of a CSS modification
- Keeping the door open for future optimization regarding runtime performance and data transfer by tailoring CSS files on server side that only contain the CSS required on the current application page

Optimization and Compression of CSS Size

For performance reasons, the SAPUI5 CSS generation can optionally remove all comments and unnecessary whitespace and can compress verbose declarations into a more compact format.

Base Theme for Generic Style

While some of the style and layout applied using CSS clearly depends on the current theme, and customers are likely to modify such style rules, there are other style rules that are required for a control to work properly and unlikely to differ for different themes. Examples are the overflow behavior, the positioning of popup elements, the mouse cursor type, the display mode, and others.

SAPUI5 promotes and supports keeping those style definitions in the so-called base theme, which serves as a common base for all themes.

Themes are built upon this base style that defines their specific visual design by applying colors and images, sizes and fonts. SAPUI5 theme generation takes care of combining the base theme with the specific theme for each generated theme CSS file. Because the specific CSS is appended to the base theme, a specific theme can always override styles defined in the base theme. If this step is required often, the respective style declaration should probably not be located in the base theme.

Any style declarations which are referencing CSS variables (at least those common ones defined in the base theme) can also be done in the base theme, and it is sufficient to do it only there: The CSS generation will apply the correct value for each respective theme. So this split between base theme and specific themes avoids duplicate creation and maintenance of CSS parts which are common for all themes and keeps the CSS files that need to be written for any new theme smaller.

Generic Right-to-Left Support

For some countries, right-to-left (RTL) text mode needs to be supported. In order to avoid the need to create a completely new set of CSS files for those countries, SAPUI5 supports generic RTL generation. Basically, this involves switching the right and left margins and mirroring everything else (including CSS3 rotations, and so on).

Everything else that is not covered by this automatic transformation can be fixed by using style rules that are only applied in the RTL case.

If you override RTL-specific attributes like text alignment, positioning and so on, you have to write the RTL equivalent into your application CSS. The attributes below are critical for RTL support:

- `float`
- `clear`
- `text-align`

The following attributes require special attention:

- `margin`
- `padding`

- `border`
- `background-position`
- `position` (right/left)
- `text-indent`

Depending on the values, these latter attributes might also need to be mirrored. As applications running in RTL mode add a `dir` attribute to the HTML tag in the DOM, custom styles that have to be written can use the following selector to handle RTL-specific styling:

```
html[dir="rtl"]  .myselector {
}
```

Creating Themable User Interfaces

There are several things you should keep in mind to ensure that an application can actually be themed.

General Aspects

Do not hard-code colors and fonts if you want your control or application to be themable in the theme designer tool.

For example, if you hard-coded the font color to black, this color cannot be adjusted when you apply a custom theme. This can be problematic if you want to apply a custom theme with a dark background color because the font color cannot be adjusted to a light color.

Stable Theme Parameters

- Themes in SAPUI5 are built with the CSS preprocessor LESS and make use of a complex parameter structure.
- You can view the entire set of basic theme parameters in this sample in the Demo Kit.
- SAPUI5 applications have access to these parameters at runtime via the API call `Parameters.get\(\)` from module `sap/ui/core/theming/Parameters`. These parameters not only differ in terms of themes, they are also not necessarily stable across different versions of SAPUI5. For this reason, you cannot rely on the completeness of the set of parameters.
- To allow developers to build theme-dependent, custom-styled applications or controls, a subset of roughly 70 parameters representing colors for different types of usage (such as border, backgrounds, charts) is available.
- The parameters in this subset can be considered as "stable", which means the following:
  - We will not change the naming for these parameters.
  - The contrast ratio between foreground- colors like text and the related background will remain stable.
- Always choose parameters that fit best from a semantic perspective, do **not** choose them based on their color value.

- This set is available across the predefined themes `sap_fiori_3_dark`, `sap_fiori_3`, `sap_belize`, `sap_belize_plus`, `sap_fiori_3_hcw`, and `sap_fiori_3_hcb` and should be handled with care. You should test your implementations for all themes to ensure the results are as expected.

  > **i** Note
  >
  > This approach can only cover the most common use cases. In addition, be aware that due to parameter value changes with future versions of SAPUI5 it might be necessary to rework your applications even when using the “stable” theme parameters described here.

- For charts, two individual sets are available, Chart Accent and Chart Semantic. Both sets are logically independent. Therefore only one of these sets is to be used consistently across individual charts.

Tips for SAPUI5 Application Developers Writing Custom HTML

- LESS parameters are not accessible via CSS and only support standard libraries.
- To ensure theme-dependent styling, only use basic theme parameters.
- Read appropriate parameter values via API and set the elements' CSS properties rather than hard-coding colors or borrowing arbitrary style classes from control sets:

```
sap.ui.require(["sap/ui/core/theming/Parameters"], function(Parameters){
    var myColor = Parameters.get("sapUiDarkBG");
});
```

Tips for SAPUI5 Control Developers

- Use the theme LESS parameters in CSS class definitions and rules.
- Use the appropriate level of parameters for the control, creating new ones as appropriate. For example, do not use color values or quick theming parameters directly in CSS rules.
- Follow the control parameter hierarchy conventions.

CSS Classes for Theme Parameters

SAPUI5 provides a set of essential adjustable colors behind the generic predefined CSS rules that enable custom content to use the respective CSS classes for the required colors.

If SAPUI5 applications define their own HTML and CSS content that is not created by any standard SAPUI5 controls, switching between different themes or adapting colors using the theme designer won't have any effect on these parts of the application.

The reason this doesn't work is because this type of styling cannot make use of the SAPUI5 theme parameters.

HTML content like that might be created as part of the following:

- JavaScript libraries that are **not** SAPUI5 libraries

- Custom/notepad controls(if they do **not** belong to a control library with a theme build)
- Plain static HTML or CSS content used in an application

To resolve this problem, you can use CSS classes. There is a number of predefined CSS classes that are supplied with color values by LESS CSS parameters of the current theme. These classes can be used in custom HTML content and in custom controls. The names of the CSS classes are generic and derived from the respective theme parameter name. This makes it easier to use the classes, and the names created are not too long or conflict-prone.

Example

The most straightforward example is the theme parameter `@sapUiText`. The theme parameter is mainly used for text colors, so the custom CSS rule sets the `color` property. Every parameter `sapUiXY` can be provided as a CSS class `sapThemeXY`. This suggests it is a theme color, and `sapTheme` is a new and reserved prefix for CSS classes.

```
   CSS
.sapThemeText {
   color: @sapUiText;
}
```

This solution is not sufficient if the same color is used for borders, for example. To support this, the color is defined for each CSS color parameter: Once as a text color, once as a background color, once as a border color, and so on. The styled CSS property name is part of the CSS class name, as a suffix:

```
   CSS
.sapThemeText-asColor {
   color: @sapUiText;
}
.sapThemeText-asBackgroundColor {
   background-color: @sapUiText;
}
.sapThemeText-asBorderColor {
   border-color: @sapUiText;
}
```

Nevertheless, if there is an obvious default CSS property, such as the (text) color for `@sapUiText` or the background color for `@sapUiPageBG`, this one is available without suffix.

If an application is using SAPUI5, and a theme is loaded into the page, any custom content like plain HTML, HTML inside HTML controls or HTML/XML views, or HTML rendered by custom/notepad controls can use theming if the respective generic CSS classes are added. If custom HTML should have the font color as defined in the current theme, the application writes:

```
<span class="sapThemeText">some custom text in custom HTML</span>
```

Whenever the theme is switched or the theme designer is used to modify the standard text color, this span automatically gets the new text color. The same is valid if a custom/notepad control is created. Just make sure the control writes the respective CSS class, for example, by calling `oRm.addClass("sapThemeText");`:

```
// the part creating the HTML:
render : function(oRm, oControl) {
oRm.openStart("div", oControl);
    oRm.style("width", oControl.getSize());
```

```
    oRm.style("height", oControl.getSize());
    oRm.class("mySquare");
    oRm.class("sapThemeText");  // here the CSS class is added which makes the
text color depend on the current theme
    oRm.openEnd();
    oRm.text(oControl.getText());
    oRm.close("div");
},
```

Related Information

List of Supported CSS Classes [page 1264]
Step 34: Custom Controls [page 170]

List of Supported CSS Classes

Overview of the CSS classes currently supported by SAPUI5.

CSS Class Name	CSS Property	sap_bluecrystal	All Themes	Description
sapThemeFontFamily	font-family	X	X	Default font
sapThemeFontSize	font-size	X	X	Default font size
sapThemeFont	font-family +font-size	X	X	Default font and font size
sapThemeText	color	X	X	Default text color
sapThemeText-asColor	color	X	X	Default text color
sapThemeText-asBackgroundColor	background-color	X	X	Default text color
sapThemeText-asBorderColor	border-color	X	X	Default text color
sapThemeText-asOutlineColor	outline-color	X	X	Default text color
sapThemeTextInverted	color	X	X	Default color of inverted text
sapThemeTextInverted-asColor	color	X	X	Default color of inverted text
sapThemeBaseBG	background-color	X	X	Base color for all backgrounds

CSS Class Name	CSS Property	sap_bluecrystal	All Themes	Description
sapThemeBaseBG-asBackgroundColor	background-color	X	X	Base color for all backgrounds
sapThemeBaseBG-asBorderColor	border-color	X	X	Base color for all backgrounds
sapThemeBaseBG-asColor	color	X	X	Base color for all backgrounds
sapThemeBrand-asColor	color	X	X	Brand color
sapThemeBrand-asBorderColor	border-color	X	X	Brand color
sapThemeBrand-asBackgroundColor	background-color	X	X	Brand color
sapThemeBrand-asOutlineColor	outline-color	X	X	Brand color
sapThemeHighlight-asColor	color	X	X	Color for highlighted elements
sapThemeHighlight-asBorderColor	border-color	X	X	Color for highlighted elements
sapThemeHighlight-asBackgroundColor	background-color	X	X	Color for highlighted elements
sapThemeHighlight-asOutlineColor	outline-color	X	X	Brand color
sapThemePageBG	background-color	X	-	Background color of mobile pages
sapThemePageBG-asColor	color	X	-	Background color of mobile pages
sapThemeBarBG	background-color	X	-	Background color for header bars in mobile pages
sapThemeBarHeading	color	X	-	Header text color for header bars in mobile pages
sapThemeBarText	color	X	-	Normal text color for header bars in mobile pages

CSS Class Name	CSS Property	sap_bluecrystal	All Themes	Description
sapThemeNegativeText	color	X	-	Semantic negative text color
sapThemeCriticalText	color	X	-	Semantic critical text color
sapThemePositiveText	color	X	-	Semantic positive text color
sapThemeLightText	color	X	-	Light text color
sapThemeMediumText	color	X	-	Medium text color
sapThemeDarkText	color	X	-	Dark text color

You can also check the availability of the `sapTheme` classes across the predefined themes *Blue Crystal* and *High Contrast Black* (HCB) in the sample in the Demo Kit.

Theming FAQ

Frequently asked questions regarding theming in SAPUI5

How can I adapt the visuals of a control?

While there is always the option to create a new theme, this is overkill for most minor style adaptations. For those minor changes, the recommendation is to include additional CSS into the page which changes the style of the respective tags of the SAPUI5 control. This allows complete, arbitrary changes of the visual design - after all it is the same technology that the UI5 controls use for their styling.

The main options are the following:

- Inspect the HTML and CSS of a control and write a similar, but adapted CSS rule for a CSS property you want to override for all controls of a type.
- Call `.addStyleClass("myStyle")` on some control instances if you want only those instances to look different from other instances - and then write CSS code that refers to the normal classes/tags and to the CSS class you just added.

> **i** Note
>
> - With this high degree of power and flexibility comes quite some responsibility. With CSS you can easily break the functionality of a control. This is not SAPUI5-specific, but when you make CSS adaptions, you should always have good knowledge of this open standard.
> - The inner structure of a control, the tag hierarchy, the IDs and CSS classes are **not** part of the public control API for which we guarantee stability. This is also the case for other UI libraries which might define some CSS classes as stable, but not everything else. As CSS can refer to the inner structures of

a control, you have to accept the risk that your style changes break when we change the inner structure. Changing the inner structure is a freedom we absolutely need to reserve, so we can fix bugs and add features of a control.

- When your CSS does not work as expected, use the developer tools in your browser to inspect the page and check which CSS rules are applied to the respective tag, and which rules might be applied but are overridden by other rules. If your rules are overridden by other rules, this is probably due to their order of appearance (last rule wins) or the CSS selector specificity (more specific CSS selectors win).

DON'Ts

- Do not adapt the style attribute of HTML elements belonging to SAPUI5 controls. When these controls are re-rendered, the changes will be lost.

How can I provide additional CSS that is not overwritten by the SAPUI5 CSS?

When SAPUI5 is used in a standard way, which means loaded by a`<script>` element in the `<head>` of a page, and all libraries declared in the respective attribute of the `script` tag), it is sufficient to just add the custom CSS to any place after the SAPUI5 `<script>` element. SAPUI5 will insert its CSS links immediately after the `<script>` tag, so any subsequent CSS will appear further down in the DOM and can thus overwrite the SAPUI5 CSS.

However, it is important to understand the precedence rules of CSS: The order of appearance is not the only factor that determines which one of two or more conflicting rules wins. Actually it is only the least important factor. The most important (and maybe least known) factor is the specificity of the selector belonging to a rule.

For example, if one rule says `button {color:red;}` to make all button texts red, and a second rule says `div > button {color:green;}` to make all button texts, which are direct children of a `<div>` element, green, the second rule always wins because it is more specific. The order of appearance in the DOM does not matter in this case. It would only matter if both rules started with an equal selector, such as `button{color:***}`.

The order of loading is completely irrelevant, only the position in the DOM counts in this case. If you load SAPUI5 without a `<script>` tag in the `<head>`, or if you do not specify all used control libraries in the `<script>` tag, but loaded some of them later on when the body was already loaded, you can still make sure a custom CSS appears further down in the DOM by loading it with `jQuery.sap.includeStyleSheet(stylesheetUrl[, id])` **after** loading SAPUI5 or the dynamically loaded control library.

Related Information

- For more information on the related part of the CSS specification, see http://www.w3.org/TR/CSS21/cascade.html#cascading-order
- For more information on specificity, see http://www.w3.org/TR/CSS21/cascade.html#specificity

Why do SAPUI5 controls not have a `style` property where I can make arbitrary changes?

A control usually does not map to **one** HTML element, but to a whole tree of HTML elements. Whatever is set for the `style` property would probably be added to the root element of this HTML tree, and only there, so there

is no `style` access to inner parts. If you just want to override the height of a button, this would actually work. But as soon as a change is a bit more complex, it will not work that easily. A more complex change is, for example, adapting the height of a `ComboBox` control. The outer `<div>` will get the proper height. And incidentally also the `<input>` tag inside, as it has 100% height set. But the dropdown arrow and the respective button-kind-of-thing has a fixed height, and the whole control will look pretty broken then.

In other cases, when HTML elements that break the CSS inheritance chain are nested, for example, `<table>` and font settings, you can change `style` to a different font and text color, but it will simply do nothing.

In general, we try to expose the obvious adaptation content in the API, for example, the button height. But the less obvious adaptations might have to be supported from inside the control to work properly, and as we cannot foresee and support everything you can do with a `style` property, we raise the bar a little bit by requiring you to write CSS (potentially using `.addStyleClass(...)` for the respective control). With CSS you can do what you cannot do with a `style` property: tweak the inner HTML components of a control.

Applications (at least the more traditional ones – currently this seems to be less of a rule, but I'm not sure it will stay like this forever) need to conform to some visual design guideline and, in general, it is not even desired that applications change the `TextField` height or use font just the way they like. As you can use CSS, UI5 still supports that, but we shouldn't make breaking the visual design a rule in our official API.

I am adding a style class, but it does not work! Why?

If you want to change some styling and use `control.addStyleClass(...)` to add a CSS class, but it does not seem to work, you first have to pin down exactly what is not working:

- Has the style class not been added to the HTML?
- Has the style class been added correctly, but the style you supplied not been applied by the browser?

You can check this by inspecting the HTML with your browser's developer tools.

- If the style class has really not been added to a control, bear in mind that some entities are not controls, but only elements (inherited from `sap.ui.core.Element`). Only some of them support `addStyleClass`.
- If the style class is available in the HTML, the bug is inside the CSS styles you supplied:
  - Are they loaded by the browser?
  - Are the selectors matching the element you want to style? You can again check in the developer tools: They mostly list all styles which apply, but some are overriding others (those are usually listed with a strikethrough). If your style is not listed at all, your CSS selector is probably not correct.
  - If your selector is fine, but other style rules override your styles (potentially those from the original UI5 theme), then the CSS precedence rules determined this. Refer to the section on additional CSS above and see http://www.w3.org/TR/CSS21/cascade.html#cascading-order for the respective part of the CSS spec and http://www.w3.org/TR/CSS21/cascade.html#specificity for more on specificity.
  - Maybe your browser does not understand the CSS styles you have written. Some browsers still display them in the developer tools, some don't, so you might want to try changing very common styles like the border to check whether selector and specificity are fine.

How can I perform a "clean" browser switch inside CSS code?

On all SAPUI5 application pages, the HTML root tag of the DOM gets the additional attribute `data-sap-ui-browser` where the value is the type and the current browser version. When browser-specific CSS needs to be written, this attribute can be used in CSS selectors.

```
html[data-sap-ui-browser="ie11"] button {  /* this rule will only be applied if
the current browser is Internet Explorer 11 */
   margin-top: 0px;
}
html[data-sap-ui-browser*="sf"] button {  /* this rule will only be applied if
the current browser is ANY version of Safari */
   padding-top: 0px;
}
```

When should I use the UI theme designer, and when should I perform manual steps?

There is not one single way to create a new theme, but there are several options. Which one you choose depends on several factors:

- How different is the desired design from an existing theme?
- Should the theme be used across several applications or just in one?
- Are sufficient CSS skills available?
- How much effort can be invested?

Depending on the answers it might be better to not even create a new theme but just adapt an existing one.

Localization

The framework concepts for text localization in SAPUI5 are aligned with the general concepts of the Java platform.

> **i** Note
>
> When connected to an SAP NetWeaver Application Server (ABAP), you can use the SAPUI5 repository to trigger the translation process on the ABAP server. For more information, see: Fallback: Translating Apps Using the SAPUI5 Text Repository [page 1517]

Identifying the Language Code / Locale

For the identification of languages, the framework uses a language code of type `string`.

The language can be set, for example, by using the following options:

- URL parameter `sap-ui-language` and configuration parameter `language`
- Script tag attribute `data-sap-ui-language`
- Global configuration variable `window["sap-ui-config"].language`
- URL parameter `sap-language`

These SAPUI5 configuration options accept the following formats:

- Language codes according to the de facto standard BCP-47, which are used by most browsers for language identification
  As of JDK 1.7 they are also supported by the Java locale class. Examples are `de`, `en-US`, `zh-Hans-CN`.
- Java locale syntax that combines a lower case ISO 639 alpha-2 or alpha-3 language code with an ISO 3166 alpha-2 country code
  Both codes are combined with an underscore. An arbitrary sequence of variant identifiers (also separated by underscores) can be appended as a third component. Examples are `de`, `en_US`, `zh_TW_Traditional`
- SAP proprietary language codes (only supported by URL parameter `sap-language`)
  SAPUI5 applications are often used to connect to ABAP-based SAP application servers. These servers use SAP proprietary language codes for compatibility reasons. These language codes often match an ISO 639 alpha-2 language code, but not in all cases. If the language code for an SAPUI5 application is specified with the URL parameter `sap-language`, SAPUI5 assumes that it is an SAP proprietary language code and converts it to a BCP-47 language tag as follows:

SAP Language Code	BCP47 Language Tag	Description
`ZH`	`zh-Hans`	`ZH` is the SAP language code for Simplified Chinese. The most generic representation in BCP47 is `zh-Hans`. `zh-CN` (Chinese, China) is another representation, but SAPUI5 decided to use `zh-Hans`.
`ZF`	`zh-Hant`	`ZF` is the SAP language code for Traditional Chinese. The most generic representation in BCP47 is `zh-Hant`. `zh-TW` (Chinese, Taiwan) is another representation, but SAPUI5 decided to use `zh-Hant`.
`1Q`	`en-US-x-saptrc`	`1Q` is a technical SAP language code used in support scenarios, for example for translation issues. When you select this language code, the technical keys are displayed instead of the actual data. As no ISO639 code for this exists, the information has been added as a BCP47 private extension to the `en-US` language tag: "trc" stands for "trace" or "traceability".
`2Q`	`en-US-x-sappsd`	`2Q` is also used as a technical SAP language code in support scenarios and displays a pseudo translation ("psd" in the private extensions name).

> **Note**
>
> Only these SAP-proprietary language codes are understood by SAPUI5. Other SAP-proprietary language codes are not automatically transformed. If you develop your app to run in the SAP Fiori launchpad, all other SAP-proprietary language codes are handled by the SAP Fiori launchpad.
>
> If you don't make use of the SAP Fiori launchpad, you may have to explicitly implement the language handling. You can use the `sap.ui.getCore().setLanguage()` method to provide both settings, a BCP47 language code and the corresponding SAP-proprietary language) in one call. SAPUI5 will then use one of the two codes where appropriate (e.g. BCP47 for the retrieval of translated texts or in HTTP Accept Headers, but the proprietary SAP language code when propagating the `sap-language` URL parameter to an OData service).

Current Language Code / Locale

SAPUI5 has the notion of a current language. It is determined during the SAPUI5 bootstrap from the following sources of information. The sources are ordered increasingly by priority and the last available user language/ locale wins:

1. Hard-coded SAPUI5 default locale `en`
2. Potentially configured browser language (`window.navigator.browserLanguage`); for Internet Explorer this is the language of the operating system
3. Potentially configured user language (`window.navigator.userLanguage`); for Internet Explorer this is the language in the region settings
4. General language information from the browser (`window.navigator.language`)
5. Android: Language contained in the user agent string (`window.navigator.userAgent`)
6. First language from the list of the user's preferred languages (`window.navigator.languages[0]`) (For more information, see https://developer.mozilla.org .)
7. Locale configured in the application coding (For more information, see API Reference: `sap.ui.core.Configuration`.)
8. Locale configured via URL parameters

After the bootstrap, the language can be changed by calling `sap.ui.getCore().setLanguage(...)`. A call to this method does not guarantee that all already existing translatable texts will be adapted. You use the configuration API to retrieve the resulting current language as follows:

```
var sCurrentLocale = sap.ui.getCore().getConfiguration().getLanguage();
```

For more information, see API Reference: `sap.ui.core.Configuration.setLanguage`.

> **Note**
>
> The syntax of the returned value depends on the syntax used for configuration. If the information source is one of the browser language properties, the returned language most likely is in BCP-47 format. If it is configured as a URL parameter, the user might have chosen the JDK Locale syntax.

> **Note**
>
> None of the `window.navigator.*` properties in Internet Explorer (IE) reflect the settings of the *Language Preference* dialog. Instead, IE returns the language of the Operating System installation as

`browserLanguage` and the language from the Operating System regional settings as `userLanguage`. As a result, the settings in the *Language Preference* dialog **cannot** be used for the current language of SAPUI5. This is often confusing for developers and a known shortcoming in IE. To circumvent this, an additional server request could be used where IE provides the corresponding setting in the`Accept-Language` header. This server request, however, requires a backend component. SAPUI5 must be able to run without a server component and, thus, the server request is not implemented.

Resource Bundles

A resource bundle file is a Java properties file (as described in the Javadoc of class `java.util.Properties`). It contains key-value pairs where the values are the language-dependent texts and the keys are language-independent and used by the application to identify and access the corresponding values.

Resource bundles are a collection of `*.properties` files. All files are named with the same base name (prefix identifying the resource bundle), an optional suffix that identifies the language contained in each file, and the fixed `.properties` extension. The language suffixes are formed according to the older JDK locale syntax. By convention, a file without a language suffix should exist and contain the raw untranslated texts in the developer's language. This file is used if no more suitable language can be found.

When a localized text is needed, the application uses the SAPUI5 APIs to load the properties file that matches the current language best. The same applies to any other localized data that can be represented as a string, for example, a date formatter string. To retrieve a text from the properties file, the application uses the (language-independent) key. If no text can be found for this key, the next best matching file is loaded and checked for the text. Finally, if no file matches, the raw file is loaded and checked.

Example

The resource bundle `sap.ui.commons.message_bundle` consists of the following individual files:

- `sap.ui.commons.message_bundle.properties`: Contains the raw texts form the developer, determines the set of keys
- `sap.ui.commons.message_bundle_en.properties`: Contains English texts (without a specific country)
- `sap.ui.commons.message_bundle_en_US.properties`: Contains texts in American English
- `sap.ui.commons.message_bundle_en_UK.properties`: Contains texts in British English
- `sap.ui.commons.message_bundle_de.properties`: Contains texts in German

To enable proper translation, you classify the texts with additional information, at least with the text type. Place the additional information in the line directly above the text element, beginning with the number sign (#). For more information, see Text Classification [page 1519].

A `properties` file can, for example, look like this

```
# SAPUI5 TRANSLATION-KEY <GUID>
#XMSG: A message to greet the world
helloWorld=Hello World
#XBUT,10: Save button text
buttonSave=Save
#XFLD,30: Greetings displayed in the upper right corner of the screen
welcome=Welcome {0}
```

> **i Note**
>
> If you are using SAPUI5 in SAP HANA, resource bundles files must have the extension `*.hdbtextbundle` instead of `*.properties`.
>
> To load this bundle, you add the following code to the `createContent` function of your view:

```
// "ResourceBundle" required from module "sap/base/i18n/ResourceBundle"
// load the resource bundle
var oBundle = ResourceBundle.create({
  // specify url of the .hdbtextbundle
  url : "i18n/messagebundle.hdbtextbundle"
});
```

Related Information

Resource Model [page 995]
API Reference: `sap.ui.model.resource.ResourceModel`

Use of Localized Texts in Applications

SAPUI5 provides two options to use localized texts in applications: The `sap/base/i18n/ResourceBundle` module and data binding.

Using sap/base/i18n/ResourceBundle

You can use the JavaScript module `sap/base/i18n/ResourceBundle` to access localized texts. The module contains APIs to load a resource bundle file from a given URL and for a given locale.

You can then use the `ResourceBundle.create` function to load the resource bundle from the given URL that is the bundle name, and for a provided locale. When no locale is specified, the default locale (en) is used. The following code snippet shows the use of the `ResourceBundle.create` function to return a Promise which resolves with a `sap/base/i18n/ResourceBundle`:

```
// "ResourceBundle" required from module "sap/base/i18n/ResourceBundle"
ResourceBundle.create({
    url : sUrl,
    locale: sLocale,
    async: true
}).then(function(oBundle) {
    // code
});
```

For more information, see `ResourceBundle` in the API Reference.

The resource bundle `sap/base/i18n/ResourceBundle` provides access to the localized texts that are contained in the resource bundle. You can use the `getText` method to access the texts in the loaded bundle by means of their key. This is shown in the following code snippet:

```
var sText = oBundle.getText(sKey);
```

Localization Test Page

The test suite provides a test page that shows how to use localized texts. This section only provides a short overview how the `sap/base/i18n/ResourceBundle` module is used there.

For a localized Web page you need the .html page itself and the .properties files of the required languages, in this example English and German.

The resource bundle `i18n.properties` is the English fallback version, which is the default version.

```
welcome=Welcome {0}. Please enter a new contact:
lastname=Last Name:
firstname=First Name:
street=Street:
zip=ZIP:
city=City:
```

The resource bundle `i18n_de.properties` contains the texts in German. The following code snippet shows the content of this file:

```
welcome=Willkommen {0}. Bitte geben Sie einen neuen Kontakt ein:
lastname=Nachname:
firstname=Vorname:
street=Straße:
zip=PLZ:
city=Ort:
```

The localization test page uses these texts to display a welcome message and a form to enter the address of a person. The coding of the test page looks as follows:

```
// "ResourceBundle" required from module "sap/base/i18n/ResourceBundle"
// "MatrixLayout" required from module "sap/ui/commons/layout/MatrixLayout"
// "Label" required from module "sap/ui/commons/Label"
// "TextField" required from module "sap/ui/commons/TextField"
// "TextView" required from module "sap/ui/commons/TextView"
var sLocale = sap.ui.getCore().getConfiguration().getLanguage();
ResourceBundle.create({url : "res/i18n.properties", locale:
sLocale}).then(function(oBundle) {
    var oMatrixLayout = new MatrixLayout();
    oMatrixLayout.setLayoutFixed(false);
    oMatrixLayout.createRow(
      new TextView({text: oBundle.getText("welcome", ["Administrator"])})
    );
    oMatrixLayout.getRows()[0].getCells()[0].setColSpan(2);
    oMatrixLayout.createRow(
      new Label({text: oBundle.getText("lastname")}),
      new TextField()
    );
    oMatrixLayout.createRow(
      new Label({text: oBundle.getText("firstname")}),
      new TextField()
    );
```

```
    oMatrixLayout.createRow(
      new Label({text: oBundle.getText("street")}),
    );
    oMatrixLayout.createRow(
      new Label({text: oBundle.getText("zip")}),
      new TextField()
    );
    oMatrixLayout.createRow(
      new Label({text: oBundle.getText("city")}),
      new TextField()
    );
    oMatrixLayout.placeAt("userForm");
});
```

With regard to localization, the code above defines the following procedure:

1. Require the `sap/base/i18n/ResourceBundle` module
2. Determine the language
3. Load the resource bundle
4. Access the text using the `welcome` key and pass the value for the placeholder (`{0}`) via an array
5. Access the text using the `lastname` key and set it as text for the `Label`

Data Binding

You can also use data binding to access localized texts. The `ResourceModel` is a wrapper for resource bundles that exposes the localized texts as a model for data binding. You use the `ResourceModel` to bind texts for control properties to language dependent resource bundle properties. You can instantiate the `ResourceModel` either with `bundleName` (name of a resource bundle that equals a SAPUI5 module name within the define/require concept), or a `bundleUrl`, which points to a resource bundle. When you use the bundle name, make sure that the file has a `.properties` suffix. If no `locale` is defined, the current language is used.

Example

```
// "ResourceModel" required from module "sap/ui/model/resource/ResourceModel"
// "Button" required from module "sap/ui/commons/Button"
 var oModel = new ResourceModel({
   bundleName:"myBundle",
   bundleLocale:"en",
   async: true
 });
 var oControl = new Button({
    id : "myButton",
    text : "{i18n>MY_BUTTON_TEXT}"
});
// attach the resource model with the symbolic name "i18n"
// The texts are resolved via databinding, once the resource bundle file was
loaded
oControl.setModel(oModel, "i18n");
```

Note

The current data binding implementation does not allow to pass parameters to your texts in the resource bundle. If you have to pass parameters, you must do this on your own. You can, however, access the resource bundle directly from the model instead of loading it:

```
oModel.getResourceBundle().then(function(oBundleInstance) {
```

```
    ...
});
```

After the instance has been created, you have a model containing the resource bundle texts as data.

For a complete overview of available methods and parameters, see `ResourceModel` in the *API Reference* in the Demo Kit

Related Information

Resource Model [page 995]

Accessibility

Accessibility features are essential for users with disabilities. In an ongoing approach, SAPUI5 controls aim to comply with various product standards such as screen reader support, high-contrast theming and keyboard handling.

The following topics provide important accessibility information related to SAPUI5 controls from an end-user perspective.

> **i** Note
>
> This documentation describes standard accessibility functionality in SAPUI5. For specific information on accessibility in products based on SAPUI5, see the documentation for the respective product. **In case of conflict between the SAPUI5 documentation and the respective product documentation describing accessibility functionality, the respective product documentation shall prevail.**
>
> Note that accessibility for web applications also depends on the browser and the operating system used. For more information, see the Accessibility Status Documents.

The following table shows the availability of the different accessibility features.

Table 39: Accessibility Feature Availability

Feature	Available as of version
Keyboard Handling	1.26
HCB Theme	1.26
Screen Reader Support	1.30
HCW Theme	1.46

Keyboard Handling for SAPUI5 UI Elements

SAPUI5 UI elements provide keyboard handling in order to improve accessibility and speed up work.

Keyboard Shortcuts for End Users

In this topic we introduce the main keyboard combinations that are used by SAPUI5 UI elements. Furthermore we describe some additional combinations that are used in specific cases.

Table 40: Main Keyboard Combinations

Key Combination	What it does	Specific Behavior
`Tab` / `Shift` + `Tab`	Focuses UI elements in order (forward / backward)	You can cycle through all interactive, enabled and visible UI elements that are part of a given dialog or other container. When the last UI element is focused, pressing the key again will move the focus to first element.
`Space`	Triggers an action (reversible)	Pressing and releasing the key triggers the action that is associated with a UI element (for example, open a link or toggle a button). → Tip If you press and hold the key, you can cancel the trigger action by pressing `Shift`.
`Enter`	Triggers an action (immediate)	Triggers the action that is associated with a UI element (for example, open a link or toggle a button). The action is triggered immediately after the key is pressed.
`Arrow Keys`	Navigates between elements	You can move the focus between elements within a complex UI element (for example a table or a list). Depending on the structure, this navigation is either one-directional (left/right or up/down) or two-directional (left, right, up, down).

Key Combination	What it does	Specific Behavior
`Home` / `End`	Navigates between elements	You can move the selection to the first/last element within a set of elements. **→ Tip** When using UI elements like sliders and rating indicators, pressing these keys will set the selected value to the maximum/minimum respectively.
`Page Up` / `Page Down`	Skips elements during navigation	You can move the selection forward/backward by several elements at a time. If the remaining number of elements is less than the step size, selection will move to the last/first element.
`Escape`	Closes element / Reverts changes	Depending on your situation, you can do the following: • Close an additionally opened element (for example, a pop-up). • Revert the execution to a previous step - one step at a time. • If you have made changes to the content of an element (for example, a text field), pressing this key will revert those changes.
`F4` / `Alt` + `Down` / `Alt` + `Up`	Opens / closes a drop-down menu	You can open the options and elements available in a drop-down menu, if the UI element in question provides this type of information.
`F6` / `Shift` + `F6`	Skips focus of UI elements (forward / backward)	UI elements within an application can be grouped together (for example, all elements in the header of an application). You can skip focusing the elements within a group by using these keys.

Table 41: Additional Keyboard Combinations for Specific UI elements

Key Combination	What it does	Specific Behavior
`Page Up` / `Page Down`	Date modification (Days)	When in input: Pressing these keys decreases/increases the date value by one day. When in Calendar UI: Pressing these keys decreases/increases the date value by one month.

Key Combination	What it does	Specific Behavior
Shift + Page Up / Shift + Page Down	Date modification (Months)	When in input: Pressing these keys decreases/increases the date value by one month. When in Calendar UI: Pressing these keys decreases/increases the date value by one year.
Ctrl + Shift + Page Up / Ctrl + Shift + Page Down	Date modification (Years)	When in input: Pressing these keys decreases/increases the date value by one year. When in Calendar UI: Pressing these keys decreases/increases the date value by 10 years.
Ctrl + Arrow Keys	When used with grouped radio buttons - Moves selection	Pressing these keys moves the focus to the next corresponding radio button in the group. The currently selected radio button does not change.
Ctrl + Arrow Keys	When editing tiles in a container - Changes tile position	Pressing these keys changes the position of the focused tile in the tile container. The remaining tiles are re-ordered accordingly.
F2	Toggles edit mode	You can toggle editing of an editable UI element like an input field or text area.
F7	Exits edit mode	When you are editing an input field in a table or a list, pressing this key will stop the editing and move the focus to the parent UI element. **i Note** Pressing F7 again will move focus back to the editable element.

Screen Reader Support for SAPUI5 Controls

SAPUI5 offers screen reader support in order to aid people with visual impairments. The implementation is based on the ARIA (Accessible Rich Internet Applications) standard.

General Information

Currently, the following libraries have screen reader support based on the ARIA standard:

- sap.f
- sap.m
- sap.suite.ui.commons
- sap.tnt
- sap.ui.commons
- sap.ui.comp
- sap.ui.core
- sap.ui.generic
- sap.ui.layout
- sap.ui.suite
- sap.ui.table
- sap.ui.unified
- sap.ui.ux3
- sap.uxap
- sap.viz

SAPUI5 controls provide the prerequisites for screen reader support based on the ARIA standard. All screen readers that implement this standard should work fine. If there are deviations in the interpretation, these need to be addressed to the screen reader vendor. If you need more information on our testing environment, see SAP Note 2564165.

> **Note**
>
> - No screen reader activation settings are necessary since the accessibility mode in SAPUI5 is switched on by default.

High Contrast Themes for SAPUI5 Controls

SAPUI5 offers two high contrast themes for controls - High Contrast Black (HCB) and High Contrast White (HCW). These themes support people with visual impairments and are required by the **Accessibility** product standard. The themes can be switched on by adding a dedicated URL parameter.

SAPUI5 library support

Currently, the following libraries support the high contrast themes:

- `sap.f`
- `sap.m`
- `sap.tnt`
- `sap.ui.comp`
- `sap.ui.layout`
- `sap.ui.suite`
- `sap.ui.table`
- `sap.ui.unified`
- `sap.uxap`
- `sap.viz`

Switching on the HCB theme

You can switch on the High Contrast Black theme by appending the `sap-ui-theme=sap_belize_hcb` URL parameter as in the following example.

> ### Example
>
> HCB Theme Enablement
>
> `http://<hostname>:<port>...?<parameter>=<value>&...&`**`sap-ui-theme=sap_belize_hcb`**

Switching on the HCW theme

You can switch on the High Contrast White theme by appending the `sap-ui-theme=sap_belize_hcw` URL parameter as in the following example.

> ### Example
>
> HCW Theme Enablement
>
> `http://<hostname>:<port>...?<parameter>=<value>&...&`**`sap-ui-theme=sap_belize_hcw`**

Drag and Drop

Drag and drop allows you to easily move, rearrange, and restructure items, for example, in a list or hierarchy structure.

Overview

Drag and drop in SAPUI5 enhances the standard browser events. A drag session (`DragSession`) is created that contains all information relevant for the drag-and-drop operation.

The central `DragAndDrop` handler manages the drag-and-drop scenarios of an application. The handler also creates the drag session for the data transfer and supports the custom dragging ghost element. The handler plugs into the pre- and post-event processing of the UI area to enhance native HTML5 drag-and-drop events.

Use

You can use drag and drop in various scenarios, for example, if you want to do the following:

- Rearrange items in a list
- Rearrange items in a hierarchy structure
- Rearrange items in a calendar, such as appointments in a planning calendar
- Move items from one control to another
- Move files from one application to another, for example, during a spreadsheet export or a file upload

Details

To drag an HTML element in HTML5, the draggable attribute must be set to `true`. This is done by `sap.ui.core.RenderManager` while the element data is being written to the Document Object Model (DOM) tree.

When the dragging of an HTML element has started, the `DragAndDrop` handler determines the responsible control and its relevant `DragInfo` class using the related `dragDropConfig` aggregation of the control. Between the pre- and post-processing of the `dragStart` event, owners of a control can decide whether to allow the dragging in their `ondragstart` handler.

> **i Note**
>
> Calling the `preventDefault` method on the `dragStart` event stops the dragging.

After that, the `dragStart` event is fired by the related `DragInfo` class of the control. At this point, application developers can change the default to prevent the dragging. Also, the drag session is now available and can be used to transfer data or to provide the custom dragging ghost element. After everything has been defined, the user can start dragging the control.

During the dragging and after the `dragEnter` event has been fired on an HTML element, the `DragAndDrop` handler determines the responsible control and its relevant `DropInfo` class using the related `dragDropConfig` aggregation of the control. Between the pre- and post-processing of the `dragEnter` event, the owner of a control can decide whether to allow the dropping in their `ondragenter` handler.

> **i Note**
>
> Marking the `dragEnter` event with the `NonDroppable` key (using the `setMark` method) prevents the dropping.

After that, the `dragEnter` event is fired from the relevant `DropInfo` class of the control. At this point, application developers can change the default to prevent the dropping. If dropping is not allowed, the user will see a non-droppable cursor. If dropping is allowed, the user will see a droppable cursor and the drop indicator depending on the `dropPosition`, `dropLayout`, and `dropEffect` properties of the first relevant `DropInfo` class of the control.

If `dragEnter` is allowed, the user can now drop an object by releasing the mouse. After that, the `drop` event gets fired for further implementation.

> **i Note**
>
> The `DragAndDrop` handler does not provide any default drop handler implementation. This is up to the application developers.

Related Information

API Reference: `sap.ui.core.dnd`
Sample

Drag-and-Drop Configuration

To use drag and drop, you have to provide the required configuration using the `dragDropConfig` aggregation in `sap.ui.core.Element`.

Overview

The `dragDropConfig` aggregation with multiplicity `0..n` is enabled for all controls and elements in SAPUI5. However, it has to be defined in the metadata first.

> **i Note**
>
> This configuration might be ignored due to metadata restrictions of `sap.ui.core.Element.extend`. For more information, see the API Reference: `sap.ui.core.Element.extend`.

To enable configuration, the following configuration entities for the aggregation are available:

- `DragInfo`
  This class can be used to enable dragging if the drop target is unknown, or if you are not the owner of the target. Additional checks can be done during the `dragStart` event, and the default behavior can be changed.
- `DropInfo`
  This class can be used as a general drop target. Incoming data might have to be validated during the `dragEnter` event. Applications have to implement the `drop` event.

Both `DragInfo` and `DropInfo` provide the `groupName` property. If this property has been specified, the `DropInfo` object only interacts with the relevant `DragInfo` classes within the same group.

- `DragDropInfo`
  This class can be used if the drag source and the drop target are closely connected, and both are known. The most common use case is rearranging items.

Related Information

API Reference: `dragDropConfig`
Drag-and-Drop Metadata [page 1284]

Drag-and-Drop Metadata

To influence the drag-and-drop behavior, use the metadata definition of a control.

Overview

You can use the `dnd` key for the drag-and-drop behavior of a control. Here is an example that shows you how the `dnd` key can be used:

```
Control.extend('my.CustomControl', {
      metadata : {
          properties : {
              value : { type : 'string' },
              width : { type : 'sap.ui.core.CSSSize', defaultValue : 'auto' }
          },
          dnd : { draggable: false, droppable: true },
          aggregations : {
              header : { type : "sap.ui.core.Control", multiple : false, dnd :
true },
              items : { type: 'sap.ui.core.Control', multiple : true, selector :
"#{id}-items", dnd : {
                        draggable: true, dropppable: true, layout: "Horizontal"
              } },
          }
     }
```

You can use the following attributes in the metadata of a control for drag and drop:

- `draggable`: Defines whether the control or aggregation is draggable
  Default value of `draggable` is `false`.
- `droppable`: Defines whether dropping is allowed for the control or within the control and/or from one of its aggregations to another one
  Default value of `droppable` is `false`.
- `layout`: Defines the arrangement of the items in the aggregation
  Possible values are `Vertical` (for example, rows in a table) and `Horizontal` (for example, columns in a table). Default value of `layout` is `Vertical`.
- `selector`: Defines the location of the aggregation in the control DOM
  This setting is recommended for the aggregation with cardinality `0..n`.

Related Information

API Reference: `sap.ui.core.Element.extend`

Drag-and-Drop Limitations

There are some known limitations when using drag and drop.

When you use drag and drop, the following limitations apply:

- Drag and drop is not supported on mobile devices.
- The transparency of the dragging ghost element and the cursor during drag-and-drop operations is not configurable.
- Defining constraints for the drag position is not possible.
- Texts in draggable controls cannot be selected. The text of input fields in draggable controls can be selected, but not dragged.
- Microsoft Internet Explorer 11 only supports the plain text MIME type for the `DataTransfer` object (DTO). Also, defining a custom dragging ghost element is not possible.

Note that drag and drop is not accessible. Applications must provide an alternative for users with special needs for drag-and-drop operations.

Spreadsheet Export

The spreadsheet export allows you to export data to an Office Open XML spreadsheet.

Overview

The spreadsheet export allows you to export your data to an Office Open XML document of category Spreadsheet (xlsx). You can export any type of content that has a tabular format, such as tables or lists.

If you use the `SmartTable` control to export data, you can also use different types of exports, the client export or the SAP Gateway export.

> → Tip
>
> The `SmartTable` control offers you all the preconfigured content you need for the export. You can either simply use it without having to define any configuration yourself, or you can adapt the content to make it fit your own specific requirements. For other entitities, you have to define the configuration manually as described below.

Prerequisites

If you want to export data manually, without `SmartTable`, you have to perform the following steps:

1. Load the `sap.ui.export.Spreadsheet` library within your controller coding.
   You can load the library during the initialization of your controller or whenever needed.
2. Define the configuration for the export for the following objects:
   - Columns
   - Data sources
   - Additional properties that are used for processing the export (optional)
   - Hierarchical data, if required
3. Start the export process.

Details

Loading During Initialization

If the library is loaded during the initialization of your controller, it is available across the whole lifecycle of the controller. All you need to do is add the library as a dependency to your existing `sap.ui.define` call. This mechanism ensures that the library is already loaded every time you use it. You don't need to take care of synchronous or asynchronous loading but the library is loaded even if an export is never triggered.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/export/Spreadsheet"
```

```
], function(Controller, Spreadsheet) {
    "use strict";

    return Controller.extend("sample.Spreadsheet", {

        // Place your controller coding here

    });
});
```

Loading On Demand

If the library is loaded on demand, it will only be available when it is actually needed (for example, when a user presses an export button). You have to request the library every time it is needed (although it will be loaded from the backend only once). This can be done by using `sap.ui.require` with a callback function. This is necessary to ensure that the library will be loaded asynchronously, but it requires more effort to implement the export because all the export steps need to be enwrapped by the callback function.

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";

    return Controller.extend("sample.Spreadsheet", {

        onExport: function() {
            // loadLibrary is only needed when the library is not added as a
dependent in bootstrap or any other dependency mechanism used in your project.
            var oExportLibLoadPromise =
sap.ui.getCore().loadLibrary("sap.ui.export", true);

            oExportLibLoadPromise.then(function() {
                sap.ui.require(["sap/ui/export/Spreadsheet"],
function(Spreadsheet) {

                    // Place your export coding here

                });
            });
        }
    });
});
```

For more information, see the API Reference: `sap.ui.require`.

Supported Data Types

The following data types are supported:

- `String`
- `Number`
- `Boolean`
- `Date`
- `DateTime`
- `Time`
- `Currency`
- `Enumeration`
- `BigNumber`

> **Note**
>
> Currency values and numbers that cannot be represented in the standard format as defined by the *IEEE (Institute of Electrical and Electronics Engineers)* in the spreadsheet file because they are too large automatically use the `BigNumber` data type as a fallback option. The number is then stored as `string` and represented using the international format with a comma as a thousands separator and a period for the decimal point.

For more information, see Data Types for Spreadsheet Export [page 1301].

Additional Settings on Export User Interface

In addition to the regular `Export` dialog, the user can use the `Export As` dialog in the `SmartTable` control, which can be selected from a dropdown list next to the *Export to Spreadsheet* button, to define additional settings for the export.

The user can define the following for the exported file:

- File name
- File format
  The file format has been predefined.
- Whether to show cells with more than one value in separate columns
  This option is used for columns based on text arrangements, currencies, and units of measure. For example, if both the name and the ID of a customer are displayed in one column, the exported file will show the customer data in two separate columns. This might result in duplicate columns if the file already contains columns with the same header name.
- Whether to show any available filter settings
  If this option is selected, the exported file has an additional *Filter* sheet showing the filters that were set on the columns as well as in the `FilterBar` control. The filters shown in the exported file currently contain the technical property that has been extracted from the binding.

The `beforeExport` event also provides the additional export settings defined by the user (`userExportSettings` parameter). This parameter is always available and contains the settings currently valid for the export, so the application developers can decide which settings to use.

> **Note**
>
> The dialog with additional export settings is available for the client export of `SmartTable` only. For the SAP Gateway export, only the regular dialog is available with no additional options for export settings.

Related Information

API Reference: `sap.ui.export.Spreadsheet`

Samples

API Reference: `sap.ui.export.EdmType`

SAP Gateway Export versus Client Export [page 1309]

Spreadsheet Export Configuration

To perform a spreadsheet export for any content other than `SmartTable`, you have to set up the configuration for the columns and data sources, and you can configure some other additional properties.

Overview

If you export data from the `SmartTable` control, the configuration is already available. You can use it without making any modifications. You can also adapt the configuration to make it suitable for your own requirements. The existing export configuration is attached as an event parameter to the `beforeExport` event provided by `SmartTable`. The event is fired once the `SmartTable` control has finished creating the export configuration. To adapt the export configuration, you have to modify the event parameter by registering an event handler and then adjusting the provided configuration.

If you want to use the export for any content other than the `SmartTable` control, you have to define your own export configuration.

The export configuration is a JSON object that contains three major parts that are required to process the data export. While the column configuration and the data source information are mandatory, additional properties that are used for the processing are optional. The export configuration needs to be valid, otherwise the export process will be cancelled.

Details

Column Configuration

The column configuration is an array of JSON objects that is assigned to the column property of the export configuration.

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                // Place your column definition here
            }
        ]
    }
}
```

First you need to identify all the columns you want to export, since there has to be a column definition object for every column that is exported. Regardless of the data source (OData or JSON array), each row represents an instance of an entity with several properties, and each row is mapped to one of these properties. It is also possible to map multiple properties to a single column.

A column definition object is a JSON object that contains at least one `property` property that maps the column to the property of the entity. Its value must be of type `string` or an array of strings and must not be empty; otherwise the column definition is invalid. The string value must contain the name of a property of the entity. If there is no property with the given name, the column in the exported Office Open XML spreadsheet will be empty.

Property Types

Apart from the `property` property, a column definition can have additional properties. The following properties are used:

- Type-independent
- Type-dependent

The following table shows the different kind of properties.

Table 42: Properties

Type-Independent Properties	Type-Dependent Properties
`property (string)`	`scale (number)`
`label (string)`	`delimiter (boolean)`
`type (string)`	`unit (string)`
`width (number)`	`unitProperty (string)`
`textAlign (string)`	`displayUnit (boolean)`
	`trueValue (string)`
	`falseValue (string)`
	`template (string)`
	`inputFormat (string)`
	`valueMap (object|Map)`
	`wrap (boolean)`

In this section, you can find out more about type-independent properties.

The `label` property is optional, and its value must be of type `string`. Its value will be used as column header for the column. If no `label` property has been provided, the value of the `property` property will be used instead.

The optional `type` property defines the data type for this column and needs to match one of the values of the `sap.ui.export.EdmType` enumeration. If the `type` property has not been defined or the enumeration does not contain its value, the default type (`sap.ui.export.EdmType.String`) is used. For more information, see the API Reference: `EdmType`.

The optional `width` property defines the column width based on the number of characters that can be visible. The Office Open XML spreadsheet standard uses a width calculation that is not equivalent to the CSS sizes. Therefore, the calculation is handled by the library. If no `width` property has been provided, or if its value is greater than *1*, the default width is used. The default width is 10 characters. If the column header text has a length that is greater than the actual width, it will override the width with the length of the column header text.

The optional `textAlign` property defines the horizontal text alignment. Its value must be of type `string` and either be `left`, `right`, or `center`. Other CSS alignments like `begin` or `end` are not supported. If no

`textAlign` property has been provided or its value is empty or not supported, the default alignment is used. The default alignment is defined by the type of the column. This is done by the application using the scenario and the generated Office Open XML spreadsheet, for example, Microsoft Excel.

> **i Note**
>
> If you set the alignment for a particular column, the cell content is not always aligned as originally defined for every data type. The application that displays the spreadsheet can ignore the alignment depending on the column's data type or even the content of the cell, for example, in right-to-left scenarios.

The following code shows you an example of a column definition:

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                property: "Firstname",
                width: 15
            },
            {
                property: "Lastname",
                width: 15
            },
            {
                property: "User",
                label: "Username",
                width: 20
            },
            {
                property: "Attempts",
                label: "Login Attempts",
                type: sap.ui.export.EdmType.Number
            },
            {
                property: "LastLogin",
                label: "Last Successful Login",
                type: sap.ui.export.EdmType.DateTime
                width: 20,
                textAlign: "center"
            }
        ]
    }
}
```

Data Source Configuration

Apart from the column configuration, data source configuration is the most important configuration for the export process. Data source configuration is mandatory. It can either be a JSON array containing all data or a JSON object with the following properties:

Table 43: Data Source Configuration Properties

Property	Type	Optional	Description
`type`	`string`		Defines the type of the data service that provides the data. If it is an OData service, the value `OData` must be assigned.
`dataUrl`	`string`		Request URL that is needed to request the data with all the filters and its order. The URL can either be relative or absolute. If the URL is relative, the current origin will be used as a host.
`serviceUrl`	`string`		URL of the data service that serves the entity which is requested by the `dataUrl`". It is usually a substring of `dataUrl`. The URL can either be relative or absolute. If `dataUrl` is relative, `serviceUrl` must not be absolute. **i Note** This property is required if OData batch requests are enabled.

Property	Type	Optional	Description
`count`	`number`	■	Indicator of the line items available through the service. During the export process there is neither a dedicated *$count* request nor is the `inlineCount` request property used. If `count` is provided, it splits the requests so that not all data is requested at once, and the progress indicator can show reliable information. **i Note** If the OData service is an analytical service, the ratio of processing time and number of queried items is not linear. For example, 200 items take up about two seconds, 1,000 items about 2.2 seconds. Therefore, it might be necessary to configure the requested size using the `sizeLimit` property.
`useBatch`	`boolean`	■	If set to `true`, the export library will use OData batch requests. Once batch requests are enabled, you'll have to provide the `serviceUrl` and `headers` properties. If batch requests are not supported by your OData service, you can disable this functionality by setting this property to `false`.

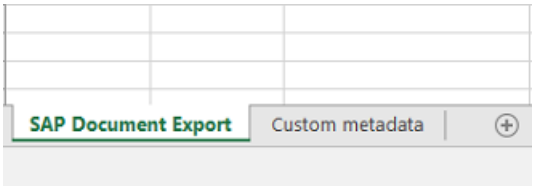
Property	Type	Optional	Description
`headers`	`object`	△	Provides additional request headers within an OData batch request. Every property you add to the `headers` object will be put into the HTTP header section of the respective GET request within the batch request. **i Note** This property is required if OData batch requests are enabled.
`sizeLimit`	`number`	■	Defines the number of records that are requested from the service with a single request. This is important to make fine adjustments.

The following code shows you an example of data source configuration:

```
/* JSON array as dataSource */
var exportConfiguration = {
    dataSource: [
        // Several line items that contain actual data
    ]
}
/* OData service as dataSource */
exportConfiguration = {
    dataSource: {
        type: "OData",
        dataUrl: "/sap/opu/odata/sap/MM_PUR_PODWNPAYT_MNTR_SRV/
C_PurOrdDownPaymentMntr(P_DisplayCurrency=%27EUR%27)/Results?sap-client=715&
$format=json&
$select=PurchaseOrder,PurchaseOrderItem,DisplayCurrency,DownPaymentsRequest,NetAm
ount,PurchasingDocumentStatus&$filter=(Supplier%20eq%20%2710300001%27)&
$orderby=PurchasingDocumentStatus%20asc",
        serviceUrl: "/sap/opu/odata/sap/MM_PUR_PODWNPAYT_MNTR_SRV",
        count: 17491,
        useBatch: true,
        headers: {
            /* Some sample headers DO NOT copy them */
            Accept: "application/json",
            Accept-Language: "en",
            sap-cancel-on-close: "true",
            DataServiceVersion: "2.0",
            x-csrf-token: "XvR_WdN7nCw83ngZnH9lZQ=="
        },
        sizeLimit: 500
    }
}
```

Context Information

Apart from the mandatory configuration, you can add context information to the generated Office Open XML spreadsheet. This part of the configuration is optional. It is provided within a JSON object that is assigned to the `context` property within the workbook object of the export configuration. Built-in properties are not visible on a data sheet and stored as document properties in the generated file while custom metadata is visible on a data sheet in the workbook.



> **i** Note
>
> Apart from the actual data that is exported, sometimes additional information is required in the spreadsheet. For example, this meta information could be the URL of the system from which the data is exported, the system ID, a timestamp of the export date, or the name of the user who exported this data.

The `sap.ui.export.Spreadsheet` library always exports the actual data to the main sheet of the Office Open XML spreadsheet but it can also add an additional sheet for meta information. The consuming applications can then assign their own specific names to both sheets.

The following code shows you an example of context configuration:

```
var exportConfiguration = {
    workbook: {
        context: {
            // Place your context information here
        }
    }
}
```

You can use the following properties:

Table 44: Context Information Properties

Property (optional)	Description	Built-in Context Information	Custom Metadata Context Information
`application (string)`	Adds information about the business application that created a particular file. We recommend to use this property because there are often several similar apps that work on the same data entity but with a different scope or view. This makes it easier to identify where the data comes from if the exported file is shared, for example, among employees and managers.		
`version (string)`	Identifies the application version through which a particular Office Open XML spreadsheet was created. This can be helpful for debugging because you can identify the version that caused the issues and compare it to previous builds. The more detailed your version information is, the easier it will be to identify your application changes within your source code management system.		
`title (string)`	Adds a self-explanatory title to the document generated. This can be useful if the exported entity is not the main entity of the application in question, for example, company codes within a purchase order application.		

Property (optional)	Description	Built-in Context Information	Custom Metadata Context Information
`modifiedBy(string)`	Adds information about the user who created this document. If you do not use this property, the `sap.ui.export.Spreadsheet` library automatically adds SAPUI5 *Document Export* as the author of the document to the generated file and generates the creation timestamp.		
`sheetName(string)`	Changes the name of the sheet that contains the exported data. If this property is not used, the default value `SAP Document Export` is used.		
`metaSheetName(string)`	Changes the name of the sheet that contains the metadata. In contrast to `sheetName`, it only affects the name of the additional sheet that contains the metadata. If no metadata has been provided, the additional sheet will not be created, and this property will not take effect.		

Property (optional)	Description	Built-in Context Information	Custom Metadata Context Information
`metaInfo` (`array`)	An array of JSON objects that follow an exact specification, the so-called meta information groups. Each meta information group has an explicit `name` property which accepts a string that is not empty. Apart from the group name, it contains an array that is assigned to the `items` property of the group. This array can contain several JSON objects that provide the `key` and `value` properties (type `string`).		

> **i Note**
>
> Properties other than the ones listed are ignored during configuration.

The following code shows you an example of context configuration with some of the properties mentioned:

```
var exportConfiguration = {
    workbook: {
        context: {
            application: "Supplier Invoices List",
            version: "6.1.0-SNAPSHOT",
            title: "Supplier Invoices",
            modifiedBy: "Doe, John",
            sheetName: "Invoices"
        }
    }
}
```

The following code shows you an example of context configuration with the `metaInfo` property:

```
var exportConfiguration = {
    workbook: {
        context: {
            metaInfo: [
                {
                    name: "Application settings",
                    items: [
                        {
                            key: "Name",
                            value: "Purchase Orders"
                        },
                        {
                            key: "Version",
                            value: "1.23.8742-p"
                        }
                    ]
                }
            ]
```

```
            }
        }
    }
```

> **Note**
>
> The reasoning behind this design is that different layers (SAPUI5 framework, SAP Fiori elements, smart controls, and applications) can add metadata without depending on each other. As long as there is at least one valid meta information group within the `metaInfo` array, the additional sheet will be shown in the generated file.

Configuration of Additional Properties

The `sap.ui.export.Spreadsheet` library offers some other additional properties that you can configure. This part of the configuration is also optional.

The following properties are available for configuration:

- `count` (type `number`)
  The value of this property must be positive. It restricts the amount of exported data, which avoids browser crashes during the transfer of very large amounts of data.
- `worker` (type `boolean`)
  The export process runs in a dedicated web worker by default. The `worker` property allows you to disable this functionality. A web worker runs in a separate thread and does not affect the performance of the browser window's main thread.
  Although it has some advantages to use a web worker, it can also cause some problems. Especially if SAPUI5 is loaded from a Content Delivery Network (CDN) and is not served by the original host, it depends on the Content Security Policy (CSP) of the server if the export can be processed in a web worker. To resolve problems related to this scenario, you should either add a `worker-src` directive to your CSP or disable the web worker functionality. If the web worker functionality is disabled, the export process will run in the main thread of the browser window. Currently, the `worker-src` directive is not fully supported by all browsers. If you can't set the `worker-src` directive, you can also adjust the fallback directive `script-src`.

  > **Note**
  >
  > The `script-src` directive will also affect all other scripts that are loaded on your page.

- `fileName` (type `string`)
  Defines a particular name for the generated export file. The file extension is added to the given file name and is always xlsx. To take effect, the property must not be empty.
- `showProgress` (type `boolean`)
  The export process shows a progress dialog by default.To prevent this from happening, you can set the `showProgress` property to `false`.

Hierarchical Data

The `sap.ui.export.Spreadsheet` library can also handle hierarchical structures.

To get exported, each object of the relevant entity must have the following:

- A distinct property containing its absolute numerical hierarchy level
- An order in which parent nodes are followed by their direct child nodes

To enable a hierarchical representation, you have to simply assign the name of the property containing the hierarchy level to the `hierarchyLevel` property of the workbook object in the export configuration.

The following code shows you an example of hierarchy data:

```
var exportConfiguration = {
    workbook: {
        hierarchyLevel: "level"
    }
}
```

> **i** Note
>
> Relational hierarchy data is not supported.

Starting the Export Process

After you have created a valid configuration, you will have to create a new `sap.ui.export.Spreadsheet` instance and initialize it with the previously created configuration. After the instance has been initialized, you can start the export process by calling the `build` method. Everything else will be handled by the export library. The result will be an Office Open XML spreadsheet which is automatically downloaded. The export library does not offer you any events to which you can register. If you need to perform additional steps after the export has been completed, you can use the `Promise` that is returned by the `build` method.

The following code sample shows the start of an export:

```
sap.ui.define([
    "sap/ui/core/mvc/Controller",
    "sap/ui/export/Spreadsheet"
], function(Controller, Spreadsheet) {
    "use strict";

    return Controller.extend("sample.Spreadsheet", {

        onExport: function() {
            var oExportConfiguration, oExportPromise, oSpreadsheet;

            /* Creates the configuration and initializes the spreadsheet export
*/
            oExportConfiguration = this.createExportConfiguration();
            oSpreadsheet= new Spreadsheet(oExportConfiguration);

            /* Starts the export and returns a Promise */
            oExportPromise = oSpreadsheet.build();

            oExportPromise.then(function() {
                // Here you can perform additional steps after the export has
finished
            });
        },

        createExportConfiguration: function() {
            var oConfiguration;

            // Create a valid export configuration

            return oConfiguration;
        }
    });
});
```

Data Types for Spreadsheet Export

Provides details about the data types supported by the spreadsheet export.

All data types supported by the spreadsheet export are listed in the `sap.ui.export.EdmType` enumeration. During the export, the values are converted to a value that is in compliance with the Office Open XML requirements.

`String`

The type `string` handles textual values within cells. Strings are usually not formatted. The internal type text is the default type that gets applied if no other type is configured for a column.

It is possible to aggregate several property values into one column, for example, `firstname` and `lastname` are combined to `fullname`. This can be achieved by providing an array of property names within the `property` property of the column definition. In addition to that, the `template` property must be made available. This additional property has to be a non-empty string and can contain placeholders. A placeholder is a numerical index enclosed by curly brackets. The index must be greater than or equal zero and less than the length of the array that is assigned to the `property` property.

Table 45: Details for `string`

Additional Property	Type	Sample	Optional	Description
`template`	`string`	`"{0} (Company code {1})"`	Yes	A textual template that can be filled with multiple values from various business objects. Each placeholder is a number within curly brackets that represents an index of a property array.
`wrap`	`boolean`		Yes	A Boolean value that indicates if the text column supports wrapping of the cell content. Apart from automatic text wrapping depending on the cell width, it automatically converts all \n to \r\n line breaks. These manual line breaks are then visible in the generated xlsx file.

Here is an example for a `string` column:

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                property: ["Firstname", "Lastname"],
                label: "Full name",
                width: 25,
                template: "{1}, {0}"
            }
        ]
    }
}


// This will result in "Doe, John" if the line item is {Firstname: "John",
Lastname: "Doe"}
```

`Boolean`

The type `boolean` handles all variations of Boolean values. It allows for displaying these Boolean values in a pre-defined format. There are additional properties that are handled by this type to format their respective values. Since a `boolean` type can be either `true` or `false`, the additional properties must be maintained for both cases for the type to take effect.

Table 46: Details for `boolean`

Additional Property	Type	Description
`trueValue`	`string`	Defines the textual representation of a Boolean type that has the value `true`.
`falseValue`	`string`	Defines the textual representation of a Boolean type that has the value `false`.

Here is an example for a `boolean` column:

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                property: "Onstock",
                label: "Availability",
                type: sap.ui.export.EdmType.Boolean,
                trueValue: "On stock",
                falseValue: "Out of stock"
            }
        ]
    }
}
```

Number

The type `number` represents a simple numeric value without any specific formatting. The value is displayed the way it is. For further adjustment use the additional properties `scale`, `delimiter`, `unit`, and `unitProperty`.

Table 47: Details for `number`

Additional Property	Output Sample	Type	Optional	Description
`scale`	1,234 1,234,567	`boolean`	Yes	Specifies if the numeric value is shown in groups of thousands. If set to `true`, the thousands delimiter is shown. Default value is `false`.
`delimiter`	1234 1234.5 1234.56 1234.567	`number`	Yes	Sets a fixed amount of decimals. The scale is applied to the whole column and displays exactly the number of decimals that is configured. If the actual value has fewer or more decimal places, it is filled with additional zeros or gets cut off to match the configured amount of decimals. This property accepts a positive integer value. Negative values are treated like zero.

Additional Property	Output Sample	Type	Optional	Description
`unit`	623 kg 89 % 120 km/h	`string`	Yes	Specifies the unit of measurement (UoM). The UoM is shown next to the numeric value. **i Note** The UoM is treated as a string and therefore has no influence on the value itself. For %, *mio*, or similar UoMs this can make a difference if the values are used in forms because multiplying by 150 % would mean x * 150 instead of x * 1.5.
`unitProperty`	623 kg 89 % 120 km/h	`string`	Yes	References a business object property that contains the UoM for this particular numeric type.

Here is an example for a `number` column:

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                property: "Weight",
                label: "Net weight (kg)",
                type: sap.ui.export.EdmType.Number,
                unit: "kg",
                scale: 3
            },
            {
                property: "Weight",
                label: "Net weight (g)",
                type: sap.ui.export.EdmType.Number,
                unit: "g",
                scale: 0,
                delimiter: true
            }
        ]
    }
}
```

`Date`, `Time`, and `DateTime`

The types `date`, `dateTime`, and `time` handle the date and time information. The application can pass additional parameters to adjust the visible representation of these types.

Table 48: Details for `date`, `time`, and `DateTime`

Type	Output Sample	Description
`date`	03/24/2017 24.03.2017	Represents a date without time-related information. Due to the use of built-in formats, `date` is displayed based on the user's locale in the operating system. This can lead to different representations for different users.
`dateTime`	08/31/2016 23:01 31.08.2016 23:01	Represents values that contain date- and time-related information. Due to the use of built-in formats, `dateTime` is displayed based on the user's locale in the operating system. This can lead to different representations for different users. The locale has no effect on any time zone formatting. All values in columns of type `dateTime` are related to UTC because it is not possible to pass time zone offset information into the Office Open XML standard representation of time stamps. For columns of type `dateTime`, a UTC suffix is automatically added to the column header.
`time`	13:21:14	Represents values that contain time-related information only. Time information can use the following units: hours, minutes, seconds, and milliseconds. Contrary to `date` and `dateTime`, the built-in formats for `time` are the same for every locale in the operating system.

Additional Property	Description
`calendar`	Allows users to choose a calendar other than the Gregorian calendar. The following values are possible: • islamic • japanese • gregorian (default) Choosing a calendar other than the Gregorian calendar overrides all type settings, such as the type or format template, for various reasons. Islamic and Japanese representation only takes effect if the displayed `date/dateTime` shows some date-related information. Therefore, it does not make sense to use it for the type `time`. Due to the fact that these special representations rely on specific formats, it is not possible to merge them with built-in formats for `date`, `dateTime`, `time`, or even a custom format.
`format`	Defines a specific format that gets applied to `date/dateTime/time`. The format overrides the default formatting of the respective type which means that you can assign a format that shows only time-related information even to a column of type `date`, which usually shows no time-related information. The format template needs to match the following regular expression to be valid: `/^[dhmsy\s-,.:/]+(AM\/PM)?$/` The list below shows some sample formats and their output: **Format Template** / **Output Sample** yyyy-mm-dd h:mm — 2007-12-24 18:21 h:mm:ss AM/PM — 9:32:24 AM d-mmm-yy — 12-Apr-17 dddd, d.mmmm yyyy — Wednesday, 22. April 2017 **Note** Office Open Spreadsheet dates can't handle time zone offset information.

Here is an example for a `date/dateTime/time` column:

```
var exportConfiguration = {
    workbook: {
```

```
        columns: [
            {
                property: "Duedate",
                label: "Due date (islamic)",
                type: sap.ui.export.EdmType.Date,
                calendar: "islamic"
            },
            {
                property: "Createdat",
                label: "Created at",
                type: sap.ui.export.EdmType.DateTime,
                format: "dddd, d.mmmm yyyy"
            },
            {
                property: "Dailymeeting",
                label: "Daily meeting",
                type: sap.ui.export.EdmType.Time
            }
        ]
    }
}
```

Currency

The type `currency` handles currencies as an aggregation of a value and a particular UoM. This type might apply various styles on cell level because the scale of each currency cell depends on the corresponding UoM which in turn might vary for various cells in a currency column. The `currency` type inherits from the `number` type but provides additional properties, including the `unitProperty` property as a mandatory property.

Table 49: Details for `currency`

Additional Property	Type	Mandatory	Description
`unitProperty`	`string`	Yes	References the business object property that contains the UoM for this particular currency. This property is required even if the UoM is not displayed.
`displayUnit`	`boolean`	No	Defines if the UoM is shown in the column. If set to `true`, the UoM is displayed after the actual value. The default value is `true`.
`scale`	`integer`	No	Property that is equivalent to the `scale` property of the internal numeric type. It applies a fixed number of decimals to all cells within the currency column regardless of the corresponding UoM.

Here is an example for a `currency` column:

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                property: "Amount",
                label: "Price",
                unitProperty: "Currency"
            }
        ]
    }
}
```

Enumeration

The type `enumeration` is used for mapping values to a particular key. This is useful if your SAPUI5 application is using formatters instead of raw data to display meaningful content because formatters are not supported directly.

Table 50: Details for `enumeration`

Additional Property	Type	Mandatory	Description
`valueMap`	`object|map`	Yes	Contains `object` as an associative array or `map`, which holds all the key value pairs that are used for mapping the raw data to an explicit value. The raw data is used as a key to look up the actual value.

Here is an example for an `enumeration` column:

```
var exportConfiguration = {
    workbook: {
        columns: [
            {
                property: "Shipping",
                valueMap: {
                    a: "Standard Shipping",
                    b: "Premium Shipping",
                    c: "Express Shipping"
                }
            }
        ]
    }
}
```

BigNumber

The type `BigNumber` is used to represent numbers that contain more than 15 digits. This data type is required because of the internal number representation of Microsoft Excel as defined by the IEEE (Institute of Electrical and Electronics Engineers). This means that all numbers that contain more than 15 digits are filled with zeros at the end. This affects precision of the values although the difference is really small compared to the total amount. The `BigNumber` type inherits from the `Currency` type and uses the same properties as `currency` and its superordinate class `Number`. This type creates a textual output which is why it is not possible to do any calculation with these values.

Related Information

API Reference: `sap.ui.export.EdmType`

SAP Gateway Export versus Client Export

To decide which type of export to use, have a look at the following criteria.

The `SmartTable` control offers the following types of exports:

- Client-side export
  This type of export is the default type.
- SAP Gateway export of the SAP Gateway Foundation
  For more information about this export, search for Excel Support in the documentation of your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

Comparison of Export Types

To find out which export suits your requirements best, check out the following table:

Table 51: SAP Gateway versus Client Export

Feature	Description	SAP Gateway	Client
Cell limitation	The number of cells that can be exported without warnings or errors.	Shows a warning if there are more than 100,000 cells or up to 500,000 cells, depending on the configuration of the session time and ABAP memory.	Shows a warning depending on device used, for 2,000,000 cells on a desktop and 100,000 cells on a mobile device. The total number of cells is limited only by the physical memory of the client and memory restrictions of the browser. Google Chrome and Mozilla Firefox have memory allocation limitations while Microsoft Edge can consume as much memory as the system provides. More memory does not automatically mean that you can export more cells. Some computers should even be able to export a higher number of cells, which also depends on how much content the cells contain.
File compression	Office Open XML spreadsheets are ZIP containers that contain a particular file structure. To reduce the file size, these ZIP containers can be compressed.	✅	✅
Header row	The exported spreadsheet contains a header row with the corresponding column labels.	✅	✅ Also, the exported spreadsheet uses the built-in auto filter for all configured columns. This will allow the user to apply filters directly in the exported file without having to change the file.

Feature	Description	SAP Gateway	Client
Localization	The location of the user exporting a file is taken into account, and the content of the exported spreadsheet will contain translated column headers and the required date, time, and Boolean representation.	 Column headers in the exported file can be shown in the user's language as defined in the back-end system and might differ from what the user sees on the UI.	 The column headers in the exported file are the same as the ones shown on the UI.
Meta information	Metadata with additional information is shown in the exported spreadsheet.	 Provides an appendix to add information to the data sheet.	 Provides an optional sheet in the workbook to attach additional information to the exported spreadsheet. This meta information is grouped and thus allows you to add information in different layers without any conflicts. A typical use case would be that, for example, `SmartTable` adds some basic information, and an application developer could enhance this by adding something on top of it. In addition to that, the client export allows you to add additional information that is not part of the sheets inside the workbook. This data is built-in information and can be processed by applications like Microsoft Excel.
Hierarchies	The exported file can contain hierarchies that are visualized when opening the file with Microsoft Excel.		

Feature	Description	SAP Gateway	Client
Data types	Office Open XML spreadsheets support various types of data that have a different visual and functional behavior. The type representation differs from the raw data and needs to be transformed accordingly.	 The types `text`, `number`, `date`, `time`, and `boolean` are supported.	The types `text`, `number`, `currency`, `date`, `time`, and `Boolean` are supported. In addition to the mere support for these data types, it is possible to pass additional configuration to create formatted text aggregations, a particular date and time output format, textual Boolean representation (for example, *in stock/out of stock* instead of `true/false`), and units of measurement.
Read Access Logging (RAL) support	If configured, Read Access Logging tracks who has access to which data at which point in time.This information might be required for audits.	 RAL is not supported because the SAP Gateway export is carried out by a server process that does not run in a specific user context.	 RAL is fully supported because the client export uses the existing OData service to request data.

 = Supported,  = Partially supported,  = Not supported

Related Information

API Reference: `sap.ui.comp.smarttable.ExportType`
API Reference: `SmartTable`

Spreadsheet Export Limitations

There are some known issues and limitations when using the spreadsheet export.

When you use the spreadsheet export, there are the following known issues:

Known Issues
In Safari on iOS , the file name and file extension get lost due to a download attribute that is not supported. This issue is caused by the iOS Safari browser in every version before iOS 13.

Known Issues
While timestamps in SAPUI5 tables are usually shown in the time of the user's time zone, exported timestamps are always shown in UTC. This has no effect on the types `date` and `time` because they show a date without any time information and vice versa (independent of a specific time zone). For columns of type `dateTime`, a suffix is added to the column header to indicate that the time is shown in UTC. If there are columns of type `date` and `time` that are related to each other, you might want to combine these into a column of type `dateTime` or add a UTC indication manually.
Column width is converted into an Office Open XML equivalent and may differ slightly from the original table. The column width is at least the same as the width of the column label.
If you use the client export within the `SmartTable` control, you have to provide additional `p13n` custom data for the custom columns. Custom columns are created by the application or SAP Fiori elements and not by the `SmartTable` control, so the `SmartTable` control does not recognize this data type.

When you use the spreadsheet export, the following limitations apply:

Limitations
The maximum number of rows that can be exported is restricted to 1,048,576.
The number of cells that can be exported has a limit of 2,000,000 on desktop and 100,000 on mobile devices.
The hierarchical depth that can be visually represented is limited to eight levels.

The following features are not supported:

Features Not Supported
Charts
Icons
Semantic cell highlighting
Custom formatters
Dates before January 1st, 1900
Aggregated rows (group headers or sum rows)

Troubleshooting

This section describes the various tools that are available for troubleshooting apps developed with SAPUI5

The first place to check for errors is the developer tools that are provided by the various browsers. They can help you examine the details of the current web page and provide you with debugging tools. For more information on how to debug SAPUI5 apps, see Debugging [page 1315] and Logging and Tracing [page 1319].

SAPUI5 also provides you with support tools that help you troubleshoot and solve issues.

Table 52: Support Tools Available in SAPUI5

Tool	Use Case Examples	How to Open
Technical Information Dialog [page 1322]	Use the *Technical Information* dialog to enable debug sources and to check which SAPUI5 version is currently running.	`CTRL` + `SHIFT` + `ALT` + `P` Gesture on mobile device: 1. Press two fingers on a noninteractive screen area (for example, a blank area) for at least 3 seconds. 2. Tap with a third finger while holding the other two fingers on the screen.
Support Assistant [page 1339]	Use the Support Assistant to check whether the application is built according to the best practices for building SAPUI5 apps.	From the *Technical Information* dialog or with the URL parameter `sap-ui-support=true`
Diagnostics [page 1326]	Use the *Diagnostics* window to enable debug sources, display the control tree, and to view and change control properties and bindings.	`CTRL` + `SHIFT` + `ALT` + `S`
UI5 Inspector [page 1374]	Use the UI5 inspector to display the control tree, and to view and change control properties and bindings on-the-fly.	Available as add-on for Google Chrome browser only

Table 53: Performance Measurement Tools

Tool	Use Case Examples
Performance Measurement Using sap/ui/performance/ Measurement Module [page 1377]	Measures the performance of your JavaScript code.
Interaction Tracking for Performance Measurement [page 1382]	Identifies performance issues in your application by tracking the interaction that is performed on the UI

For help with specific problems see our First-Aid Kit [page 1386].

Get Help

If you're stuck and need help with a development task, you can also post a question in the SAPUI5-related forums, for example in the SAP Community or on Stack Overflow.

Related Information

Troubleshooting Tutorial [page 194]

Debugging

When developing apps, searching for bugs is an inevitable part of the process. To analyze an issue, you can use the developer tools of your browser and built-in SAPUI5 tools. In this section, we give an overview of the SAPUI5 tools you can use when debugging. To learn more about the developer tools of your browser, check the documentation of the browser.

> **i** Note
>
> For information on browser debugging for ABAP developers, see Browser Debugging for ABAP Developers [page 1531].

Loading Debug Sources

For performance reasons, the SAPUI5 files are loaded in a minified version, this means that all possible variable names are shortened and comments are removed. This makes debugging harder because the code is less readable.

For debugging, you first have to load the *Debug Sources*. You have the following options:

- URL parameter `sap-ui-debug=true`
- Select the *Use Debug Sources* in the *Technical Information Dialog*
  For more information, see Technical Information Dialog [page 1322].

If you only want to load the debug sources for **specific packages**, you have the following options:

- Add the module names to the `sap-ui-debug` URL parameter, separated by a comma. For example, `sap-ui-debug=sap/ui/core/Core.js,sap/m/InputType.js` loads the debug sources for the `sap.ui.core.Core` and `sap.m.InputType` libraries.
- Choose the *Select specific modules* link in the *Technical Information Dialog*.
  For more information, see Technical Information Dialog [page 1322].

After reloading the page, in the *Network* tab of the browser's developer tools you can see that the controls and framework assets are now loaded individually and have a `-dbg` suffix. These are the source code files that include comments, the uncompressed code of the app, and the SAPUI5 artifacts.

Choose `Ctrl` + `O` (Windows) or `Command` + `O` (macOS) and type the name of an SAPUI5 artifact to view its source code in debug mode.

> **i Note**
>
> Turning on debug sources also increases the log level. For more information, see Logging and Tracing [page 1319].
>
> To improve performance, you must deactivate the debug sources once you're done with debugging.

Switching the SAPUI5 Version

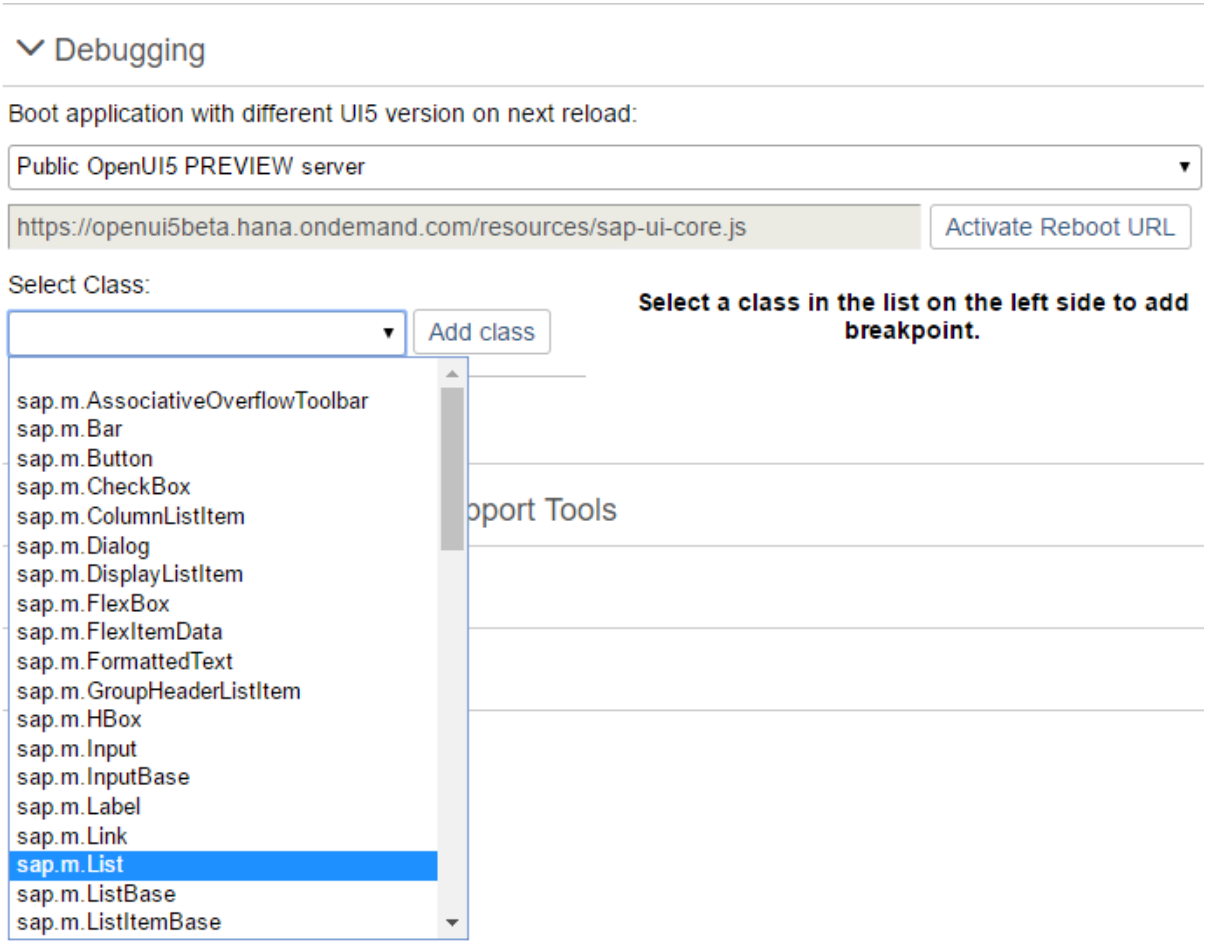
Open the *Diagnostics* window with the shortcut `CTRL` + `SHIFT` + `ALT` + `S`.

At the top of the *Debugging* view, you can configure a custom URL from which the application should load SAPUI5 the next time that the app opens.

Either select a known SAPUI5 installation from the dropdown box, or enter a different URL that points to the `sap-ui-core.js` file within a complete SAPUI5 runtime.

Once you have entered the URL, press *Activate Reboot URL*. When you then reload the application page, the application loads SAPUI5 from the alternative URL. This only happens for the next single reboot; after that, SAPUI5 is loaded again from the standard URL referenced within the app.

This feature can be used to test an application against a newer or older version of SAPUI5 as part of compatibility testing, or for verifying a bug fix or regression.



Setting Breakpoints

Breakpoints are helpful when you debug the event handling of an SAPUI5 object. You can either set breakpoints in the developer tools of your browser, or use the *Diagnostics* window.

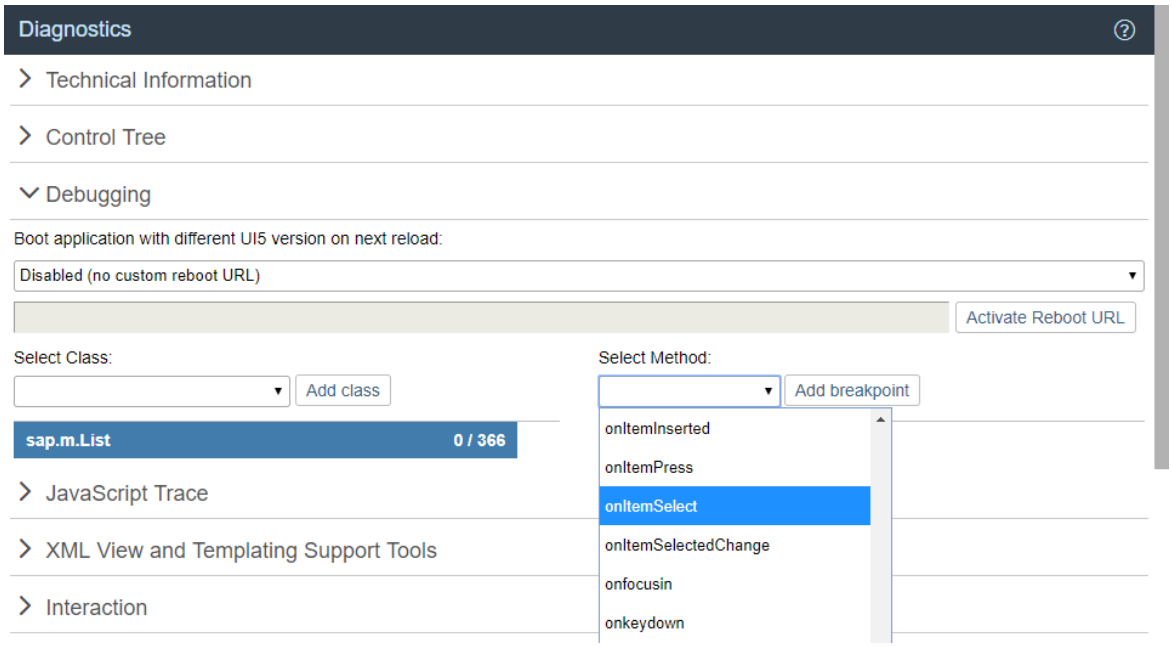
For more information, see Diagnostics [page 1326].

Breakpoints on the Class Level

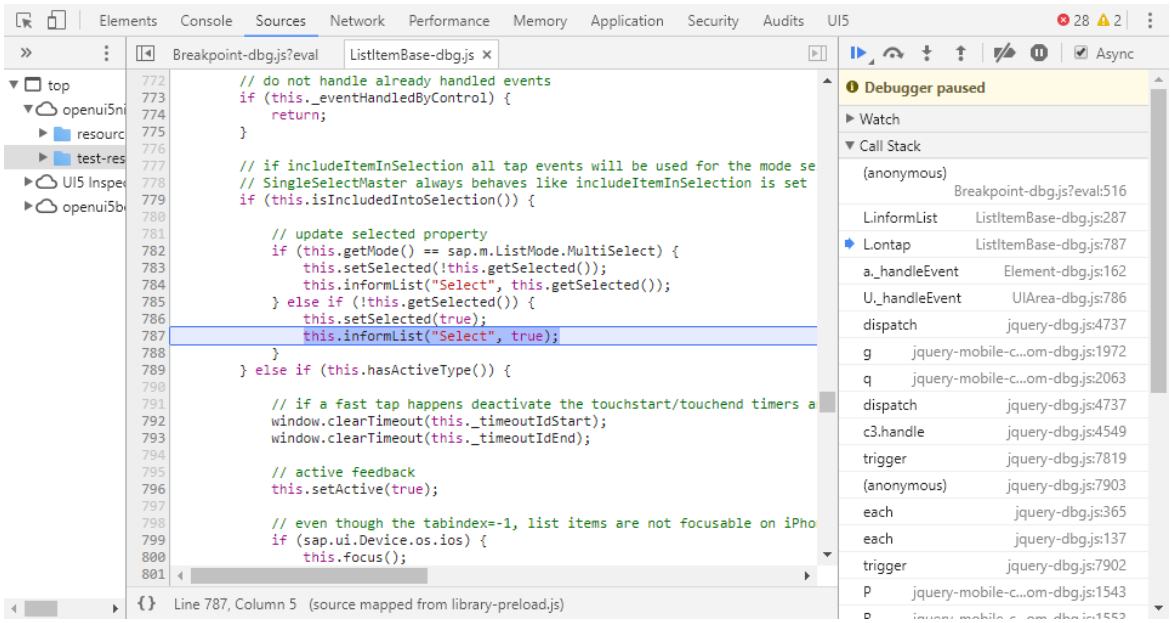
In the *Debugging* section of the *Diagnostics* window, you can set breakpoints on the class level.

1. Open the *Debugging* view of the *Diagnostics* window.
2. Select a class from the dropdown list or enter the name of the class and choose *Add Class*.
   The selected class is now visible below the dropdown list.
   The number next to the method name shows the number of methods that belong to the class and the number of methods for which a breakpoint is set.
3. Select the class. On the right side of the view, you can now select methods of the selected class from a dropdown list.

4. From the dropdown list, select the method for which you want to set the breakpoint and choose *Add breakpoint*.
   The selected methods are listed below the dropdown list.
5. Open the developer tools of your browser. Whenever the selected methods are called for any instance of the selected control, the code execution is paused in the debugger.



In the call stack you find the method for which you set a breakpoint.

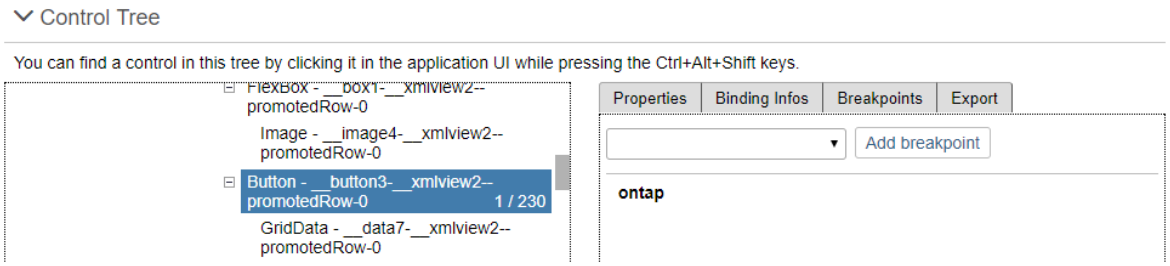


6. To remove a breakpoint, select the red x.

Breakpoints on the Object Level

In the *Control Tree* of the *Diagnostics* window, you can set breakpoints on the object level.

1. Open the *Control Tree* view of the *Diagnostics* window.
2. Select a control in the tree.
   You can also press and hold `Ctrl` + `Shift` + `Alt` and select a control in your app to select it in the tree.
3. Select the *Breakpoints* tab on the right.
4. From the dropdown list, select the method for which you want to set the breakpoint and choose *Add breakpoint*.
   The selected methods are listed below the dropdown list.
5. Open the developer tools of your browser. Whenever the selected methods are called for any instance on the control, the code execution is paused in the debugger.



6. To remove a breakpoint, select the red x.

Logging and Tracing

Use the built-in SAPUI5 logging mechanisms to debug and analyze applications or framework errors.

You can view the log in the console in the developer tools of your browser. A log entry contains a timestamp, a log level, a message with optional details, and component information.

Log messages can be written by the framework or the application code. Which log messages are contained in the log is defined according to their severity and the log level that is currently set.

Specific log messages only appear in the console when the severity of the issue is equal to or higher than the currently set log level. For example, when opening an SAPUI5 app without any additional configuration, the default log level (1 = `ERROR`) is applied. Only messages with severity `FATAL` and `ERROR` will be written to the console; other messages are not logged for performance reasons.

> → Tip
>
> In productive applications, you should keep the system defaults for log levels for performance reasons.

Table 54: Severities and Log Levels

Severity	Log Level	Description	Example
`NONE`	-1	No messages are written to the console	

Severity	Log Level	Description	Example
`FATAL`	0	Unrecoverable situations	A parse error occurred while processing a JavaScript file or an XML view
`ERROR`	1 (Default)	Erroneous but recoverable situations	Loading a requested module failed
`WARNING`	2	Unwanted but foreseen situations	Wrong property format was passed to a control
`INFO`	3	Purely informative	A configuration parameter was set
`DEBUG`	4 (Debug Mode)	Information necessary for debugging	A framework event was fired
`TRACE`	5	Tracing the program flow	A certain position in the code was reached
`ALL`	6	Messages of all severity categories are written to the console	

Tracing Errors

During development, or when you're troubleshooting application errors, you can increase the log level for debugging . When you enable debug mode, for example in the *Technical Information* dialog, the log level is automatically increased to 4 (`DEBUG`) and you will see messages in the console that might be helpful for finding bugs.

You can also set the log level manually, either by calling `in the application code or by setting the framework parameter <level>) setting the framework parameter <level>)` sap-ui-loglevel= (see<level sap-ui-loglevel=<level>Configuration Options and URL Parameters [page 703]). Depending on the log level, you might get a large amount of log messages in the console. The error might not be clearly visible from a single log message and might need further analysis.

Log entries are written to the console in chronological order. When you identify a message in the console that is connected to the error you are tracing, there might be other related log messages written shortly before or after the current message.

A dash separates the log message and the component (or feature) that logged it.

To find out which messages are related to a specific error, you can filter the messages in the console. Narrow down the log messages by selecting a severity or doing a full-text search, for example for a specific keyword or an SAPUI5 component.

Adding Your Own Log Messages

You can define your own log messages in your code to help tracing errors and understanding the application flow. You can use the `Log` API to create and manage log entries (see the API Reference: `sap/base/log`).

Generic Log Entries

To log a message, simply call `Log.*` with the specific method that corresponds to the log levels described above. The following code line issues a log statement with severity *Error*. The browser highlights the statement in red to indicate an application error:

```
// "Log" required from module "sap/base/Log"
Log.error("This should never have happened!");
```

The next code line issues a log statement with severity *Information*. The browser does not display this statement if your log console is set to filter for errors only:

```
// "Log" required from module "sap/base/Log"
Log.info("Something has happened");
```

Component Logging

The log statements above do not contain the component. A log component can be used to semantically group log entries that belong to the same software component (or feature).

It can be specified as an additional argument to the log function or as a default for all log entries written by a certain component. With `Log.getLogger("<component>")`, you can retrieve a logger that automatically adds the given component as component parameter to each log entry.

```
// "Log" required from module "sap/base/Log"
this._oLogger = Log.getLogger("sap.ui.demo.MyComponent");
this._oLogger.info("Something has happened");
```

> **i Note**
>
> The log entries can also be accessed programmatically. You can also register a listener to the log that will be notified whenever a new entry is added to the log.

Assertions

For logical checks in your application flow, you can use assertions. With `sap/base/assert` an error message is logged when a given condition is not met.

```
// "assert" required from module "sap/base/assert"
assert(aValues.length === 10, "There are 10 values stored in the array")
```

> **i Note**
>
> Assertions might be removed when the JavaScript code is optimized during a build. Therefore, callers should not rely on any side effects of this method.

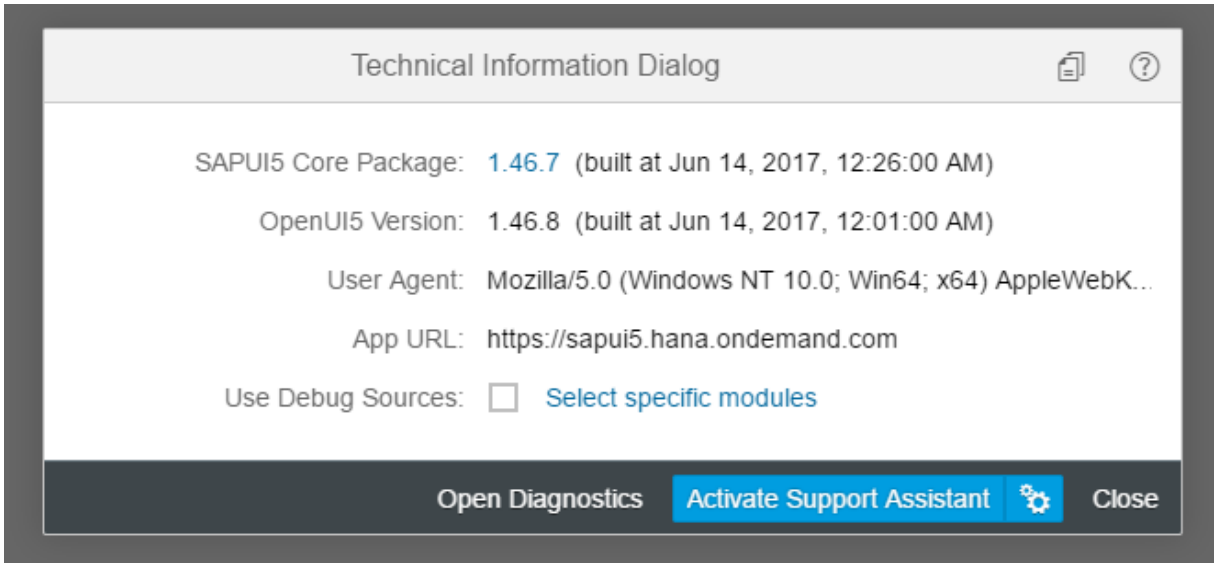
Related Information

Debugging [page 1315]
Troubleshooting Tutorial Step 1: Browser Developer Tools [page 196]

Technical Information Dialog

The *Technical Information* dialog shows details of the SAPUI5 version currently being used in an app built with SAPUI5. You can use the *Technical Information* dialog to enable debug resources and open additional support tools to debug your app.

To open the technical information dialog from within a supported desktop browser, use the following shortcut: `CTRL` + `SHIFT` + `ALT` + `P`.



The dialog contains the following information:

- The **version number** of the currently loaded SAPUI5 distribution and the underlying OpenUI5 version that represent the core libraries of the framework and their build timestamps
  For more information, see Versioning of SAPUI5 [page 29] and SAPUI5 vs. OpenUI5 [page 37]
- The **user agent** that is used for detecting the device's capabilities and device adaption
- The root **URL** of the currently loaded app

You can download the technical information, so that you can attach it to a ticket for example, by clicking *Copy*.

Loading Debug Sources

For performance reasons, the SAPUI5 files are loaded in a minified version, this means that all possible variable names are shortened and comments are removed. This makes debugging harder because the code is less readable.

For debugging, you first have to load the *Debug Sources*. You have the following options:

- URL parameter `sap-ui-debug=true`
- Select the *Use Debug Sources* in the *Technical Information Dialog*
  For more information, see Technical Information Dialog [page 1322].

If you only want to load the debug sources for **specific packages**, you have the following options:

- Add the module names to the `sap-ui-debug` URL parameter, separated by a comma. For example, `sap-ui-debug=sap/ui/core/Core.js,sap/m/InputType.js` loads the debug sources for the `sap.ui.core.Core` and `sap.m.InputType` libraries.
- Choose the *Select specific modules* link in the *Technical Information Dialog*.
  For more information, see Technical Information Dialog [page 1322].

After reloading the page, in the *Network* tab of the browser's developer tools you can see that the controls and framework assets are now loaded individually and have a `-dbg` suffix. These are the source code files that include comments, the uncompressed code of the app, and the SAPUI5 artifacts.

Choose `Ctrl` + `O` (Windows) or `Command` + `O` (macOS) and type the name of an SAPUI5 artifact to view its source code in debug mode.

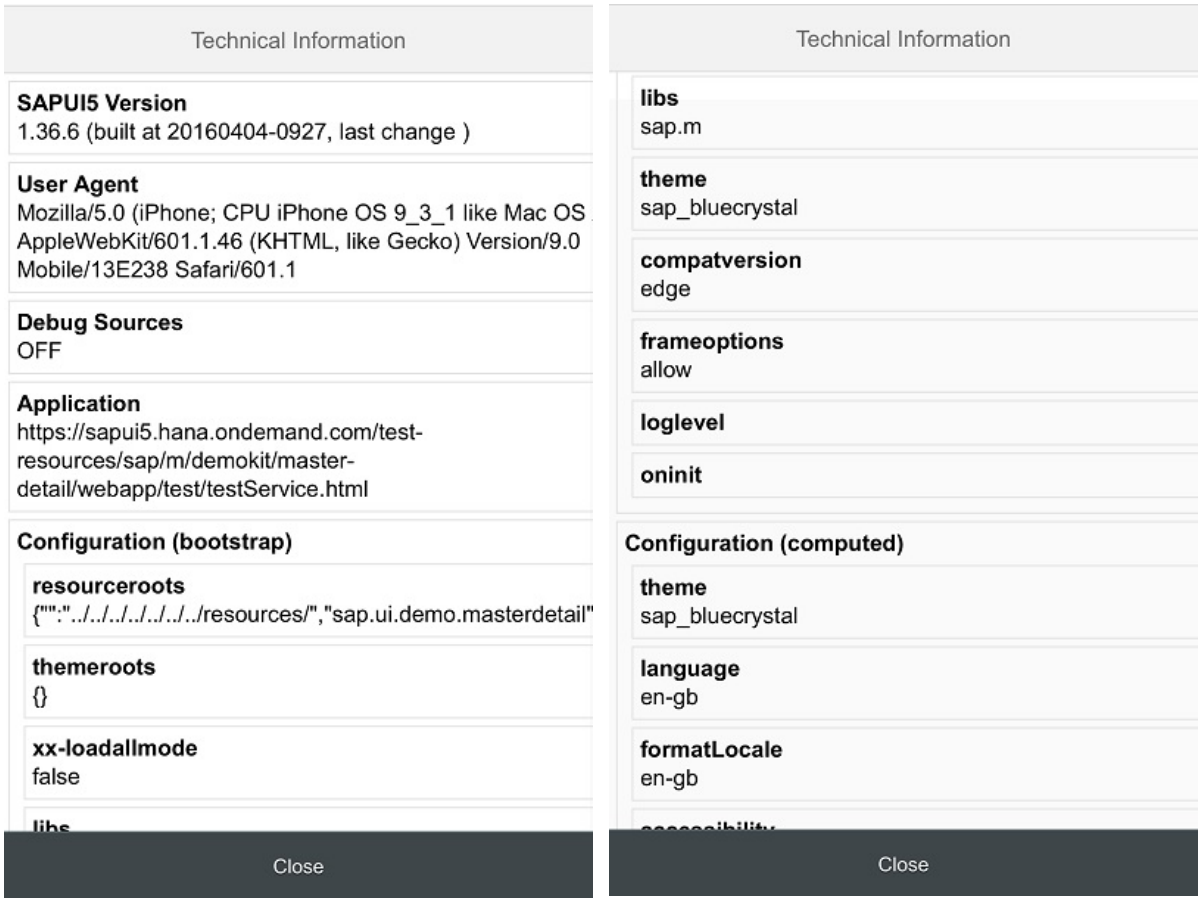
> **i Note**
>
> Turning on debug sources also increases the log level. For more information, see Logging and Tracing [page 1319].
>
> To improve performance, you must deactivate the debug sources once you're done with debugging.

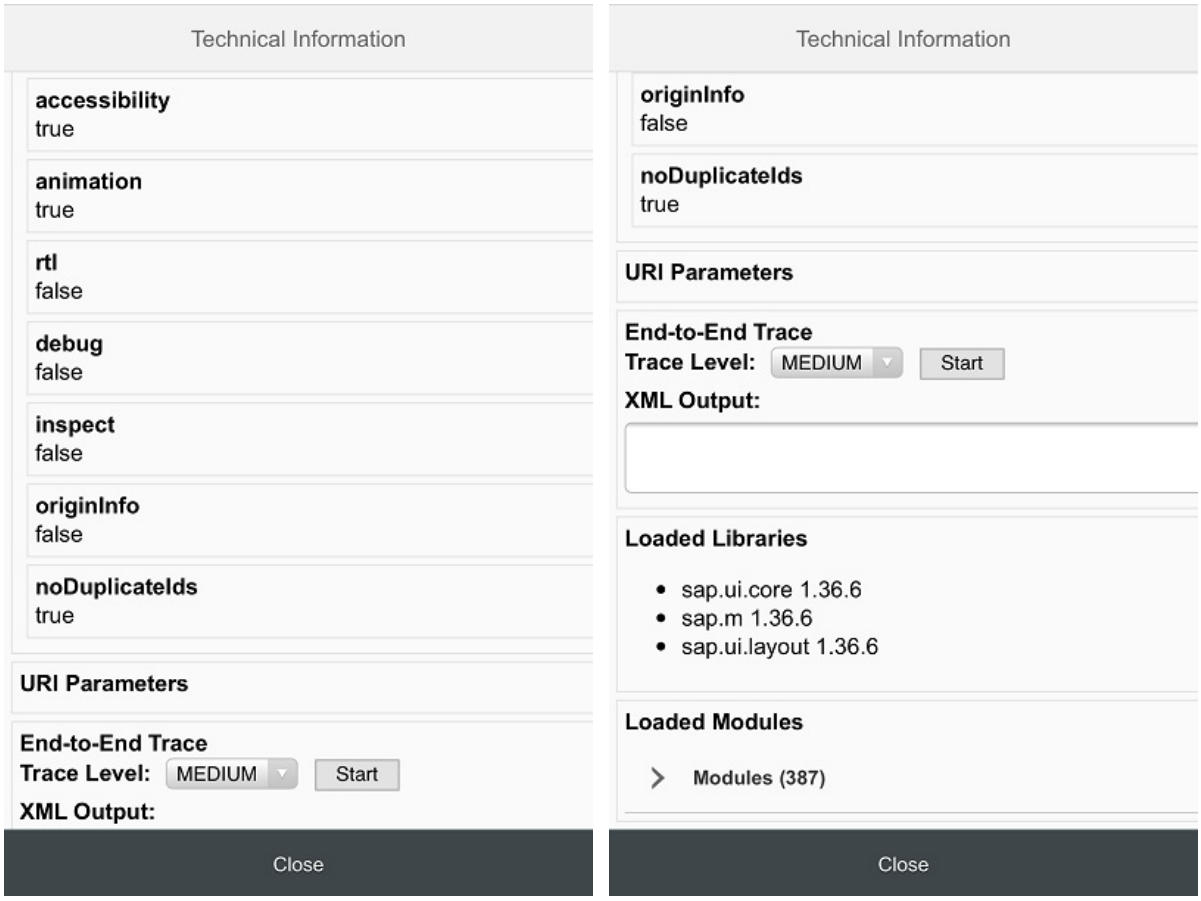
Technical Information Dialog on Mobile Devices

On mobile devices, the *Technical Information* dialog provides some additional features.

- *Configurations (bootstrap)* - Displays a list of bootstrap parameters.
- *Configurations (computed)* - Displays a list of computed parameters.
- *URI parameters* - Displays the variables passed from the URI query string.
- *End-to-End Trace* - A function that traces communication to a different part of the app.
- *Loaded Libraries* - Displays a list of the currently loaded libraries.
- *Loaded Modules* - Displays a collapsible list of the currently loaded modules.

Table 55: Technical Information Dialog on Mobile Devices





Accessing the Technical Information Dialog on Mobile Devices

To open the *Technical Information* dialog on your mobile device, proceed as follows:

1. Press two fingers on a noninteractive screen area (for example, a blank area) for at least 3 seconds.
2. Tap with a third finger while holding the other two on the screen.

> **!** Restriction
>
> - The *Technical Information* dialog can only be opened on mobile devices that support multi-touch.

Figure 221: Gesture for opening the technical information dialog

Using the End-to-End (E2E) Trace Function

The *E2E Trace* is used to create an XML file that traces the communication to a different part of your app.

To start the *E2E Trace*, proceed as follows:

1. Select the detail level from the *Trace Level* dropdown list.
2. Choose *Start*.
3. Navigate to a different part of the application.

As a result, a dialog box opens indicating that your transaction has finished. Choose *OK* to continue running the current trace. Choose *Cancel* to stop the trace and display the result in a new window.

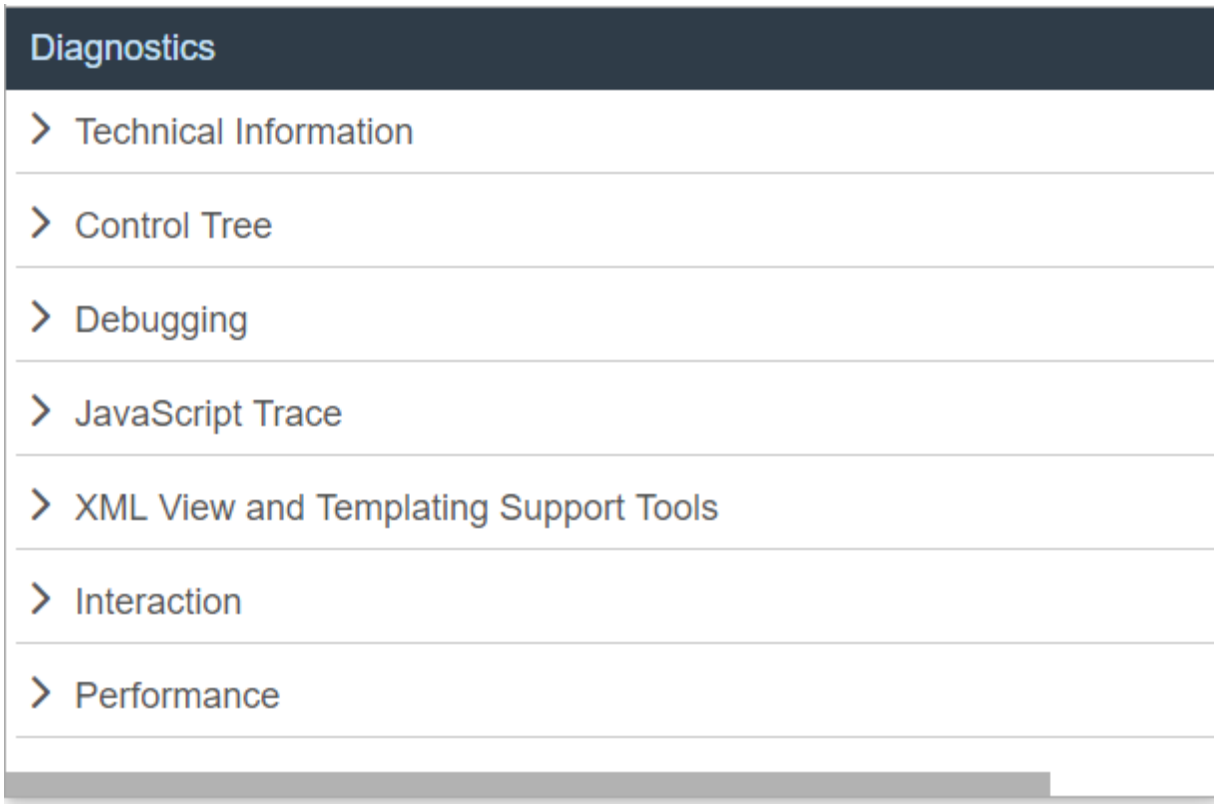
After stopping the trace, you can view the result and (optionally) upload it to a server by entering a `host name` and a `port number`, and then choosing *Submit*.

> **i Note**
>
> The result of the last completed trace is also visible in the *XML Output* field when you open the *Technical Information* dialog again.

Diagnostics

The *Diagnostics* window available in SAPUI5 is a support tool that runs within an existing SAPUI5 app.

To open *Diagnostics*, use the following shortcut: `CTRL` + `SHIFT` + `ALT` + `S` in the app.



Technical Information

This section provides the same features as the technical information dialog. You can view the technical details of the app, and turn on the debug sources. For more information, see Technical Information Dialog [page 1322].

In addition, you can see the following information:

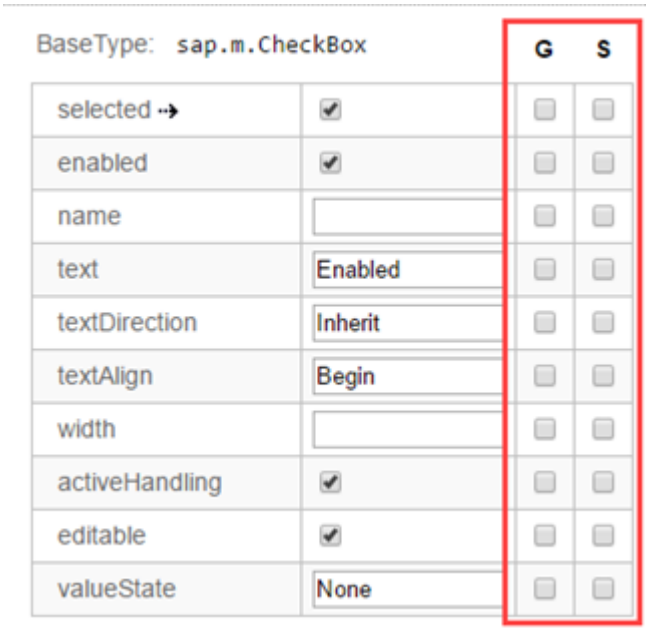
- The jQuery version that is loaded from the server. If you want to use a specific jQuery feature, you should check whether the feature is supported in the loaded version.
- The bootstrap configuration, where you can check, for example, the resource root, the theme, or the libraries. The libraries should be listed in the `libs` configuration parameter or in the descriptor file (recommended), see Descriptor for Applications, Components, and Libraries [page 734]. To improve performance, remove unused libraries and add the libraries that you use and are not yet listed. These libraries are loaded as a preload file. We recommend to also add the `async` configuration option to the bootstrap. This configuration option enables asynchronous loading of modules and preload files and can, thus, further improve performance. For more information, see Performance: Speed Up Your App [page 1434].
- The computed configuration
- The version of each library that is available and of the libraries that are loaded
- A list of all loaded modules
- URI parameters that are set
- End-to-end (E2E) trace function
  The *E2E Trace* is used to create an XML file that traces the communication to a different part of your app. Start the E2E trace and navigate in the app. Afterwards, a dialog opens indicating that your transaction has finished. Choose *OK* to continue running the current trace. Choose *Cancel* to stop the trace and display the result in a new window.
  After stopping the trace, you can view the result and (optionally) upload it to a server by entering a `host name` and a `port number`, and then choosing *Submit*.

Control Tree

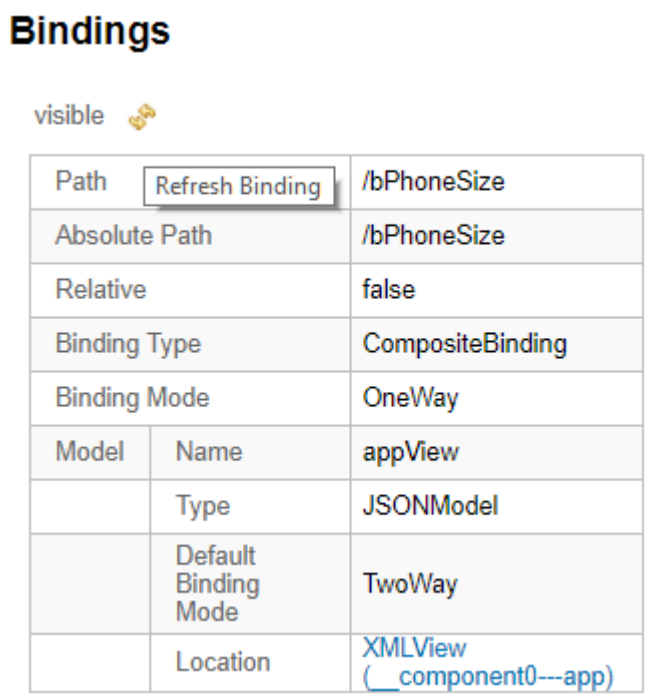
The control tree shows all controls that are used in the app. You can select controls either directly in the app by choosing `CTRL` + `SHIFT` + `Alt` and clicking on the control, or by selecting the control in the control tree.

The following functions are available in the dialog:

- On the *Properties* tab, you can change the defined properties of the selected control, and you can add or remove breakpoints. Use the respective checkbox to add or remove a breakpoint for the get and set method of a control property.



- The *Binding Infos* tab shows all existing bindings for the selected control together with additional information. To update the binding, choose *Refresh Binding*.



You can also see the binding context for the selected control. To navigate to the respective controls, use the hyperlinks.

- On the *Breakpoints* tab, you can add or remove breakpoints for methods on object level. You can either select the method from the dropdown box, or use auto-completion. To set the breakpoint, select the method and choose *Add breakpoint*. To remove a breakpoint, select the red x.
  For more information, see Breakpoints on the Object Level [page 1319].
- Many code samples are written in JavaScript. To facilitate the conversion of these code samples into XML or HTML, SAPUI5 provides a generic conversion tool. To run the tool, proceed as follows:

1. Select the root UI area in the tree on the left-hand side.
2. Open the *Export* tab and choose *Export*.
3. Open the ZIP archive and extract the files to your file system.

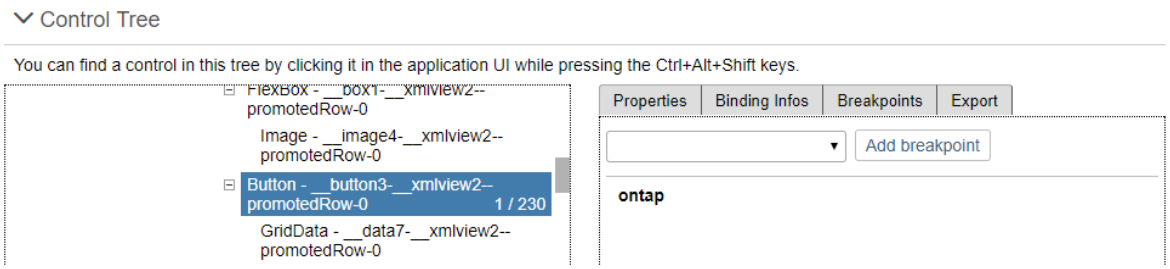
If your app does **not** contain views, the content is put in one view in the output. If your app contains views and all views are loaded, the content is output as separate files.

> **i Note**
>
> The conversion captures the runtime status of the app. This can differ from the build declaration.

Breakpoints on the Object Level

In the *Control Tree* of the *Diagnostics* window, you can set breakpoints on the object level.

1. Open the *Control Tree* view of the *Diagnostics* window.
2. Select a control in the tree.
   You can also press and hold `Ctrl` + `Shift` + `Alt` and select a control in your app to select it in the tree.
3. Select the *Breakpoints* tab on the right.
4. From the dropdown list, select the method for which you want to set the breakpoint and choose *Add breakpoint*.
   The selected methods are listed below the dropdown list.
5. Open the developer tools of your browser. Whenever the selected methods are called for any instance on the control, the code execution is paused in the debugger.



6. To remove a breakpoint, select the red x.

Debugging

The *Diagnostics* window provides you with the following features that help you when debugging your app:

- You can switch the SAPUI5 version you want to debug with.
- You can set breakpoints for methods on class level.

Switching the SAPUI5 Version

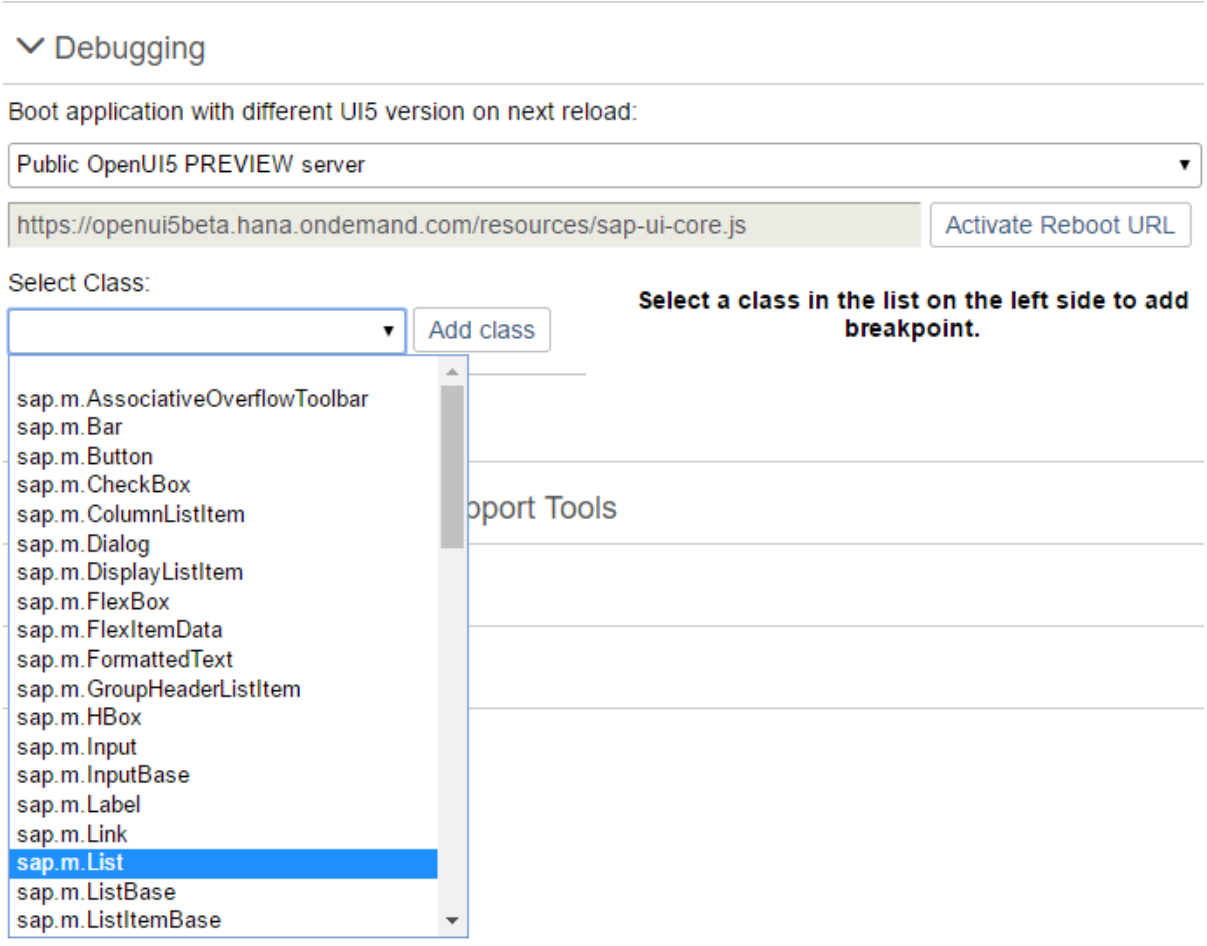
Open the *Diagnostics* window with the shortcut CTRL + SHIFT + ALT + S.

At the top of the *Debugging* view, you can configure a custom URL from which the application should load SAPUI5 the next time that the app opens.

Either select a known SAPUI5 installation from the dropdown box, or enter a different URL that points to the `sap-ui-core.js` file within a complete SAPUI5 runtime.

Once you have entered the URL, press *Activate Reboot URL*. When you then reload the application page, the application loads SAPUI5 from the alternative URL. This only happens for the next single reboot; after that, SAPUI5 is loaded again from the standard URL referenced within the app.

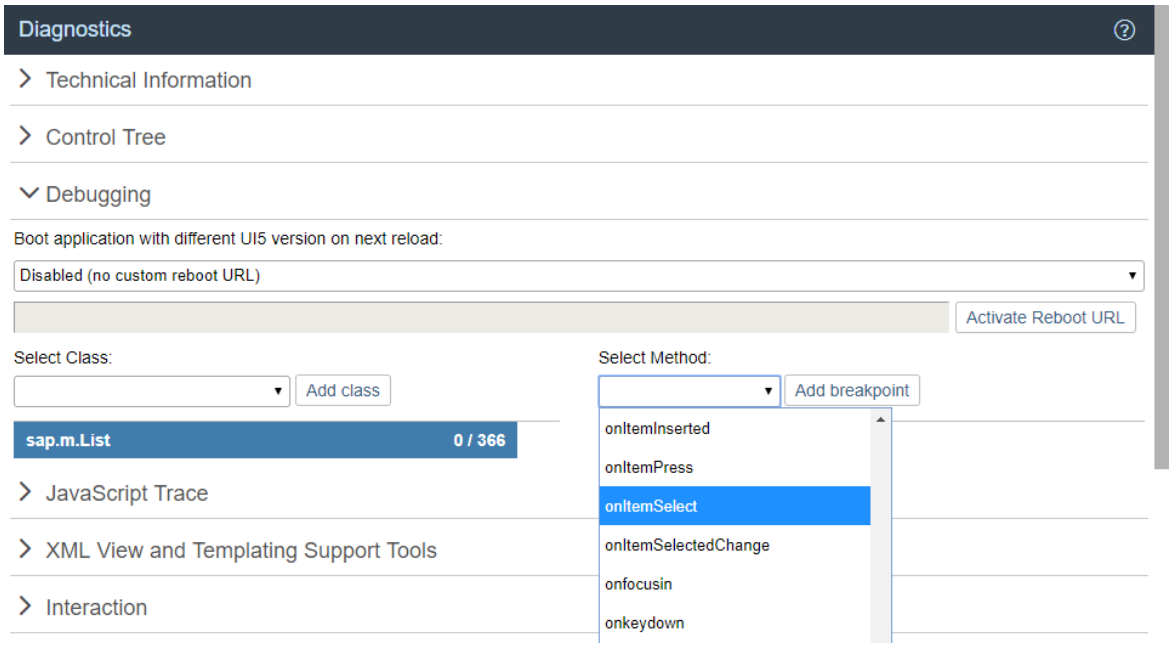
This feature can be used to test an application against a newer or older version of SAPUI5 as part of compatibility testing, or for verifying a bug fix or regression.



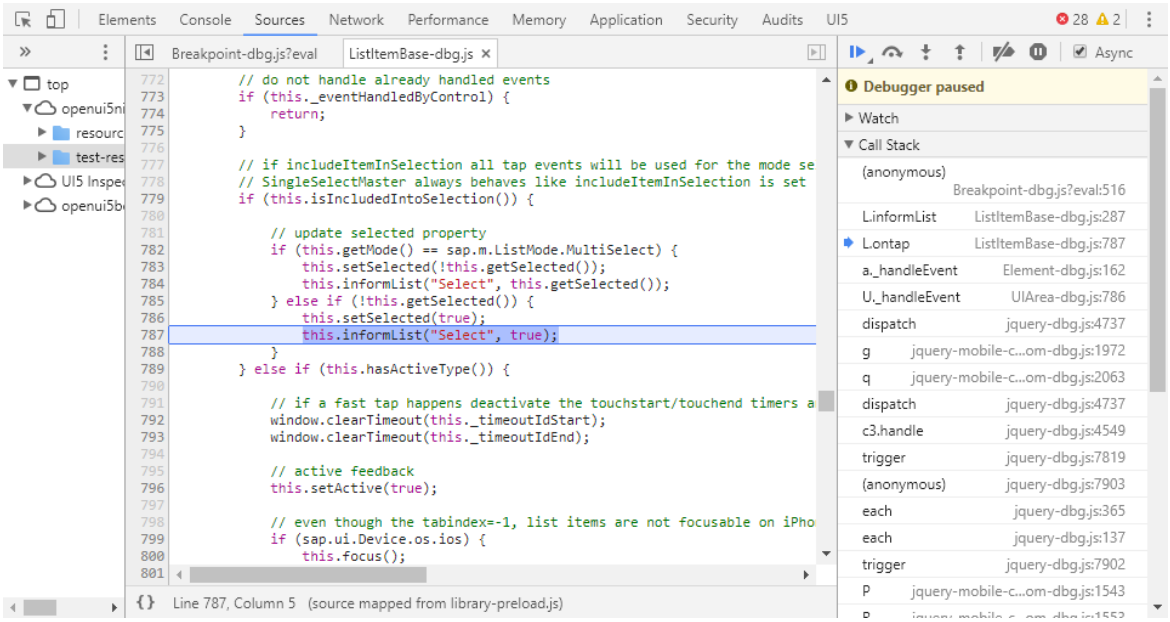
Breakpoints on the Class Level

In the *Debugging* section of the *Diagnostics* window, you can set breakpoints on the class level.

1. Open the *Debugging* view of the *Diagnostics* window.
2. Select a class from the dropdown list or enter the name of the class and choose *Add Class*.
   The selected class is now visible below the dropdown list.
   The number next to the method name shows the number of methods that belong to the class and the number of methods for which a breakpoint is set.
3. Select the class. On the right side of the view, you can now select methods of the selected class from a dropdown list.
4. From the dropdown list, select the method for which you want to set the breakpoint and choose *Add breakpoint*.
   The selected methods are listed below the dropdown list.
5. Open the developer tools of your browser. Whenever the selected methods are called for any instance of the selected control, the code execution is paused in the debugger.



In the call stack you find the method for which you set a breakpoint.



6. To remove a breakpoint, select the red x.

XML View and Templating Support Tools

This section of *Diagnostics* shows the code of the loaded XML view exactly as you would see it in your development environment. This way, you can check and test your XML code without the need to switch environments. If your app is connected to a remote service or a back-end system, you can also view the XML metadata.

Before you can use this feature, restart your app in Support Mode (with the `sap-ui-support=true` URL parameter added to the URL or your app).

XML Metadata

You can display XML metadata by expanding the related nodes in the tree. This helps you to better understand how the data is stored in the back end and which properties it has. This is especially important when investigating binding issues.

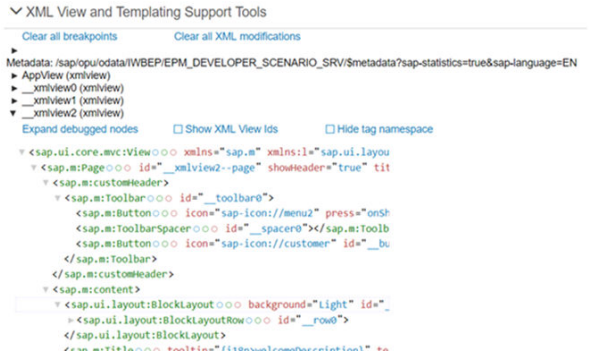
XML metadata of a service with three main entity sets: `Product`, `ProductCategory`, and `FeaturedProduct`

XML Code

When you expand an XML view in the tree, you can display the XML code.

You can choose the following options for this view:

- You can choose which kind of IDs you want to see:
  - IDs as they are in the DOM (option *Show Real IDs*)
  - IDs as they are defined in the XML view (option *Show XML View ID*)
    If no stable ID is defined in the view, the ID tag is empty (`id=" "`), if there is an ID, the value is set to `true` (`id="true"`).
- You can show or hide the namespaces to improve readability of the code.



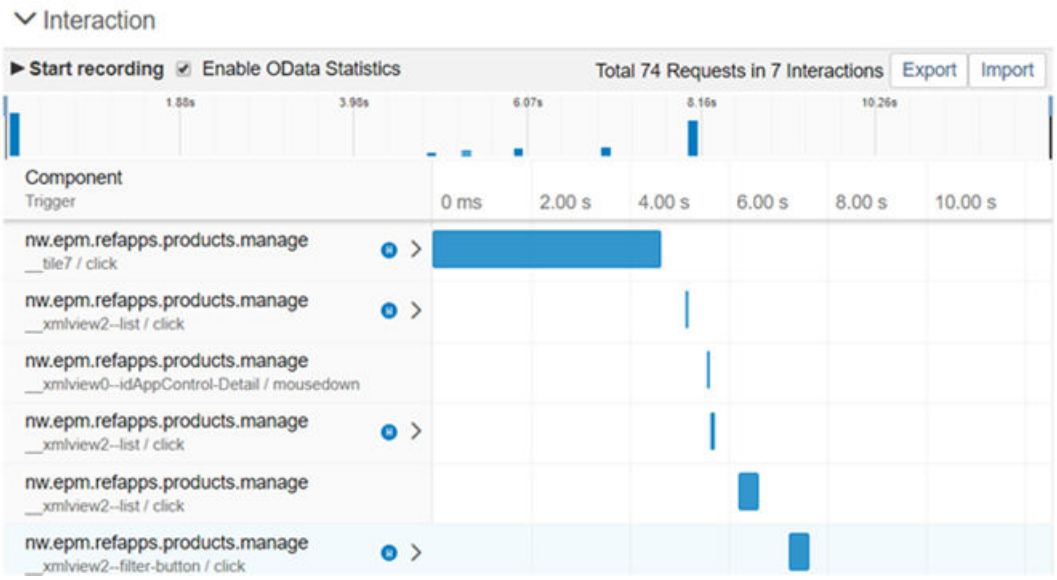
When you select a control in the code, the following information is displayed:

- Name of the control with a link to the API Reference and its ID in the DOM tree
- Instances that are **cloned** in the control with their IDs
- Attributes and properties of the control with their values
  Similar to the *Control Tree* section of *Diagnostics*, you can change those values here for testing purposes.
- Methods that are available for the control.

> **Note**
>
> When you select, for example, an aggregation, you see the name of the control which has this aggregation or the name of the parent.

Visualizing User Interaction

With this feature, you can collect and visualize the performance data collected for the interaction steps in an easy and intuitive way. In addition, you can enable statistics for OData calls that give you information about the app processing time taken by the OData back end.



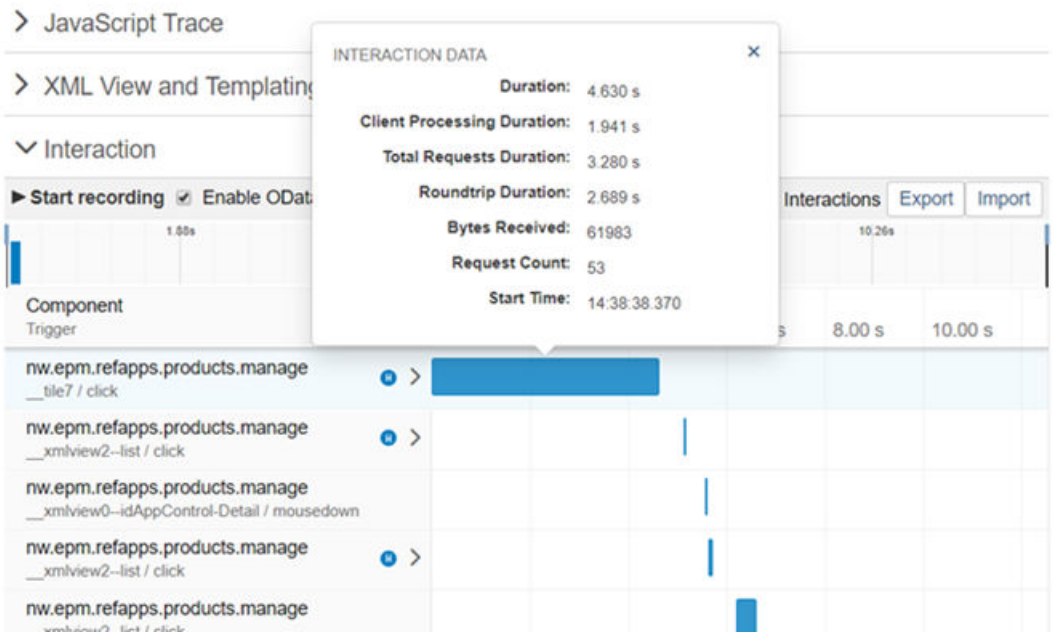
Activation

You can start the interaction data collection in the following ways:

- Enable interaction steps recording:
  - To measure the initial loading of the app, add the query string parameter `sap-ui-xx-fesr=true` at the end of the application URL and reload.
  - To measure the interaction performance, choose *Start recording* from the *Interaction* panel and then switch back to the app to do the steps that you want to record. Each user activity, such as clicking buttons or list items or scrolling a list, triggers an interaction. The end of an interaction is when the UI is fully updated by the app. The collected data is displayed once you choose *Stop Recording*.
- You can enable the OData statistics by using query string URL parameter `sap-statistics=true`, or from the UI by selecting *Enable OData Statistics*.

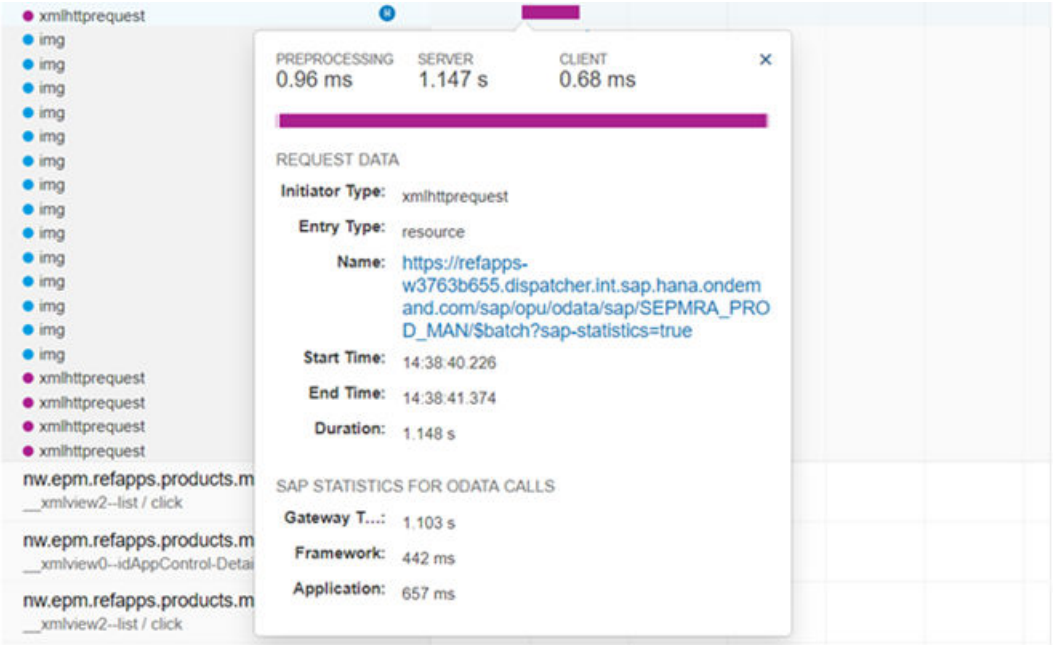
Output

- **Interaction details** - Selecting an interaction step bar, opens a dialog with details about the selected interaction: end-to-end duration, total number of requests, the size of the transferred data, and so on.



- **Interaction requests** - All interactions contain 0 to N requests, which can be displayed in a list by selecting the expand icon.
- **Request details** - Selecting a request opens a dialog with details, such as request type, URI, overall duration and how it is spread across client, server, and connection establishing processing. When OData statistics are enabled and the request is processed by the SAP NetWeaver OData gateway, in the details dialog there is a section with OData times – *Gateway Total*, *Framework*, and *Application* processing times.

> **i** Note
>
> The OData time (*Gateway Total*) is included in the total server time processing. Such requests and their interaction are identified with a blue icon H.



Additional features

- **Export** - Collected data can be exported as a ZIP archive for easy distribution through e-mail.
- **Import** - Already collected data stored as an archive can be visualized again, using the *Import* function. The input can be either an already exported archive file or directly the JSON file that contains the performance data. This enables remote analysis of app data by another team or expert.

Related Information

Interaction Tracking for Performance Measurement [page 1382]

SAP Fiori Launchpad Configuration

This feature is only available when an SAP Fiori launchpad has been loaded. You can quickly view the SAP Fiori launchpad configuration on a specific client.

You can search for parameter names.

The parameters are displayed in a tree structure. You can filter the tree by clicking on the categories that are displayed below the search field. These categories correspond to the first-level entries in the tree.

> **i Note**
>
> This tool displays the complete startup configuration, including launchpad configuration parameters that might be deprecated or that might be subject to change. Any changes that occurred after startup are not reflected.

Back-End Infrastructure

With this feature, you can check whether and to what extend your SAP NetWeaver installation can run and support SAPUI5 apps.

> **i Note**
>
> To use this feature, you need to be authorized in the back-end system (authorization object `S_DEVELOP` for `OBJTYPE` = '*DEBUG*' and '*ACTVT*' = '*03*') and ICF node `/sap/bc/ui2/check_app_infra` needs to be activated.

You can display the relevant messages for the following topics:

- Configuration and availability of the virus scanner with the profiles `/UI5/UI5_INFRA_APP/REP_DT_PUT`, `/SCET/GUI_UPLOAD`, and `/SCET/GUI_DOWNLOAD`
- Status of the background job scheduled for report `/UI5/APP_INDEX_CALCULATE`, which calculates the SAPUI5 application index
- Potential error messages in the log of this index
- Status of the standard ICF nodes used by SAPUI5 apps, components, and libraries

In addition, you can access the SAP NetWeaver version and software component as well as other information.

Flexibility

With this feature, you can check whether there are SAPUI5 flexibilty changes for the controls used in an app, and you can analyze these changes.

For example, you can see the layer of the changes in the layered repository, their type, or whether they're active. Active and erroneous changes are only evaluated for the controls currently in the DOM, in the current runtime.

Prerequisites

This tab only displays apps that use the `sap.ui.fl` library.

SAPUI5 needs to be in debug mode.

List of Applications

The *Flexibility* panel displays a list of applications that have been handled by the `sap.ui.fl` library in this session.

For each app, you can download a JSON file containing the data that has been applied to an application as well as relevant runtime information.

The JSON file contains changes on all layers. Personalization changes that have been saved to the USER layer are only collected for the current user.

You can send this JSON file to SAP support for further investigation, or you can open the *UI Flexibility Diagnostics* application to investigate yourself.

UI Flexibility Diagnostics Application

In the *UI Flexibility Diagnostics* application, upload a JSON file that you downloaded from the *Flexibility* panel in the *Diagnostics* window.

The *UI Flexibility Diagnostics* application displays all changes that have been loaded for the application. The arrows visualize dependencies between these changes. You can quickly spot which changes have been applied by checking their color:

Color	Description
Green	This change has been applied to the control in the current system.
Red	The change could not be applied, and an error was raised.
White	There was no attempt to apply the change either because required controls were not present, or because preconditions were not fulfilled.

For more information, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152].

SAP Fiori Elements

With this feature, the system helps you collect the data related to issues you encounter when creating an SAP Fiori elements app.

> **i Note**
>
> This feature is available only in SAP Fiori elements applications. You can use it only for list report, object page, worklist, and analytical list page applications.

After you have opened the *Diagnostics* window, perform the following steps:

1. Choose *Copy*.
   By default, the system copies plain text to the clipboard. You can also choose to copy HTML.
   The system automatically collects the following relevant data and copies them to your clipboard:
   - Data collected from the application:
     - Used UI5 version and build date
     - Absolute URL to manifest.json of the current application
     - Application component (technical name)
     - Used SAP Fiori elements floorplan
     - Data sources
     - Absolute URL to metadata document
     - Absolute URLs to local and backend annotations
   - Data which is added when copying:
     - Absolute application URL
     - Date and time the data was collected
     - Status of application (for example, loading, rendered, or failed)
     - Error message if the application did not finish loading
2. Paste the data from your clipboard into a new ticket.
3. To complete the ticket, under *Provide*, enter the login credentials for the system, as well as the steps to reproduce the issue.

Support Assistant

The Support Assistant enables developers to check whether their apps are built according to the SAPUI5 best practices and guidelines.

Goals

The tool aims to reduce maintenance and consulting times and to streamline SAPUI5 app development. It uses a set of predefined rules to check all aspects of an application, for example, accessibility, performance, data binding, usability. With a simple click, you can check the current state of your app. After execution, you can analyze the results and apply corrective measures based on the outcome.

Check out the Support Assistant highlights video for an overview of its main functionalities:

Getting Started

The Support Assistant can be started using a URL or a Technical Information Dialog.

From a URL Parameter

The Support Assistant is enabled with the following URL parameter: `sap-ui-support=true`. The tool then appears as a toolbar in the footer of the app.

> → Tip
>
> If you want to run the Support Assistant in a separate window, use the parameter `sap-ui-support=true,window`

Figure 222: Support Assistant Toolbar

From the Technical Information Dialog

You can also start the Support Assistant from the Technical Information Dialog.

1. Open the Technical Information Dialog by using the following shortcut: `CTRL` + `SHIFT` + `ALT` + `P`.
2. Choose *Activate Support Assistant*.

Starting the Support Assistant from here allows you to run it with a different SAPUI5 version. You can find more details on this topic in Running the Support Assistant on an Older SAPUI5 Version [page 1356].

Selecting *Rules* will show you the available rulesets. You can then select your rules and start the analysis of the app.

Persisting Rules and Settings

All scopes and temporary rules can be stored in the local storage of your browser. This will allow you to continue with your work even after you have closed the browser window. To enable this feature, choose *Settings* () on the banner and select the checkbox *I agree to use local storage persistency for*.

> → Tip
>
> You can delete your already persisted data by choosing *Delete Persisted Data*.

Features

- To learn more about rules and rule management see: Rules Management [page 1341]
- To learn more about result processing and reporting see: Results and Analysis [page 1344]
- To learn more about creating your own rules see: Rule Development Guide [page 1359]

Related Information

Step 3: Support Assistant [page 202]

Using the Support Assistant

The user interface of the Support Assistant allows you to view the available rules and load additional rulesets for an active application. You can also run an analysis and view the issues identified. The results are available in the form of a consolidated report, generated as an HTML document.

Rules Management

The user interface of the Support Assistant lets you choose which rules you can load for a library. It also allows you to organize your Rules view according to your preference and to import and export predefined rule selections.

Available and Additional Rulesets

The *Available Rulesets* tab contains the list of the currently loaded rulesets used by your application. On the left, there is a list of the available rules per library. On the right, you can see more details on the currently selected rule. By selecting the checkbox in front of each rule, you determine which rules are executed in the analysis. The list of available rulesets is dynamic and changes based on the libraries used for the current state of your application.

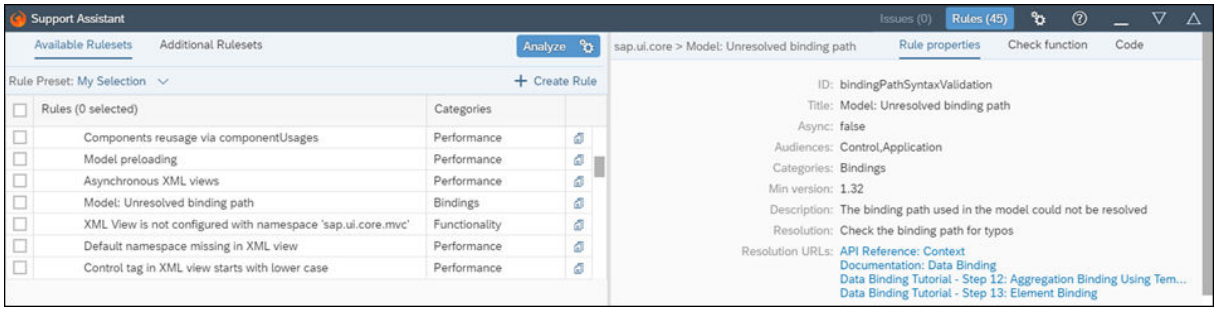


Figure 223: Support Assistant Available Rulesets

The *Additional Rulesets* tab shows rules for libraries that are not used by the application at the particular moment.

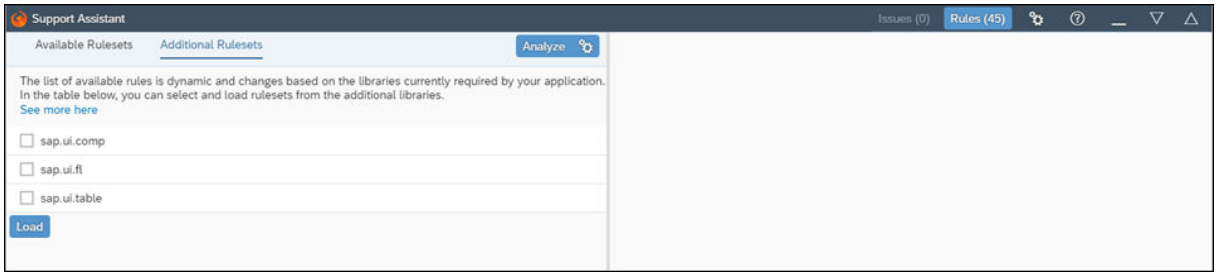


Figure 224: Support Assistant Additional Rulesets

You can select the rules from the *Additional Rulesets* tab and choose *Load* to move them to the set of *Available Rulesets* and use them in your analysis.

Rule Presets

The Support Assistant allows you to export and import subsets of preselected rules or *Rule Presets*. The rule presets are semantically grouped selection variants which you can export and save for future analyses of your apps. There are two general types of rule presets - *Custom Presets* and *System Presets*.

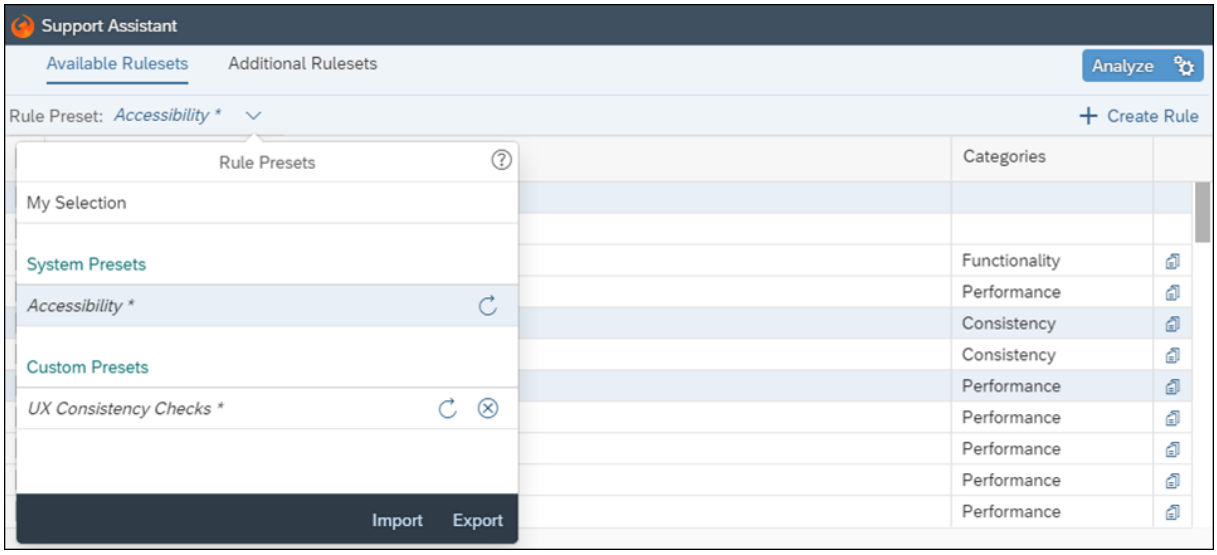


Figure 225: Support Assistant Rule Presets

Custom Presets

Custom presets are rule selections which you can group according to your particular analysis purposes and export as `.json` files. You can give the exported file a title, ID and description that are meaningful to you.

The ID is an alphanumeric string which, although not mandatory, is useful if you collect data and generate reports on your rule presets. It makes it easier to identify and report on specific rule preset executions which are of interest to you. In case you don't create an ID, the system will automatically generate one. For more information, check Analysis Report [page 1347].

Once exported and saved, your rule presets can be imported and used again. They are listed in a dropdown menu allowing you to easily switch between them.

> → Tip
>
> In *My Selection* you can find your most current selection of rules. To preserve it for your next analyses together with your choice of imported presets, go to the settings menu on the Support Assistant toolbar and tick the checkbox in front of *I agree to use local storage persistency for*.

System Presets

For your convenience, the Support Assistant is also equipped with ready-to-use system-defined rule presets. They contain selections of rules related within the context of a scenario, functional area, or other aspects of the

app UI that can be verified using support rules. The first system preset which has been added is for Accessibility-related rules.

You can't delete system presets but you have the option to modify and export them as you do with your custom ones. Although the rules selected within one system preset are grouped according to their relevance to a certain scenario, they can belong to different categories.

> → Tip
>
> If you modify the rule selection within a system preset, an asterisk appears next to its name as an indication of changes that haven't been saved. You can undo these changes by selecting the refresh icon opposite the preset name .

Columns Personalization

You have the option to personalize the Rules view by choosing which columns to be displayed. Just click on the column header and select or deselect the columns you want to use. You can also sort the content of all four columns or filter by keyword.

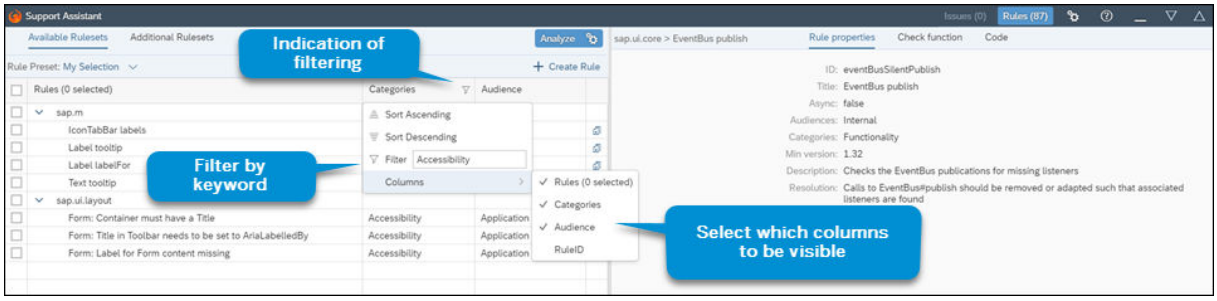


Figure 226: Support Assistant Columns Personalization

> i Note
>
> Choosing *I agree to use local storage persistency for* from the settings menu will also preserve your choice of visible columns.

Creating Rules

Selecting the *Create Rule* button allows you to create a new rule. You can create a completely new rule (*Create Rule*) or use an existing one as a template by clicking the *Copy* ( ) icon next to *Categories*. For each rule, you need to fill out the *ID*, *Categories*, *Audiences*, *Title*, *Description*, *Resolution*, *Min version* and, if available, *Resolution URLs* in the *Rule properties* tab. You also have the option to select if the *Async* value should be `true` or `false`. This value determines whether the rule check function will contain asynchronous operations. By default, it is set to `false`.

> **i Note**
>
> Keep in mind that if you set *Async* to true, you need to use `fnResolve` as the 4th parameter in your check function to indicate that the asynchronous check function has finished. The asynchronous function waits 10 seconds before it times out.
>
> Here is an example of the Async check function:
>
> ```
> function(issueManager, oCoreFacade, oScope, fnResolve) {
>   // Some async operation
>   setTimeout(function () {
>     …
>     fnResolve();
>   }, 2000);
> }
> ```

Additionally you need to provide or modify the JavaScript check function that implements the rule in the *Check function* tab. You can directly test the newly added or modified rule on the already loaded page.

The newly created rule remains *temporary* until you submit and assign it to a library.

> **→ Remember**
>
> Don't forget to copy and paste the resulting new rule and submit it separately in the IDE of your choice. You can select all the code from the *Code* tab.

Executing Rules

Once you load your rulesets or select a rule preset, you can run an analysis with them. To do this, select *Analyze*. For more information about the execution scope, you can refer to Execution Scope [page 1346]

Related Information

Create a Ruleset for a Library [page 1360]
Create a Rule [page 1363]

Results and Analysis

After an analysis run, you can view a list of all triggered rules, their description, resolution steps, and a control tree with highlighted problematic elements.

Below you can see an example of how the results are displayed.

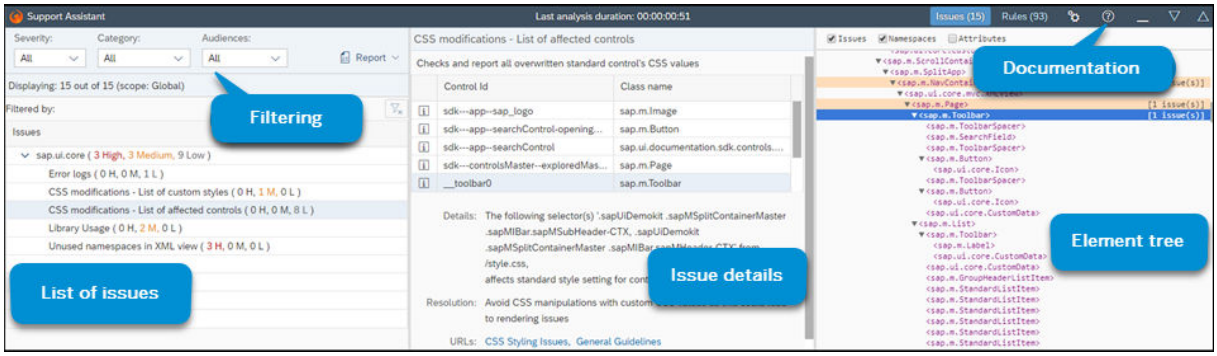


Figure 227: Support Assistant Issues View

Issues List

The left side shows a list of the triggered rules sorted by severity (High, Medium, Low). You can use the dropdown menus to filter on *Severity*, *Category*, or *Audience*. You can clear your filtering by choosing *Clear Filtering* ( ).

Selecting *Report* generates an HTML report with the current set of rules and scope. You can view it in a new tab or download it as a `.ZIP` file.

Issue Details

The middle part shows a detailed view of the selected rule. It contains the following:

- Description - general description of the rule
- Resolution - steps to resolve the issue
- URLs - useful links (for example, API Reference or Documentation)
- Severity - the severity of the identified issue
- Element ClassName and ID - the namespace of the element

Element Tree

The right side shows the element tree of the application. The root of the tree is called `<WEBPAGE>`. All rules that are not specific for a given control are mapped to this element. If these rules are triggered, the resulting issues will be mapped to the `<WEBPAGE>` element.

- Hovering on an element in the tree highlights it in the application (if it is visible).
- Selecting an element with issues in the element tree loads those issues in the issues list.
- The table in the issues list shows all other elements that have triggered the same rule. The list also shows you the severity of each issue.

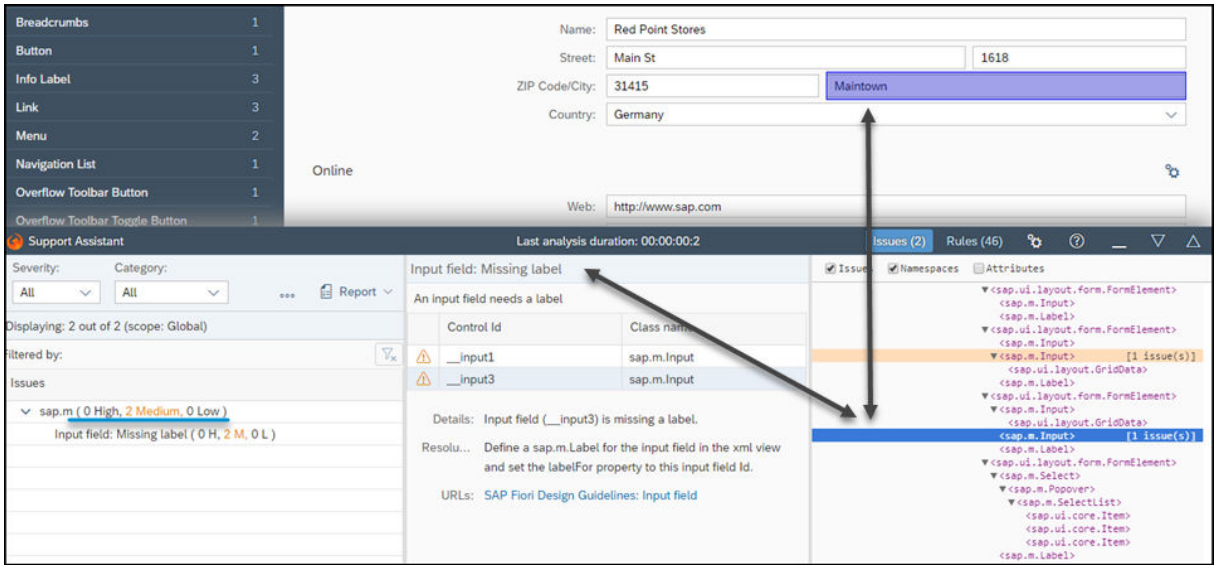


Figure 228: Support Assistant DOM Tree

Execution Scope

SAPUI5 apps consist of multiple views, components and fragments. If you run the rules on the complete app with all loaded elements and components, you may not get the desired results. Therefore, the Support Assistant allows you to change the scope of an analysis run and thus narrow down your result set. This helps you focus on specific issues, components or controls within your app.

Execution Scopes

To change the analysis scope, select the gears icon next to the *Analyze* button.

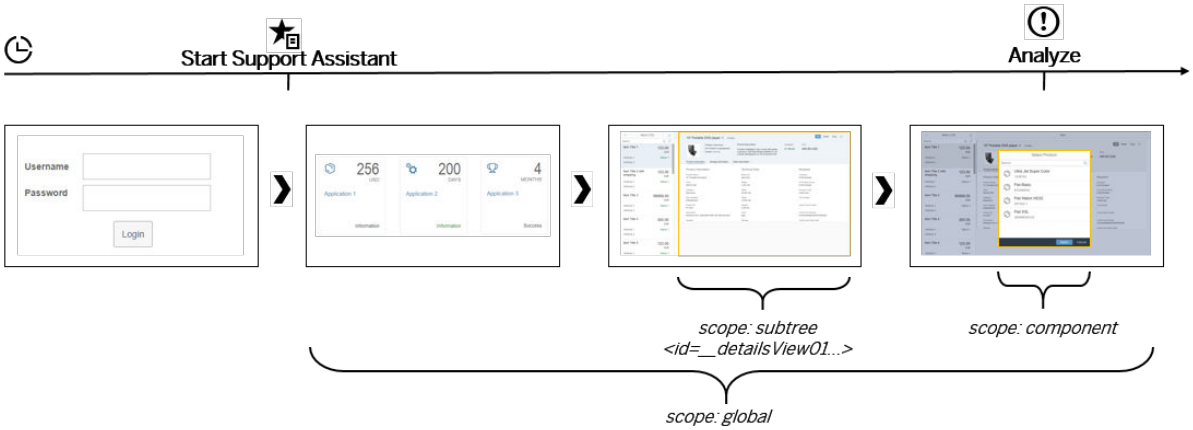


Figure 229: Support Assistant - Execution Scopes

There are three available execution scopes:

- *Global* - The rules are executed on the whole application, including all loaded components, elements and previously opened pages.

- *Sub-tree* - The rules are executed on the specified subtree root element and all its subelements (which are not components themselves).
- *Component(s)* - The rules are executed on the specified set of loaded components/fragments.

> **i** Note
>
> The scoping information will also be taken into account when generating the analysis report.

Analysis Report

The information from the execution of the Support Assistant is available as a separate HTML document. It can be viewed from the *Report* dropdown menu.

The report contains information from the loaded components, a detailed list of the technical information and a list with all issues. The following image shows the report with collapsed sections.

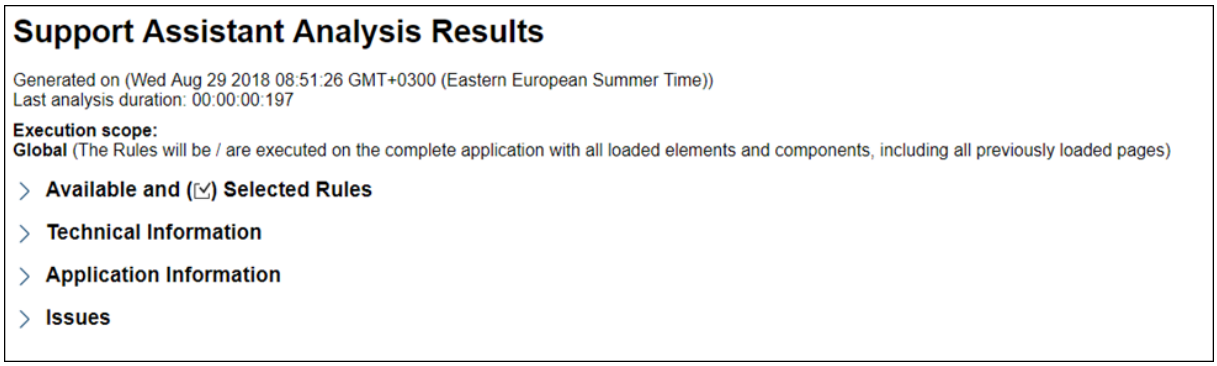


Figure 230: Analysis Results: Collapsed View

The report contains the following elements:

1. A time stamp for when it has been generated.
2. Last analysis duration showing the time taken by the last analysis.
3. An execution scope with short description.
4. Available and selected rules.
   Here you can see all available rules per library. All rules that have been selected for the analysis are marked with a checkmark (☑). This section also gives you information about the used rule preset and its ID.

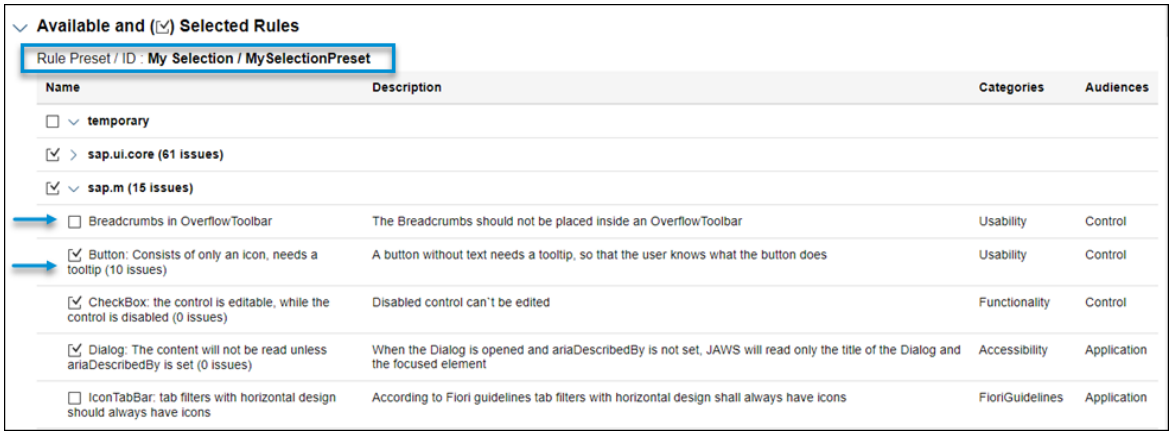


Figure 231: Analysis Results: Rules

5. Technical information section. Here you can see the version of the Support Assistant and the location from which it has been loaded as well as more information about the app.

Figure 232: Analysis Results: Technical Information

6. Application information section.

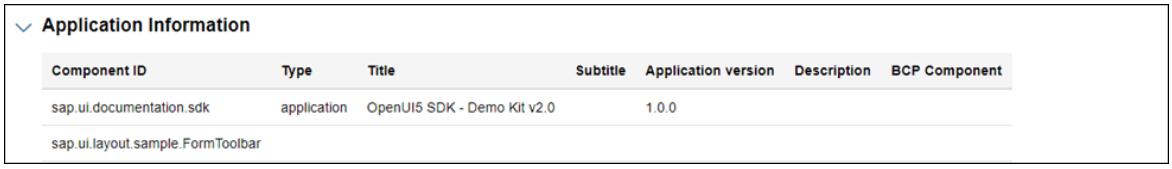


Figure 233: Analysis Results: Application Information

7. Issues section.
   The issues are grouped by library and rule. In the following example, there is one library (`sap.m`) with 5 rules. They have generated 18 issues in total - 16 with medium severity and 2 with low severity.

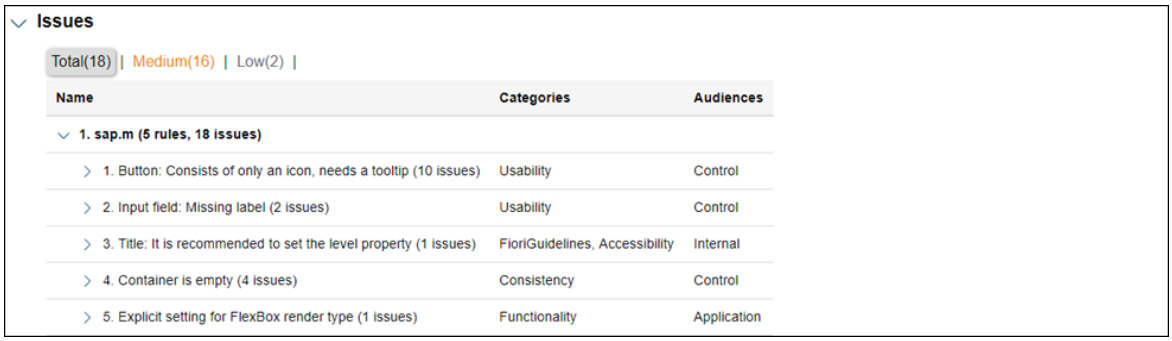


Figure 234: Analysis Results: Issues

You can also download the report by selecting *Download* (below the *View* button). The report is going to be downloaded in a ZIP format containing the following files:

- The report HTML (*report.html*)
- A JSON file with all loaded components *(appInfos.json*)
- A JSON file with all issues (*issues.json*)
- A JSON file with all technical information (*technicalInfo.json*)

Integrating the Rules in OPA Tests

The Support Assistant can be used as part of an existing OPA test to cover more test aspects of the application.

Context

The Support Assistant can be used in OPA tests to check if there are issues in the different states of the application. To do that, you need to use the Support Assistant OPA extension. This extension is available as of version 1.48. It provides three assertions:

- `noRuleFailures` - Analyzes the current state of the application, and if errors are found, the assertion will fail. A non-mandatory `options` object can be passed to the assertion containing the following properties:
  - `failOnAnyIssues (boolean)` - Determines if the assertion should fail if issues of *any* severity type are found.
  - `failOnHighIssues (boolean)` - Determines if the assertion should fail if issues of severity type *high* are found. Warning - this parameter will ignore issues of severity types: *medium* and *low*.

    > **i Note**
    >
    > This parameter overrides `failOnAnyIssues`.

  - `rules (Array)` - Determines a subset of rules to check. By default if this property is not set, all rules are checked. The rules have two properties `libName` (for example, `sap.ui.core`) and `ruleId` (for example, `orphanedElement`).
  - `executionScope (Object)` - The execution scope defines the scope of the analysis. Can be of type *global*, *subtree*, *components*.

    > **i Note**
    >
    > If types *subtree* or *components* are selected, the `selectors` property should also be set to define the IDs of the subtree/components.

- `getFinalReport` - If there are issues found, the assertion fails and a report is created as part of the message of that assertion.
- `getReportAsFileInFormat` - Collects the past history analysis and stores it in `window._$files` array for further usage. The main purpose of this assertion is to allow the OPA extension to serve the history to external services like Jenkins job or other services so that the data can be stored on the filesystem.
  The assertion can be called with two optional parameters: `historyFormat` - The format into which the history object will be converted. Possible values are listed in `sap.ui.support.HistoryFormats.` and `fileName` – the name of the file in which the history will be stored.

In addition, if you pass `sap-skip-rules-issues=true` as a URL parameter to your OPA test, the assertion results of `noRuleFailures` and `getFinalReport` assertions will be `true`, overriding the actual results.

This special URL parameter could be used temporarily in cases when you extend an existing OPA test to run the Support Assistant rule checks initially but you don't want the entire OPA journey to fail immediately. After you gain experience and clean up any check issues, you can set it to `false` or omit passing it and use once again the desired `onError` behavior.

> **i Note**
>
> When the `sap-skip-rules-issues` URL parameter is set, it affects all tests globally, unlike the `FailOnAnyIssues` parameters, which only affect a specific test level.

Procedure

1. Enable the Support Assistant OPA extension in the OPA configuration file.

   You need to change two parameters:
   - `extensions` - You need to include the Support Assistant OPA extension path (`sap/ui/core/support/RuleEngineOpaExtension`).
   - `appParams` - You need to add `sap-ui-support` with a value of `true,silent`. This will start the application in support mode and will start the Support Assistant in silent mode (without UI).

   The configuration file will look like this:

```
sap.ui.define([
...
], function(Opa5, Arrangement) {
...
extensions: ["sap/ui/core/support/RuleEngineOpaExtension"],
appParams: {
  "sap-ui-support": "true,silent"
}
...
});
});
```

2. Add additional assertions to the OPA configuration file.

   Add generic or specific assertions - depending on the use case. For example:
   - `iShouldSeeNoHighSeverityErrors` - This assertion calls `noRuleFailures` with a few parameters set, as you can see in the example code below. It checks for high issues and ignores medium and low. The rules checked are `preloadAsyncCheck`, `orphanedElement`, `deprecatedEntities` and the scope is set to *global*.
   - `iShouldGetSupportRuleReport`- This assertion calls `getFinalReport` and if there are any issues after all the analysis, it fails and a report is created as part of the message.

   The configuration file should look like this:

```
assertions: new Opa5({
...
iShouldSeeNoHighSeverityErrors: function() {
  return this.waitFor({
    success: function() {
```

```
      Opa5.assert.noRuleFailures({
        "failOnHighIssues": true,
        rules: [{
          libName: "sap.ui.core",
          ruleId: "preloadAsyncCheck"
        }, {
          libName: "sap.ui.core",
          ruleId: "orphanedElement"
        }, {
          libName: "sap.ui.core",
          ruleId: "deprecatedEntities"
        }],
        executionScope: {
          type: "global"
        }
      });
    }
  });
},
iShouldGetSupportRuleReport: function()
{
  return
this.waitFor({

    success: function()
{


Opa5.assert.getFinalReport();

    }
  });
}
...
```

3. The added assertions can now be used inside the journeys.

   Knowing the flow of the tests, choose the right place in your OPA test journey to add the needed assertion:

```
...
opaTest("Should see no Support Assistant issues with high severity", function
(Given, When, Then) {
    Then.iShouldSeeNoHighSeverityErrors();
});
...
```

> **i Note**
>
> Put these assertions after the web page being tested has been rendered and displayed with a stable UI.

4. Repeat the extended OPA test and see how your specific Support Assistant assertions are triggered.

You can see a detailed report for each run. The report is tabular and lists all executed rules with their details, followed by a list of the issues generated by that rule. It looks like this:

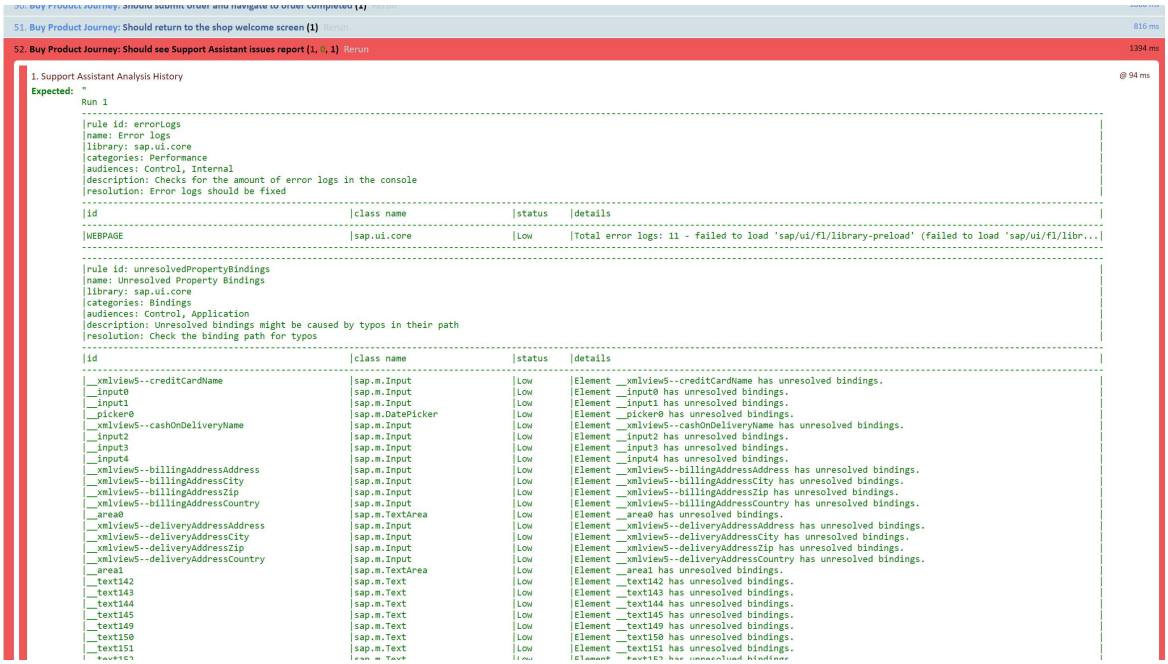


Figure 235: OPA Test Results

Related Information

Execution Scope [page 1346]
Integration Testing with One Page Acceptance Tests (OPA5) [page 1182]
Samples: `Running OPA tests with Support Assistant checks`

Support Assistant API

The Support Assistant can also run in silent mode and accept calls through its API. This way it can be integrated in more complex automated scenarios.

General Information

The Support Assistant is currently separated into two main parts:

- Core plug-in in SAPUI5
- UI client running in an iFrame or separate window, or programmable clients via an API

In the following diagram you can see how the Support Assistant is connected to the individual application layers.

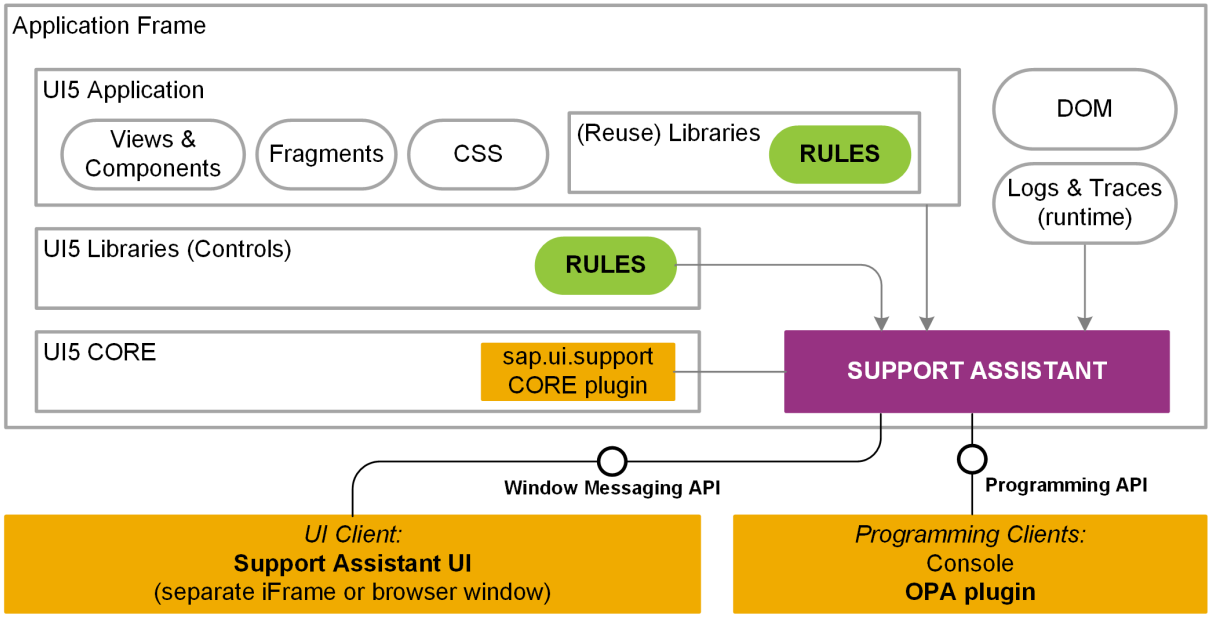


Figure 236: Support Assistant Architecture

There are two different use cases for its integration:

- Using browser window messaging protocol for agents in other window frames;
- Using the `sap.ui.support.RuleAnalyzer` module (for example, from the console or as part of OPA tests).

We will take a closer look into these use cases in the following sections.

Window Messaging API

The window messaging API is an asynchronous API based on the browser low-level `postMessage/onMessage` APIs. It is enabled by using a custom generic communication bus component - `WindowCommunicationBus`, delivered with the Support Assistant. The `WindowCommunicationBus` is used for implementing the remote UI interaction between the Support Assistant and the application `iFrame`.

Programming API

This section illustrates how to use the Support Assistant programming API through specific `sap.ui.support.RuleAnalyzer` **API examples**.

Add a Temporary Rule

After the Support Assistant has been started, if in silent mode, you can add a new temporary rule by using the `addRule` method. Then you can run an analysis with this rule.

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
```

```
         var oRule = {
            id: "Temp rule id",
            title: "Temp rule title",
            ...
         };
         RuleAnalyzer.addRule(oRule);
});
```

For more information about rule properties, see Guidelines and Best Practices [page 1369].

Run the Analysis

The Support Assistant API allows you to:

- Run a complete analysis on all components and rules. This analysis returns all issues.

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
   RuleAnalyzer.analyze().then(function() {
      var oHistory = RuleAnalyzer.getLastAnalysisHistory();
      ...
   });
});
```

- Run a complete analysis, using custom metadata. The analysis history will contain this metadata.

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {

var oMetadata = {
    …
};

   RuleAnalyzer.analyze(null, null, oMetadata).then(function() {
      var oHistory = RuleAnalyzer.getLastAnalysisHistory();
      ...
   });
});
```

- Narrow down the analysis to a specific part of the app, for example only a sub-tree.

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
         var oExecutionScope = {
            type: "subtree",
            parentId: "panelId"
         };
         RuleAnalyzer.analyze(oExecutionScope).then(function() {
            var oHistory = RuleAnalyzer.getLastAnalysisHistory();
         ...
         });
});
```

  For more information, see Execution Scope [page 1346].

- Check for issues using specific rules.

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
         var oExecutionScope = {
            type: "subtree",
            parentId: "panelId"
         };
         var aRules = [{
            ruleId: "inputNeedsLabel",
            libName: "sap.m"
```

```
        }];
        RuleAnalyzer.analyze(oExecutionScope, aRules).then(function() {
           var oHistory = RuleAnalyzer.getLastAnalysisHistory();
        ...
        });
     });
```

- Check for specific rules using rule presets.
  The rule presets are semantically grouped rules which can be custom or system defined. They can be imported and exported as `JSON` files. For more information, see Rules Management [page 1341].
  - Here is an example of running an analysis by using a **custom preset**:

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
         var oExecutionScope = {
            type: "subtree",
            parentId: "panelId"
         };
         var oCustomPreset = {
            id: "CustomPreset",
            title: "Custom",
            description: "Custom rules",
            selections: [{
               ruleId: "inputNeedsLabel",
               libName: "sap.m"
            }]
         };
         RuleAnalyzer.analyze(oExecutionScope,
oCustomPreset).then(function() {
            var oHistory = RuleAnalyzer.getLastAnalysisHistory();
         ...
         });
      });
```

> **i Note**
>
> You can get the preset definition from an available `JSON` file and pass it as a second parameter of the `analyze` function instead of defining it in your code.

  - Here is an example of running an analysis by using a **system preset**:

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
         var oExecutionScope = {
            type: "subtree",
            parentId: "panelId"
         };
         RuleAnalyzer.analyze(oExecutionScope,
"Accessibility").then(function() {
            var oHistory = RuleAnalyzer.getLastAnalysisHistory();
         ...
         });
      });
```

  - Here is an example of running an analysis with system preset by accessing it through the `sap.ui.support.SystemPresets` enumeration:

```
sap.ui.require(["sap/ui/support/RuleAnalyzer"],
      function (RuleAnalyzer) {
         var oExecutionScope = {
            type: "global"
         };
         RuleAnalyzer.analyze(oExecutionScope,
sap.ui.support.SystemPresets.Accessibility).then(function() {
```

```
            var oHistory = RuleAnalyzer.getLastAnalysisHistory();
        ...
        });
    });
```

View the Results

- `RuleAnalyzer.getAnalysisHistory()` - Returns all the analysis history objects.
- `RuleAnalyzer.getLastAnalysisHistory()` - Returns the last analysis history.
- `RuleAnalyzer.getFormattedAnalysisHistory(sap.ui.support.HistoryFormats)` - Returns the history in the format that has been passed. The default format is string.

Related Information

Integrating the Rules in OPA Tests [page 1349]

API Reference: `sap.ui.support`

API Reference: `sap.ui.support.RuleAnalyzer`

Running the Support Assistant on an Older SAPUI5 Version

In some cases you may want to run the Support Assistant against a different version of SAPUI5. You can do so by following a few steps.

Prerequisites

The minimum SAPUI5 version in which the Support Assistant is available is 1.44.17.

Procedure

1. Open the *Technical Information Dialog* using the shortcut `CTRL` + `SHIFT` + `ALT` + `P`.
2. Choose the settings button for the Support Assistant ().
3. Select a predefined version from the dropdown, or select *Custom Location* to paste a custom URL in the input field.

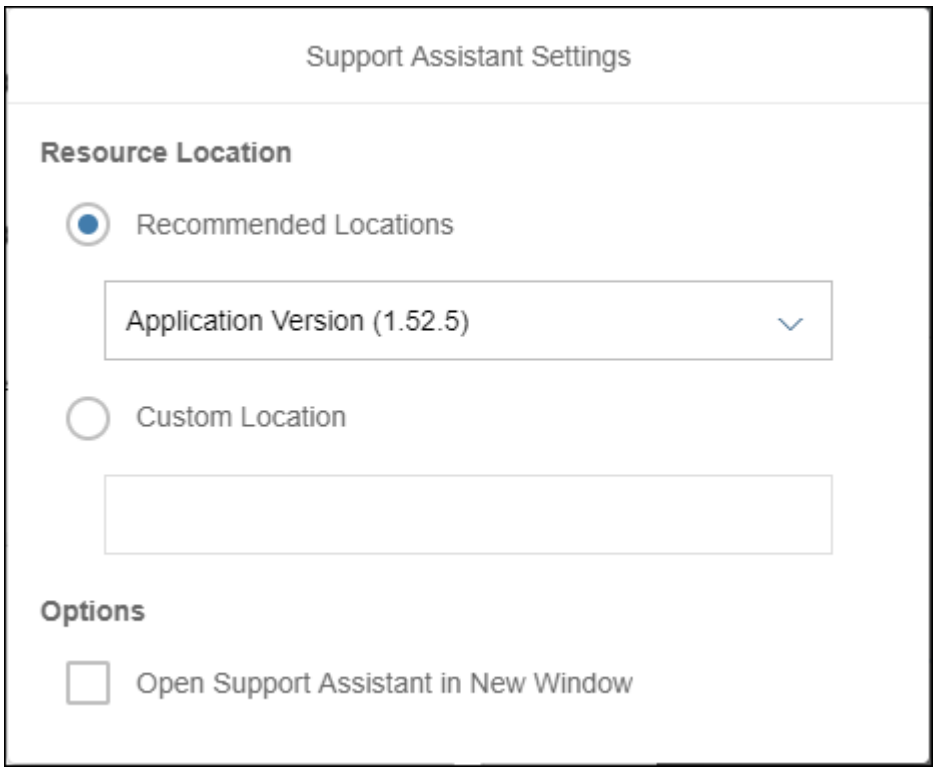


Figure 237: Technical Information Dialog: Support Assistant Settings

> **i Note**
>
> When you choose a custom location, keep in mind that the URL should match the protocol of the application. For example, if the application is HTTP, the location should also be HTTP. If it is HTTPS, the location should be HTTPS. The URL should also end in `sap/ui/support/`.

- Under *Options* you can select if the Support Assistant should be opened in a separate window.

  > **i Note**
  >
  > Additional window popups may be blocked by your browser settings.

4. Select *Activate Support Assistant*.

   Your application will reload and the Support Assistant will start.

   In the following diagram, you can see how the different SAPUI5 versions interact with the Support Assistant.

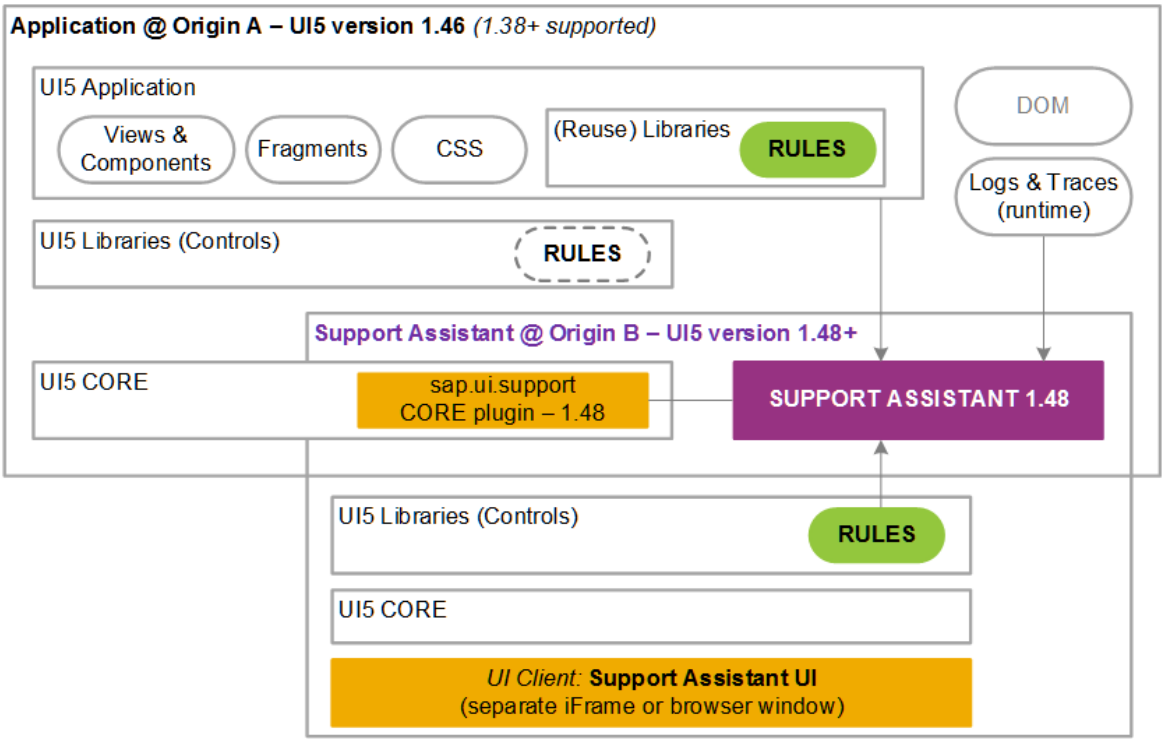


Figure 238: Support Assistant - Multi-Version Support

Results

You are now able to run the Support Assistant on the version that you selected.

> **i** Note
>
> Rules with a higher `minVersion` than the one currently loaded are not checked.

> **→** Remember
>
> These settings are stored in your local storage (if selected) and are reused on consecutive runs.

Related Information

Technical Information Dialog [page 1322]

Troubleshooting the Support Assistant

There are certain scenarios in which the Support Assistant does not behave as expected. In this section you can find tips on how to recognize and resolve some of these cases.

Support Assistant Behavior	Root Cause	Solution
What does it mean when the following errors appear in the browser console? "Access to XMLHttpRequest at <URL> from origin <ORIGIN> has been blocked by the CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource."	This usually happens when serving SAPUI5 from a different origin. The Support Assistant tries to load support rule definitions from there, but cannot load them - a 404 response is sent by the origin server. The issue is usually caused by missing or misconfigured CORS headers on the 404 response specifically. Besides some support rules not being loaded, this should not affect normal operation of Support Assistant.	SAPUI5 library owners are encouraged to provide a `.supportrc` as specified by Support Assistant documentation to avoid causing 404 responses. For more information, see Create a Ruleset for a Library [page 1360] and Content Security Policy [page 1481].
When you choose a custom location, you get an error message: 'The syntax of the location address is incorrect. The correct syntax is ... '.	URL doesn't match the protocol of the application.	If the application is HTTP, the location should also be HTTP. If it is HTTPS, the location should be HTTPS. Also, the URL should end in sap/ui/support/.

Rule Development Guide

The Support Assistant allows you to create custom rules and rulesets and test them on your apps.

Rules can check different aspects of an app depending on the desired result. Each rule belongs to a specific ruleset. The ruleset is a JavaScript file named `library.support.js` that defines a set of rules. It consists of all the rules for an SAPUI5 library. Before creating a rule, you first need to create the ruleset. You can only have one ruleset per library.

The next subsections contain more detailed information on how to create new rules and test them. You can also see examples of best practices and common types of rules.

Create a Ruleset for a Library

The Support Assistant allows you to create your own ruleset.

Choose Your Ruleset Location

A ruleset is a `library.support.js` file that defines an object `name` and `niceName` and returns a set of rules. Before you create a rule, you need to create a ruleset at a specific location. You can follow these steps as an example:

1. Open the respective library project. Let's say that your library is part of OpenUI5, open the project in `openui5([openui5.git]/src/sap.ui.support/src/sap/m)`.
2. Create a JavaScript file with name `library.support.js` in the root folder where `library.js` is placed.
3. Add an extensions property in the `initLibrary()` function of the `library.js` file and there add `sap.ui.support` extension.
   If your library contains public rules it will look like this:

```
…
extensions: {
    //Configuration used for rule loading of Support Assistant
    "sap.ui.support": {
      publicRules: true
    }
…
```

4. Create a folder to hold the rules. For example, if the library name is `sap.m`, the folder structure, if there isn't one already created, should be `src/sap/m/rules`.

Here is an example of folder structure depending on the location of your ruleset:

Table 56: Ruleset Folder Structure

Location	Folder Structure
OpenUI5	[openui5.git]/src/sap.m/src/sap/m/
Reuse library	[project]/src/[library path]

Add `.supportrc` file

Each library should have a `.supportrc` file placed at its root folder. It is a simple JSON file specifying availability of public and internal rules per library. The `.supportrc` file defines whether to load ruleset files of the respective library. This reduces the number of redundant requests to load the respective ruleset library files.

Here is an example of `.supportrc file`:

```
{
  "publicRules": true,
```

```
  "internalRules": true
}
```

> **i Note**
>
> If a `.supportrc` file doesn't exist, a library is considered not to have any rulesets. Therefore, all ruleset developers should add a `.supportrc` file to their libraries root folders. Have in mind that if a ruleset property is missing, its value is considered to be false.

Add `SupportLib`

Once you choose the correct location for the `library.support.js`, the next step is to add the `SupportLib`. It provides a set of constants and enumerations you can use to define the rules inside the `library.support.js`. After creating your rules, return an object holding all of them and a `name/niceName` to specify their library.

Here is an example of how to add and use the `SupportLib`:

```
sap.ui.define(["jquery.sap.global", "sap/ui/support/library"],
  function(jQuery, SupportLib) {
    "use strict";
    var rule1 = {
      ...
      audiences: [SupportLib.Audiences.Control]
      categories: [SupportLib.Categories.Usability]
        ...
      check: function(oIssueManager, ...) {
        ...
        oIssueManager.addIssue({
          severity: SupportLib.Severity.Medium,
          ...
        });
      }
    };
    var rule2 = {…};
    return {
      name: "sap.ui.core",
      niceName: "UI5 Core Library",
      ruleset: [
        rule1,
        rule2
      ]
    };
  }, true);
```

Create Helper Functions (Optional)

When creating a more complex ruleset, you may need to create helper functions. It is recommended that those helper functions are separated into a different file that ends in `.support.js` and is located in the same folder as the ruleset.

> **i Note**
>
> Helper files must be required by a relative path such as `./CoreHelper.js` so that when the Support Assistant is loaded from a different origin, the file will be required from the correct place.

Split `library.support.js` (Optional)

You can also split a `library.support.js` into multiple files.

When creating a ruleset for a bigger library, there may be too many rules and the ruleset will become very big. To avoid this, the rule definitions can be split into multiple files. For example, we might want to split the ruleset of the `sap.m` library by creating a file with rules for each control.

If the `library.support.js` contains rules for `sap.m.Button` and `sap.m.Label`, you can create `Button.support.js` and `Label.support.js` files. After that, the `library.support.js` can require all the rules from those files and create a ruleset.

This is an example of a `library.support.js` before the split:

Example

```
sap.ui.define(["jquery.sap.global", "sap/ui/support/library"],
  function(jQuery, SupportLib) {
    "use strict";
    var buttonRule = {…};
    var labelRule1 = {…};
    Var labelRule2 = {…};
    return {
      name: "sap.ui.core",
      niceName: "UI5 Core Library",
      ruleset: [
        buttonRule,
        labelRule1,
        labelRule2
      ]
    };
  }, true);
```

In this example there are three rules - one for button and two for label. Splitting these rules to different files is done in the following way:

1. Create a `Button.support.js` and `Label.support.js` files.
   `Button.support.js`:

   ```
   sap.ui.define(["jquery.sap.global", "sap/ui/support/library"],
     function(jQuery, SupportLib) {
       "use strict";
       var buttonRule = {…};
       return buttonRule;
     }, true);
   ```

`Label.support.js`:

```
sap.ui.define(["jquery.sap.global", "sap/ui/support/library"],
  function(jQuery, SupportLib) {
    "use strict";
    var labelRule1 = {…};
    var labelRule2 = {…};
    return [labelRule1, labelRule2];
  }, true);
```

> **i Note**
>
> You can return a single rule or an array of rules, as shown in the second example.

2. Require the newly created files in `library.support.js`:

```
sap.ui.define(["jquery.sap.global", "sap/ui/support/library",
    "./Button.support", "./Label.support"
  ],
  function(jQuery, SupportLib, ButtonSupport,
    LabelSupport) {
    "use strict";
    return {
      name: "sap.ui.core",
      niceName: "UI5 Core Library",
      ruleset: [
        ButtonSupport,
        LabelSupport
      ]
    };
  }, true);
```

> **i Note**
>
> The ruleset property is an array which can contain both rule objects and arrays of rules. In the example, `LabelSupport` returns an array of two rules.

Related Information

Create a Rule [page 1363]

Create a Rule

A rule consists of properties that test and advise on how possible issues can be resolved and a check function that tests the application for a specific issue. To create a rule, you need to set the properties and add a check function.

For more information on how to create rules in the user interface, see Rules Management [page 1341].

You can find best practices on how to create rules in Guidelines and Best Practices [page 1369] .

Properties

You need to set the following properties :

Property	Description
ID	The ID of the rule. It must be a valid camelCase string consisting of between 6 and 50 alphabetic characters.
Title	The name of the rule in a readable format. It must be a valid string consisting of between 6 and 200 characters.
Audiences	Describes what audiences the rule is intended for. You can have multiple audiences selected.
Categories	Describes what the rule tests. You can have multiple categories selected.
Min version	The minimum version the rule should be checked at. Possible values are <empty string> and versions like 1.28, 1.44, etc.
Async	Defines if the rule check function will contain asynchronous operations.
Description	A short description of the rule.
Resolution	A short advice on what to do to fix the issue generated by the rule.
Resolution URLs	An array of key/value pairs of texts and URLs providing the links to documentation where the user can find how to fix the issue generated by the rule. You can have multiple resolution URLs. Key is `text` and value is `href`.
Check function	Function that checks the application against the rule. It is described in more detail in the next section.

Check Function

The check function has three main and one optional parameters. The main ones are `oIssueManager`, `oCoreFacade` and `oScope`, and the optional one is `fnResolve`. Here is more information about them:

- `oIssueManager` - allows you to add new issues with the `addIssue()` method. The issue object has the following properties:
  - Severity - the possible values are:
    - `sap.ui.support.Severity.Low`
    - `sap.ui.support.Severity.Medium`
    - `sap.ui.support.Severity.High`
  - Details - free text, may include any type of details related to the issue.
  - Context - an object, which has an `ID` property. The `ID` should belong to the element, which generates the issue.

- `oCoreFacade` - gives you access to the different elements provided by the SAPUI5 core framework:
  - `getMetadata()`
  - `getUIAreas()`
  - `getComponents()`
  - `getModels()`
- `oScope` - retrieves elements in the scope with the following methods:
  - `getElements()` - returns all the elements.
  - `getElementsByClassName(className)` - the className can be, for example, `sap.m.Button`. The function returns all elements of type `sap.m.Button`.
  - `getPublicElements()` - returns all elements that are part of public API aggregations.
  - `getLoggedObjects(type)` - returns all logged objects. The method provides access to the logs and traces in the browser Console. Note that it is possible to enhance the log traces with an extra `fnSupportInfo` object which you can then analyze in the rule check function. The `fnSupportInfo` function is called only when support mode is turned on with the URL parameter `sap-ui-support` set to `true` (`sap-ui-support=true`). For more information, see Rules for the Console Traces and Logging [page 1367].

Prior to version 1.54, `getElements` was not accepting any arguments and was returning all elements registered with the core. Now it accepts one query object parameter. This allows you to select only a specific subset of elements valid for your use case. The three parameters of the method are `type`, `public`, and `cloned`.
Here is an example format:

```
var queryObject = {
  type: "sap.m.Button", // String property specifying the type to select
  public: true, // Boolean property specifying whether only public elements
should be loaded
  cloned: false // Boolean argument specifying if cloned elements are needed
}
```

When the `public` parameter is set to true, the `getElements` function only explores public aggregations. It is useful if, for example, you have a composite control comprising of public and internal subelements, and you only want to check the public ones.
The `cloned` parameter allows you to filter out elements that are clones of list bindings. If you don't want to explore issues associated with multiple cloned elements, for example repeated table cell content, set `cloned: false` and the results will include only one representative instance.
Here is an example of a check function that checks all Input controls which are part of the public aggregation and have no parent set:

```
function(issueManager, oCoreFacade, oScope) {
  var mElements = oScope.getElements({
    type: "sap.m.Input",
    public: true,
    cloned: false
  });
  for (var n in mElements) {
    var oElement = mElements[n];
    if (!oElement.getParent()) {
      issueManager.addIssue({
        severity: sap.ui.support.Severity.Medium,
        details: "The element " + oElement.getId() + " has no parent.",
        context: {
          id: oElement.getId()
        }
      });
```

```
    }
  }
}
```

- `fnResolve` - an optional parameter. It is passed to the check function only when the rule property `async` is set to `true` to allow you, as the rule developer, to resolve an asynchronous operation. The rule times out if it takes more than 10 seconds to resolve.
  Here is an async rule code example:

```
function(issueManager, oCoreFacade, oScope, fnResolve) {
  // Some async operation
  setTimeout(function () {
    …
    fnResolve();
  }, 2000);
}
```

> → Remember
>
> Make sure to call **`issueManager.addIssue()`** in your check function so that issues can be seen in the analysis results.

Related Information

API Reference: `sap.ui.support.ExecutionScope`
Common Rule Patterns [page 1367]

Test a Rule

After you create a rule, you can test it manually on an app.

Manual Testing on an App

Test your rule on locally running apps

If you want to test a Support Assistant rule on a locally running app, there are several steps you need to follow:

1. Start your server with the Support Assistant and support rules code loaded locally.
2. Start the app from your local server and call the Technical Information Dialog (available as of version 1.44).
3. Press *Activate Support Assistant* and it will load from your local server. If the Support Assistant is not loading, click the gear icon  next to the button, select *Recommended locations*, and from the dropdown list choose the current version app.
4. After it loads, if you have correctly entered your data when creating the rule, you should see the title of your rule in the *Available Rulesets* tab.
5. To execute your rule, select it from the list and press *Analyze*. Your check function will be invoked and then you can put a debugger in your code or insert a breakpoint and see if you are getting the right results.

6. After your rule was executed, you will get the analysis of all results and the issues that it generated.

Test your locally modified rule on a remotely running app

To test your locally modified rule on other apps:

1. Start your server with the Support Assistant and support rules code loaded locally.
2. Start the app that you want to test your locally modified rule on and call the Technical Information Dialog (available as of version 1.44).
3. Select the gear icon  next to the *Activate Support Assistant* and select *Custom Location*.
4. In the input field, write the path to the Support Assistant location from your local server. For example, `http://localhost:8080/testsuite/resources/sap/ui/support/`. This way you can load your version of the Support Assistant with the newly created or modified rule. If the remote app is hosted on an HTTPS server, the local server should run on and support HTTPS protocol.
5. Close the popup and select *Activate Support Assistant*. You will be able to load the local version of the Support Assistant with the app that you want to test it on.
6. To execute your rule, select it from the list and choose *Analyze*. Your check function will be invoked and then you can put a debugger in your code or insert a breakpoint and see if you are getting the right results.
7. After your rule was executed, you will get the analysis of all results and the issues that it generated.

Common Rule Patterns

The Support Assistant checks verify different aspects of your web application - from the view/elements structure and control properties to the dynamic, data and event-driven interactions. You can traverse the DOM tree, look at error logs during startup or check the CSS.

Rules for the Rendered HTML and the SAPUI5 Element Tree

With these rules you can check how your application is rendered and how properties of the controls affect the rendering. Here is an example:

```
var mElements = oScope.getElements();
for (var n in mElements) {
  ...
}
```

Rules for the Console Traces and Logging

Rules that analyze the console trace allow you to react to dynamic events while the application is loading. They can help you catch common errors in the binding and bootstrapping of the application. One such rule (*Error Logs*) is already created and catches all the errors from the console.

With version 1.46, the logging API has been enhanced to allow additional objects to be added to logs produced by any module. An additional callback function in a log statement can provide such additional objects, which can be stored with the log entry.

For more information, see the API Reference: `sap/base/log`.

Here is an example of how to log a warning with additional support information:

```
// "Log" required from module "sap/base/Log"
//enable to log additional support information, this is automatically turned on
if the url parameter: sap-ui-support is set to true
Log.logSupportInfo(true)
Log.warning("Log this text", function() {
  //return additional object for further processing in support tooling
  return {
    type: "sap.mylib.supportType", //type can be used to filter the logs in
support tooling
    elementId: "sap.mylib.Class or ID" //can be given to narrow the scope of
support tooling to a certain element ID, normally used as control ID.
    mylogobject: {
      context: "Identify Context"
    }
  }
});
```

Recommendations for Writing CSS Rules

To set custom design preferences, you need to overwrite or manipulate the CSS styling rules. However, this bears a risk because inappropriate changes in the standard styles might provoke rendering or representation issues.

Below, you can find examples of specific Support Assistant rules to check in such situations.

Getting all loaded style sheets

To get all loaded style sheets, you simply need to call the `document.styleSheets` method. This method returns a list of all inline and external CSS rules. You can further filter or search for a specific style depending on your needs.

Getting a list of all custom CSS file paths

By custom CSS files we mean all files and styles that are not included within a standard `library.css` file. Here is an example function that filters all loaded styles and returns only those specific file paths. Once you have a list of all custom CSS files, you can do your further analysis.

```
getExternalStyleSheets: function() {
  return Array.from(document.styleSheets).filter(function(styleSheet) {
    var themeName = sap.ui.getCore().getConfiguration().getTheme(),
      styleSheetEnding = "/themes/" + themeName + "/library.css",
      hasHref = !styleSheet.href || !styleSheet.href.endsWith(styleSheetEnding),
      hasRules = !!styleSheet.rules;
    return hasHref && hasRules;
  });
},
```

Determining if a specific style sheet is in an `inline` or `external` file

If you iterate over `document.styleSheets` elements, you will see that each element has a property `href` containing the full path to the style sheet. If it's empty, it means that the style is `inline`. Here is an example

function that returns the full path to a specific style sheet when loaded externally and `inline` if the applied style is added by a `<style>` tag:

```
getStyleSheetName: function(styleSheet) {
  return styleSheet.href || "Inline";
},
```

Determining if a specific CSS rule is applied on a node

Each style sheet object contains a property called `rules`. This property is a `CSSRuleList` of all `CSSStyleRules` that are inside this style sheet. Each rule has its own property `selectorText` that contains a selector of a rule. To get all nodes that contain that selector, we can use `document.querySelectorAll(rule.selectorText)`. Here is a simple example where we get all rules and find all nodes that contain each rule selector:

```
Array.from(styleSheet.rules).forEach(function(rule) {
  var selector = rule.selectorText,
    matchedNodes = document.querySelectorAll(selector);
});
```

Guidelines and Best Practices

There are some general guidelines for writing succinct and meaningful rules to ensure high quality, consistency and better usability of the reported issues.

Rule Property Values

The following table contains guidelines and examples on how to set the rule property values.

Field	Guidelines / Explanation	Example / Clarification
ID	• CamelCase • Start with small caps	*hardcodedTextValues*
Async	Defines if the rule check function will contain asynchronous operations. It can be true or false. The default value is false.	Make sure you use the resolve function in your rule check function as a 4th parameter.
Audiences	• Control - rule is relevant for control developers • Internal - rule is relevant for an SAP internal developer • Application - rule is relevant for app developers	Choose one.

Field	Guidelines / Explanation	Example / Clarification
Categories	A list of categories checked that show which aspects of the application are affected by the rule. Examples: • Performance • Model Bindings • Memory	Choose one or more, or add your own.
Min version	The minimum SAPUI5 version required so that the rule can produce valid results. The Support Assistant considers applicable rules according to their `minversion` value. If you have rules in your custom library, keep in mind that the rule `minversion` will still be compared against the underlying SAPUI5 version. If you are not aware of the version, you can put in the `minversion` field "*", "-" or whitespace (" ") to make sure these rules are executed.	for example, 1.44 for SAPUI5 version comparison or "*", "-", or whitespace (" ") to avoid version filtering.
Max version	The maximum SAPUI5 version required to run the rule.	(currently not taken into account)
Title	• As short as possible, as descriptive as possible. • (Where applicable) - `<Control Name (no namespace)>:` Description of issue • Sentence case.	*Page: invalid background design property*
Description	• Briefly explain what the rule does/checks. • Ideally one sentence. No period. • Avoid explaining how to fix the issue.	*Dialogs with content should have ariaLabelledBy association set*
Resolution	• Explain how to fix the issue. • Use imperative. • Ideally one sentence. No period. • Could be left out if it is trivial and already explained in the Description.	*Set property upperCase to false or add icons to IconTabFilters*
Details	• Contains technical details on a rule that was triggered for a specific element. • References specific errors and should not be a resolution hint.	*Element{0} has no icon but its parent Element{1} has property upperCase set to true.*

Field	Guidelines / Explanation	Example / Clarification

URLs	• Ideally every rule should point to a topic or document in the DevGuide, API Reference, Samples, or SAP Fiori Design Guidelines.
	• Use the following text values when referring to specific parts of the documentation:
	  ○ Developer Guide - `Documentation: <Title of topic>`
	  ○ API Reference - `API Reference: <Name of control + path to method/property>`
	  ○ SAP Fiori Guidelines - `SAP Fiori Guidelines: <Name of control>`
	  ○ External Link - no description text, just a URL
		• *Documentation: Element Binding*
		• *API Reference: ComboBox #getSelectedItem*
		• *SAP Fiori Design Guidelines: RadioButton*

Check function	Check function code	`function (issueManager, oCoreFacade, oScope) { ... }`

Check Function Guidelines

Here are some general guidelines that you should consider when creating a new rule:

- **Create very specific rules**
  It is important that the rules are as specific as possible. Avoid too generic or unspecific rules that would produce excessive number of issues difficult to digest or follow up. The rules should focus on one issue and provide a resolution for it.
- **Reduce the number of issues generated**
  Do not overload the user with a large number of issues. When appropriate, produce one issue where multiple texts are concatenated with `\n` delimiter.
- **Write clear descriptions and resolutions**
  Use the guidelines in Create a Ruleset for a Library [page 1360]

Test Recorder

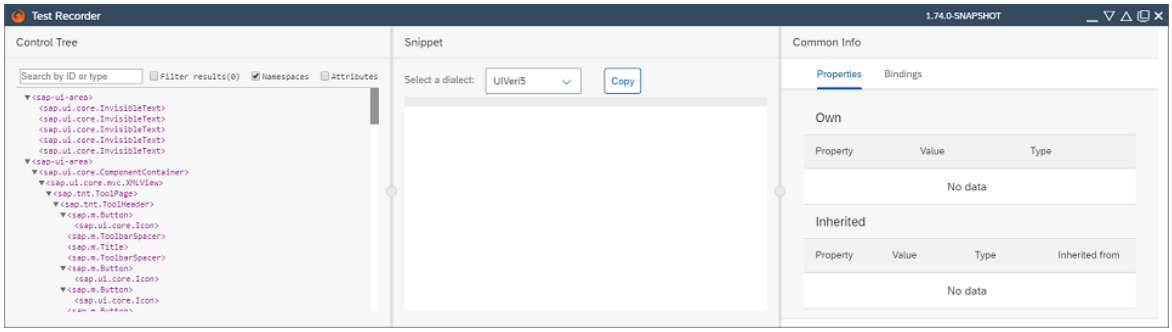
The Test Recorder tool supports app developers who write integration and system tests.

The Test Recorder is part of the SAPUI5 framework and is available in all browsers. As of version 1.74, you can use the tool in any SAPUI5 app to inspect the rendered user interface (UI), view the control properties, and gain hints on writing tests. The Test Recorder is aligned with the two official SAPUI5 testing tools – OPA5 and UIVeri5.

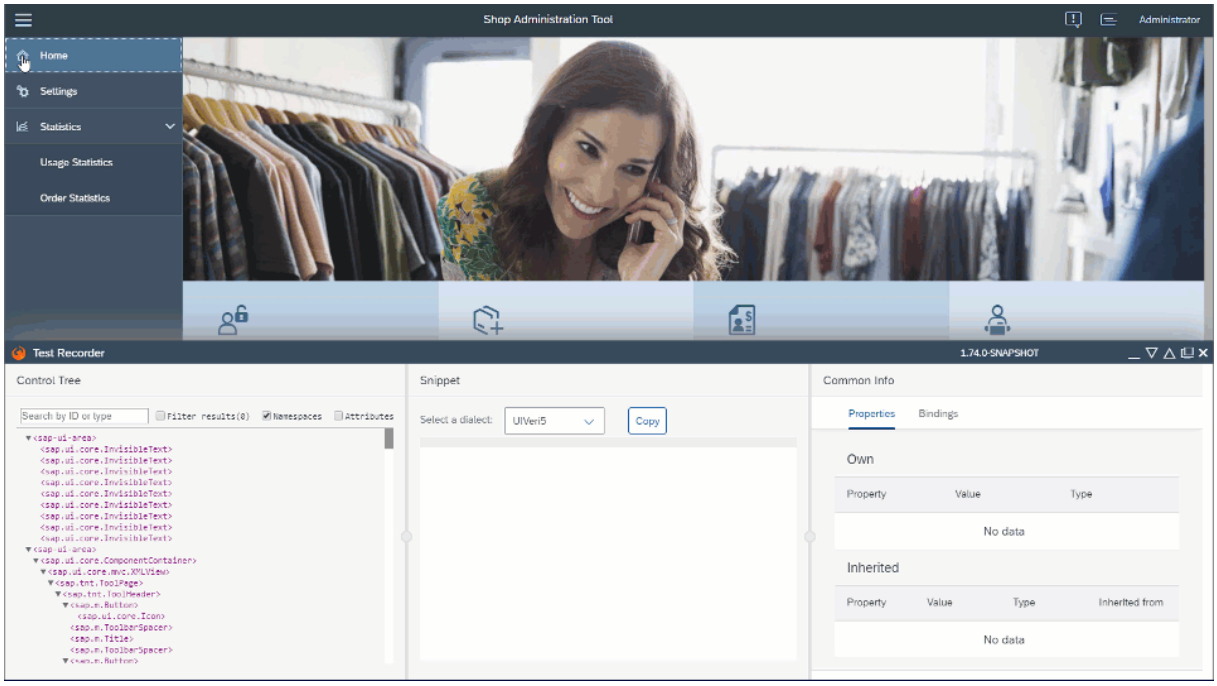
Getting Started

There are two ways to open the Test Recorder:

- In a separate window: Press CTRL + SHIFT + ALT + T
- In an IFrame: Press CTRL + SHIFT + ALT + P to display the *Technical Information Dialog* and then choose *Activate Test Recorder*



The main sections of the tool are *Control Tree*, *Snippet*, and *Common Info*.



From the navigation actions at the top bar of the Test Recorder, you can minimize, resize, open it in a new window, or close the tool.

Control Tree and Common Information

In the *Control Tree* section, you can see the DOM structure of the current app page. When navigating to another page or view, the tree is automatically updated.

You can display more information in the *Control Tree* by selecting the *Namespaces* and *Attributes* checkboxes. Entering text in the *Search* field highlights all elements that (partially) match by namespace, control name, or attribute values.

> **i Note**
>
> Elements in the *Control Tree* get highlighted if there's a match by namespace or attribute value even when the *Namespaces* and *Attributes* checkboxes aren't selected and the information isn't visible.

There are three general types of testing-relevant information that you can gather for any control:

- Position in the rendered control tree, type and ID – displayed in the *Control Tree* section. To see the control type, select the *Namespace* checkbox. To see the ID, select the *Attributes* checkbox.
- Properties (either own or inherited) – displayed in the *Common Info* section on the *Properties* tab.
- Bindings (binding context, properties, and aggregations) – displayed in the *Common Info* section on the *Bindings* tab.

> **i Note**
>
> - If an ID is not stable (because it was generated automatically), it's not suitable for tests. Unstable IDs start with a double underscore.
> - Many controls can have the same property or binding values. Therefore, when you use them in a control locator, the test finds multiple controls. This is a valid scenario, but it's always more reliable to locate only one control with a highly specific locator.

Actions

You can perform the following actions on controls, either from the *Control Tree* or from the rendered UI of the app:

- **Highlight:** Generates a code snippet for finding the control, which can be used to assert the control state.
- **Press:** Generates a code snippet for pressing on the control.
- **Enter Text:** Generates a code snippet for entering text into the control.

To perform an action from the *Control Tree*, right-click and choose *Press* or *Enter Text* in the context menu. If you want to *highlight* the respective control in the rendered UI, simply select the desired element in the tree.

To perform an action from the app page, right-click on any control and select the desired action from the context menu (the respective control is highlighted in the *Control Tree*).

> **i Note**
>
> A *Press* or *Enter Text* action snippet is generated irrespective of whether the control accepts such interactions. Keep in mind that such a snippet is not suitable for tests.

Snippets

The code snippets generated by the Test Recorder usually contain a function invocation that locates one control on the app page. The function receives one argument – a control locator. The control location is a JSON object containing a specific combination of conditions and matchers.

The code snippet can be directly copied and pasted into your test code and is already aligned with the supported tools for testing – *OPA5* and *UIVeri5*. To choose the tool for which to generate a code snippet, select an option from the *Dialect* dropdown menu. The *raw selector* option gives you just the control locator with no function invocations.

Related Information

Integration Testing with One Page Acceptance Tests (OPA5) [page 1182]
Stable IDs: All You Need to Know [page 1442]
API Reference: `sap.ui.test.Opa5`
GitHub: `ui5-ui5veri5`

UI5 Inspector

The UI5 Inspector is an open source Chrome DevTools extension that helps app developers to inspect, analyze, and support SAPUI5-based apps. It is supported for apps based on SAPUI5 version 1.28 and higher.

Check out the UI5 Inspector video on YouTube for a quick overview of the most common use cases.

Key features:

- Inspect SAPUI5 controls and review their properties, bindings, and data model
- Modify control properties on the fly and see how this affects rendering and behavior
- Find relevant framework information for your SAPUI5 app

How to get it?

You can download the UI5 Inspector as a standard extension from the Chrome Web Store at https://chrome.google.com/webstore/detail/ui5-inspector/bebecogbafbighhaildooiibipcnbngo?hl=en .

Features

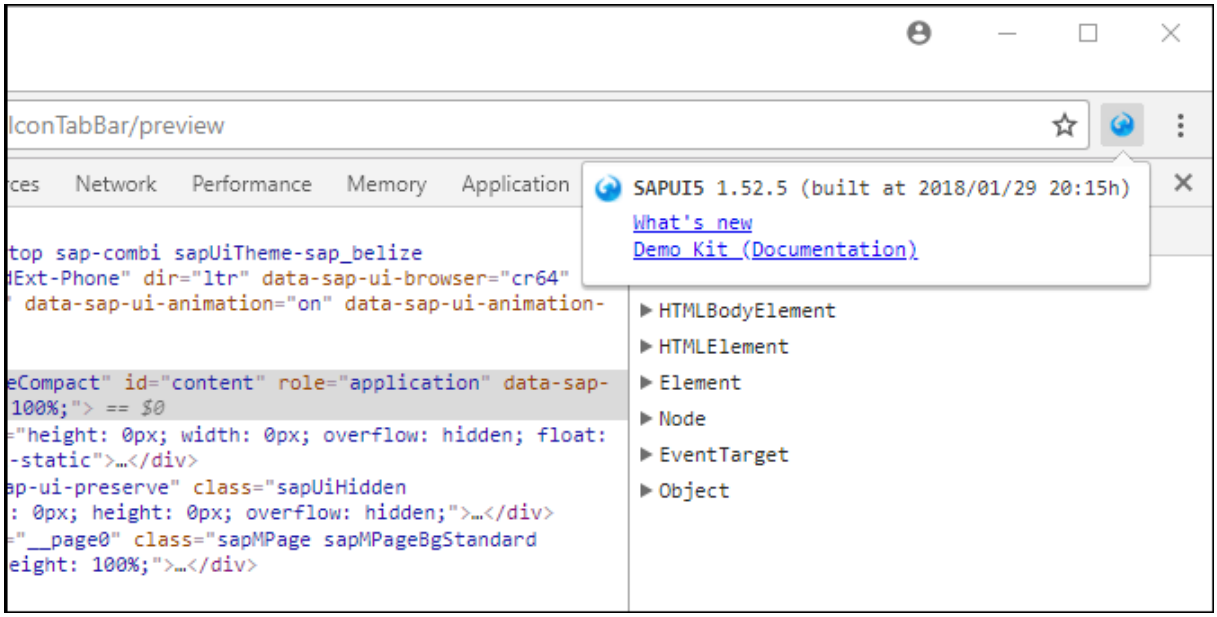
Once installed, the UI5 Inspector is available in Chrome DevTools (by choosing `F12`). It becomes active when an SAPUI5 app is loaded.

Browser Action

Clicking the browser action icon in the address bar provides you with:

- Information on the used SAPUI5 version
- Links to the *What's New in SAPUI5* section and the SAPUI5 documentation

> **i Note**
>
> Your version of SAPUI5 may be older than the latest and the features described in the documentation may not be available for you.

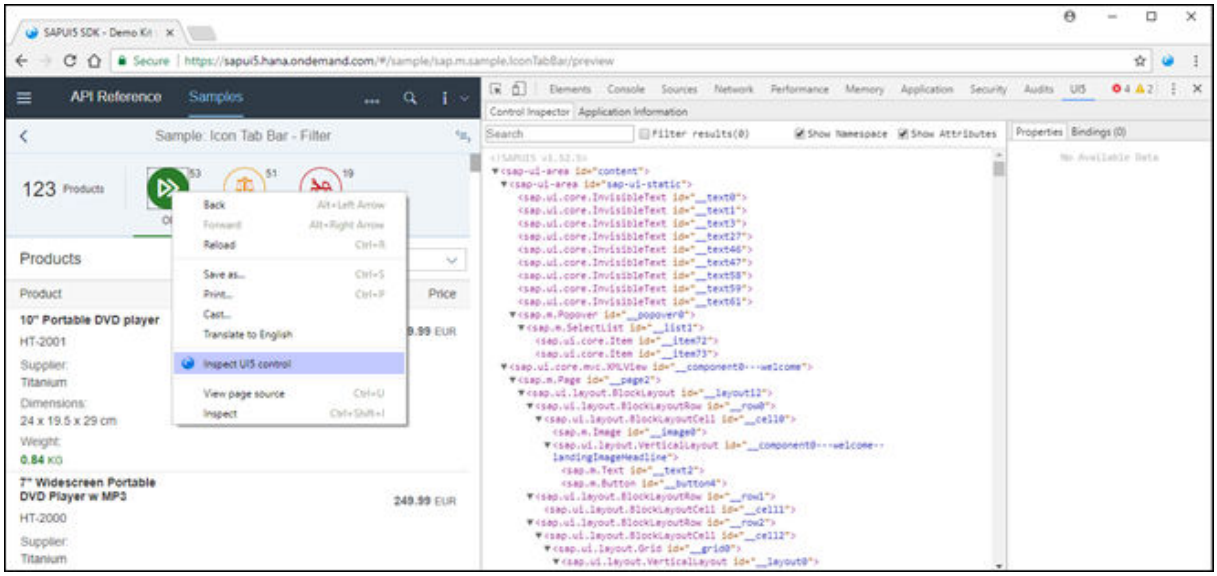


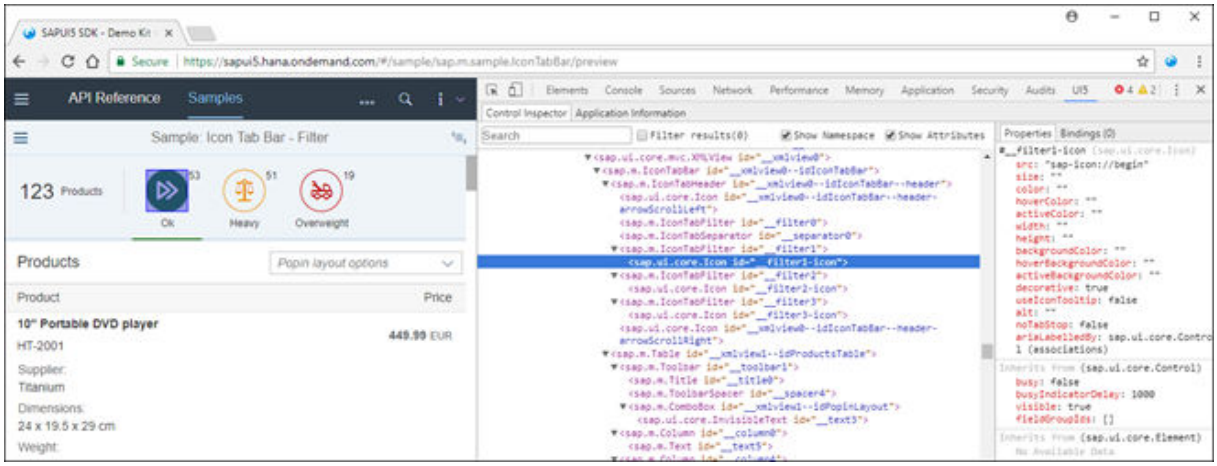
Control Inspector

This tab shows the structure and nesting of the SAPUI5 controls. You can search and filter for specific controls. You have the options to show/hide the control's namespace and attributes in the tree.

Hovering over a specific branch of the tree highlights the corresponding control in the app.

Additionally, you can right click on any SAPUI5 control from the app and select *Inspect UI5 control*. This automatically selects the control in the tree and you can review its properties directly.





Properties

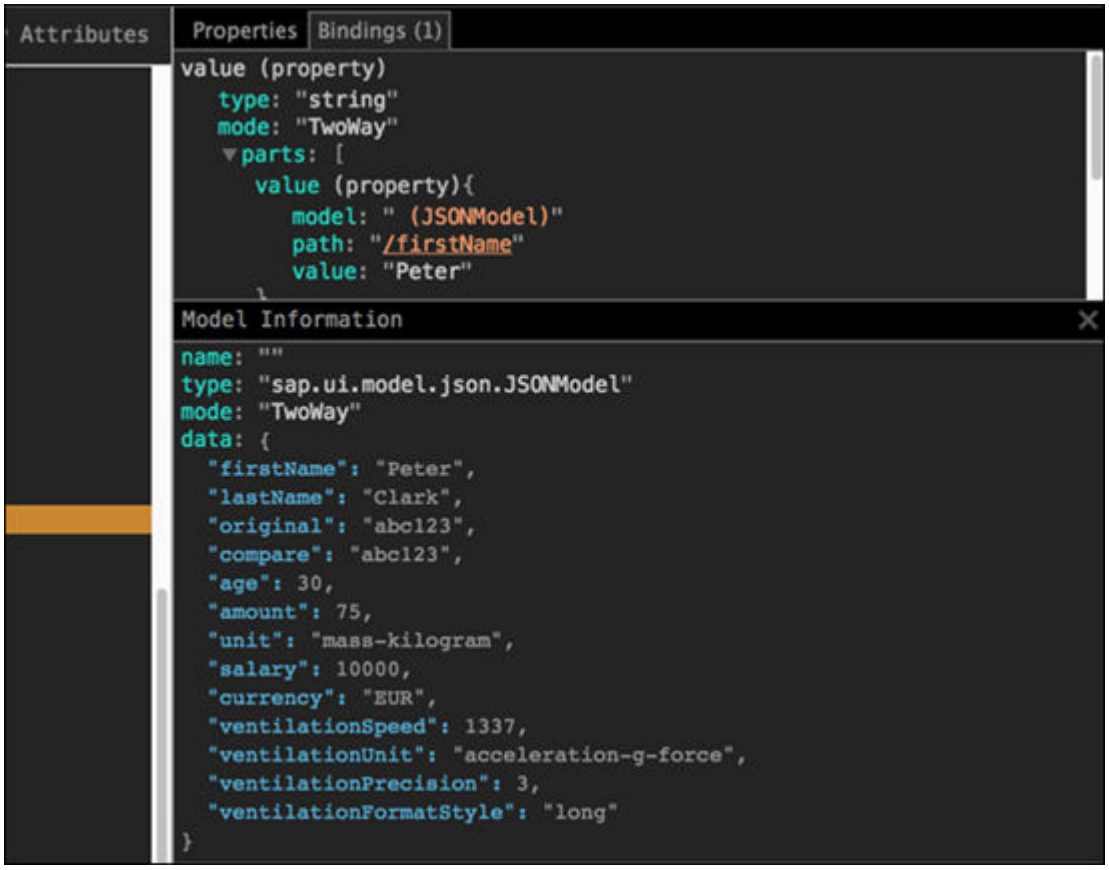
In this tab, you can see the properties that have been set for the selected control. Additionally, the inherited properties are also listed.

You can change the values of the properties. The changes will be validated against the framework and rendered on the fly.

> **i Note**
>
> Errors from incorrectly set values are displayed in the Chrome browser's console.

Bindings

In this tab, you can see the bindings for a specific control. The number of bindings is displayed in parentheses in the tab title. Selecting the tab gives you more information about individual models, paths and values. The model property holds a link to the corresponding binding file. Clicking on the link opens the *Model Information* section with details about all values.



> → Tip
>
> UI5 Inspector supports a dark theme in case you are using the Google Developer Tools in dark mode.

Application Information

In this tab, you can see overall information for your app – for example, the exact SAPUI5 version you are running, the version of your browser, and the app URL. The information on the loaded libraries and modules is collapsed by default.

Performance Measurement Using sap/ui/performance/Measurement Module

You can use `sap/ui/performance/Measurement` to measure the performance of your JavaScript code.

For each measurement, the result is a time and a duration. The time are the milliseconds (ms) from starting the measurement till its end. The duration is the effective milliseconds, pause phases are not counted here.

You can measure the categories that are used by the SAPUI5 core classes as listed in the following table:

Category	Description
`javascript` (default)	Default measurement category if no category is provided
`require`	Identifies the duration of `jQuery.sap.require` for lazy loading of JavaScipt classes including the loading and parsing times for a class
`xmlhttpreques t`	Identifies the duration of an `jQuery.ajax` call
`render`	Used for all rendering-related measurements that trigger core rendering of controls within the `RenderManager` class With the `render` category there comes an additional set of categories to distinguish between different phases of rendering
`control`	Identifies the duration for HTML rendering provided with the `ControlRender.render` method
`after`	Identifies the duration for calls on the control's `onAfterRendering` method
`preserve`	Identifies the duration needed to find out whether rendering can be preserved

Procedure

1. Activate performance measurement

By default, `Measurement` is disabled to avoid unnecessary code execution during runtime. Therefore, you first have to activate the measurement using one of the following options:

- Use URL Parameter `sap-ui-measure=true` to measure an initial request.
- Use the diagnostics window in your app with `Ctrl` + `Alt` + `Shift` + `S`. In this window, you can also see a visualization of the results.
- Activate measurement in the browser's console by calling `Measurement.setActive(true)`
- Create a measurement in your code with:

```
// "Log" required from module "sap/base/Log"
// "Measurement" required from module "sap/ui/performance/Measurement"
Measurement.setActive(true);
Measurement.start("myId","Measurement of myId");
Log.info("foo");
Measurement.end("myId");
```

You can use methods as listed in the following table:

Action	Method
Start measurement	`Measurement.start(sId,sInfo, [categories])`

Action	Method
End measurement	`Measurement.end(sId)`
Pause	`Measurement.pause(sId)`
Resume	`Measurement.resume(sId)`

To activate measurement for certain categories only, you have the following options:

- Provide a URL Parameter with categories `sap-ui-measure=category1,category2`
- Add the category as parameter to the call of the `Measurement.setActive` as in the following example:

```
// Measure only "require" category
// "Measurement" required from module "sap/ui/performance/Measurement"
Measurement.setActive(true,"require")
```

To assign a measurement to a specific category, just add the category to the `start` function.

```
// "Measurement" required from module "sap/ui/performance/Measurement"
Measurement.start("myId","Measurement of myId", **\["foo"\]**);
```

> **i** Note
>
> If you also use the `start` or `average` method, make sure that the same categories are passed on, otherwise no measurement is started.

2. Retrieve the results

You can view the results in the *Performance* section of the diagnostics window (`Ctrl` + `Alt` + `Shift` + `S`). Here, you can also refresh the result list, if the performance measurement is still running.

You can retrieve the results via API with one of the following commands:

Command	Returns
`Measurement.getAllMeasurements()`	Array of all measures (running and completed)
`Measurement.getAllMeasurements(true)`	Array of completed measures
`Measurement.getAllMeasurements(false)`	Array of running (not completed) measures
`Measurement.getMeasurement(string)`	One specific measurement by ID
`Measurement.filterMeasurements(func)`	Array of all measures based on the result of the filter function (running and completed)
`Measurement.filterMeasurements(func, true)`	Array of completed measures based on the result of the filter function
`Measurement.filterMeasurements(func, false)`	Array of running measures based on the result of the filter function

In Google Chrome, for example, you can also display the results in a table in the console by using:

```
console.table(Measurement.getAllMeasurements(true)) //table with completed
measurements
```

3. Interpret the results

Each entry in the resulting array provides an object of the following structure:

- `id`: `string`
  The unique ID of the measurement as provided in the `start` or `average` method
- `info`: `string`
  Additional information as provided in the `start` or `average` method
- `duration`: `float`
  Duration or average duration in ms
- `count`: `int`
  Number of calls counted of an average
- `average`: `boolean`
  Indicates whether the result is an average
- `categories`: `string[]`
  Categories as provided in the `start` or `average` method

4. Clear results

To clear all measurements call the `Measurement.clear` method.

Specific Use Cases

Averages

For repeatedly occurring operations, you can calculate an average duration with the `Measurement.average` method.

```
// "Log" required from module "sap/base/Log"
// "Measurement" required from module "sap/ui/performance/Measurement"
Measurement.setActive(true);
for (var i=0;i<1000;i++) {
    Measurement.average("myId","Average of myId");
    Log.info("Foo " + i);
    Measurement.end("myId");
}
```

Based on the ID, all measurement calls are counted and the average duration is calculated and provided in the result, together with the complete duration and the number of calls:

```
// "Log" required from module "sap/base/Log"
// "Measurement" required from module "sap/ui/performance/Measurement"
Log.info("1000 calls: " + Measurement.getMeasurement("myId").count === 1000); //
true
Log.info("Average time: " + Measurement.getMeasurement("myId").duration);
```

Measurement of Object Methods

You can register an average measurement without changing the original source code. For this, you use the following APIs:

- `Measurement.registerMethod`
- `Measurement.unregisterMethod`
- `Measurement.unregisterAllMethods`

To measure the average time a method of an instance, you can use the following example code:

```
// "Button" required from module "sap/m/Button"
// "Measurement" required from module "sap/ui/performance/Measurement"
var oButton = new Button();
Measurement.registerMethod("oButton.setText", oButton, "setText",
["instance"]); //register to oButton instance on method setText
Measurement.setActive(true,["instance"]); //measure only category "instance"
for (var i=0;i<1000;i++) {
    oButton.setText("MyButton" + i);
}

Measurement.unregisterMethod(oButton, "setText");
// or Measurement.unregisterAllMethods();
Measurement.getAllMeasurements();
```

To measure the average time a method of a class, you can use the following example code:

```
// "Button" required from module "sap/m/Button"
// "Measurement" required from module "sap/ui/performance/Measurement"
Measurement.registerMethod("oButton.setText", Button.prototype, "setText",
["class"]); //register to Button class on method setText
Measurement.setActive(true,["class"]); //measure only category "class"
for (var i=0;i<1000;i++) {
    var oButton = new Button();
    oButton.setText("MyButton" + i);
}

Measurement.unregisterMethod(oButton, "setText");
//or Measurement.unregisterAllMethods();

Measurement.getAllMeasurements();
```

Filtering

You can also use the categories listed above as filters for the result list or to define measurements for one or more specific categories with the `filterMeasurements` method.

To filter the categories that are measured, you use, for example:

```
// Filter for category1
Measurement.filterMeasurements(function(oMeasurement) {
    return oMeasurement.categories.indexOf("category1") > -1;
});
```

To filter the results, you can use a code like this:

```
// Filter for duration > 500ms
Measurement.filterMeasurements(function(oMeasurement) {
    return oMeasurement.duration > 500;
});
```

Related Information

API Reference: `jQuery.sap.measure.html`
Performance: Speed Up Your App [page 1434]
Performance Issues [page 1467]

Interaction Tracking for Performance Measurement

You can identify performance issues in your application by tracking the interaction that is performed on the UI.

Interaction in this context means a closed step in a sequence of actions that a user performs on the UI, for example, everything that happens between two clicks on two different buttons.

To **start** interaction tracking, use `Interaction.setActive(true)` from module `sap/ui/performance/trace/Interaction`. To map the interaction data to the data of `sap/ui/performance/Measurement`, you have to explicitly set `sap-ui-measure=true`.

To **retrieve** the result of the interaction measurement, use `Interaction.getAll()` from module `sap/ui/performance/trace/Interaction`. This returns an array of all interactions that occurred and their measurement.

You can use `Interaction.filter` from module `sap/ui/performance/trace/Interaction` to filter the interaction measurements according to a filter function (`fnFilter`).

Example of an Interaction Measurement

```
InteractionMeasurement = {
    event: "click",             // event which triggered interaction
    trigger: "Button1",         // control which triggered interaction
    component: "my.Component",  // component or app identifier
    start : 0,                  // interaction start
    end: 0,                     // interaction end
    navigation: 0,              // sum over all navigation times on the critical
path
    roundtrip: 0,               // time from first request sent to last received
response end
    processing: 0,              // client processing time
    duration: 0,                // interaction duration
    requests: [],               // Performance API requests during interaction
    measurements: [],           // sap/ui/performance/Measurement measurements
    sapStatistics: [],          // SAP Statistics for OData
    requestTime: 0,             // sum over all requests in the interaction
    networkTime: 0,             // request time minus server time from the header
    bytesSent: 0,               // sum over all requests bytes
    bytesReceived: 0,           // sum over all response bytes
    requestCompression: false,  // true if all responses have been sent gzipped
    busyIndication: 0           // summed GlobalBusyIndicator duration during
this interaction
}
```

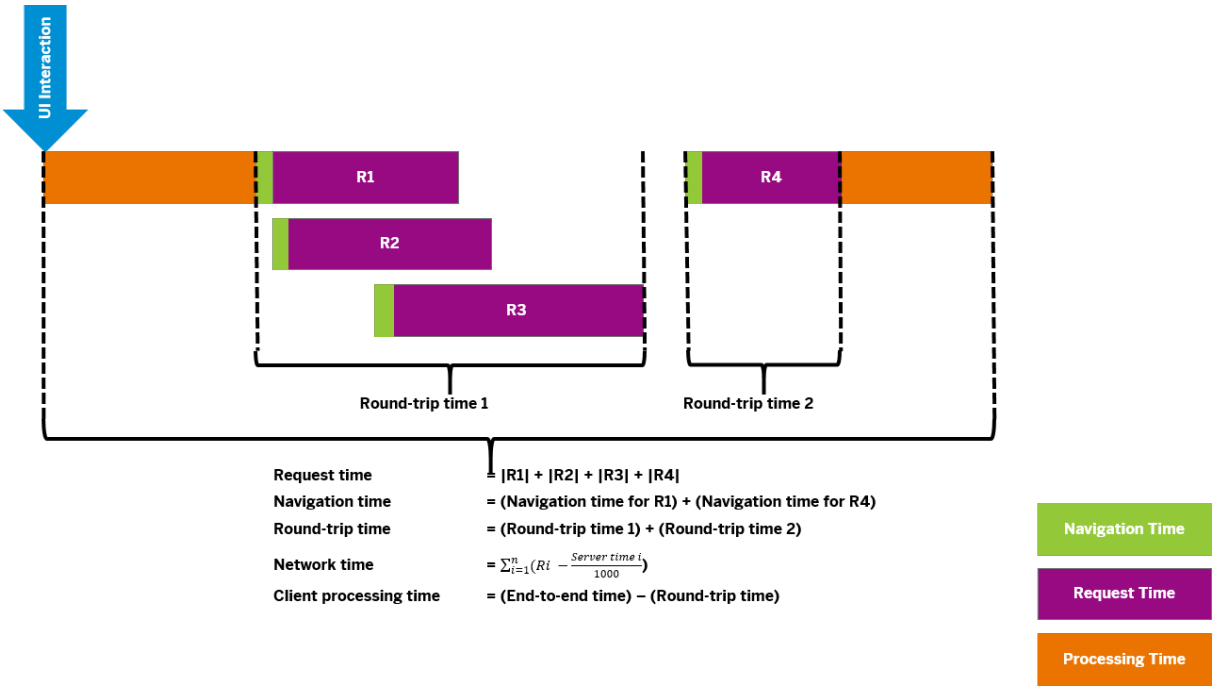
Properties of Interaction Measurements

Table 57: Properties of Interaction Measurements

Property	Type	Description
`event`	`String`	Event type which triggered the interaction. Allowed types are: • `mousedown` • `mouseup` • `click` • `keydown` • `keyup` • `keypress` • `touchstart` • `touchend` • `tap` • `mousewheel` • `scroll`
`trigger`	`String`	ID of the element that triggered the action
`component`	`String`	ID of the app or name of the Component that contains the triggering element
`start`	`Number`	Time stamp when interaction was started (in ms)
`end`	`Number`	Time stamp when interaction has been finalized (in ms) **i Note** This is not always the start time plus the duration. The duration is determined depending on the heuristic determination of the processing time.
`navigation`	`Number`	Navigation time for all requests, calculated as difference from `startTime` to `connectEnd` of a `PerformanceTiming` (in ms) Requests that are started while another request is already in progress are ignored (see figure below).
`roundtrip`	`Number`	Roundtrip time for a request, calculated as difference from `requestStart` to `responseEnd` of a `PerformanceTiming` (in ms)
`processing`	`Number`	JavaScript processing time of an interaction. This is the time consumed when no requests are active. Although we also have JavaScript being processed while asynchronous requests are active, we only consider those to be relevant (in ms)
`duration`	`Number`	If a processing time could be determined duration is navigation plus roundtrip plus processing time. Otherwise it is navigation time plus roundtrip time, or end time minutes start time if network requests last longer than the actual interaction (in ms)

Property	Type	Description
`requests`	`PerformanceTiming[]`	All requests that occurred during the interaction, taken from the `NavigationTiming` API
`measurements`	`Measurement[]`	Performance measurements (see Performance Measurement Using sap/ui/performance/Measurement Module [page 1377])
`sapStatistics`	`Object[]`	Map of request URL to corresponding `sap-statistics` header as String (format: `{ url: "https://somehost.com/sap/data...", statistics: "total=167,fw=167,app=0,gwtotal=167,gwhub=160,gwrfcoh=0,gwbe=7,gwapp=0,gwnongw=0" })`
`requestTime`	`Number`	Sum over all request durations of this interaction, from `startTime` to `responseEnd` (in ms)
`networkTime`	`Number`	Average latency of the requests that occurred during the interaction, calculated using the `sap-perf-fesrec` header that is sent (if available) by the back end with each response (in ms)
`bytesSent`	`Number`	Sum over all bytes sent with requests (content plus headers)
`bytesReceived`	`Number`	Sum over all bytes received with responses (content plus headers)
`requestCompression`	`Boolean`	Indicates if all requests during an interaction have been received in GNU zip format ("gzipped")
`busyDuration`	`Number`	Time how long a `GlobalBusyIndicator` was rendered and hence blocking the UI during an interaction

Calculation of Times



Related Information

API Reference: `jQuery.sap.measure`

Performance Measurement Using sap/ui/performance/Measurement Module [page 1377]

`NavigationTiming` API on `https://developer.mozilla.org`

Navigation Timing on `https://www.w3.org/`

First-Aid Kit

This section contains the most common issues that you might face when developing SAPUI5 apps and how to solve them.

An Empty Page Comes Up

You find yourself in one of these situations:

- The browser shows an empty page: there's no content and no error message is displayed
- An `Uncaught Error` message is shown in the developer console

Preview

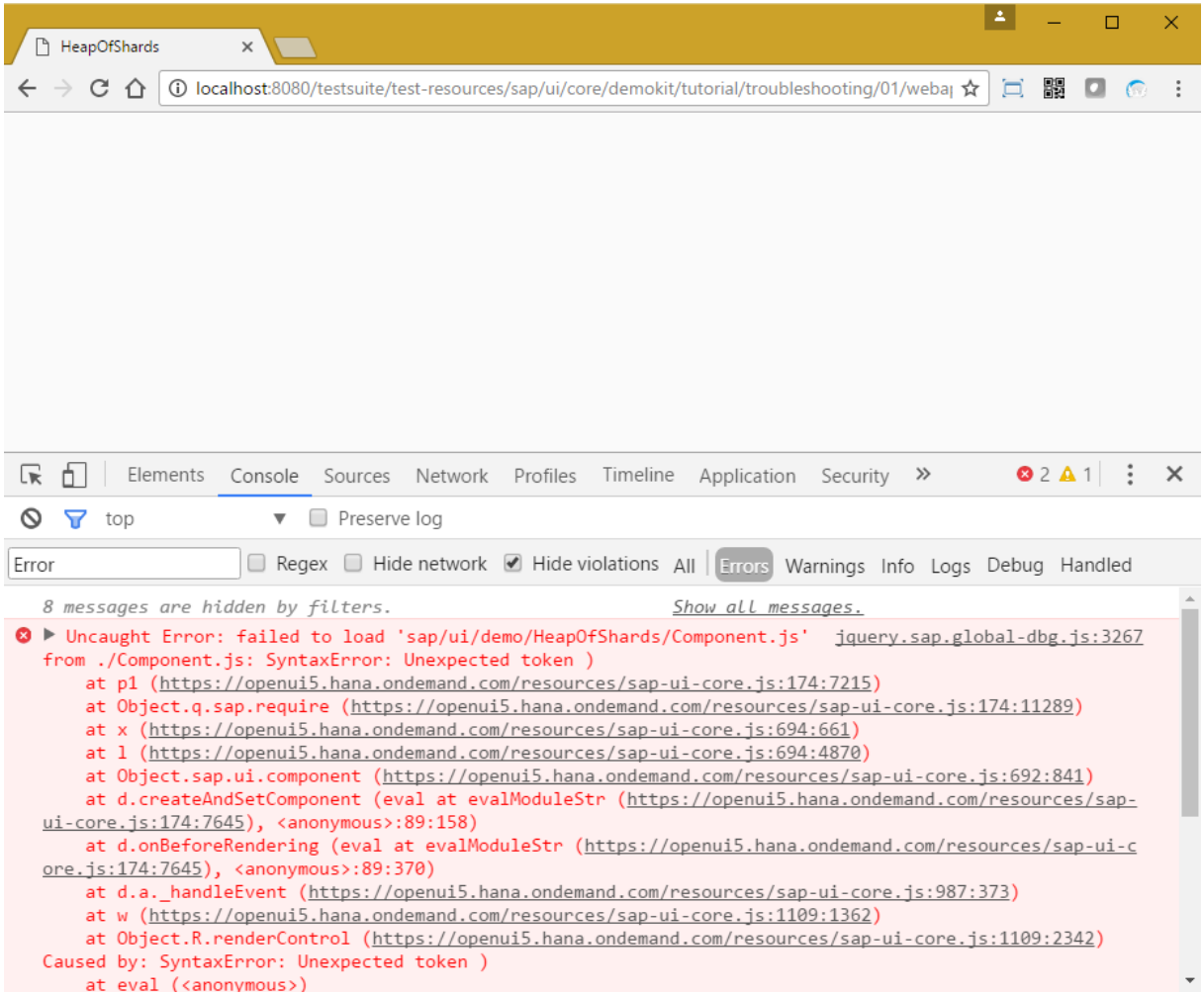


Figure 239: The browser displays an empty page and an `Uncaught Error` is issued in the console

Root Cause

This can happen for one of the following reasons:

- A critical reference error is prohibiting the app from starting.
- A syntax error is stopping the execution of your application code.
- A parsing error has occurred in an XML view.
- The tag of the control is written with lowercase letters.
- The root view is missing a root control.

Resolution

Console shows `"ReferenceError: sap is not defined"`

Have a look at the `resource` path in the bootstrap of the HTML page you are trying to open. The path to the file `sap-ui-core.js` is probably incorrect and needs to point to the path where the SAPUI5 resources are located (typically globally under `/resources` or locally under `resources`).

If you are running the code in SAP Web IDE, you have to configure the `neo-app.json` project descriptor (see Create a neo-app.json Project Configuration File [page 46]).

Other development environments might need the resources to be copied to the server and referenced relatively to the app (see Standard Variant for Bootstrapping [page 694]).

Alternatively, you can use the CDN version (see Variant for Bootstrapping from Content Delivery Network [page 696]).

Console shows `SyntaxError: <error details>`

A JavaScript error in the application code throws an exception and stops all subsequent execution. Take a look at the error details: In most cases, the root cause is mentioned in the first line of the error message.

The stack trace can provide more context on the execution scope. Analyze it from thoroughly to find a line referencing your application code and start debugging there.

Console shows `Error: Invalid XML`

If the XML view to be displayed cannot be parsed, SAPUI5 stops the execution and throws a parse error. Check the XML view for namespace issues, typos, and missing closing tags. Do a schema validation with an XML validator tool.

Console shows `Uncaught Error: failed to load 'sap/m/xxxxx.js'`

During the development on Microsoft Windows, your app works fine, but a soon as you deploy it on a Linux system, only an empty page comes up.

This could happen if you wrote the tag of the control with lowercase letters, because Linux systems use case-sensitive file names.

Correct Example	Incorrect Example
`<Button text="Click me" />`	`<button text="Click me" />` Error message: `Uncaught Error: failed to load 'sap/m/button.js'`

> → Tip
>
> Control tags always start with capital letters after the namespace like `<Button>`, `<l:FixFlex>`, `<f:SimpleForm>`.
>
> Aggregations always start with lowercase letters like `<content>`, `<l:fixContent>`, `<f:content>`

Console shows no error

Your root view is missing a root control. In the context of SAPUI5, `sap.m.App` or `sap.m.SplitApp` function as root controls. Please check your root view (for example `App.view.xml`) and add the missing root control.

Content or Control Is Not Visible

You find yourself in the situation that a control or the content of a control is not visible, but you don't see an error message in the console.

Root Cause

This can happen for one of the following reasons:

- The element is not properly bound
- The `visible` property is set to `false`
- The `height` or `width` dimension is set to `0`

Resolution

First, you should check if your control was rendered properly by using the developer tool of your browser to check the DOM element. For information about how to use your browser tools, see the documentation of you browser or check our Troubleshooting Tutorial Step 1: Browser Developer Tools [page 196].

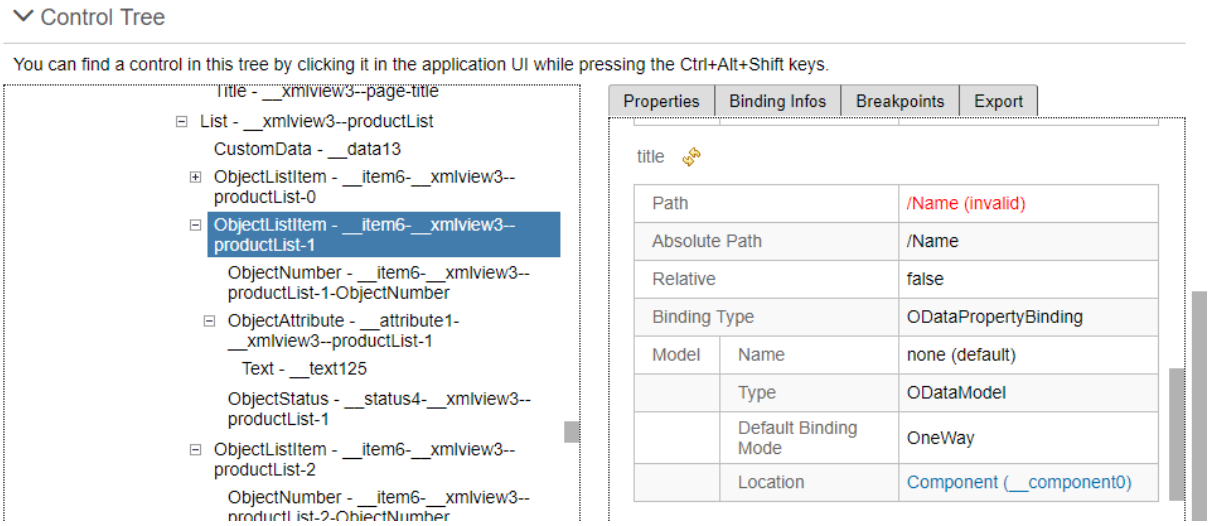
Wrong binding

If you bound your control to a source, for example, an image control, the binding may not be resolved properly. This can be caused by minor mistakes such as typos. We recommend using *Diagnostics* to debug your bindings. For more information, see Diagnostics [page 1326].

In the *Diagnostics* window, you can check whether you used a relative binding instead of an absolute one or vice versa.

If you, for example, use a `List` control, you bind the list itself to an absolute path like `items="{/Products}"` whereas the aggregations are bound to a relative path like `title="{Name}"`. The actual path of the `title` property is now `{/Products/*Product_Index*/Name}`.

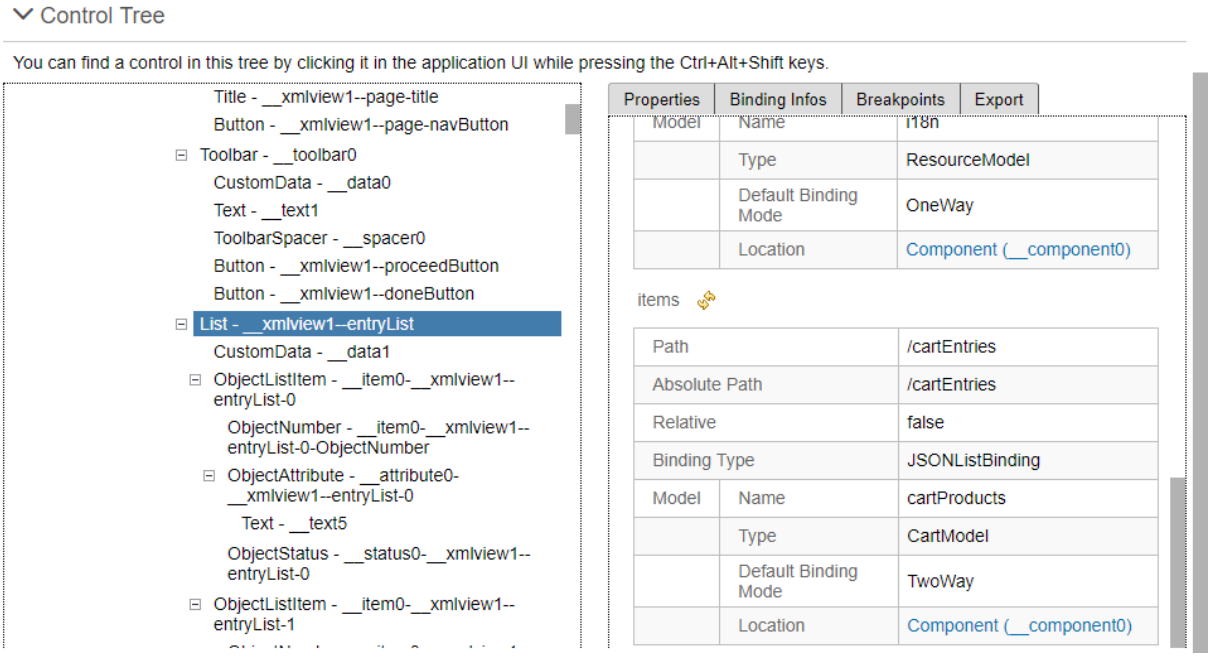
If you used an absolute binding path like `title="{/Name}` for an aggregation instead of a relative one, the result in the window would look like this:



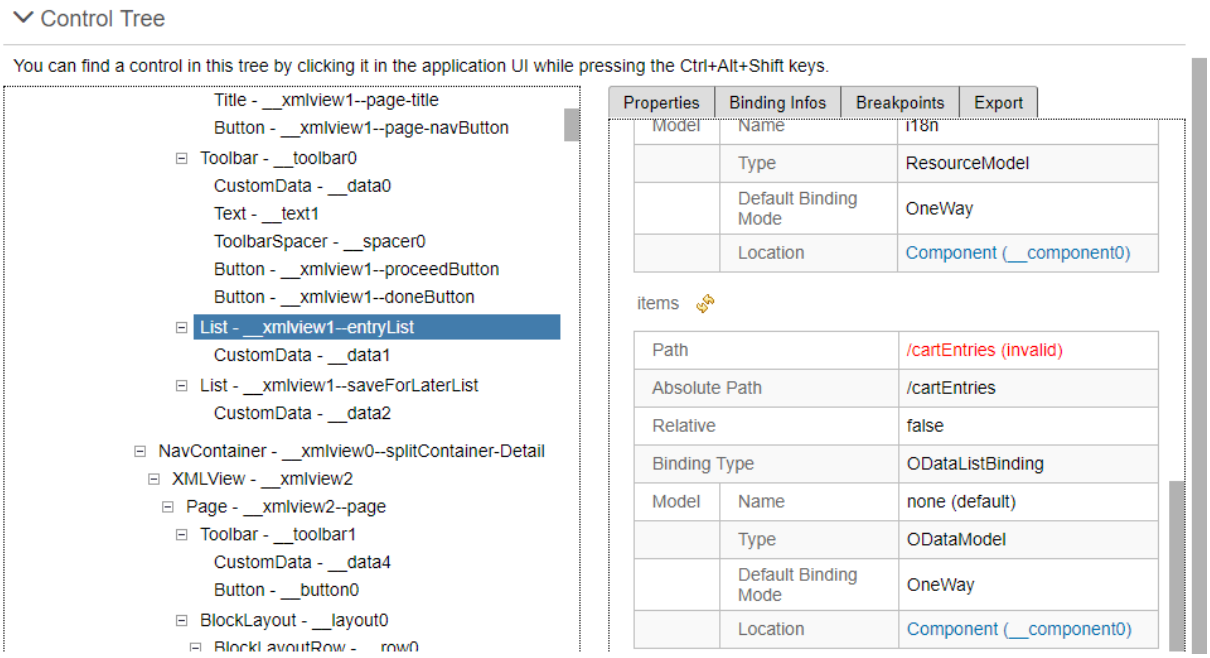
Another common error related to binding is to refer to the default model instead of referring to a specific model. This happens, for examples, if you forgot to add the model name to the binding declaration.

For example, you have two models in your application: the default model, which has no name and another model named `cartProducts`. To bind to the `cartProducts` model you have to write the model name explicitly like `items="{cartProducts>/cartEntries}"`.

If you used the binding correctly *Diagnostics* displays the following:



If the model name is missing, you see the following:



`visible` property set to `false`

If you set the `visible` property of a control to `false`, it will not be rendered at all.

Nested controls inherit the value of the `visible` property from their parents. Therefore, if the control that you are missing is nested in a parent control that is set to invisible, the nested control will also not be rendered.

You can fix this by setting the `visible` property of the parent control to `true` or by moving your missing control in the XML view so that it is not longer nested inside an invisible control.

Dimensions set to `0`

Most controls have the properties `width` and `height`. If one of them is explicitly set to `0` some controls may not be displayed at all. Similar to the `visible` property, the value of `width` and `height` are also inherited from parent controls, as long as you don't set an explicit value for these dimensions. If you, for example, set one of the dimension values for a control to `100%` it will have the same size as the parent control. And if the parent's width is `0` the nested control will also be `0`.

As with the `visible` property, you can solve this by either increasing the size of the parent or setting fixed values for the child (for example, `100px`) instead of a relative value.

Request Fails Due to Same-Origin Policy (Cross-Origin Resource Sharing - CORS)

If you use a remote URL in your code, for example a remote OData service, such as the publicly available Northwind OData service, the browser may refuse to connect to a remote URL. Due to the same-origin policy, browsers deny AJAX requests to service endpoints in case the service endpoint has a different domain/ subdomain, protocol, or port than the app.

Preview

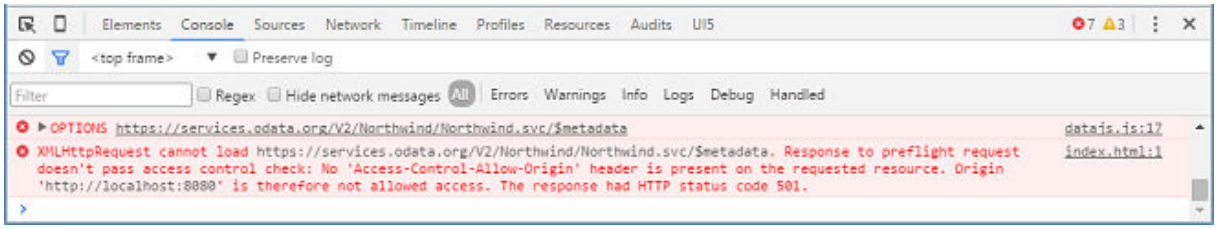


Figure 240: Violations of the same-origin policy in Google Chrome

Root Cause

Normally, the remote system would be configured to send the cross-origin resource sharing (CORS) headers to make the browser also allow direct access to remote URLs. However, if you, for example, use a Northwind OData service, you cannot modify the publicly available service. Then when you try to execute XHR requests (`XMLHttpRequest`) the browser prevents the call due to the same-origin policy.

Resolution

To solve the issue, you have the following options:

- SAP Web IDE: Configure a destination as described below (recommended)
- Local Development: Configure a local proxy (CORS anywhere)
- Workaround: Disable the same-origin policy in the browser for local testing (not recommended, only for testing)
- Set the CORS-relevant response headers on the remote system (if possible)

SAP Web IDE: Configure a destination

SAP Web IDE and the SAP Cloud Platform offer destinations that allow you to easily connect to remote systems. The destination to the Northwind OData service is an internet proxy made available inside the app at `<protocol>://<domain>/destinations/northwind/*`. Any request that is sent to this location is forwarded to `https://services.odata.org` automatically.

Create Destination in SAP Cloud Platform Cockpit

Requested URL	Forwarded To
`/destinations/northwind/V2/Northwind/Northwind.svc/$metadata`	`https://services.odata.org/V2/Northwind/Northwind.svc/$metadata`

Requested URL	Forwarded To
`/destinations/northwind/V2/Northwind/Northwind.svc/Invoices`	`https://services.odata.org/V2/Northwind/Northwind.svc/Invoices`

The destination itself is configured inside the SAP Cloud Platform Cockpit. For more information, see Create a Northwind Destination [page 49].

neo-app.json

For SAP Web IDE, a `neo-app.json` file is needed to make sure that the destination and resource mapping are available in the app. It has to be located in the root folder (`webapp`), on the same level as the `user.project.json` file that is automatically created.

If it does not exist yet, create a `neo-app.json` file and reference the Northwind destination there. Just copy the content of the code into that file and try to run the app again.

```
{
  "routes": [
    {
      "path": "/destinations/northwind",
      "target": {
        "type": "destination",
        "name": "northwind"
      },
      "description": "Northwind OData Service"
    }
  ]
}
```

> **i Note**
>
> If the file already exists, for example, because you already created it to map the SAPUI5 resources, just append the destination to the existing `route` definitions.

manifest.json

In the `manifest.json` descriptor file of your app, you can now change the data source to use the remote destination, for example:

```
{
  "_version": "1.12.0",
  "sap.app": {
    ...
    "dataSources": {
      "invoiceRemote": {
        "uri": "/destinations/northwind/V2/Northwind/Northwind.svc/",
        "type": "OData",
        "settings": {
          "odataVersion": "2.0"
        }
      }
    }
  },
  "sap.ui": {
    ...
  },
  "sap.ui5": {
    ...
```

```
    }
}
```

After this change, you can run the app in SAP Web IDE without disabling the same-origin policy of your browser. The destination now manages the connection to the remote service.

Local Development: Configure a local proxy (CORS anywhere)

A proxy is simply a service end point on the same domain of your app to overcome the restrictions. It receives requests from the app, forwards them to another server, and finally returns the corresponding response from the remote service.

Follow the below steps to configure such a proxy in your poject.

Prerequisites: NodeJS is installed on your machine.

package.json

```
{
    "name": "Sample-Package",
    "version": "1.0.0",
    "description": "Sample package.json",
    "scripts": {
        "proxy": "node proxy.js"
    },
    "devDependencies": {
        "cors-anywhere": "^0.4.1"
    },
    "dependencies": {
    }
}
```

Add the `devDependency` called `"cors-anywhere": "^0.4.1"` to your existing `package.json`. Run `node install` to install the npm module. Add the `proxy` script to the `scripts` section in the `package.json` so that you can run a script via `npm run <script_name>`.

proxy.js (new)

```
var cors_proxy = require('cors-anywhere');

// Listen on a specific IP Address
var host = 'localhost';

// Listen on a specific port, adjust if necessary
var port = 8081;

cors_proxy.createServer({
    originWhitelist: [], // Allow all origins
    requireHeader: ['origin', 'x-requested-with'],
    removeHeaders: ['cookie', 'cookie2']
}).listen(port, host, function() {
    console.log('Running CORS Anywhere on ' + host + ':' + port);
});
```

Create a new file `proxy.js`, and copy the above script into your project directory. This is the pre-configured proxy server we are going to use to prevent the occurrence of **same-origin policy error**. We can start it by running the command `node proxy.js` or `npm run proxy. It runs a local proxy on port` in the console.

manifest.json

```
{
    "sap.app": {
        ...
        "dataSources": {
            "northwind": {
                "uri": "http://localhost:8081/https://services.odata.org/V2/
Northwind/Northwind.svc/",
                "type": "OData",
                "settings": {
                    "odataVersion": "2.0"
                }
            }
        }
    }
}
```

To use a service in the `local ui5 application` we have to change the `uri` in the `manifest` file.

> **i** Note
>
> The `uri` must start with `http://localhost:<port>`.

> **i** Note
>
> By default, you can't run the request in your browser with the `proxy.js` script. It throws the following exception: `exception Missing required request header. Must specify one of: origin,x-requested-with`. If you want to test the service in your browser, you can temporarily comment out the line `requireHeader: ['origin', 'x-requested-with']` from your `proxy.js`.

For more information on CORS Anywhere, see https://www.npmjs.com/package/cors-anywhere

Workaround: Disable the same-origin policy in the browser (not recommended, only for testing)

. It runs a local proxyIn Google Chrome, you can easily disable the same-origin policy of Chrome by running Chrome with the following command: `[your-path-to-chrome-installation-dir]\chrome.exe --disable-web-security --user-data-dir`. Make sure that all instances of Chrome are closed before you run the command. This allows all web sites to break out of the same-origin policy and connect to the remote service directly.

> ⚠ Caution
>
> **This approach is not recommended for productive apps. Running Chrome this way for surfing on the internet poses a security risk.** However, it allows you to avoid the need of setting up a proxy at development time or for testing purposes.

App or Control Looks Odd

You find yourself in a situation that your app or a control looks different than you expected.

Root Cause

This can happen for one of the following reasons:

- An HTML file is missing the `DOCTYPE` specification (this leads, for example, to exceptionally high table headers)
- Custom styles aren't working properly
- The theme you are using does not support the used libraries

Resolution

To solve the issue, you have the following options:

- Check whether the `<!DOCTYPE html>` tag is placed at the beginning of each HTML file, before the `<html>` tag.
- Check if you have used a custom CSS in your app.
  If you have used a custom CSS, it is probably interfering with the styling in the standard SAPUI5 theming libraries.
  Use the developer tools of your browser to inspect the element that has the wrong styling. In the HTML tab, you can usually see which styles are applied to a DOM element. If you have styles in the list that are added by your app, disable these styles in the debugger to see whether this solves the problem.

  > **i Note**
  >
  > SAPUI5-specific CSS classes and IDs all have an `sapUi` prefix, for example, `sapUiButton`.

  If this does not solve the issue, check for inline styles that are applied to the element in the HTML code. You can also try to isolate the control from the app to see whether there is an issue with the control instead of a collision of styles.
- Check whether the theme you that you are using is supported in combination with the libraries that you are using in your app. For more information, see Supported Combinations of Themes and Libraries [page 27] and Deprecated Themes and Libraries [page 34].

Developing Apps

Create apps with rich user interfaces for modern web business applications, responsive across browsers and devices, based on HTML5.

Project Setup

Before you start developing apps with SAPUI5, you start by setting up the development environment of your choice. You can find our recommendations under Development Environment [page 41].

After that, you define the project setup. If you work in a team, we recommend using a continuous integration setup as described under Continuous Integration: Ensure Code Quality [page 1398]. If you work alone, you should at the very least set up an automated testing environment.

Development

When faced with developing an app, you have several ways to get started ranging from app templates or a make-em-completely-from-scratch approach. But which approach is right for your situation? In most cases, it's your level of expertise or need for flexibility and freedom that will decide.

Think of the app templates described in this section as a kind of best practice for app development. They incorporate our latest recommendations and can be used as a starting point for developing apps according to the SAP Fiori design guidelines. They include generic application functionality and tests that can be easily extended with custom functionality if needed. There are separate templates for Worklist and Master-Detail application patterns, and we have an empty basic template. The templates are described under App Templates: Kick Start Your App Development [page 1399].

If you're skilled at coding and want the freedom and flexibility, you can build an app completely from scratch. In the chapters in this section, you will find some guidance on how to deal with crucial concepts such as accessibility, security, device adaptation.

Be sure to check out information about things that you really should avoid doing while coding: Coding Issues to Avoid [page 1458].

You can also use SAP Fiori elements or the reuse components of Analysis Path Framework. For more information, see Developing Apps with SAP Fiori Elements [page 1535] and Developing Apps with Analysis Path Framework (APF) [page 2040].

Packaging and Deployment

How you deploy your app when you're finished depends on the tools you use and the platform where your app is going to run. See the related documentation for details.

For example, it's quite easy to deploy an app to SAP Cloud Platform with SAP Web IDE. For more information about SAP Web IDE, see the documentation for SAP Web IDE on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_Web_IDE.

> **i Note**
>
> For more information about packaging apps, read the blog post Optimizing OpenUI5/SAPUI5 Apps.

Continuous Integration: Ensure Code Quality

This section describes the setup of a development project where multiple developers work together on the same code.

When you develop an app, you want, of course, to deliver high code quality. To do so, you have various tools at hand:

- Unit and integration tests to make sure that your change doesn't break the app or other tests (see Unit Testing with QUnit [page 1159] and Integration Testing with One Page Acceptance Tests (OPA5) [page 1182])
- Code analyzer (or "linter") to check whether the code follows the code style conventions that apply to your project
- A code coverage analyzer to check the test coverage of your code
- A code collaboration tool to let other experts review your code
- An automation server that orchestrates the automated task

If you work in a team where you share the same code with your colleagues, or if you work in a continuous delivery context where an app is updated on a regular basis, you need the support of automated continuous integration processes.

Continuous integration makes sure that only code that has passed various automated and manual checks is merged to productive use.

In an SAPUI5 project, the process, for example, can look like this:

1. When the code is ready, the developer commits the code to the source code management system (SCMS, for example, *Gerrit Code Review*) and assigns experts that should review the code.
2. The SCMS calls an automation server (for example, *Jenkins*).
3. The developer triggers the peer code review and the automation server starts a voter job that triggers, among others:
   - Static code checks (for example, *ESLint*) to check the code style
   - Unit and integration test automated by a test runner (for example, *Karma*)
   - Analysis to check whether all parts of the code are covered with automated tests (for example, with *Karma* plug-in *Istanbul Code Coverage*)
4. When the voter job and the human reviewer have both given their OK, the change can be merged.

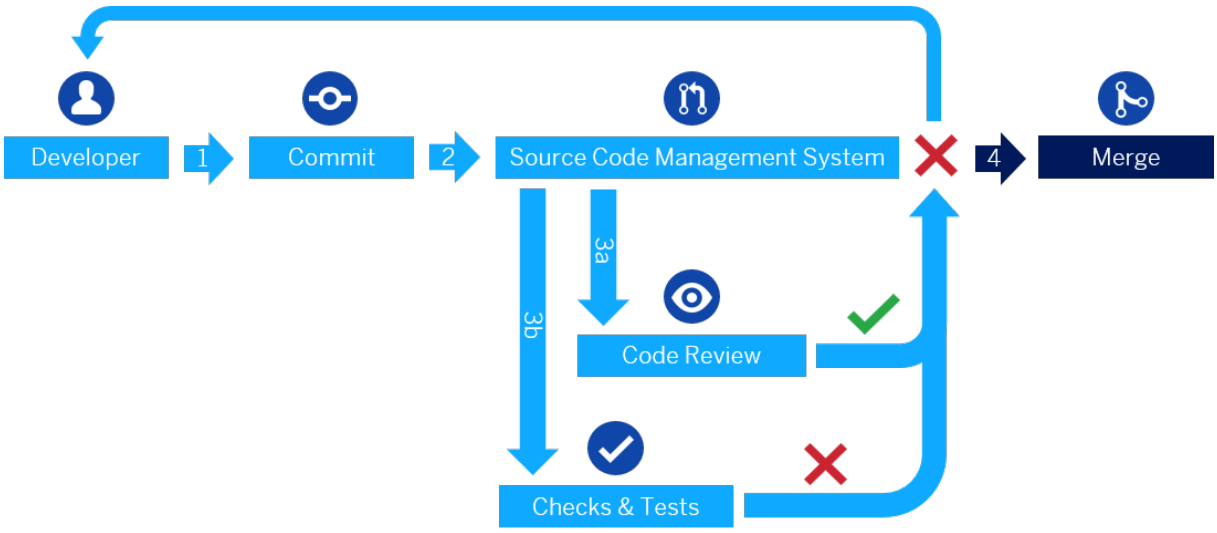


Figure 241: Code cannot be merged because automated tests failed

Related Information

Testing [page 1158]
Gerrit Code Review Home Page
Jenkins Home Page
ESLint Home Page
Karma Home Page
Istanbul Code Coverage Home Page

App Templates: Kick Start Your App Development

The app templates documented here are a kind of "best practice" for your app development.

They incorporate our latest recommendations and can be used for example as a starting point for developing apps. They include generic application functionality and tests that can be easily extended with custom functionality if needed.

There are separate templates for the **Worklist** and **Master-Detail** application patterns, which are closely aligned with the SAP Fiori design guidelines. The subsections of this chapter explain the concepts and features of each of these templates. See the SAP Fiori Design Guidelines.

> **i Note**
>
> These templates were primarily created for developing SAP Fiori apps in accordance with the SAP Fiori design guidelines, but can also be used in any other SAPUI5 context.

The basic template serves as a starting point when you want to start from scratch, but also want to make sure that everything is set up properly.

Where Can I Find the Templates?

All templates are available under Demo Apps, in SAP Web IDE, and in the SAP Repository on GitHub.

Template	Demo App	Name in SAP Web IDE	Name of GitHub Repository
Basic template	**Basic Template**	**SAPUI5 Application**	`openui5-basic-template-app`
Worklist template	**Worklist Template** **Worklist (FLP) Template**	**SAP Fiori Worklist Application** **SAP Fiori Worklist Application - OData V4**	`openui5-worklist-app`
Master-Detail template	**Master-Detail Template** **Master-Detail (FLP) Template**	**SAP Fiori Master-Detail Application**	`openui5-masterdetail-app`

Related Information

Demo Apps [page 671]
Development Environment [page 41]

Worklist Template

The **SAP Fiori Worklist Application** template implements a typical worklist floorplan, one of the patterns that is specified by the SAP Fiori design guidelines.

A worklist displays a collection of items to be processed by the user and usually involves reviewing details of a list item and taking action. If the data needs to be organized into columns or the overview of the items is more important than showing the item details directly, this template can be used as a starting point. For more information about worklist floorplans, see the SAP Fiori Design Guidelines.

> **i** Note
>
> You have two options: You can use this template to build an **app for the SAP Fiori launchpad (FLP)** or to build **standalone apps**.
>
> - If the app runs in FLP it also contains additional features like *Save as Tile* or *Share in SAP Jam* that depend on FLP at runtime. This app cannot be run standalone, meaning no `index.html` file is created but only files for testing the app in the FLP sandbox.
> - Only standalone apps contain an `index.html` file that is used to start the app.

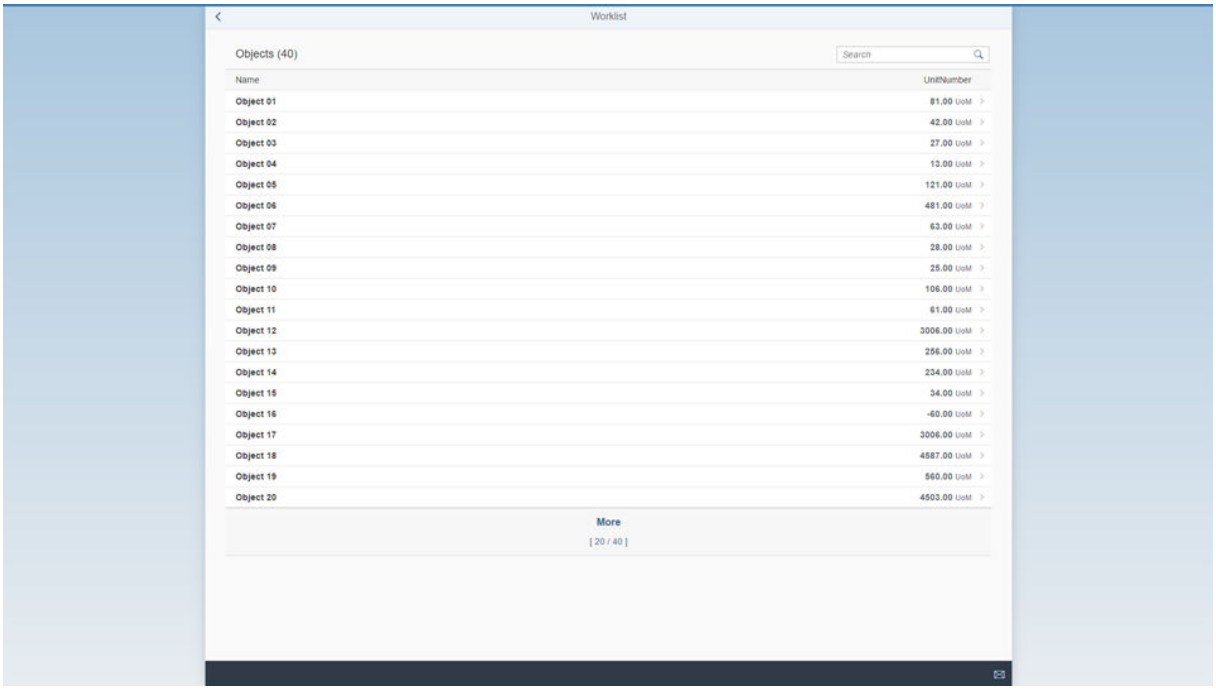


Figure 242: Screenshot of the Worklist App

The *Worklist* view is the main view that is initially displayed in this app. When a user clicks or taps an item in the table, the *Object* view is displayed, showing more details for the selected item. We use the semantic `FullscreenPage` control as the page for both. A `SemanticPage` is an enhanced `sap.m.Page` that contains controls with a semantic meaning and displays them according to the SAP Fiori Design Guidelines, for example. For more details about semantic controls, see the sample in the Demo Kit.

The table in the *Worklist* view displays a header area that shows the current amount of items in the worklist and a search field. The number of items are updated automatically and the search filters for a preconfigured column of the table.

> **i Note**
>
> As the use cases for apps using a worklist pattern differ greatly, we only show a basic scenario in our template as a starting point for your individual development activities. For more information, see How Do I Enhance the Template? [page 1402]

Where Can I Find the Worklist Template?

You can find the template in the following places:

- **SAP Fiori Worklist Application** (for OData V2 models) and **SAP Fiori Worklist Application - OData V4** (for OData V4 models) templates in SAP Web IDE
  For more information about SAP Web IDE, see the documentation for SAP Web IDE on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_Web_IDE.
- **Worklist Template** and **Worklist (FLP) Template** under Demo Apps.
- `openui5-worklist-app` in the SAP Repository on GitHub

For more information on how to clone or download the template from GitHub, refer to the template documentation on GitHub .

Tutorial

See the Worklist App [page 447] tutorial for an example of how this application can be extended. The result of this tutorial can be seen as the *Manage Products* app in the *Demo Apps* section of the Demo Kit.

How Do I Enhance the Template?

In our template, we use a simple layout that you can use as a basis for enhancements. For example, if you want to use an object page with a dynamic header, you can use one of the page-type Object Page Layout samples. All you have to do is replace the relevant content in the template with the content from the sample.

You can find more information about the possibilities of object pages at SAP Fiori Design Guidelines - Object Page.

Related Information

Demo Apps [page 671]
Development Environment [page 41]
Worklist App [page 447]

Navigation

The navigation flow of the Worklist app is very simple as it only contains two main views and the *not found* pages that are displayed as a message to the user in case of navigation errors.

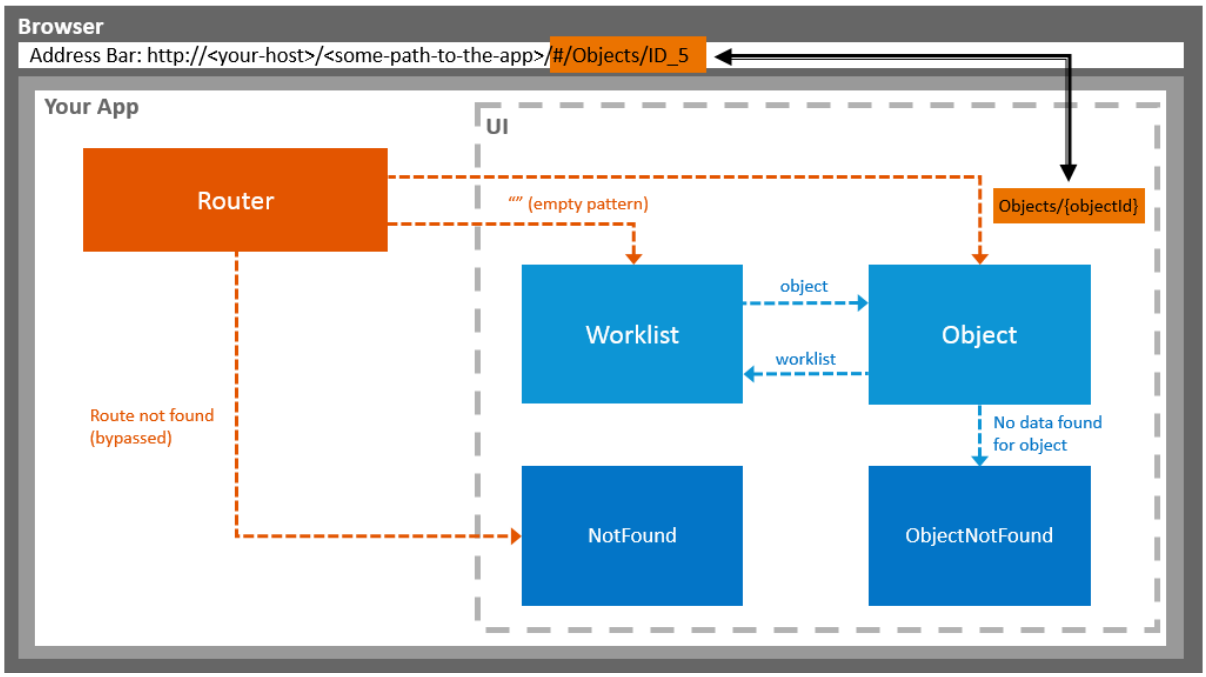


Figure 243: Navigation Flow of the Worklist App

The two main views *Worklist* and *Object* each have a route and a target configured. When the route matches the URL, the target is displayed and the corresponding view is created. For more information, see Routing and Navigation [page 1072].

Here is a sample implementation for navigating from the worklist to the object page. First you have to implement a press handler on the `ListItem`. Inside, you extract the current ID of the object pressed by the user by using its `bindingContext`. Since we want to navigate to the "object" route, you need to supply the mandatory `objectId` parameter and pass it to the `navTo` function, as described in the sap.ui.core.routing.Routing#navTo section of the *API Reference* in the Demo Kit and shown here:

```
/**
 * Event handler when a table item gets pressed
 * @param {sap.ui.base.Event} oEvent the table selectionChange event
 * @public
 */
onPress : function (oEvent) {
   // The source is the list item that got pressed
   this.getRouter().navTo("object", {
      objectId: oEvent.getSource().getBindingContext().getProperty("ObjectID")
   });
},
// more controller code
```

After calling `navTo`, the hash of the browser is updated and you get an event on the `ObjectController` when the route "object" matches the current hash. In the event handler, you extract the `objectId` using the `Event.getParameter` function. You then bind the data to the view:

```
// init function of the object controller
```

```
onInit : function () {
   var oView = this.getView();
   var oModel = oView.getModel();
   this.getRouter().getRoute("object").attachPatternMatched(function (oEvent) {
      var sObjectId =  oEvent.getParameter("arguments").objectId;
      oModel.metadataLoaded().then(function() {
         var sObjectPath = oModel.createKey("Objects", {
            ObjectID :  sObjectId
         });
         oView.bindElement({
            path: ("/" + sObjectPath)
        });
      });
   });
    …
   // more init code
},
…
// more controller code
```

notFound (similar to an HTTP 404 "not found" status code)

The *not found* pages are implemented using a sap.m.MessagePage. They display an error message according to the SAP Fiori UX specifications. There are different "not found" cases that each have a separate target and a *notFound* view.

If you have the following URL, no route will match: `index.html/#/thisIsInvalid`. This means that the *notFound* view will be displayed, as the target *notFound* is defined in the bypassed section.

The code sample below shows the relevant parts of the configuration. For a full implementation of a *not found* page, see Step 3: Catch Invalid Hashes [page 300].

```
"routing": {
    "config": {
        …
        "bypassed": {
              "target": "notFound"
        }
    }
    …
  "targets": {
    …
    "notFound": {
        "viewName": "NotFound",
        "viewId": "notFound"
    }
}
```

objectNotFound

If the object route matches – an ID is passed (for example `#/Objects/1337`) but the back end does not contain an object with the ID `1337`, then you need to display the *objectNotFound* page. This is achieved by

listening to the “change” event of a binding. Inside this, you check if there is no data and tell the router to display the *objectNotFound* target, as shown in the sample code below:

```
// inside of a controller
this.getView().bindElement({
    path: “/Objects/1337”,
   change: function () {
       // there is no data
       if (!this.getView().getElementBinding().getBoundContext()) {
        this.getRouter().getTargets().display("objectNotFound");
                    return;
                }
       // code handling the case if there is data in the backend
      …
   };
});
```

The routing configuration for this navigation flow is set up in the descriptor for applications (`manifest.json` file), as shown here:

```
"routing": {
  "config": {
    "routerClass": "sap.m.routing.Router",
    "viewType": "XML",
    "viewPath": "sap.ui.demo.worklist.view",
    "controlId": "app",
    "controlAggregation": "pages",
    "bypassed": {
      "target": "notFound"
    }
  },
  "routes": [
    {
      "pattern": "",
      "name": "worklist",
      "target": "worklist"
    },
    {
      "pattern": "Objects/{objectId}",
      "name": "object",
      "target": "object"
    }
  ],
  "targets": {
      "worklist": {
        "viewName": "Worklist",
        "viewId": "worklist",
        "viewLevel": 1
      },
      "object": {
        "viewName": "Object",
        "viewId": "object",
        "viewLevel": 2
      },
      "objectNotFound": {
        "viewName": "ObjectNotFound",
        "viewId": "objectNotFound"
      },
      "notFound": {
        "viewName": "NotFound",
        "viewId": "notFound"
    }
  }
}
```

For more information, see Routing and Navigation [page 1072], the sap.m.routing.Router section of the *API Reference* documentation in the Demo Kit, and the sap.ui.core.routing.Router sample within the Demo Kit.

Busy Indication

The Worklist app implements a busy indication concept as specified by the SAP Fiori Design Guidelines.

Calling the app will result in the following:

- Only initially a global busy indicator is displayed that overlays the whole app until the metadata of the service is loaded.
- A local busy indicator is displayed on the worklist table or on the page of the object view while the data from the service is loading.
- When controls are loading additional data or getting refreshed, a local busy indication is displayed automatically.

By default, the busy indicator delay is set to one second for all controls. This would first show the UI for a second, then show a busy indication until the data is loaded. To avoid this behavior initially and show the busy indicator immediately without delay the following concept is implemented in the app: The `busyIndicatorDelay` and `busy` properties of certain controls (`AppView`, `Table` on the *Worklist* page, `FullScreenPage` on the *Object* page) are bound to the local view model and manipulated in the controllers of the app. The delay is initially set to "0" for displaying the busy indicator immediately, and reset to the previous value after the initial loading is done.

> **i Note**
>
> You can find more information about busy indicators, busy states, and busy handling in general in the SAP Fiori Design Guidelines.

Model Instantiation

The app configures several data models that are used throughout the app to update the views or to store additional configuration options.

The service model and the resource bundle are instantiated automatically by the app's component during startup and described in the first section. The local view models and helper models such as the device model are set up as JSON models and described in the second section.

Automatic Model Instantiation

The templates instantiate the service and resource model automatically using the following configuration entries in the descriptor. When the component of the app is initialized, these models will be made available under the configured name throughout the app.

An external service is defined in the `dataSources` section of the `sap.app` namespace. In the example shown below, we configure an OData V2 model and the alias `"mainService"` in the `manifest.json` descriptor file:

```
{
    …
    "sap.app": {
        …
        "i18n": "i18n/i18n.properties",
        …
        "dataSources": {
            "mainService": {
                "uri": "/here/goes/your/serviceUrl/"
                "type": "OData",
                "settings": {
                    "odataVersion": "2.0",
                    "localUri": "localService/metadata.xml"
                }
            }
        },
        …
    },
    …
}
```

> **i Note**
>
> If you use the OData V4 template, you set the `odataVersion` accordingly.

In the models section of the `sap.ui5` namespace we define two models that will be instantiated automatically. The resource model is a named model (i18n) and the OData model is the default model so it has no name. The OData model also receives additional URL parameters via the `metadataUrlParams`. The parameters `sap-server`, `sap-client`, and `sap-language` are passed to the service automatically by SAPUI5, as shown in the following `manifest.json` code snippet:

```
{
    …
    "sap.ui5": {
        …
        "models": {
            "i18n": {
                "type": "sap.ui.model.resource.ResourceModel",
                "settings": {
                    "bundleName": "sap.ui.demo.masterdetail.i18n.i18n"
                }
            },
            "": {
                "dataSource": "mainService",
                "preload": true
            }
        },
        …
    }
}
```

> **i Note**
>
> Before SAPUI5 version 1.30, all models were defined and instantiated in the component's `init` method. We recommend removing all manual model creation code and switching to the automatic model instantiation instead. The "device model" however is still a local model that has to be instantiated manually.

Additional Models for the App

The following models are created as local JSON models in the app and can be referenced by its model name where needed:

- **device**
  The device model provides an easy access to the device API and is used to configure certain view settings according to the user's device.
- **FLP**
  The FLP model is a helper module to configure SAP Fiori launchpad (FLP) integration and is used to control the sharing options of the app.
- **worklistView**
  A local view model for the worklist view that stored configuration options that are bound to controls in the view.
- **objectView**
  A local view model for the object view that stored configuration options that are bound to controls in the view.

Send Email

The **Send Email** feature is a sharing option that can be found in the share menu of each view.

This feature simply triggers a `sap.m.URLHelper` action that will show a new email with preconfigured texts in the default client of the user.

The placeholder texts are located in the resource model and can be adjusted to your use case. The texts already include the current context and the current location. The texts may vary for each view, therefore they are configured with the local view model and updated when the business object context has changed.

For more information about `sap.m.URLHelper`, see the sample in the Demo Kit.

Testing

The templates include basic testing features, unit tests as well as integration tests for a basic test coverage of the initial app. The tests are written independently of the actual data displayed in the app.

The `webapp` folder of the template app contains a `test.html` file which serves as an overview for the different test pages. You can run the app with or without mock data and run the unit and integration tests. This section describes which application tests are provided and how they are structured.

Custom Mock Data

To run the automated tests below, we need a stable reference between the item on the master page and the items displayed in the line item table on the detail page. The mock server we use to simulate a backend system

is capable of creating random entities suitable for testing, but it cannot foresee the dependency between the entity sets.

After creating this template, we therefore need to create mock data to successfully run all automated tests delivered with this template.

In SAP Web IDE, you can easily create mock data that fulfill the conditions above: Right-click on the `metadata.xml` file in the `localService` folder of your template project, and select *Edit Mock Data*. Select the entity sets that you have chosen for the master page and the detail page, and choose *Generate Random Data*. The generated files will be put in the folder `localService/mockdata`.

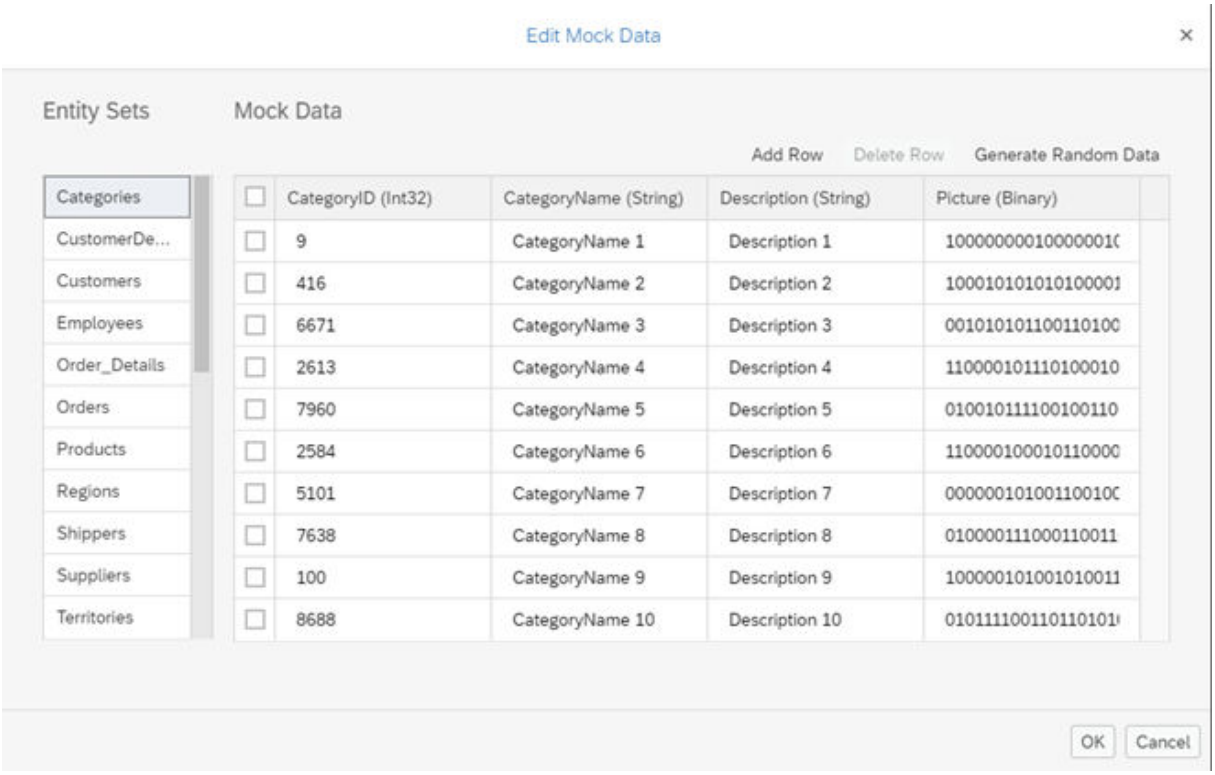


Figure 244: Screenshot of the Edit Mock Data dialog

You can also manually download a set of json files from the actual service by appending **`$format=json`** to the corresponding URL. Download the resulting files, and put them in the `localService/mockdata` folder within the template project.

For more information, see Step 2: Custom Mock Data [page 456] in our Worklist App tutorial.

Integration Tests

The integration tests shipped with the template cover all basic functionality and provide several "journeys". Journeys include a series of OPA tests that belong to the same functionality and should be executed together:

- `NavigationJourney`: This journey will trigger user interactions and navigate through the application. The routing configuration, basic navigation events, and error handling are tested here.
- `NotFoundJourney`: Several "not found" cases of the application are tested here. Faulty navigation scenarios are introduced intentionally to simulate errors.

- `WorklistJourney`: A series of tests for the *Worklist* page that check the busy indication and sharing features built into the app.
- `ObjectJourney`: A series of tests for the *Object* page that check the busy indication and sharing features built into the app.
- `FLPIntegrationJourney`: This journey is available if you have enabled SAP Fiori launchpad (FLP) for your app. It tests the FLP integration features *Save as tile* and *Share on SAP Jam*.
- `AllJourneys`: This is a convenience journey that will call all the other journeys specified above and is used in the test suite file.

You can execute all journeys by calling the test suite file `opaTests.qunit.html` in the `webapp/test/integration` folder or selecting the *run all integration tests* link in the `test.html` file in the app's root folder.

For more information, see Integration Testing with One Page Acceptance Tests (OPA5) [page 1182] and sap.ui.test.Opa5 in the *Samples* within the Demo Kit.

Unit Tests

In the `unit` subfolder you can find all unit tests for our application. They are structured similarly to the structure of the `webapp` folder. For example, controller tests are located in the `controller` folder whereas formatter tests are located in the `model` folder.

Unit tests are included for the following functionality:

- App controller tests
- Worklist controller tests
- Formatters
- Device model

As with the integration tests, you can execute all unit tests by calling the test suite file `unitTests.qunit.html` in the `webapp/test/unit` folder or selecting the *run all unit tests* link in the `test.html` file in the app's root folder.

For more information, see Unit Testing with QUnit [page 1159], https://qunitjs.com/ and http://sinonjs.org/ .

Device Adaptation

The following section outlines the best practices for ensuring your worklist apps adapt to different kinds of devices in the best way possible.

Content Density

The app templates include a mechanism to adjust the content density of the controls according to the device features. On devices that feature touch support, the controls are automatically displayed larger. For more information, see How to Use Densities for Controls [page 1146].

Stable IDs

Setting stable IDs is crucial if your app is used in combination with certain functions.

Most controls in the template apps (except for aggregations that are created dynamically, such as list items) are assigned a stable ID to identify the controls in integration tests, extensibility tools like key user adaptation, as well as interactive inline help tools.

Related Information

SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152]
Extending Apps [page 2143]
Stable IDs: All You Need to Know [page 1442]

Master-Detail Template

The **SAP Fiori Master-Detail Application** template implements a flexible column layout, one of the design patterns that is specified by the SAP Fiori design guidelines.

The flexible column layout is a layout control that displays multiple floorplans on a single page. This allows faster and more fluid navigation between multiple floorplans than the usual page-by-page navigation. The flexible column layout offers different layouts with up to three columns. In the template, we use two colums (master and detail). For more information about flexible columns and master-detail apps, see the SAP Fiori Design Guidelines.

> **i** Note
>
> You have two options: You can use this template to build an **app for the SAP Fiori launchpad (FLP)** or to build **standalone apps**.
>
> - If the app runs in FLP it also contains additional features like *Save as Tile* or *Share in SAP Jam* that depend on FLP at runtime. This app cannot be run standalone, meaning no `index.html` file is created but only files for testing the app in the FLP sandbox.
> - Only standalone apps contain an `index.html` file that is used to start the app.

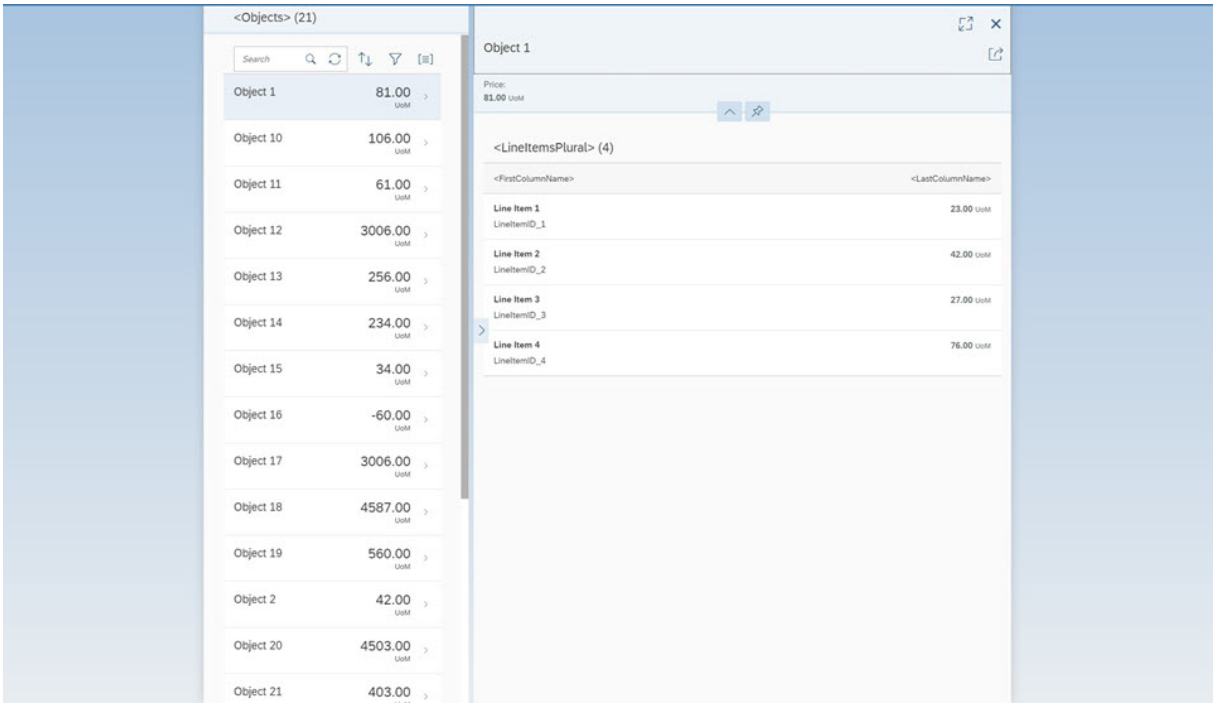


Figure 245: Screenshot of the Master-Detail App

The main control of this app is the `sap.f.flexibleColumnLayout` control. This control first displays only the *Master* view with a list of objects. When the user selects an object in the master list, the *Detail* view is displayed on the right side, showing the details for the selected item.

The *Master* view shows the current number of items and a search field that can be used to search through the list items. The number of items are updated automatically and the search filters for a preconfigured field of the list. Functionality for sorting, filtering, and grouping the list is also included in the template as well.

The *Detail* page contains a dynamic page header displaying more details for the selected object, an `sap.m.Overflowtoolbar` that can be enriched with custom content, and a table of line items that are associated to the selected object in the data model.

The master list and the line item table are set to `growing` mode so that initially only the first few items are displayed for performance reasons. Using the `scrollToLoad` feature, the user can display more items by scrolling down or pressing the trigger at the end of the list.

We use the semantic `MasterPage` and `DetailPage` controls for the content aggregations of the `sap.f.FlexibleColumnLayout` control. A `SemanticPage` is an enhanced `sap.f.DynamicPage` that contains controls with semantic-specific meaning and displays them according to the SAP Fiori design guidelines. For more details about semantic controls, see the sample in the Demo Kit.

> **i Note**
>
> As the use cases for apps using a master-detail pattern differ greatly, we only show a basic scenario in our template as a starting point for your individual development activities. For more information, see How Do I Enhance the Template? [page 1413]

Where Can I Find the Master-Detail Template?

You can find the template in the following places:

- **SAP Fiori Master-Detail Application** template in SAP Web IDE
  For more information about SAP Web IDE, see the documentation for SAP Web IDE on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_Web_IDE.
- **Master-Detail Template** and **Master-Detail (FLP) Template** under Demo Apps.
- `openui5-masterdetail-app` in the SAP Repository on GitHub
  For more information on how to clone or download the template from GitHub, refer to the template documentation on GitHub .

How Do I Enhance the Template?

In our template, we use a simple layout that you can use as a basis for enhancements. For example, if you want to use an object page with a dynamic header, you can use one of the page-type Object Page Layout samples. All you have to do is replace the relevant content in the template with the content from the sample. To add a third column to your app, create a new view and have a look at the `sap.f.FlexibleColumnLayout` examples to see how to configure it.

You can find more information about the possibilities of object pages at SAP Fiori Design Guidelines - Object Page.

Related Information

Demo Apps [page 671]
Development Environment [page 41]

Navigation

The navigation flow of the Master-Detail app considers both the *Master* and *Detail* pages, and is therefore slightly more complex than a typical full-screen scenario.

With an empty hash in the URL, only the master view is shown initially. When the user enters the app with an object id in the hash, both views are loaded at the same time, and methods in the controller logic make sure that the pages are in sync. Additional *not found* pages display a message to the user in case of any navigation errors that occur for the master and the detail page.

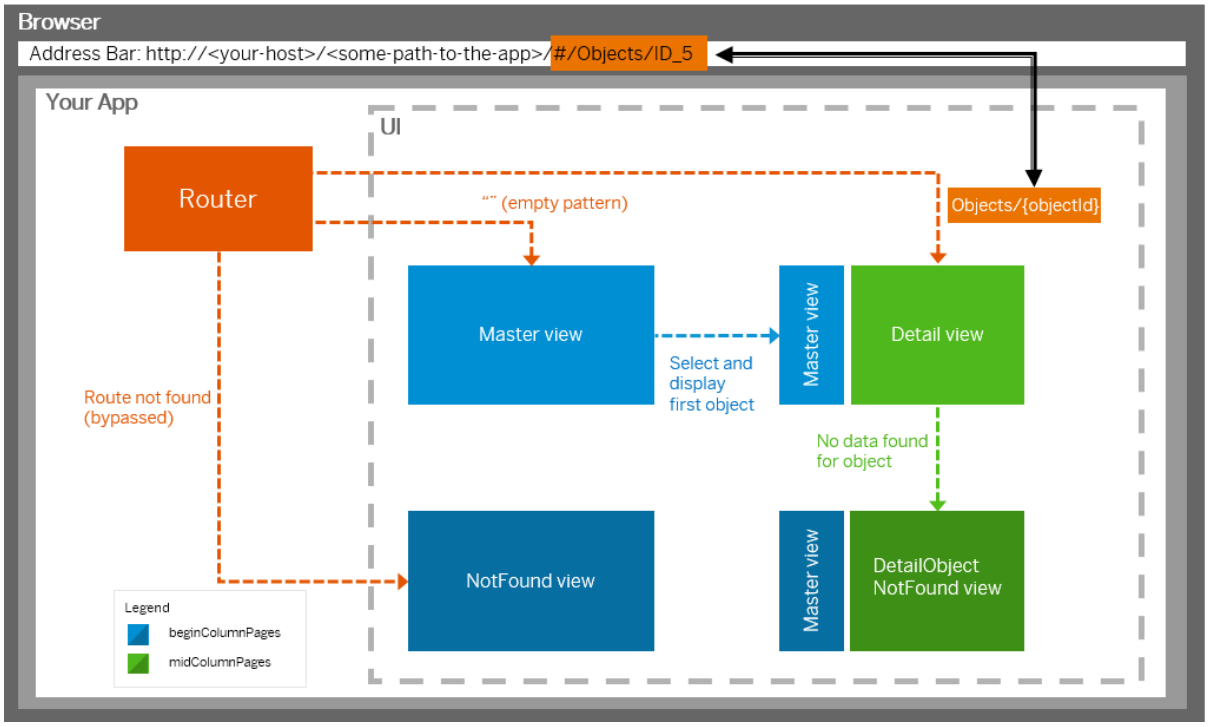


Figure 246: Navigation Flow of the Master-Detail App

The two main views *Master* and *Object* each have a route and two targets configured. When the route matches the URL, both targets are displayed and the corresponding views are created. The target master puts the created view in the `beginColumnPages` aggregation of the `sap.f.FlexibleColumnLayout` control. All other targets put their created views in the `midColumnPages` aggregation. For more information, see Routing and Navigation [page 1072].

Here is a sample implementation for navigating from the *Master* to the *Object* page. The below `_showDetail` method is called by the `selectionChange` event handler of the `sap.m.List` control. We need to change the layout parameter of the `sap.f.FlexibleColumnLayout` to `TwoColumnsMidExpanded` and navigate to the `object` route. Then, we extract the current ID of the object pressed by using its binding context. We supply this parameter to the mandatory `objectId` parameter and pass it to the `navTo` function, as described in the sap.ui.core.routing.Routing#navTo section of the *API Reference* in the Demo Kit and shown here:

```
...
/**
   * Shows the selected item on the detail page
   * On phones an additional history entry is created
   * @param {sap.m.ObjectListItem} oItem selected Item
   * @private
   */
_showDetail : function (oItem) {
      var bReplace = !Device.system.phone;
      // set the layout property of FCL control to show two columns
      this.getModel("appView").setProperty("/layout", "TwoColumnsMidExpanded");
      this.getRouter().navTo("object", {
         objectId : oItem.getBindingContext().getProperty("ObjectID")
      }, bReplace);
},
...
```

After calling `navTo`, the hash of the browser is updated, and you get an event on the `DetailController` when the route `object` matches the current hash. In the `_onObjectMatched` handler that we register in the `init`

method of the controller, we extract the `objectID` from the event arguments and create a valid model path with the help of the `createKey` method of our OData model. We then bind the data to the view:

```
...
/**
  * Binds the view to the object path and expands the aggregated line items.
  * @function
  * @param {sap.ui.base.Event} oEvent pattern match event in route 'object'
  * @private
  */
_onObjectMatched : function (oEvent) {
      var sObjectId =  oEvent.getParameter("arguments").objectId;
      this.getModel("appView").setProperty("/layout", "TwoColumnsMidExpanded");
      this.getModel().metadataLoaded().then( function() {
        var sObjectPath = this.getModel().createKey("Objects", {
          ObjectID :  sObjectId
        });
        this._bindView("/" + sObjectPath);
      }.bind(this));
},
...
```

notFound (similar to an HTTP 404 "not found" status code)

The *not found* pages are implemented using an sap.m.MessagePage. They display an error message according to the SAP Fiori UX specifications. There are different "not found" cases that each have a separate target and a *notFound* view.

If you have the following URL, no route will match: `index.html/#/thisIsInvalid`. This means that the *notFound* view will be displayed, as the target *notFound* is defined in the bypassed section.

The code sample below shows the relevant parts of the configuration. In addition, we set the layout property for the `sap.f.FlexibleColumnLayout` to `OneColumn` in the controller of the *notFound* page so that only a single column is displayed in this case. For a full implementation of a *not found* page, see Step 3: Catch Invalid Hashes [page 300].

```
"routing": {
    "config": {
    …
    "bypassed": {
    "target": "notFound"
    }
    }
    …
    "targets": {
    …
    "notFound": {
      "viewName": "NotFound",
      "viewId": "notFound"
    }
}
```

detailObjectNotFound

If the object route matches – an ID is passed (for example `#/Objects/1337`) but the back end does not contain an object with the ID `1337`, then you need to display the *detailObjectNotFound* page. This is achieved by listening to the "change" event of a binding. Inside this, you check if there is no data and tell the router to display the *detailObjectNotFound* target, as shown in the sample code below:

```
// inside of a controller
this.getView().bindElement({
    path: "/Objects/1337",
    change: function () {
      // there is no data
      if (!this.getView().getElementBinding().getBoundContext()) {
          this.getRouter().getTargets().display("detailObjectNotFound
");
      return;
                }
      // code handling the case if there is data in the backend
      …
};
});
```

Busy Indication

The Master-Detail app implements a busy indication concept as specified by the SAP Fiori Design Guidelines.

Calling the app will result in the following:

- Only initially a global busy indicator is displayed that overlays the whole app until the metadata of the service is loaded.
- Afterwards, a local busy indicator is displayed on the master list and the detail page.
- When the detail page is loaded, the line item table on the detail page is set to busy until the line items are loaded with a separate service call.
- When controls are loading additional data or getting refreshed, a local busy indication is displayed automatically.

By default, the busy indicator delay is set to one second for all controls. This would first show the UI for a second, then show a busy indication until the data is loaded. To avoid this behavior initially and show the busy indicator immediately without delay, the following concept is implemented in the app: The `busyIndicatorDelay` and `busy` properties of certain controls (`AppView`, `List` on the *Master* page, `DetailPage` and `Table` on the *Detail* page) are bound to the local view model and manipulated in the controllers of the app. The delay is initially set to "0" for displaying the busy indicator immediately, and reset to the previous value after the initial loading is done.

You can simulate server delays to test this implementation running with mocked application data by using the URL parameter `serverDelay=true` in the hash. The default is set to 1000ms.

> **i Note**
>
> You can find more information about busy indicators, busy states, and busy handling in general in the SAP Fiori Design Guidelines.

Model Instantiation

The app configures several data models that are used throughout the app to update the views or to store additional configuration options.

The service model and the resource bundle are instantiated automatically by the app's component during startup and described in the first section. The local view models and helper models such as the device model are set up as JSON models and described in the second section.

Automatic Model Instantiation

The templates instantiate the service and resource model automatically using the following configuration entries in the descriptor. When the component of the app is initialized, these models will be made available under the configured name throughout the app.

An external service is defined in the `dataSources` section of the `sap.app` namespace. In the example shown below, we configure an OData V2 model and the alias `"mainService"` in the `manifest.json` descriptor file:

```
{
    …
    "sap.app": {
        …
        "i18n": "i18n/i18n.properties",
        …
        "dataSources": {
            "mainService": {
                "uri": "/here/goes/your/serviceUrl/"
                "type": "OData",
                "settings": {
                    "odataVersion": "2.0",
                    "localUri": "localService/metadata.xml"
                }
            }
        },
        …
    },
    …
}
```

> **i Note**
>
> If you use the OData V4 template, you set the `odataVersion` accordingly.

In the models section of the `sap.ui5` namespace we define two models that will be instantiated automatically. The resource model is a named model (i18n) and the OData model is the default model so it has no name. The OData model also receives additional URL parameters via the `metadataUrlParams`. The parameters `sap-server`, `sap-client`, and `sap-language` are passed to the service automatically by SAPUI5, as shown in the following `manifest.json` code snippet:

```
{
    …
    "sap.ui5": {
        …
        "models": {
            "i18n": {
```

```
                "type": "sap.ui.model.resource.ResourceModel",
                "settings": {
                    "bundleName": "sap.ui.demo.masterdetail.i18n.i18n"
                }
            },
            "": {
                "dataSource": "mainService",
                "preload": true
            }
        },
        …
    }
}
```

> **i** Note
>
> Before SAPUI5 version 1.30, all models were defined and instantiated in the component's `init` method. We recommend removing all manual model creation code and switching to the automatic model instantiation instead. The "device model" however is still a local model that has to be instantiated manually.

Additional Models for the App

The following models are created as local JSON models in the app and can be referenced by its model name where needed:

- **device**
  The device model provides an easy access to the sap.ui.Device API and is used to configure certain view settings according to the user's device.
- **FLP**
  The FLP model is a helper module to configure SAP Fiori launchpad (FLP) integration and is used to control the sharing options of the app.
- **masterView**
  A local view model for the `master` view that stored configuration options that are bound to controls in the view.
- **detailView**
  A local view model for the `detail` view that stored configuration options that are bound to controls in the view.
- **appView**
  A local view model for the `app` view that stored configuration options that are bound to controls in the view.

Related Information

Resource Bundle API
ODataModel V2
OData V2 Model [page 883]
Descriptor for Applications, Components, and Libraries [page 734]

Master List Filtering

You can use the following best practices when implementing search, sorting, filtering and grouping functions for a master list in your Master-Detail apps.

A search field is displayed in the master list to filter the list items for a custom keyword. In the header toolbar of the master list, options for sorting, filtering, and grouping are displayed. When searching or using one of the options in the header, the list content is updated automatically, and the search result is displayed.

All four options adjust the master list content (search, sort, filter, group) and are managed and applied in the logic of the master controller. This section describes the implementation details for these four options.

Search

The search is implemented in a manual mode and the list operation mode is "server". This means that the search has to be triggered explicitly by pressing enter or the search button, and the results are always fetched from the server.

The search function is implemented using the standard SAPUI5 `sap.ui.model.Filter` objects. The options are added to an internal state object of the controller and applied together with the filters that can be selected in the filter options. The type of these filters is "Application", and these filters are added on top of the predefined filters from the framework of type "Control".

The *Search* field also displays a *Refresh* button. Pressing this button triggers a simple refresh for the list binding.

Sorting, Filtering and Grouping

Sorting, filtering, and grouping can be implemented by using a semantic button that opens a `sap.m.ViewSettingsDialog` containing options for sorting, grouping, and filtering.

The event handlers that are called when selecting a sorting and grouping option are similar. They are implemented as an XML fragment with a `sap.m.ViewSettingsDialog` in a fragment. Therefore, we process the selected options in the handler of the dialog's `confirm` event. The event handlers create a `sap.ui.model.Sorter` object on the *key* field of the selected item. For the grouping functionality, a custom grouper is loaded and applied to the selected entry. Both sorting and grouping options are applied together on the binding of the master list. A `sap.ui.model.Filter` object is created for each filter option that has been selected in the dialog and applied together with the search option on the master list.

The filter message is automatically updated with the chosen filter texts. It is displayed on top of the master list and can be clicked to reopen the filter settings.

Related Information

SAP Fiori Design Guidelines: Master List

Send Email

The **Send Email** feature is a sharing option that can be found in the share menu of each view.

This feature simply triggers a `sap.m.URLHelper` action that will show a new email with preconfigured texts in the default client of the user.

The placeholder texts are located in the resource model and can be adjusted to your use case. The texts already include the current context and the current location. The texts may vary for each view, therefore they are configured with the local view model and updated when the business object context has changed.

For more information about `sap.m.URLHelper`, see the sample in the Demo Kit.

Testing

The templates include basic testing features, unit tests as well as integration tests for a basic test coverage of the initial app. The tests are written independently of the actual data displayed in the app.

The `webapp` folder of the template app contains a `test.html` file which serves as an overview for the different test pages. You can run the app with or without mock data and run the unit and integration tests. This section describes which application tests are provided and how they are structured.

Custom Mock Data

To run the automated tests below, we need a stable reference between the item on the master page and the items displayed in the line item table on the detail page. The mock server we use to simulate a backend system is capable of creating random entities suitable for testing, but it cannot foresee the dependency between the entity sets.

After creating this template, we therefore need to create mock data to successfully run all automated tests delivered with this template.

In SAP Web IDE, you can easily create mock data that fulfill the conditions above: Right-click on the `metadata.xml` file in the `localService` folder of your template project, and select *Edit Mock Data*. Select the entity sets that you have chosen for the master page and the detail page, and choose *Generate Random Data*. The generated files will be put in the folder `localService/mockdata`.

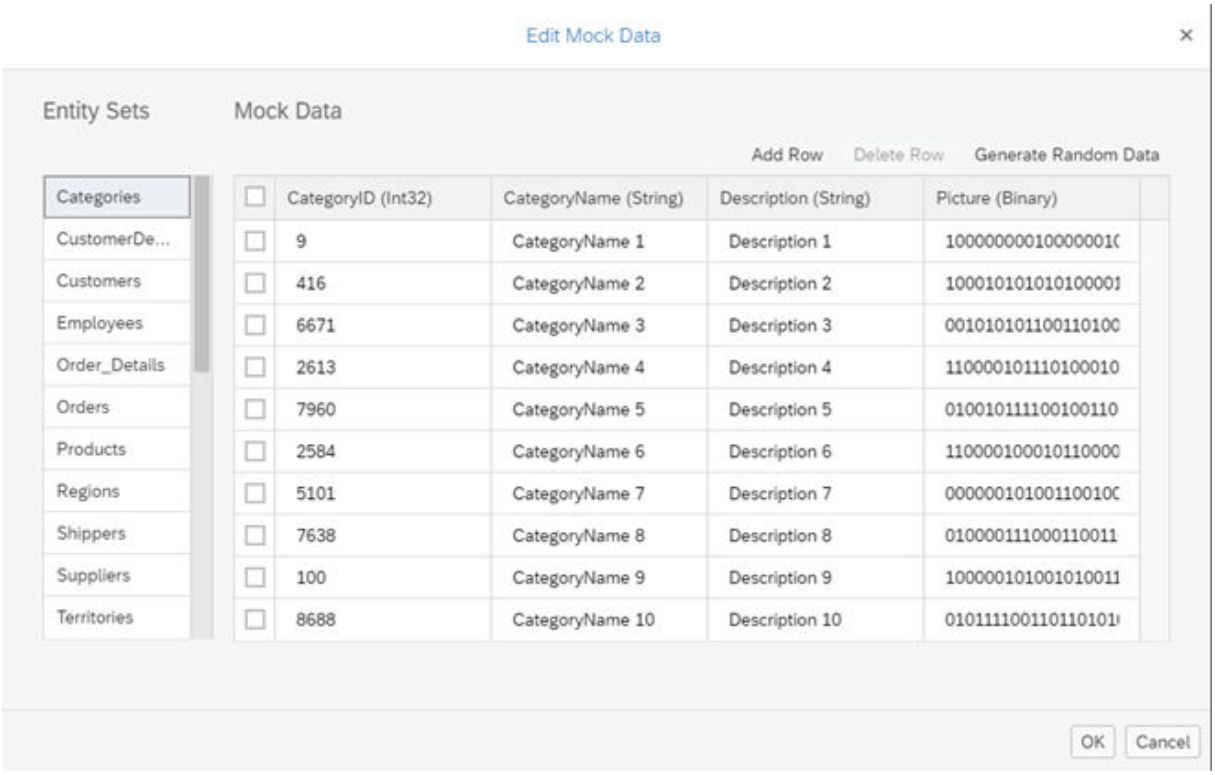


Figure 247: Screenshot of the Edit Mock Data dialog

You can also manually download a set of json files from the actual service by appending **`$format=json`** to the corresponding URL. Download the resulting files, and put them in the `localService/mockdata` folder within the template project.

For more information, see Step 2: Custom Mock Data [page 456] in our Worklist App tutorial.

Integration Tests

The integration tests shipped with the template cover all basic functionality and provide several "journeys". Journeys include a series of OPA tests that belong to the same functionality and should be executed together. Some of the journeys are implemented for both phone and desktop use cases to test device-specific interaction steps:

- `BusyJourney` / `BusyJourneyPhone`: This journey tests the busy indication features of the app for phone and other devices.
- `NavigationJourney` / `NavigationJourneyPhone`: This journey will trigger user interactions and navigate through the application. The routing configuration, basic navigation events, and error handling are tested here.
- `NotFoundJourney/NotFoundJourneyPhone`: Several "not found" cases of the application are tested here. Faulty navigation scenarios are introduced intentionally to simulate errors.
- `MasterJourney`: Tests for the *Master* page that check the search, sorting, filtering and grouping features built into the app.
- `FLPIntegrationJourney`: This journey is available if you have enabled SAP Fiori launchpad (FLP) for your app. It tests the FLP integration features *Save as tile* and *Share on SAP Jam*.

- `AllJourneys`: This is a convenience journey that will call all the other journeys specified above and is used in the test suite file.

You can execute all journeys by calling the test suite file `opaTests.qunit.html` or `opaTestsPhone.qunit.html` in the `webapp/test/integration` folder or selecting the *run all integration tests* link in the `test.html` file in the app's root folder.

For more information, see Integration Testing with One Page Acceptance Tests (OPA5) [page 1182] and sap.ui.test.Opa5 in the *Samples* within the Demo Kit.

Unit Tests

In the `unit` subfolder you can find all unit tests for our application. They are structured similarly to the structure of the `webapp` folder. For example, controller tests are located in the `controller` folder whereas formatter tests are located in the `model` folder.

Unit tests are included for the following functionality:

- ListSelector tests
- Formatters
- Device model

As with the integration tests, you can execute all unit tests by calling the test suite file `unitTests.qunit.html` in the `webapp/test/unit` folder or selecting the *run all unit tests* link in the `test.html` file in the app's root folder.

For more information, see Unit Testing with QUnit [page 1159], https://qunitjs.com/ and http://sinonjs.org/ .

Device Adaptation

The following section outlines the best practices for ensuring your master-detail apps adapt to different kinds of devices in the best way possible.

Content Density

The app templates include a mechanism to adjust the content density of the controls according to the device features. On devices that feature touch support, the controls are automatically displayed larger. For more information, see How to Use Densities for Controls [page 1146].

Stable IDs

Setting stable IDs is crucial if your app is used in combination with certain functions.

Most controls in the template apps (except for aggregations that are created dynamically, such as list items) are assigned a stable ID to identify the controls in integration tests, extensibility tools like key user adaptation, as well as interactive inline help tools.

Related Information

SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152]
Extending Apps [page 2143]
Stable IDs: All You Need to Know [page 1442]

Basic Template

The basic template is intended for all developers who want to start developing their own SAPUI5 app from scratch.

With this basic template you have a blank canvas to start coding right away. The basic file structure is set up according to our best practices.

> **i Note**
>
> This template does not include SAP Fiori launchpad (FLP) features and is intended for standalone use. If you want to convert it to a launchpad app you have to add some features manually, such as the *Save as Tile* feature.

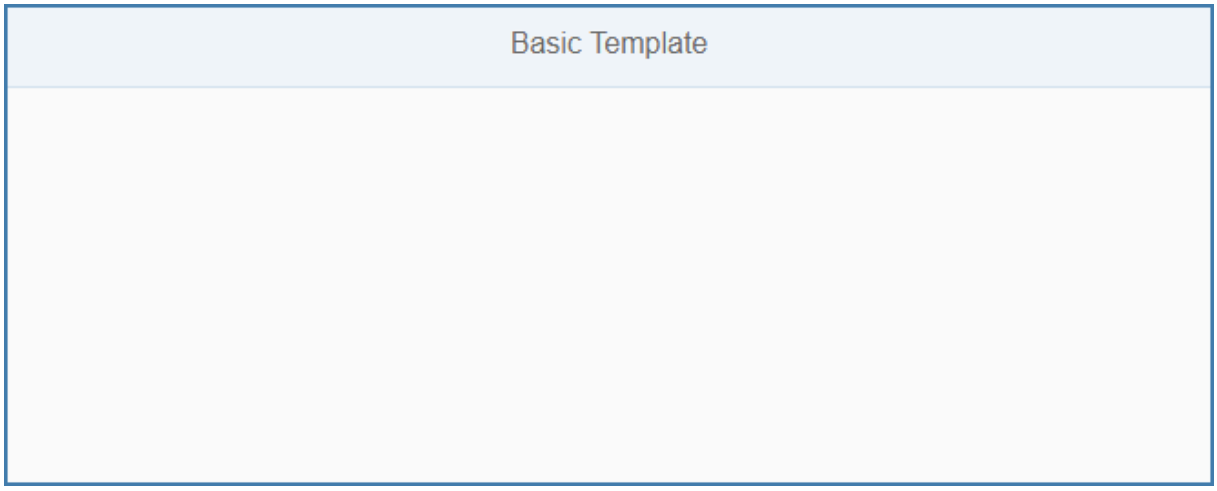


Figure 248: Screenshot of the Basic App

The `index.html` file defines the page that is displayed when the app is started. It is located in the `webapp` folder. It contains an XML view with a header and a title from the `sap.m` library as a starting point. You can easily modify the app to add more functionality.

Integrated Tests

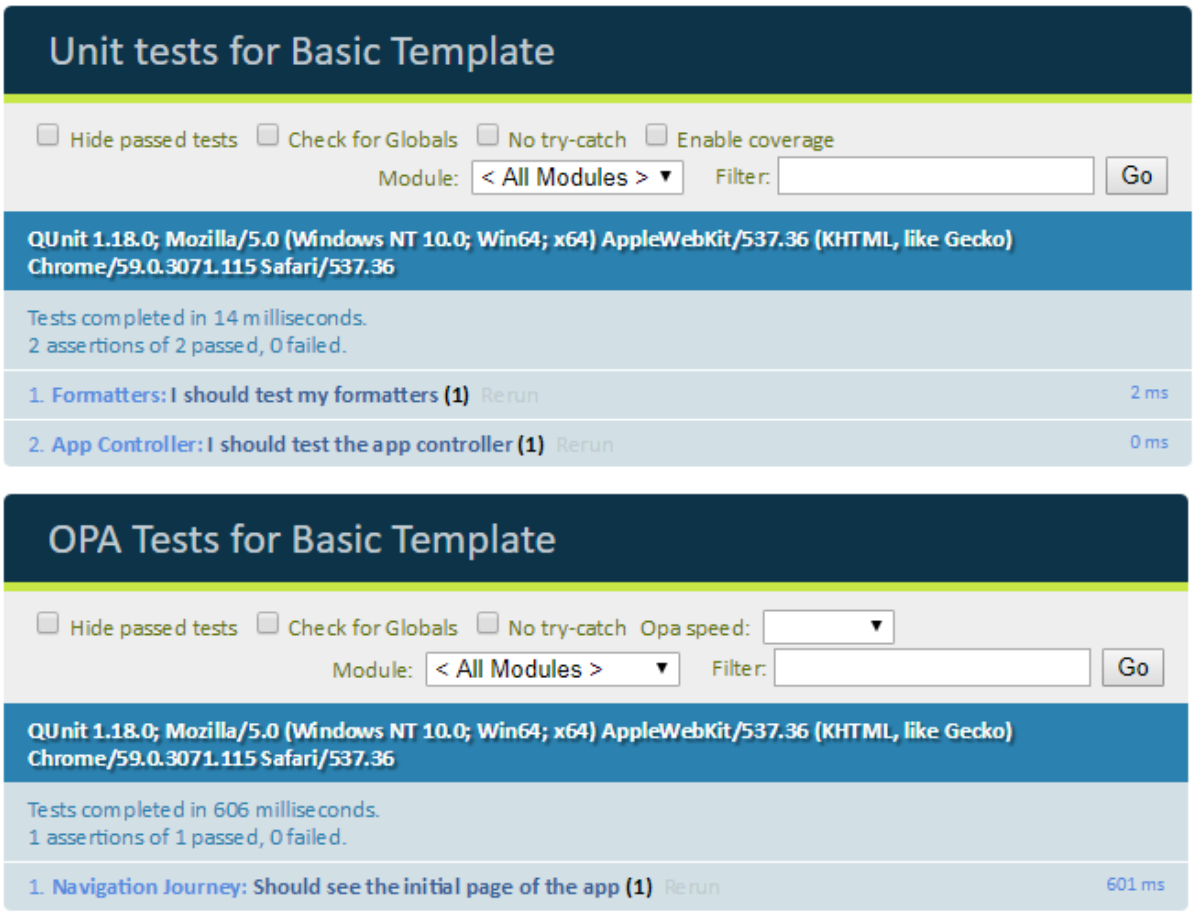


Figure 249: Test for the Basic App

An important best practice is to have unit and integration tests for your app. With this template, we have included sample tests that you can use: Tests on formatters and the app controller are the basic tests any app should cover. You can find them in the `test` subfolder of the `webapp` folder.

Where Can I Find the Basic Template?

You can find the template in the following places:

- **SAPUI5 Application** in SAP Web IDE
  For more information about SAP Web IDE, see the documentation for SAP Web IDE on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_Web_IDE.

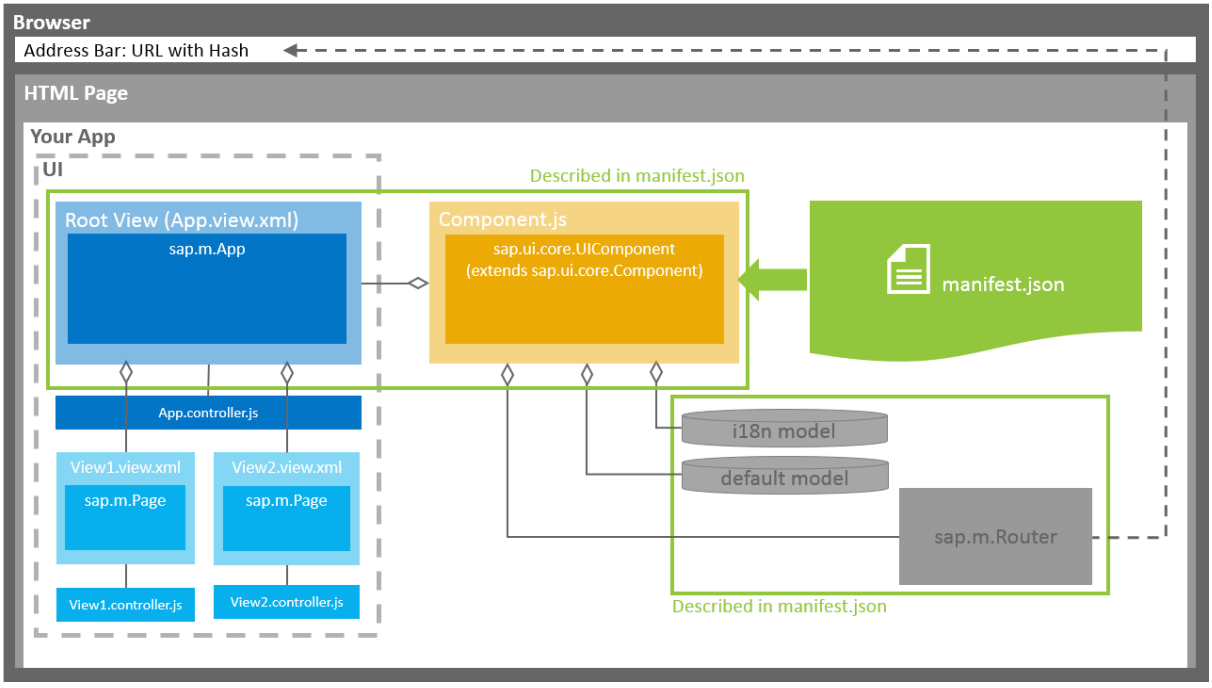
- **Basic Template** under Demo Apps.
- `openui5-basic-template-app` in the SAP Repository on GitHub .
  For more information about how to clone or download the template from GitHub, refer to the template documentation on GitHub .

Related Information

Demo Apps [page 671]
Development Environment [page 41]

App Overview: The Basic Files of Your App

We recommend creating at least three files for your app: the descriptor (`manifest.json`), the component (`Component.js`), and the main view of the app (`App.view.xml`).



Descriptor (`manifest.json`)

We recommend that you use the `manifest.json` file to configure the app settings and put all important information needed to run the app in there. Using this approach means you need to write less application code, and you can already access the information before the app is instantiated.

Some attributes in the descriptor are just for information purposes, such as the minimum SAPUI5 version (`minUI5version`), others help external components (for example the SAP Fiori launchpad (FLP)) to integrate

the application correctly, but most of the attributes are actually used to configure specific aspects of the app that are needed frequently.

The most important configuration settings are:

- **Models**. Examples of models are the configuration of the OData service (default model) and language files (i18n model). All models described in the `manifest.json` file are automatically instantiated when the app is started.
- **Libraries** and **components** that are used in the app and have to be loaded during app initialization.
- The **root view** of your application.
- **Routing** configuration that defines the navigation between views.

Root View (`App.view.xml`)

The `App.view.xml` file defines the root view of the app. In most cases, it contains an `App` control or a `SplitApp` control as a root control.

SAPUI5 supports multiple view types (XML, HTML, JavaScript, JSON). We recommend using XML views, as for these you have to separate the controller logic from the view definition in a controller file (for example `App.controller.js`).

We also recommend creating a separate view file for each view you want to use in your app.

Component (`Component.js`)

The `Component.js` file holds the app setup. The `init` function of the component is automatically started by SAPUI5 when the component is instantiated.

> ⚠ Caution
>
> Your component extends `UIComponent`. If you are overriding the init function of your component, you have to make sure that you call the `init` function of `UIComponent` and initialize the router afterwards.

In the metadata section of the component, you define a reference to the descriptor file. When the component is instantiated, the descriptor is loaded and parsed automatically.

HTML Page

All apps are started using an HTML page that loads SAPUI5 and the component. You have two options: You can build an app for the FLP or build a standalone app.

- **App for FLP**
  The FLP instantiates the component based on the information given in the descriptor file. The FLP can contain multiple apps at the same time. Each app can define local settings, such as supported themes or supported devices.

This app cannot be run standalone, meaning no `index.html` file is created but only HTML files for testing the app in the FLP sandbox.
For more information, search for *Embedding SAPUI5 Applications* in the documentation of your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

- **Standalone app**
  If you want to run your app standalone, you need to create an `index.html` file. Within this file, you instantiate the component.

Related Information

Folder Structure: Where to Put Your Files [page 1428]
App Initialization: What Happens When an App Is Started? [page 1427]
Descriptor for Applications, Components, and Libraries [page 734]
Model View Controller (MVC) [page 784]
Controller [page 807]
Views [page 787]
Models [page 882]
Routing and Navigation [page 1072]
Components [page 720]

App Initialization: What Happens When an App Is Started?

When a user starts an app (in the SAP Fiori launchpad (FLP) or using an HTML page), several steps will be performed in the background.

1. Component container loads the component (`Component.js`) of the app.
2. Component loads descriptor (`manifest.json`) that is referenced in the `Component.js` file.
3. Component creates models that are defined in the descriptor (default model and resource models (`i18n`)).
4. `init` function of `Component.js` is executed.
5. `init` function of the component calls init function of parent `UIComponent`. **(This has to be implemented by the app developer!)**
   `init` function of `UIComponent` creates router and root view as defined in descriptor.
6. Root view creates root control.
7. `init` function of component initializes router. **(This has to be implemented by the app developer!)**
8. Router creates necessary views depending on the hash in the URL with which the app has been started.
9. View loads corresponding controller.
10. `init` function of controller is executed.
11. Router places views in root control, now also models available within the view and its controller.
12. Bindings of views are evaluated.
13. Data is retrieved from model.
14. Views are updated.
15. User can interact with the app.

> **i Note**
>
> When a user closes the app, the `destroy` function of the component is called. All models and the router are destroyed. The router will take care of destroying the views.
>
> If a controller has created resources that need to be destroyed explicitly, for example non-aggregated controls, the app developer has to use the `onExit` function of the controller to free up resources. For more information, see Controller [page 807].

Related Information

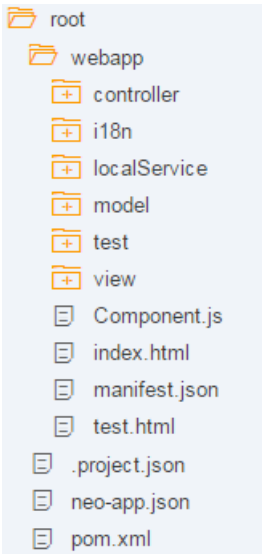
App Overview: The Basic Files of Your App [page 1425]

Folder Structure: Where to Put Your Files

The details described here represent a best practice for structuring an application that features one component, one OData service and less than 20 views. If you're building an app that has more components, OData services and views, you may have to introduce more folder levels than described here.

The 3 Main Folders

The 3 main folders in an application are the `root` folder, the `webapp` folder and the `test` folder. Regarding their structure, the `webapp` folder should be inside the `root` folder, and the `test` folder should be located inside the `webapp` folder, as shown below:



> **i Note**
>
> The image above shows a screenshot taken from SAP Web IDE, and is only meant to serve as an example. This applies to all images contained within this topic.

The `root` Folder

The root folder should contain files that are not part of your application coding. Examples are build configuration files, such as a `pom.xml` for maven or a `Gruntfile.js` for node/grunt, and documentation files like `readme.md` or `txt`. These files may also be grouped in folders if needed. For example, you could group all documentation files into a `doc` folder.

The `webapp` Folder

The `webapp` folder contains all the code that is related to the application. This means running and extending the application using the **extensibility** mechanism offered by SAPUI5. This includes the JavaScript files for the logic, view files written in xml, html, json or js format, and also files for **localization**, such as `i18n.properties` files. Any files that are only relevant for testing should be put inside the `test` folder. For more details about the `webapp` folder, see the section below. For more information about extensibility and localization, see Extending Apps [page 2143] and Localization [page 1269] respectively.

The `test` Folder

The `test` folder contains all of the files needed for running automated tests for your application, as well as for launching your application in a sandbox mode so that you can do manual testing. For more details about the `test` folder, see the section below.

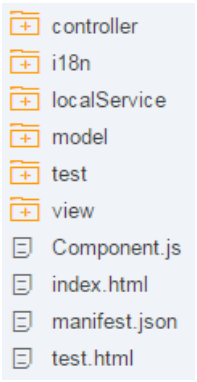
Why Use 3 Separate Folders?

To achieve sound performance when loading your application, the code you deploy to your production servers should only contain a `component-preload.js` and a `manifest.json` file. This means that when you create a package with a build, it is easier if all the files you really want to deploy are inside **one** folder. This is true no matter which build framework you use. We recommend using the `webapp` folder for this. Nothing inside the `root` folder is needed for running the app, so it's not included on a production server serving your application. The content of the `test` folder has to be executed in design time and during the automated test execution on a central server. We choose to include it inside the `webapp` folder, to be able to reference resources of the `webapp` folder relatively to the `test` folder. This folder has to be excluded when you are building a `component-preload.js`. You should never reference resources of the `test` folder from your application, because when you deploy to a productive environment, the resource cannot be loaded. For more information about the `manifest.json` file, see Descriptor for Applications, Components, and Libraries [page 734].

The `webapp` Folder in Detail

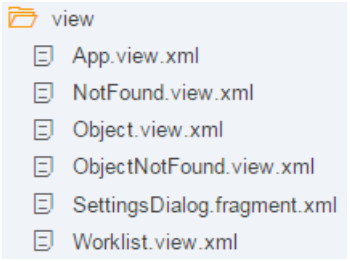
Aside from the `test` folder, the `webapp` folder contains 3 folders related to the MVC (model, view, controller) pattern used in SAPUI5, as well as a localization folder and a local-services folder used for emulating OData services. Each of these folders is outlined below.

For standalone app, this folder also contains an `index.html` file that is used to start the app and to instantiate the component. If your apps is built for the SAP Fiori launchpad no `index.html` file is created but only files for testing the app in the FLP sandbox.



The `view` Folder

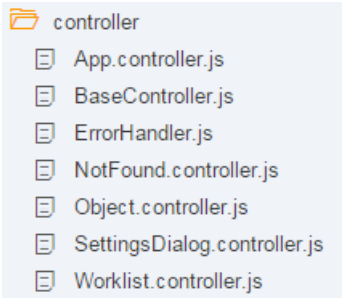
In the `view` folder, you should put all SAPUI5 **views** and **fragments**. This folder should not contain any application logic, so no JavaScript files unless you are using JavaScript views. JavaScript views are not recommended because it is easier to mix controller logic when building up a view. In declarative views this is not possible. In the example shown below, the `view` folder contains a mixture of views and fragments. If this folder gets too big, you might consider adding subfolders to group views by their semantics. In this example for instance, you could add a `detail` folder and move all views that are related to the detail area of your application to this subfolder.



For more information about views and fragments, see Views [page 787] and Reusing UI Parts: Fragments [page 1004] respectively.

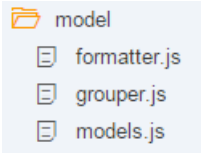
The `controller` Folder

The `controller` folder contains all the controllers used by your views, and might also contain additional logic files that are used by one or more controllers. The structure of the `controller` folder should mirror the `view` folder. If a view is in a subfolder, the controller of the view should also be in the corresponding subfolder.



The `model` Folder

The `model` folder is where you put any files needed for creating models and logic relating to model data. This includes grouping, filtering and formatting data.



In the above example, `models.js` is a factory for creating models that are used by our application.

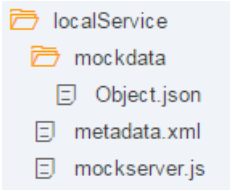
Localization Folder - `i18n`

We also have one dedicated folder for **localization** files. An SAPUI5 app will potentially be translated into many languages. Each of those languages has its own `.properties` file. Note that although the `ResourceModel` is an SAPUI5 model from a technical point of view, the localization folder is **not** part of the `model` folder. This is because the `.properties` files have a different semantic since they are used for translation. The code needed to instantiate the `ResourceModel` is located in the `model` folder. For more information about localization, see Localization [page 1269]. For more information about the `ResourceModel`, see the API Reference in the Demo Kit.

> **i Note**
>
> The path to the `i18n` file must not exceed 100 characters.

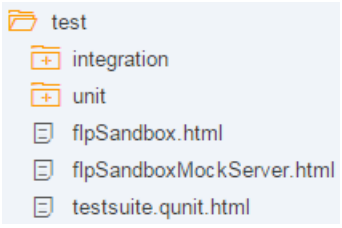
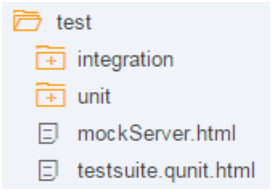
The `localService` Folder

The `localService` folder is used to emulate OData services for tests or as a preview mode for your application. It is also intended for design-time tools since it contains the `metadata.xml` file, which describes the backend connection of your application. You need to have one `metadata.xml` file per OData service, which exactly matches the remote service's metadata. The location of this file also needs to be maintained in the data sources section of the `manifest.json` file. For more information, see Descriptor for Applications, Components, and Libraries [page 734].

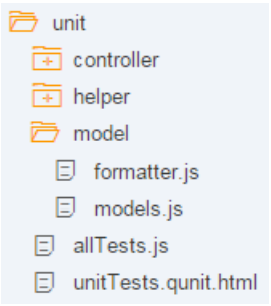


For integration tests, it is helpful if you are able to mock your backend with stable data. A second use case for this is for running an application in a **preview** mode so that it serves data locally instead of connecting to a backend. This is why this folder also contains files necessary for starting up a mock server. The data served by the mock server is put inside the `mockdata` folder. If you need to, you can also include multiple sets of mock data here, by giving each set its own folder. For more information about mock servers, see the API Reference in the Demo Kit.

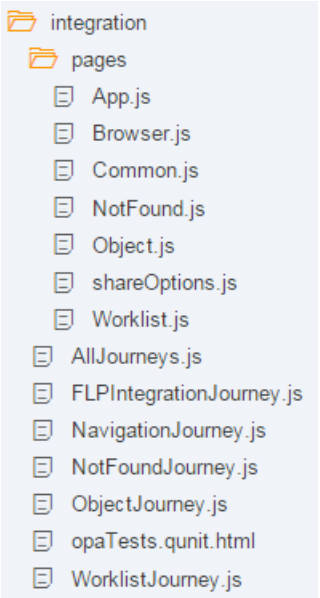
The `test` Folder in Detail

`test` folder for apps that are build for the SAP Fiori launchpad	`test` folder for standalone apps
	

The `test` folder contains three sets of files: files related to unit tests, files related to integration tests, and html files for either launching the tests or for testing the application manually. Inside the `unit` folder, the structure of the `webapp` folder is replicated for the files that are being tested.



In this example shown above, the files being tested are `webapp/model/formatter.js` and `webapp/model/models.js`. You may be using JavaScript files or html files to run your tests, depending on the runners you are executing your tests with. We recommend using `.js` files for writing your tests, so that you can run them with tools such as karma for instance.



The `integration` folder contains the OPA tests of your application. We decided to separate our unit and integration tests, since the execution time of the integration tests is much longer. While the unit tests run in less than 10 seconds, the integration tests run for over 2 minutes. These times will vary a lot depending on the size of your project. If the project grows and grows, the difference in the execution time will also grow. For this reason, we recommend making both kind of tests separately executable so that developers can choose which sets of tests they want to run during design time.

Device Adaptation: Using Device Models for Your App

Depending on the capabilities of the device on which the app is running, the functionality and the design of the application might differ. By introducing a local JSON model holding the device-dependent data, we can bind properties of our views to the device's capabilities.

As an example, on big screens (if the device is detected as a desktop device), it is not necessary to show the *Back* button on a detail view in a master-detail scenario, because the master and detail view are shown at the same time. You can control the visibility of the *Back* button with a property from the device model.

You need to make the `sap.ui.Device` API available in a JSON model once in your component to allow controls to be adapted to the current platform using data binding. The code below shows an example of how to achieve this:

`Component.js`

```
sap.ui.define([
    "sap/ui/core/UIComponent",
    "sap/ui/model/json/JSONModel",
    "sap/ui/Device"
], function (UIComponent, JSONModel, Device) {
[…]
init: function () {
// set the device model
    var oDeviceModel = new JSONModel(Device);
    oDeviceModel.setDefaultBindingMode("OneWay");
    this.setModel(oDeviceModel, "device");
[…]
}
```

This instantiates a named JSONModel (`"device"`) which contains all of the properties of the `sap.ui.Device` class, like browser, device type, or the current orientation of the screen. You can then bind the model properties in your views as follows:

`Master.view.xml`

```
<Page showNavButton="{device>/system/phone}" />
...
<PullToRefresh visible="{device>/support/touch}" />
```

If you want to negate a value of the device model or make a simple case decision, you can use the expression binding syntax as shown below:

`Master.view.xml`

```
<SearchField showRefreshButton="{= !${device>/support/touch} }" />
```

For more information, see sap.ui.Device in the *API Reference* in the Demo Kit, and the documentation under The Device API [page 1137].

Performance: Speed Up Your App

If a web app has performance issues, finding the cause can be both a time-consuming and nerve-consuming task. To help you avoid and solve performance issues in your app, here are some good practices we've discovered while dealing with SAPUI5 apps.

SAPUI5 apps are basically JavaScript files sent to a client by a server and interpreted by the browser. So it's not only the coding of the app that can cause slow performance. It often turns out, for example, that the configuration is wrong. Slow networks or servers may also have a heavy impact on the performance of a web app. Let's have a look at the most common issues that impact performance.

Enable Asynchronous Loading in the Bootstrap

Configuration issues are often caused by an old bootstrap or a wrong usage of the activated features. Here's an example of what a bootstrap should look like for an up-to-date SAPUI5 app:

```
<script
    id="sap-ui-bootstrap"
    src="/resources/sap-ui-core.js"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-compatVersion="edge"
    data-sap-ui-async="true"
    data-sap-ui-onInit="module:my/app/main"
    data-sap-ui-resourceroots='{"my.app": "./"}'
>
```

The most important setting is `data-sap-ui-async="true"`. It enables the runtime to load all the modules and preload files for declared libraries asynchronously, if an asynchronous API is used. Setting `async=true` leverages the browser's capabilities to execute multiple requests in parallel, without blocking the UI.

The attribute `data-sap-ui-onInit` defines the module `my.app.Main`, which will be loaded initially.

> **i Note**
>
> Configuration of the bootstrap can only be done for standalone applications and when the bootstrap is under control of the developer. The bootstrap of applications from a Fiori Launchpad is managed by the Launchpad.

> **i Note**
>
> The `data-sap-ui-async="true"` configuration option requires extensive testing as well as cooperation on the application side to ensure a stable and fully working application. It is, therefore, **not** activated automatically, but needs to be configured accordingly. If you encounter issues or want to prepare your application for asynchronous loading, see Is Your Application Ready for Asynchronous Loading? [page 689] The bootstrap attribute `data-sap-ui-async="true"` affects both modules **and** preload files. If it is not possible to load the modules asynchronously (e.g. for compatibility reasons), use `data-sap-ui-preload="async"` to configure at least the preloads for asynchronous loading. For further information, see Standard Variant for Bootstrapping [page 694].

If you listen to the `init` event as part of your `index.html` page, make sure that you implement the asynchronous behavior also here, as shown in the following code snippet:

```
<script>
    sap.ui.getCore().attachInit(function() {
        sap.ui.require(["sap/ui/core/ComponentContainer"],
function(ComponentContainer) {
            new ComponentContainer({
                name: "your.component",
                manifest: true,
                height: "100%",
                componentCreated: function(oParams) {
                    var oComponent = oParams.getParameter("component");
                    // do something with the component instance
                }
            }).placeAt("content");
        });
    });
</script>
```

> **Note**
>
> Please note that this variant with inline scripting is not CSP-compliant. It is better to create a module with `sap.ui.define` which contains the startup code and load it via `data-sap-ui-onInit="module:my/app/main"` (this usually also requires a declaration of `data-sap-ui-resourceroots`, e.g.: `data-sap-ui-resourceroots='{"my.app": "./"}'`).

> **Note**
>
> Applications without a descriptor file can declare additional dependencies explicitly via the bootstrap parameter `data-sap-ui-libs`. If those dependencies are not listed, such as transitive dependencies that are inherited from a listed library, SAPUI5 will load them automatically, but then has to first read the configured libraries and find out about these dependencies. This can take time as the application might benefit less from parallel loading.

Additional Information:

- For more information about bootstrap attributes, see Bootstrapping: Loading and Initializing [page 692]
- Walkthrough tutorial, Step 2: Bootstrap [page 72]
- Standard Variant for Bootstrapping [page 694]
- Best Practices for Asynchronous Loading in UI5

Ensure that Root View and Routing are Configured to Load Targets Asynchronously

Please check the `rootView` of the application's `manifest.json` file for an `async=true` parameter. This allows the root view to be loaded asynchronously.

To configure the targets for asynchronous loading, please also check the Routing Configuration [page 1074] for the `async=true` parameter.

```
"sap.ui5": {
    "rootView": {
```

```
        "viewName": "sap.ui.demo.walkthrough.view.App",
        "type": "XML",
        "id": "app",
         "async": true
    },
    "routing": {
        "config": {
            "routerClass": "sap.m.routing.Router",
            "viewType": "XML",
            "viewPath": "sap.ui.demo.walkthrough.view",
            "controlId": "app",
            "controlAggregation": "pages",
            "async": true
        }
    },
...
```

Additional Information:

- Walkthrough tutorial, Step 10: Descriptor for Applications [page 91]

Make Use of Asynchronous Module Loading (AMD Style)

If modules follow the Asynchronous Module Definition (AMD) standard and the bootstrap flag `data-sap-ui-async` is set to `true`, custom scripts and other modules can also be loaded asynchronously when a preload is not available. It will help you in the future to enable asynchronous loading of individual modules combined with the usage of HTTP/2 or AMD-based module bundlers. It also ensures proper dependency tracking between modules.

But it isn't enough to write AMD modules. You also need to prevent access to SAPUI5 classes via global names. For instance, do not use global namespaces like `new sap.m.Button()` but require the `Button` and call its constructor via the local AMD reference instead.

For more information, see the API Reference: `sap.ui.define`.

Always avoid usages of `sap.ui.requireSync` and `jQuery.sap.require` ! In order to enable modules to load asynchronously, use `sap.ui.define` to create modules (e.g. controllers or components) or `sap.ui.require` in other cases.

Please follow the Best Practices for Loading Modules [page 1100].

Use `manifest.json` Instead of the Bootstrap to Define Dependencies

Don't specify a link to the CSS in the bootstrap of your app; use the `manifest.json` descriptor file instead.

Please use the `manifest.json` application descriptor file to declare dependencies. This has several advantages:

- In the manifest, the dependency information is reusable; it works when the app runs standalone and when it is embedded in the Fiori Launchpad or some other launcher.
- Moving the dependencies to the manifest loads them later and can therefore make the first rendering happen earlier. Obviously, that first rendering cannot come from the component then.

- Design-time tools or runtime back-end services (e.g. AppIndex in ABAP systems) can use the manifest entries to determine the transitive closure of dependencies and thereby further optimise the parallel loading of dependencies. If the dependencies are maintained in the bootstrap, developers can do this by hand, but will have to update the information on each version upgrade.

Make sure that you don't load too many dependencies. In most apps it's enough to load the libraries `sap.ui.core` and `sap.m` by default, and add additional libraries only when needed.

If you want to make additional libraries generally known in your app, without directly loading them during the app start, you can add them to the dependency declaration in the `manifest.json` file with the `lazy` loading option. This makes sure that the libraries are only loaded when they are needed:

```
"sap.ui5": {
    "dependencies": {
        "minUI5Version": "1.70.0",
        "libs": {
            "sap.ui.core": {},
            "sap.m": {},
            "sap.ui.layout": {
                "lazy": true
            }
        },
...
```

If a library preload contains reuse components and this preload is configured to be loaded lazily (via `"lazy": true` in the dependencies of the `manifest.json`), the library is not available upon creation of the related component.

In this case you need to use `sap.ui.getCore().loadLibrary("my.library")` before creating the component (e.g with `Component.create({ name: "my.component" })` or component usage `myComponent.createComponent("myUsage")`).

An indicator that a component is inside a library is the existence of an entry `sap.app/embeddedBy` in its `manifest.json` file.

Additional Information:

- Descriptor for Applications, Components, and Libraries [page 734]

Load SAPUI5 from the Content Delivery Network (CDN)

In order to ensure that all static SAPUI5 resources are served with the lowest possible latency in SAP Cloud Platform scenarios, you can load the resources from the Content Delivery Network (CDN) cached by AKAMAI. Especially when running your app in the cloud, you benefit from the global distribution of servers. For other scenarios, it is possible to configure a custom CDN of choice as an external location.

Additional Information:

- Variant for Bootstrapping from Content Delivery Network [page 696]

Ensure that all Resources are Available to Avoid 404 Errors

Provide i18n files for all languages used in your application. See: Identifying the Language Code / Locale [page 1269]

Use "manifest first" to Load the Component

Load the `manifest.json` descriptor file of the component first to analyze and preload the dependencies when loading the component. For more information, see Manifest First Function [page 735].

```
// "Component" required from module "sap/ui/core/Component"
// load manifest.json from default location and evaluate it before creating an
instance of the component
Component.create({
  name: "my.component",
});
```

Ensure that Library Preloads are Enabled

If the library preloads are disabled or not found, every module is loaded separately by an own request. Depending on the server and network infrastructure, this can take a lot of time. Except for debugging reasons, it is always recommended to make sure library preloads are used. Fortunately, the library preloads are active by default if the files are present.

In some cases it may happen that preloads are disabled:

- The `data-sap-ui-preload` bootstrap attribute is empty or set to an invalid value. The attribute is optional and only necessary if the loading behavior (sync / async) needs to be overwritten manually.
- Debug sources are enabled in the bootstrap (`data-sap-ui-debug=true`) or via the URL (`sap-ui-debug=true`).

Ensure that Application Resources are Loaded as Component Preload

Application modules (e.g. components, controllers, views or resource bundles) should be loaded asynchronously via the component preload file. Check (e.g. via the Network tab in the Google Chrome developer tools) if a component preload (`Component-preload.js`) is missing. If the application is not configured to load modules asynchronously, required application files may be loaded synchronously.

> **i Note**
>
> If a component preload does not exist yet, the bundle needs to be created. For example, you may use the UI5 Build Tooling .

Check the Network Requests

To quickly check the network load caused by your app, look at your browser's developer tools, for example the Network tab in the Google Chrome developer tools (`F12`). You'll see an overview of all requests being sent. Possible issues here may be:

Synchronous requests that block each other

In this case, use the `data-sap-ui-async="true"` setting in the bootstrap.

Too many requests

You can use the UI5 Build Tooling to bundle and minimize all relevant component files by creating a component-preload file.

If you're using apps with grunt as a web server, you can use the `openui5_preload` task; for more information see Optimizing OpenUI5/SAPUI5 Apps in the SAPUI5 Developer Center on SAP SCN.

If you're using SAP Web IDE, refer to Application Build in the SAP Web IDE documentation.

Back-end related performance issues

- Slow database service (e.g. OData)
- Slow web server or CDN issues (e.g. serving of static resources)
- Slow network infrastructure (e.g. mobile network)
- The h2 protocol is not supported (only HTTP/1.1); ideally, the h2 protocol should be supported by the web server

Additional Information:

- To determine the minimum required bandwidth when using UI5-based applications, you can find further information in SAP Note 2240690 on front-end network bandwidth sizing.

Migrate `jquery.sap.*` Modules to their Modularised Variants

Since UI5 version 1.58, the global `jquery.sap.*` modules are deprecated. Please use the modularised variant of the module. If you are still using the `jquery.sap.*` variants, a so-called "stubbing layer" may load the old module synchronously!

You can find a list of modules in the Legacy jQuery.sap Replacement [page 1109] documentation.

The usages can either be replaced manually or by the UI5 Migration Tool .

> **i Note**
>
> Please make sure to declare the required modules in `sap.ui.define` or `sap.ui.require` to ensure that they get loaded asynchronously.

Migrate Synchronous Variants of UI5 Factories to Asynchronous Variants

Check if the application uses synchronous UI5 factories. Many asynchronous variants are available, e.g. for Components, Resource Bundles, Controllers, Views and Fragments. Please visit the following overview:Legacy Factories Replacement [page 1124].

Use the OData V2 Model Preload

Components can preload models for which modules are already loaded; otherwise a warning will be shown. The ODataModel V2 benefits especially, because the metadata can be loaded in parallel during a component load.

```
"sap.ui5": {
  ...
  "models": {
      "mymodel": {
          "preload": true,
...
```

For more information, see Manifest Model Preload [page 781].

Use OData V2 Metadata Caching

To ensure fast loading times for **SAP Fiori applications started from the SAP Fiori launchpad**, the OData metadata is cached on the web browser using cache tokens. The tokens are added with the parameter `sap-context-token` to the URL of metadata requests. Please check via the developer tools of your browser (e.g. the Network tab in the Google Chrome developer tools) if the token has been appended to the request URL.

> **i** Note
>
> This feature is only supported by OData V2 for SAP Fiori applications.

> **i** Note
>
> Please consider switching to the OData V4 Model [page 918] for improved performance.

Additional Information:

- Cache Buster for OData Metadata of SAP Fiori Apps
- Scheduling Update of OData Metadata Caching

Check Lists and Tables

The performance limits are reached differently depending on the used browser, operating system and hardware. Therefore, it is important to be mindful about the amount of controls and data bindings. This applies especially to lists and their variants (e.g. `sap.m.Table` or `sap.ui.table.Table`):

- If a table needs to display more than 100 rows, please use `sap.ui.table.Table` instead of `sap.m.Table` The reason for this is that `sap.m.Table` keeps every loaded row in memory, even if not visible after scrolling. To choose the right table variant for your requirements, check out the documentation about Tables: Which One Should I Choose? [page 2286]
- If the table rows contain multiple controls and/or custom-data fields, please check if they are required, or if another control can replace them. For example, another list like a ComboBox inside of a table cell may create many controls for every row, which can be very expensive.
- Check tables for hidden columns and load only the visible ones, if possible.

Additional Information:

- Performance of Lists and Tables [page 2350]

Further Code Optimization

You can further optimize your code by doing the following:

- Use asynchronous view loading as described here: Instantiating Views [page 805].
- Use the OData V4 model, which has an improved performance over the OData V2 model.
  Visit the OData V4 Model [page 918] documentation and ensure that all required features are available.
  For a quick start, follow the OData V4 [page 261] tutorial.
- If you use data binding with an OData V2 service as a back end, you should consider switching your OData model to our more updated OData V2 model. For more information, see OData V2 Model [page 883].
- Optimize dependent bindings as described here: Optimizing Dependent Bindings [page 893].
- Avoid the usage of `setTimeout()` calls with values greater than `0`. This usually indicates an anti-pattern in application code that is used as a workaround and should be avoided. For more information, see also JavaScript Code Issues: Don't use timeouts [page 1463].
- Don't use visibility for lazy instantiation. For more information, see Performance Issues: Don't use visibility for lazy instantiation [page 1467].
- Please ensure the application does not block the rendering while waiting for back-end requests to respond. Waiting for data before rendering anything is not the favored user experience. It is recommended to load data asynchronously and already render the page while the request is pending. Mostly, the requests won't fail, and if they do, it is better to show an error or to navigate to an error page.
- If an `XML Preprocessor` is used, we recommend to use the XML View Cache [page 797]. If configured in the XML View and with a properly implemented key provider (for invalidation), it is able to cache already processed XML View Preprocessor results.

Related Information

Coding Issues to Avoid: Performance Issues [page 1467]
Performance Measurement Using sap/ui/performance/Measurement Module [page 1377]
Blog: SAPUI5 Application Startup Performance – Best Practices
Blog: SAPUI5 Application Startup Performance – Advanced Topics

Stable IDs: All You Need to Know

Stable IDs are IDs for controls, elements, or components that you set yourself in the respective `id` property or attribute as opposed to IDs that are generated by SAPUI5. *Stable* means that the IDs are concatenated with the application component ID and do not have any auto-generated parts.

Background

If you don't define IDs, SAPUI5 generates them dynamically. These IDs are not static and might differ from program run to program run. For example, the page and table in the following XML view could have the generated IDs `__page0` and `__table0` at runtime:

```
<mvc:View
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page>
        <content>
            <Table>
            </Table>
        </content>
    </Page>
</mvc:View>
```

The generated IDs change whenever the control structure of the app changes. The sequence of instantiation also plays a role: If there are two views with unstable IDs in the app, depending on the order the views are opened, they get the generated IDs `__view0` and `__view1`. This is an issue for the following features that require stable IDs:

- SAPUI5 flexibility
  Allows you to adapt apps based on your requirements, for example, by creating variants or changing the user interface at runtime. Stable IDs are used to identify the controls that are to be adapted. For more information, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152].
- Automated tests
  To check the behavior of apps at runtime, these tests find controls by searching for stable IDs. If you use OPA in SAPUI5, you're able to find controls via other criteria like control type, display name and others. For more information, see Integration Testing with One Page Acceptance Tests (OPA5) [page 1182].
- Inline help tools
  These tools display user assistance information directly in the app and depend on stable IDs (example: Web Assistant).

> → Tip
>
> Stable IDs are an important prerequisite for SAPUI5 flexibility services, automated testing, and inline help tools, such as Web Assistant. Apps with stable IDs are of high quality and offer customers more functionality. Therefore, we strongly recommend that you use stable IDs whenever possible (some technical controls don't need stable IDs, such as `CustomData`).

How to Set IDs Manually to Keep Them Stable

→ Tip

Using the rule *Stable control IDs are required for SAPUI5 flexibility services* in the Support Assistant, you can check whether all controls use stable IDs. For more information, see How to Check If All Your IDs Are Stable [page 1449].

Views	• Views in the descriptor for applications, components, and libraries
	The standard use case is that you use stable IDs for the view that the router navigates to. Ideally, instead of creating the views yourself, you create them with routing targets and declare the view ID in the manifest.json file as shown in the example below. For more information, see Routing and Navigation [page 1072] and Descriptor for Applications, Components, and Libraries [page 734].
	Example:
	`{ "sap.ui5": { "rootView": { "viewName": "<your namespace>.view.Root", "id" : "rootView", "async": true, "type": "XML" } ... "routing": { ... "targets": { "myTarget": { "viewName": "MyView", "viewId": "myView" } } } } }`
	• Embedded views
	If you embed your view, set its ID (such as `myEmbeddedView`).
	Example:
	`<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc"> <Page id="page"> <mvc:XMLView id="myEmbeddedView" viewName="MyView" async="true" /> </Page> </mvc:View>`
	• Programmatic creation
	If you create the view programmatically, provide the ID as one of the parameters to the constructor or factory function. Make sure to prefix the view ID with the component ID using the `createId` method of the owner component.
	Example:
	`// "XMLView" required from module "sap/ui/core/mvc/XMLView" XMLView.create({ id: <component>.createId("myProgrammaticView"), viewName : "<your namespace>.view.ProgrammaticView" }).then(function(oView){ // code });`
	For more information, see namespace sap.ui.

Extension points	If you use extension points, use stable IDs for nested views and prefixes for nested controls of a fragment.

Controls

- Controls in XML views
  The XML view prefixes the control IDs (only the defined IDs, not the automatically created ones) with its own ID. This allows you to use the same control ID for different views and the same view multiple times. For more information, see Support for Unique IDs [page 814].
  If the following XML view is instantiated using the ID `myView`, the contained page and table would have the IDs `myView--page` and `myView--table` at runtime:

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <Page id="page">
        <content>
            <Table id="table">
            </Table>
        </content>
    </Page>
</mvc:View>
```

- Programmatic creation
  For JavaScript views and JavaScript-generated controls you must use the `createID` method of the view or component. Here's how it could look like when you're creating a control directly in the control code:

```
// "Button" required from module "sap/m/Button"
new Button({
    id : oView.createId("ConfirmButton"),
    text : "Confirm"
});
```

Components

> **Note**
>
> The following is only relevant if you do not use the SAP Fiori launchpad because it instantiates components for you and provides IDs.

For example, if you instantiate a component inside an HTML page, set the ID of the component as shown below. The reason for this is that components could be displayed more than once on a page. To get unique IDs for the views and controls inside the component, they must be prefixed with the component ID. All views in the component that are created by the framework are automatically prefixed with the component ID. As described above, for the programmatically generated components, you must do it yourself.

Example:

```
// "Shell" required from module "sap/m/Shell"
new Shell({
   app: new ComponentContainer({
      height : "100%",
      name : "sap.ui.demo.worklist",
      settings: {
         id: "worklist"
      }
   })
}).placeAt("content");
```

> **Note**
>
> Only if there's more than one component in an app, the component container requires a stable ID by setting the component container to `autoPrefixId`. For more information, see sap.ui.core.ComponentContainer.

Embedded Components

If you want to add an embedded component with a stable ID, you have two options:

1. Option: Add a component re-use entry in the application component's manifest.json. Let's say you want to add an embedded component with the name `embeddedComponent.name`. You define it as follows in the application component's manifest.json file:

```
"sap.ui5": {
   "componentUsages": {
      "reuseName": {
         "name": "embeddedComponent.name",
         "settings": {
            "id": "embeddedComponentID"
         }
      }
   }
}
```

   Now you can use the re-use entry name defining the component container in XML:

```
<core:ComponentContainer
   usage="reuseName"
   async="true"
   id="embeddedComponentContainerID"
   propagateModel="true" //to propagate models from the
application component
/>
```

2. Option: Add an embedded component independently from the manifest and mention the correct namespace using the `name` property. Also, when instantiating the component, make sure that the `id` property is set during component instance creation:

```
<core:ComponentContainer
   name = "embeddedComponent.name"
   async="true"
   id="embeddedComponentContainerID"
   propagateModel="true" //to propagate models from the
application component
/>
```

   Inside the embedded component's `constructor()` (with the `embeddedComponent.name` namespace) add :

```
[...]
constructor: function() {
                    arguments[0].id = "embeddedComponentID";

UIComponent.prototype.contructor.apply(this, arguments);
}
[...]
```

   Alternatively, you could use `sap.ui.core.Component.create()` and specify the `id` property as part of the arguments. For more information, see the API Reference: `sap.ui.core.Component/methods/sap.ui.core.Component.create`.

> **Note**
>
> In order to support SAPUI5 flexibility features, all embedded components should have a stable ID. For more information, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152].

XML fragments

If you are using XML fragments in your app, make sure they are instantiated using the correct view ID prefix.

Example:

```
// "Fragment" required from module "sap/ui/core/Fragment"
Fragment.load({
    id: this.getView().getId(),
    name: "my.fragment.SampleFragment"
});
```

> **Note**
>
> If some controls have disappeared after a software upgrade or the way in which they can be identified has been changed, this has a direct impact on the functions that depend on stable IDs. For this reason, the IDs, which are part of the public API of the app, must be kept stable over the life cycle of the app.

How to Name Stable IDs

Choose names for your stable IDs that describe the semantics of your views and controls, such as *page* or *table*.

> **Note**
>
> For the allowed sequence of characters, see the namespace sap.ui.core.ID. But bear in mind not use hyphens (-) as separators in your names as they would interfere with the ones that are added automatically by the framework.

Example:

If you build an app using the following stable IDs for the component and the views using the SAP Fiori Worklist Application template, here's what the concatenated IDs that are generated at runtime look like:

Component	Views	Contained views	Concatenated IDs
myProducts	**worklist**	**page**	myProducts---worklist--page
		table	myProducts---worklist--table

Component	Views	Contained views	Concatenated IDs
	product	**page**	myProducts---product--page
		objectHeader	myProducts---product--objectHeader

For more information about the SAP Fiori Worklist Application template, see Worklist Template [page 1400].

How to Check If All Your IDs Are Stable

With the Support Assistant, you can analyze whether there are any issues with the stable IDs used in your app. Here's how you can check this:

1. Open your app in a browser.
2. Choose CTRL + SHIFT + ALT + P to start the Support Assistant.
3. In the *Technical Information Dialog*, choose *Activate Support Assistant*.
4. In the table on the left, deselect all rules.
5. Click on the *Rules* column.
6. Filter for **stable** and choose *Enter*.
7. Select the *Stable control IDs are required for SAPUI5 flexibility services* rule.
8. Choose *Analyze*.

If any generated IDs are found, set the IDs for these controls manually as described here [page 1443].

Related Information

Support Assistant [page 1339]

Reacting on User Input Events

A handler can be used to validate, parse, and format issues.

You register the handler and can then use the following functions of `sap.ui.getCore()`:

- `attachFormatError`
- `attachParseError`
- `attachValidationError`
- `attachValidationSuccess`

You can also register for these events directly on the control or any parent control where the event is fired. The corresponding event bubbles up to the Core if it is not canceled in the event handler.

Related Information

Validation Messages [page 1065]

SAPUI5 Flexibility: Enable Your App for UI Adaptation

Here's what you have to consider when developing apps that support UI adaptation.

UI adaptation is a feature of SAPUI5 flexibility that allows key users without technical knowledge to easily make UI changes for all users of an app, end users to personalize controls, and developers to extend the UIs of SAPUI5 applications. For more information, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152].

You can enable your application for UI adaptation by following a few simple steps.

Use the `flexEnabled` flag

This flag in the `manifest.json` indicates that your application is enabled for UI adaptation. As of app descriptor schema version 12 (`_version: 1.11`), which was introduced with SAPUI5 version 1.56, the `flexEnabled` flag in the `sap.ui5` section of the `manifest.json` indicates if the application supports UI adaptation.

If you set this flag, make sure that you provide stable IDs and follow the rules and guidance on this page.

Example

Please note that the `_version` property must be at least `1.11.0`.

```
{
    "_version": "1.11.0"
    […]
    "sap.ui5": {
        "flexEnabled": true,
    […]
}
```

> ⚠ Caution
>
> If you set the flag to `false`, your application won't work with any UI adaptation tool. In certain scenarios it can make sense to do this, for example, for a beta release when you still expect major changes for the app. However, do not change the `flexEnabled` flag to `false` **after** your application has been delivered. This would lead to incompatibilities and regressions if a key user has adapted the UI in the meantime.

→ Tip

In case you want certain parts of your app to remain unchanged, you can restrict adaptation for the corresponding controls, see Restrict Adaptation for Certain Controls [page 1455].

Very important: Use stable IDs

Stable IDs are used to identify the controls that can be changed via UI adaptation or personalization. For this reason, every control and view that you use must have a stable ID. For more information, see Stable IDs: All You Need to Know [page 1442]. Here, you also find information on how to check if all your IDs are stable.

If your app is based on SAP Fiori elements, it automatically generates IDs for its controls. You only have to set stable IDs if you use extension points to modify it, for example to add an additional view.

⚠ Caution

Never change the IDs when you're creating the next version of your app if the app was already delivered. Otherwise you'd risk inconsistencies and the loss of UI changes that a key user might have done in the delivered version.

→ Tip

With the `Stable control IDs are required for SAPUI5 flexibility services` rule in the Support Assistant, you can check whether all controls have stable IDs. For more information, see Stable IDs: All You Need to Know [page 1442].

Use SAPUI5 controls supported by UI adaptation

More than 60 controls, such as forms, pages, tool bars and input elements support UI adaptation, and the number is growing.

If you want to know whether a control is enabled for key user adaptation, go to the Samples. Pick a control, open the sample, and choose (Adapt UI) at the top right. If UI elements in the sample are highlighted when you hover over or select them, it's an indicator that the control is enabled for UI adaptation. You can also try out the UI adaptation features directly in the sample.

i Note

The table, filter bar and chart controls from the `sap.ui.comp` library do not support key user adaptation. However, end users can personalize these controls on the UI and save individual views. If you use a `sap.ui.comp.VariantManagement` control, your users can also share personalized views with other end users. For more information see Smart Variant Management [page 2457].

Controller Code

Normally, UI changes should win over binding and application coding. However, there is no distinct point in time when all controls are available. This consequently means that there is also no single event when all changes have been applied, because, for example, new changes can be introduced through user interactions.

The following mechanisms ensure that changes usually win over application coding:

- For controls in the XML view, changes are applied during the XML view processing.
- For controls that are created by your controller code, for example controls that are part of a group and are later placed inside an existing control, the changes are applied when the control is added to the SAPUI5 control tree.

However, with too many complexities or dynamics in the controller code, you can inadvertently overrule changes done by the key user. Example: If a key user removes (i.e., hides) a control and your controller code overwrites its visibility property afterwards, the key user change doesn't have an effect. If you cannot avoid this case because of some app-specific reasons, you should at least restrict the availability of certain actions as described under Restrict Adaptation for Certain Controls [page 1455].

Keep in mind that the positions of controls can change if the key user moves them on the UI. Controls might end up in different aggregations and at different index positions from what you specified. Therefore, you should only access controls by their ID.

> **i Note**
>
> If you access `ObjectPageSections` and controls within `ObjectPageSections`, they might be stashed when the key user removes them. This means that the control is not processed and therefore not instantiated. Your code shouldn't break in this case.

It's okay to create additional controls dynamically, for example, as a result of user interaction. If you dynamically add controls to the UI, make sure to set all properties before you add them to an aggregation of a parent control, ideally as part of the constructor. With this, you prevent that changes that key user made to the control are overridden.

> **→ Tip**
>
> Use data binding whenever possible to express UI state changes.

How can I change my app without breaking the key user's changes?

In general, you can change your app without any problems, as long as you keep the following in mind:

- Don't change control IDs. If you change control IDs, the app will still start, but any existing changes related to this ID will not be applied anymore.
- Don't change view hierarchies, because the control IDs depend on the view IDs or fragment IDs.
- Don't remove controls that had a stable ID before.

Something isn't working like it should?

Please check the troubleshooting information.

> ⚠ Caution
>
> UI changes, similar to any UI coding, cannot replace back-end validations and authorizations. For example, when an action is hidden via a change, the change processing could be blocked or the action could be reached by the browser console, so that the action could still be triggered.

For more information, see Troubleshooting [page 1457].

You can also check out our Enabling UI Adaptation: Other Things to Consider [page 1453] page for additional tips and tricks..

For more information about how key user adaptation works for key users and how it gets enabled in the SAP Fiori launchpad, search for **`Adapting SAP Fiori UIs at Runtime`** and **`Enabling UI Adaptation at Runtime`** in the documentation for your SAP NetWeaver version on the SAP Help Portal.

Related Information

SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152]
Descriptor for Applications, Components, and Libraries [page 734]
Stable IDs: All You Need to Know [page 1442]
Adaptation Projects for SAP Fiori Elements-Based Applications

Enabling UI Adaptation: Other Things to Consider

Find out how you can enable UI adaptation exactly how you need it.

How to improve the performance

Use asynchronous loading of views

To enable processing of UI changes directly on XML views, use the asynchronous loading of views. For more information, see Instantiating Views [page 805].

> → Tip
>
> Using the rule `Asynchronous XML views in the Support Assistant`, you can check whether asynchronous loading is used for all views. For more information, see Support Assistant [page 1339].

Use view caching

If you use view caching, our views are stored in the cache after UI changes have been applied. The view (including the UI changes) is then loaded from the cache. For more information, see XML View Cache [page 797].

Configure the variant management control

If users should be able to save their UI changes as different views (control variants), you have to configure the `sap.ui.fl.variants.VariantManagement` control.

All you have to do is add this control to an appropriate location in your app, and assign the desired UI container as a target in a `for` association.

Enable the key user to add additional UI elements

To enable the key user to make additional properties of an OData entity visible (typically when the key user adds fields or custom fields), you have to do the following:

- Use data binding in the container where you want to enable this.
- Make sure that the related OData model is the default model of your app component. For more information, see Assigning the Model to the UI [page 998].
- If a property of an OData entity shouldn't come up in key user adaptation, for example, because it's a technical field, you should set the annotation `sap:visible=false`. If the property is only relevant under certain circumstances, you can provide the `field-control` property and set the field to be hidden. For more information, see the official annotations documentation.
- If you don't have OData models or if you want to give the key user the option to enable more complex UI parts, you can deliver hidden controls (`visible="false"`). These can then be made visible via key user adaptation.

Reuse Components

If you are using reuse components, dialogs or popovers inside your application, they must have the application component as owner component (`type=component`). Make sure to instantiate it using a `runAsOwner` function.

About Reuse Components

During key user adaptation, only the specific UIs/applications are adapted, but not the reuse components themselves. Key user adaptation follows a WYSIWYG approach. Since the use cases can vary from app to app, the UI changes done inside a reuse component of one app won't be applicable for another app where the same reuse component is used. Therefore, any changes to the UIs inside a reuse component will be stored only in the context of the app in which it was embedded.

Restrict adaptation for certain controls

It can be a good idea to exclude certain controls from key user adaptation. Examples:

- Controls for standard actions that should not be **removed** (for example, the *Close* button)
- Controls that should not be **changed**, because their properties are changed dynamically by the application coding, which will consequently overwrite key user changes

To achieve this, you need instance-specific design time metadata. You attach these metadata via the `sap.ui.dt` namespace and the `designtime` attribute to the `xml` node of the control you want to restrict, or via `CustomData`. There are three types of restrictions:

- The control cannot be **changed** on the respective instance level (recommended, for example, if the control properties are changed dynamically by the application coding).
  Required metadata: `not-adaptable`
- The control cannot be **changed** on the respective instance level as well as on all children of that instance.
  Required metadata: `not-adaptable-tree`
- The control cannot be **removed** or **revealed** (i.e. added) (recommended, for example, for standard actions).
  Required metadata: `not-adaptable-visibility`

> **i Note**
>
> Controls that have already been delivered with previous app versions should *not* be switched to `not-adaptable`, `not-adaptable-tree`, or `not-adaptable-visibility` later. Reason: Setting these metadata later will not affect existing changes and might even cause regressions.

The process to define the design time metadata depends on your scenario:

XML View

In this case, you need to specify metadata via the `sap.ui.dt` namespace. Example:

```
<core:View
    ...
    xmlns:sap.ui.dt="sap.ui.dt"
>
    ...
    <SomeControl sap.ui.dt:designtime="<path>/<name.designtime" />
    <SomeOtherControl sap.ui.dt:designtime="not-adaptable" />
    <AnotherControl sap.ui.dt:designtime="not-adaptable-visibility" />
    <ContainerControl sap.ui.dt:designtime="not-adaptable-tree" />
        <ChildControl> <!-- this is also not adaptable -->
            <AnotherChildControl /> <!-- this is also not adaptable -->
        </ChildControl>
    </ContainerControl>
    ...
</core:View>
```

Control is instantiated by JavaScript code

In this case, you need to provide the instance-specific design time metadata as custom data. Example:

```
new SomeControl({
    //other settings
    customData :  [new CustomData({
        key : "sap-ui-custom-settings",
```

```
        value : {
            "sap.ui.dt" : {
                "designtime": "not-adaptable" || "not-adaptable-visibility" ||
"not-adaptable-tree"
            }
        }
    })];
});
```

When are key user changes applied?

As described under Controller Code [page 1452], two mechanisms apply:

- For controls in the XML view, changes are applied during the XML view processing.
- For controls that are created by your controller code, such as controls that are part of a group and are later placed inside an existing control (e.g. `placeAt` to place a control inside a node of the DOM), the changes are applied when the control becomes part of the SAPUI5 control tree.

Can I check for stable IDs during automatic testing?

Yes. You can integrate corresponding check from the Support Assistant rule into your Opa5 tests as described in Integrating the Rules in OPA Tests [page 1349].

Can I do manual testing?

To test key user adaptation in your SAP Fiori launchpad sandbox (locally or in the SAP Web IDE preview), include the following script in the HTML file that you use for sandbox testing. With these configurations, you should be able to start key user adaptation as usual.

```
<script type="text/javascript">
    window["sap-ushell-config"] = {
        defaultRenderer : "fiori2",
        bootstrapPlugins: {
            "RuntimeAuthoringPlugin" : {
                "component": "sap.ushell.plugins.rta",
                config: {
                    validateAppVersion: false
                }
            }
        }
    }
</script>
```

Troubleshooting

Click the sections below, to find tips and solutions to the most common problems.

Cannot add OData property

Check the following:

- Is the field control invisible? If yes, the field doesn't get displayed in the list of available fields.
- Ensure that the appropriate entity type is bound to the control.

I've changed an UI element, but the change is not everywhere applied

UI elements that you change in one place might not be changed in all other places where they are used. Reason: Key user adaptation allows to change controls, not data or metadata. For example, if you rename a label defined in the oData metadata (annotations), the label is not changed everywhere in the app, and you must rename each occurrence individually.

The control has a stable ID, but the new group I've created doesn't

If you create a control ID within fragments or controller code, the ID of the parent view might not be part of the control ID. Thus, the control can have a stable ID, but the view doesn't. As the view ID is needed to create the ID for a new container when using *Create Group*, the view ID has to be stable (otherwise the container ID is not stable).

UI Adaptation cannot be started, because the application version is missing or incorrect

This error refers to the `applicationVersion` attribute in the app descriptor (`manifest.json` file). If this mandatory attribute is missing or invalid, the UI cannot be adapted.

For more information about the app descriptor and its attributes, see Descriptor for Applications, Components, and Libraries [page 734]

My key users and/or end users get a message that says the app is not enabled for adaptation or personalization

Reason: In the `sap.ui5` section of the `manifest.json`, the `flexEnabled` flag is set to `false`. To enable adaptation/personalization for your users, set this flag to `true`.

> → Tip
>
> For more information about the FAQ for users of key user adaptation, search for **`Something Isn't Working like It Should?`** in the documentation for your SAP NetWeaver version on the SAP Help Portal.

Related Information

SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]

Coding Issues to Avoid

This section lists some of the most important issues that should be avoided when creating applications using SAPUI5, split into different categories for reasons of simplicity.

JavaScript Code Issues

This section lists some of the most important issues that should be avoided when writing JavaScript code in SAPUI5.

Don't use methods or properties that are not public

Don't use or override "private" methods or properties. Private functions are typically (but not always) prefixed with "_".

Use "protected" methods or properties only if you access it from the object itself or an object that extends that object. (For example as we do in Step 19: Reuse Dialogs [page 114] of the *Walkthrough* tutorial.) In the API Reference, protected functions are indicated by a label *Visibility: protected* below the description of the function.

Always double check in the API Reference. If SAPUI5 changes the implementation in a future release, your code will break if you fail to follow this guideline.

Table 58: Examples

Bad Examples	Good Example
`var sText = oControl.mProperties["text"];`	`var sText = oControl.getText();`
`oSelectDialog._oList.setGrowing(false);`	
`var sPart = oEvent.oSource.oBindingContexts.description.sPath.split('/')[3];`	

For more information, see Compatibility Rules [page 17] and the API Reference.

Don't use references to global names

Use only local variables inside the AMD factory function, do not access the content of other modules via their global names, not even for such fundamental stuff like `jQuery` or `sap.ui.Device`. You can't be sure that the modules are already loaded and the namespace is available.

Bad Example

Access the modules directly:

```
sap.ui.define(['sap/m/Button'],
function(Button) {
    var fnCreateContent = function() {
        // global reference on
sap.m.Input, which might not be
loaded yet
        return new sap.m.Input({
            color: ...,
        });
    };
});
```

Good Example

Declare a dependency to `sap.m.Input` within `sap.ui.define`:

```
sap.ui.define(['sap/m/Input'],
function(Input) {

    var fnCreateContent = function() {
        // reference sap.m.Input via
a dependency
        return new Input({
            color: ...,
        });
    };
});
```

Exceptions

SAPUI5 provides a couple of static modules and (factory) functions that can be referred to via their global name:

- `sap.ui.define`
- `sap.ui.require`
- Factory functions and core references:
  - `sap.ui.getCore`
  - `sap.ui.component`
  - `sap.ui.fragment`
  - `sap.ui.htmlfragment`
  - `sap.ui.jsfragment`
  - `sap.ui.jsview`

- `sap.ui.template`
- `sap.ui.view`
- `sap.ui.xmlfragment`
- `sap.ui.xmlview`

- Commonly used names (However they can also be used as AMD references via `sap/ui/Global`):
  - `sap.ui.getVersionInfo (Global.getVersionInfo())`
  - `sap.ui.lazyRequire`
  - `sap.ui.resource`
  - `sap.ui.version`

Don't use deprecated APIs

Entities marked as “deprecated” in the API Reference documentation (this includes properties, methods, events, and their parameters as well as entire controls and other APIs) are no longer intended to be used. They will not get feature updates in the future. Alternatives, if available, are described in the API Reference documentation.

One prominent example is the old `jQuery.sap.device` API that has been replaced with `sap.ui.Device`.

For more information, see the Deprecated APIs.

Don't override or add control methods

If you override methods like `onBeforeRendering`, `onAfterRendering`, or getters and setters, the original methods will no longer be called. You have to make sure that you call them in your method explicitly. Even if they are not implemented right now, they could be added in the future. This applies to control inheritance in particular.

Instead, you should consider using delegates.

Table 59: Examples

Bad Examples	Good Example
`oControl.onAfterRendering = function() { // do something };`	`oControl.addEventDelegate({ onAfterRendering:function() { // do something } });`
`oControl.prototype.setText = function() { ... };`	

See also: sap.ui.core.Element - addEventDelegate.

Don't manipulate the DOM structure within controls

Manipulating the DOM structure of controls rendered by SAPUI5 can result in undesired behavior and only has a temporary effect. Changes will be overridden after the next rerendering or the DOM might change in a future version of SAPUI5, which can break your code. In addition, your DOM changes could break the code of the SAPUI5 control if it relies on a certain structure.

If you need to manipulate the DOM of an SAPUI5 control, attach a delegate to the `afterRendering` hook of the control, safeguard your code against DOM changes, but still be prepared to have to rework your code at any time when the DOM structure (which is in no way guaranteed to remain stable!) changes. The adaptation should be covered by your automated tests.

Even `onAfterRendering` may not be called when a control handles certain property changes without complete rerendering.

Table 60: Examples

Bad Examples	Good Example
`oControl.$().find(".sapMLabel")[0].innerHTML = "reallybad";`	`oControl.addEventDelegate({ "onAfterRendering": function() { var $label = oControl.$().find(".sapMLabel"); if (/* sanity check whether the change still makes sense */) { // TODO: re-test after UI5 updates, create automated test $label.text("Better"); } }});`
`oControl.$().find(".sapMLabel").remove();`	

Don't attach DOM event handlers

Use `attachBrowserEvent()` if you need to listen to any DOM event on SAPUI5 controls. An even better approach is to use `addEventDelegate()` for the most important event types instead, as it avoids additional event registrations and listens to the regular SAPUI5 event dispatching.

If you are creating event handlers in custom controls, you can use listen to DOM events directly, but make sure that the listeners are properly deregistered in `onBeforeRendering()` and in `exit()`, and registered in `onAfterRendering()`.

Good example for arbitrary events:

```
oControl.attachBrowserEvent("mousemove", function() {
    // do something
});
```

Good example for wide but limited selection of browser events:

```
oControl.addEventDelegate({
```

```
        onmouseover:function() {
            // do something
        }
    });
```

See also: sap.ui.core.Control - attachBrowserEvent and sap.ui.core.Element - addEventDelegate.

Don't create global IDs (when running with other views or apps)

When you create JSViews or applications that will be running together with views or applications from other sources (that are not owned by you), or JSViews that will be instantiated several times in parallel, you must not create stable IDs for your controls, fragments, or views in SAPUI5. Doing so might result in duplicate ID errors that will break your app. Especially when running together with other apps, there could be name clashes or other errors.

Use the `createId()` function of a view or controller instead. This is done automatically in XMLViews and JSONViews. The `createId()` function adds the View ID as a prefix, thus recursively ensuring uniqueness of the ID (for example: `__page0--__dialog0`).

Table 61: Examples

Bad Example (Inside a JSView)	Good Example (Inside a JSView)
`createContent: function(oController) { var btn = new sap.m.Button("myBtn", {text: "Hello"}); return btn; }`	`createContent: function(oController) { var btn = new sap.m.Button(this.createId("myBtn"), {text: "Hello"}); return btn; }`

See also: sap.ui.core.mvc.View - createId.

Don't forget about control lifecycle management

SAPUI5 controls are kept alive until they are destroyed, so lifecycle management of controls is important since multiple apps can be opened and closed in the same user session. Controls that are not destroyed cause memory leaks and may slow down the browser after prolonged use.

Also clean up internal structures in controllers, views and your custom controls.

See also: sap.ui.core.Element - destroy (for applications) and sap.ui.core.Element - exit (for custom control implementation).

Don't hard code or concatenate strings that need to be translatable

Hard coding UI strings will exclude them from translation. In addition, concatenating translatable strings in applications might lead to errors in internationalization: the texts in question might have a different translation order in other languages and will then be syntactically wrong.

Table 62: Examples

Bad Example	Good Example
Using separate texts like " you selected " and " items " in the translation file to construct sentences like: " you selected " + 10 + "items ". This would lead to a wrong word order in languages where the verb needs to be at the end of the sentence, for example.	Using a complete sentence including a placeholder in the translation file: " you selected {0} items ". This allows translators to change the word order and the position of the inserted placeholder value.

Don't forget about proper "this" handling

For developers new to JavaScript, it is often confusing to understand how the "this" keyword behaves. In event handlers in particular, but also for other callback functions, the "this"-pointer must be used correctly, so make sure you check what it actually refers to. Without proper usage of the execution context, unexpected results can occur (this-pointer might be the global window object or a different control).

Don't use `console.log()`

There is a native browser API available for logging errors and warnings in the developer console of your browser (`console`). Calling it directly is not recommended as it doesn't allow control over the amount of log entries that are created and it provides no criteria to associate a log entry with a specific topic or software component. Instead, add a dependency to the `sap/base/Log` module and use its methods to write log entries, for example `Log.error` or `Log.warning`. Create a dedicated logger for a topic or use the `sComponent` parameter of the log calls to assign the log entry to a topic. Use `Log.setLevel()` to define the minimum severity to be logged.

Note that most errors and warnings in the developer console thrown by the SAPUI5 framework are potential bugs in your application and must be analyzed thoroughly!

Table 63: Examples

Bad Example	Good Example
`console.error("Logon failed");`	`Log.error("Logon failed", "", "connectivity";)`

See also: Namespace sap/base/Log.

Don't use timeouts

Executing logic with timeouts is often a workaround for faulty behavior and does not fix the root cause. The timing that works for you may not work under different circumstances (other geographical locations with greater network latency, or other devices that have slower processors) or when the code is changed. Use callbacks or events instead, if available.

Table 64: Examples

Bad Example	Good Example
`jQuery.ajax("someData.json");` `setTimeout(fnProcessResults, 300);`	`jQuery.ajax("someData.json").done(fnProcessResults);`

Don't build apps without reasonable automated tests

This should not come as surprise, but it is very difficult to refactor or modify apps that do not have any (or have bad) automated test cases. There are substantial risks when QUnit and OPA tests are missing in applications.

CSS Styling Issues

This section lists some of the most important rules relating to CSS styling in SAPUI5.

SAPUI5 controls are styled with CSS, and as applications can provide their own CSS, they can adapt the styling. However, the deeper such adaptations are, the more likely is it that they break with future SAPUI5 updates or with other libraries and apps getting involved. If you follow the rules listed below, you will reduce the risk of this happening.

Don't override control class styling directly

SAPUI5 does not guarantee the stability of style class names across versions. If the naming of style classes is changed in future versions, styling rules will no longer be applied to targeted elements. In addition, overriding control class styles directly might lead to style clashes when applications are run in shared runtime environments (like SAP Fiori launchpad).

Add your own namespaced classes instead.

Table 65: Examples

Bad Example	Good Example

`.sapMInputBaseError {`
`  font-weight: bold;`
`}`
	Add a custom CSS class to the control in those situations where you want additional styling:
	`oButton.addStyleClass("poaAppError");`
	Then provide the style for this class:
	`.poaAppError {`
	`  font-weight: bold;`
	`}`

Don't style DOM element names directly

Styling DOM elements directly will lead to unpredictable results, as SAPUI5 does not guarantee the stability of the inner-control DOM-tree over time. In addition, this might lead to styling clashes when applications run in shared runtime environments (like SAP Fiori launchpad) or when custom HTML is added. It is better to limit styling changes to specifically used CSS classes.

Table 66: Examples

Bad Example	Good Example
`div { width: 120px; }`	`.myStyleClass { width: 120px; }`

Don't use generated IDs in CSS selectors

SAPUI5 applications can create dynamic IDs for elements. Do not use these IDs as selectors in custom CSS as they can change over time. It is better to add and use CSS classes instead.

Table 67: Examples

Bad Example	Good Example
`#__view1__button0 { font-weight: bold; }`	Add a style class as described above and then define the following: `.myEmphasizedButton { font-weight: bold; }`

Don't create selectors that are not namespaced

Custom selectors and CSS classes that are not namespaced might lead to style clashes in shared runtime environments like SAP Fiori launchpad, or when other JavaScript libraries are included that might use the same CSS class names.

Table 68: Examples

Bad Example	Good Example
`.title { font-weight: bold; }`	`.poaAppTitle { font-weight: bold; }`

Don't use hard-coded colors, font sizes and images if the app should be themable

Themability of applications relies on LESS calculations within the SAPUI5 theme CSS. Hard-coding colors, fonts and images in applications and custom controls means that these colors are not modified by theming, which leads to design issues or accessibility issues (for example, in the High Contrast Black (HCB) theme). You can use special CSS classes instead that are supplied by these LESS calculations.

Table 69: Examples

Bad Example	Good Example
`.myCustomHTML { color: #FFF; background-color: blue; }`	Add the CSS classes `sapThemeTextInverted` and `sapThemeHighlight-asBackgroundColor` to your custom HTML element.

See also: CSS Classes for Theme Parameters [page 1262].

Don't use theming parameters for attributes they were not intended for

SAPUI5 applications come with a built-in set of parameters which are used for theme-dependent styling, mainly for colors. They are accessible using the `sap.ui.core.theming.Parameters.get()` API (and for library builds using the OpenUI5 build mechanism, also in the *.less files in control libraries). These theme parameters have descriptive names, meaning that by looking at a parameter name, you can see the usage it has been defined for.

To ensure that you do not use combinations of theme colors which may clash after future theme changes, do not use background colors for fonts or vice versa, for example, and do not use border colors for anything else but borders.

Table 70: Examples

Bad Example	Good Example
`var sColor = sap.ui.core.theming.Parameters.get("sapUiButtonBorderColor"); $(oSomeDomElement).css("background-color", sColor);`	`var sColor = sap.ui.core.theming.Parameters.get("sapUiButtonBorderColor"); $(oSomeDomElement).css("border-color", sColor);`

See also: Namespace sap.ui.core.theming.Parameters.

Performance Issues

This section lists some of the most important issues that should be avoided to improve performance in SAPUI5 applications.

Don't use visibility for lazy instantiation

When an application has areas that are not visible initially, or if only one of multiple options is visible at a time, **do not** create all UI controls and set most of them to non-visible! If you do, SAPUI5 will instantiate and initialize all of those controls, which consumes unnecessary time and memory, even when they are not rendered. On top of this, the data binding will also be initialized, which may trigger back-end requests that are not needed at this stage. The impact is particularly big when the parts of the UI that are not visible initially are complex or numerous.

Please note that lazy loading of views can be achieved with routing. For more information, see Routing and Navigation [page 1072] and Step 10: Implement "Lazy Loading" [page 338] of the Navigation and Routing tutorial.

> ### Example
>
> An application needs to display a `Panel` containing a `Table` in **display mode**, but the user can switch to **edit mode** to modify data, in which case a different `Panel` needs to be shown. Especially when using XML views, it is tempting for application developers to specify two panels in the view XML and set the `Panel` with the editable table to `visible="false"`. The *Edit* button could then just toggle visibility of both panels.

The following XML view is easy to handle, but leads to suboptimal performance when the `editPanel` has a lot of content.

View:

```
<mvc:View xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
controllerName="my.own.controller">
    <Page>

        <Panel id="displayPanel" headerText="Display Data">
            <Table...>
        </Panel>

        <!-- edit panel is initially hidden, but still instantiated -->
        <Panel id="editPanel" headerText="Edit Data" visible="false">
            <Table...>
        </Panel>

        <Button text="Edit" press="toEditMode"/>
    </Page>
</mvc:View>
```

Controller code:

```
toEditMode: function() {
    this.byId("displayPanel").setVisible(false);
    this.byId("editPanel").setVisible(true);
}
```

The following code is better in terms of initial performance because the second table is created lazily when the user switches to edit mode.

View:

```
<mvc:View xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m"
controllerName="my.own.controller">
    <Page>

        <!—only the initially needed display panel -->
        <Panel id="displayPanel" headerText="Display Data">
            <Table...>
        </Panel>

        <Button text="Edit" press="toEditMode"/>
    </Page>
</mvc:View>
```

Additional fragment named `EditPanel.fragment.xml` for content that is initially hidden:

```
<Panel xmlns="sap.m" id="editPanel" headerText="Edit Data" visible="false">
    <Table...>
</Panel>
```

Controller code:

```
toEditMode: function() {
    this.byId("displayPanel").setVisible(false);

    var oEditPanel = this.byId("editPanel");
    if (!oEditPanel) { // load and instantiate the edit panel lazily
        // instantiate the Fragment:
        // giving the View ID as ID will prefix the IDs in the Fragment and
allows using this.byId(…) in the Controller
        // giving "this" (the Controller) allows using controller methods from
within the Fragment
        oEditPanel = sap.ui.xmlfragment(this.getView().getId(),
"myApp.EditPanel", this);
        this.byId("myPage").insertContent(oEditPanel, 0); // for sake of
simplicity inserts at position 0
    }
    oEditPanel.setVisible(true);
}
```

In other scenarios, at the time of developing you may not know which UI part is displayed initially. In this case, you can define that the UI is empty (showing none of the panels) in the view definition, and the controller's `onInit()` method decides which fragment to instantiate and display initially:

```
onInit: function() {
    var oPanel;
    if (bEditMode) {
        oPanel = sap.ui.xmlfragment("myApp.EditPanel");
    } else {
        oPanel = sap.ui.xmlfragment("myApp.DisplayPanel");
    }
    this.byId("myPage").insertContent(oPanel, 0);
}
```

> **i Note**
>
> Although the example above shows an XML view and an XML fragment, the problem and the solution apply to all view types.

Please also note that this guideline is not set in stone: If the hidden UI elements are just small or few in number, using fragments would not help but add additional overhead instead. Having said that, creating several big tables and displaying only one of them is **not** a good idea. There is no definite rule where to draw the line, it depends on many factors like application size, number of libraries being loaded, and additional data requested by those hidden controls. If in doubt, you can test the performance using the performance tracing tools in the browser's developer console with the controls in question being hidden, against them being removed.

See also: Reusing UI Parts: Fragments [page 1004].

Related Information

Performance: Speed Up Your App [page 1434]

Securing Apps

The following section provides information about security aspects of SAPUI5. The information is intended for SAPUI5 application and control developers, as well as to system administrators running applications based on SAPUI5.

SAPUI5 is a client-side JavaScript library, so while the library itself is designed and tested to be secure, it cannot ensure the application to be secure. Unlike WebDynpro, where the application is built against an abstract programming model and the framework handles the HTML rendering, JavaScript code and communication with the browser, in SAPUI5 the application controls the HTML output and provides its own JavaScript code. This code is executed on the client and it handles client/server communication.

While this brings a lot of freedom and possibilities for the application, it comes with a lot of responsibility with regards to security. Application developers need to understand the security threats and actively prohibit exploitation. Also important is the correct configuration of the used HTTP server.

Moreover, common security mechanisms, which are usually taken for granted, like user authentication, session handling, authorization handling, or encryption are not part of SAPUI5 and need to be handled by the server-side framework and/or custom code of the application.

Further Reading

SAPUI5 is not bound to any server implementation or server-side programming language and can, thus, be used with SAP NetWeaver AS for ABAP, Java, HANA XS Engine, or any standard web server. Therefore, the corresponding Security Guides also apply to SAPUI5.

> **i Note**
>
> We highly recommend that you implement SAP Note 1582870 for ABAP XSS escaping support.

Browser Security

Browser security comprises several topics such as cross-site scripting, clickjacking, and local storage.

A browser is, by design, an untrusted client: A server cannot rely on any information sent from a browser, as a malicious user can use a JavaScript debugger to tamper with the client code, or a proxy server like fiddler to modify request data. Input validation on the client is just for convenience purposes as the server always has to validate the data again that is received from the client.

Browsers also offer possible attack vectors, such as Cross-Site-Scripting (XSS). The application has to take care of these.

Cross-Site Scripting

Cross-Site-Scripting has become the most prominent security issue of web applications within the last years and also the most dangerous one, as it allows several ways of exploitation. Once malicious code is running within your browser, it can be used to steal your session cookies, to trigger requests within the current session, or even to exploit a known browser vulnerability to do native code execution.

For SAPUI5 applications, XSS vulnerabilities can exist on different levels:

- Within the HTML page or custom data transports sent to the browser from the server
- Within the JavaScript code of the application, which is processing server responses
- Within the HTML renderers of SAPUI5 controls

SAPUI5 can only prevent cross-site scripting in the processing and rendering of controls. For that purpose, input validation exists for all typed element properties and output encoding is done in the renderer class of controls. However, there are exceptions for controls that are especially built to include arbitrary HTML, for example, `sap.ui.core.HTML`.

The application is responsible for the following tasks:

- Proper output encoding of all content embedded in the HTML page itself
- Encoding JSON or XML data sent to the client
- Secure processing of the JSON/XML data
- Security of custom controls provided by the application

For more information, see Cross-Site Scripting [page 1475].

Clickjacking

Clickjacking, or UI redressing, tricks the user into triggering actions within an application by redirecting clicks. This is done, for example, by using an invisible iFrame that is positioned above a fake UI. When the user clicks on something on the fake UI, the content of the invisible iFrame handles the click.

SAPUI5 provides a way to prevent clickjacking since version 1.28.0. This has to be configured, as needed, by the application. For more information, see Frame Options [page 1478].

> **i Note**
>
> As of version 1.28.0, you no longer need to use the Business Add-In `/UI5/BADI_CONFIG_HTTP_HANDLER` to configure the X-Frame-Options response header (SAP Note 2075016). The SAPUI5 framework now handles clickjacking prevention and the add-in solution won't work on all browsers.

For more detailed information on clickjacking, refer to the following SAP Note: 2319727

HTML5

HTML5 offers a lot of new functionality, which also brings a lot of potential new security issues. This is just an overview of some of the new features and possible security issues when they are used.

Local Storage

All browsers are now offering a local storage API. This API can be used to store a limited amount of data on the browser. Access to this data is limited to JavaScript code running from the same domain as it has been stored. SAPUI5 offers helper functions to access the local storage on different browsers.

The local storage of browsers is not a secure storage, so while it can be used for static data, like enumerations, applications must not store any user or application data within the local storage.

SAPUI5 is using the local storage of the browser for the history-capability of dropdown boxes and combo boxes.

WEBGL

While more and more browsers are supporting WEBGL by default, WEBGL allows accessing the graphics API of the computer on a very low level, which may also lead to low level exploits. This is the main reason why some browsers have no support for WebGL at all.

SAPUI5 is currently not using WEBGL.

WebSockets

While WebSockets offer great new possibilities for the client/server communication of web applications, there have been many security issues rising while the first implementations were done by the browser vendors. Standardization of WebSockets has reached a stable state with RFC 6455 and is now implemented beginning with Chrome 16, Firefox 11 and Internet Explorer 10. Even if the browser implementations themselves prove to be secure, using WebSockets may require additional security measures on the client.

SAPUI5 is currently not using WebSockets.

Postmessage/Onmessage

This is another feature in the HTML5 area, which can lead to massive security issues when not used correctly. `postMessage` allows inter-window-communication between windows from different domains. This opens a hole in the same origin policy currently implemented in the browser. As soon you subscribe to the `onMessage` event, you can receive messages from any other browser window. The application is responsible to check the originating domain and to process only messages that have been sent by trusted domains.

SAPUI5 uses postMessage for its debugging and tracing functionality.

Transport Security

Transport security comprises topics such as encryption and session security.

Security on the client and server side is not sufficient if the data transport between client and server can be read, intercepted, or even modified by an attacker. Per default, HTTP communication is stateless and unencrypted and this makes it necessary to configure it in a way that it uses encrypted connections and to add session handling on top using either cookies or URL rewriting.

Encryption

Sending the HTTP protocol over a SSL secured connection is not only standardized, but also required for SAP applications.

SAPUI5 fully supports the use of HTTPS, but there are some restrictions regarding the CDN version of SAPUI5 when HTTPS is used. It is recommended to enable or at least to test SSL connections in an early stage of application development, as usually switching to HTTPS causes some issues. First of all, when the application is started using HTTPS, the SAPUI5 library also has to be loaded from an HTTPS server. Second, Internet Explorer 8 and 9 have some additional restrictions regarding cross-origin requests with HTTPS, which are related to the security zone concept.

Session Security

Even if the data transport is secured using SSL or TLS, there are possibilities to hijack such a secure connection and send malicious requests from the client. Cross-site request forgery and session fixation are two of the prominent examples of this class of attacks.

SAPUI5 does only provide XSRF prevention for the data, which is sent to the server by SAPUI5. Currently this only happens in the OData Model, where a XSRF token is read from the server and used for subsequent write requests.

The application is responsible for using the XSRF header or other mechanisms to prevent XSRF for all other server communication triggered by the application.

Server Security

Server security comprises topics such as cross-origin resource sharing and resource handlers.

SAPUI5 contains only a small server-side part to support loading of resources by the client framework. The use of the resource handlers is not mandatory, SAPUI5 also offers a static version of the libraries, which can be used with an arbitrary HTTP server.

Cross-Origin Resource Sharing

Usually the `XMLHttpRequest` for security reasons only allows accessing resources from the same domain as the originating document. As there are a lot of web-based services available today, starting with RSS or Atom feeds, WebServices or OData services, there is a need to be able to also access data sources from different domains within the browser, which was addressed with the CORS (Cross-Origin Resource Sharing) standard. This allows the server to set special headers on their responses, which are telling the `XMLHttpRequest` object, whether it is allowed to process the requested data or not.

This CORS capability also plays an important role in SAPUI5 based applications. In case the application itself and the data visualized are coming from different servers, the CORS header has to be configured correctly on the data providing server, to allow the application server domain to access the data.

SAPUI5 is using CORS header on its CDN based library to be able to load additional scripts, styles, and resources from the CDN server.

Third-Party Libraries

For the third party libraries shipped with SAPUI5, security-related issues have to be observed.

SAPUI5 ships with third-party libraries. jQuery is mandatory as SAPUI5 is based on it, and datajs is needed in case OData services should be used.

jQuery

jQuery does not have any security-related documentation on their site, but they are known to be aware of security and usually reacting quickly in case security issues are found within their library.

SAPUI5 includes different versions of jQuery together with their own libraries, so also has the possibility to add custom security fixes to jQuery, if necessary.

datajs

datajs does not have any security-related documentation on their site.

SAPUI5 includes the datajs library and can add custom security fixes, if necessary.

Libraries Included by the Application

Applications based on SAPUI5 are allowed from a technical point of view to include arbitrary custom libraries within their application. SAPUI5 can, of course, not give any statement about the security of third-party libraries and can not ensure security of third-party libraries. The application has full responsibility for doing a

security assessment of third-party libraries before using them and for embedding and using them in a secure manner.

Secure Programming Aspects

The secure programming guide introduces topics that developers should note.

Input Validation

From the application point of view, the validation of user input must be done on the server and, optionally, on the client. This can be achieved by using two-way data binding and model types.

From the control point of view, the input of control properties must be validated, so that integer properties only accept integers and enumeration properties only accept an existing enumeration value. While this sounds obvious, in JavaScript it is not. The type system of JavaScript does not do type validation on assignment.

Output Encoding

All data sent from the server must be properly output encoded according to the context they are contained in. For more information, see Cross-Site Scripting [page 1475].

Content, which is created on the client side either for display within the browser or for data transport, needs to be properly output encoded with the encoding methods provided by SAPUI5. There are methods for encoding HTML, XML, JavaScript, CSS and URI components.

All controls in SAPUI5 libraries properly encode their data, except for HTML-control and XMLView. The latter two are explicitly built to display arbitrary HTML content. If applications use these two controls and provide unsecure HTML content, they have to check/validate the content on their own.

> **i Note**
>
> Using an XMLView with application controlled secure HTML content together with standard SAPUI5 controls (other than HTML and XMLView) containing potentially unsecure data is also safe. Only untrusted HTML content is critical.

For more information on SAPUI5 HTML code cleanup, see HTML5 Sanitizer [page 1476].

URL Validation

URL validation should take place on the server-side when possible. In case URLs are entered on the client-side or are loaded from an external service, SAPUI5 offers a URL validator, which can be used to validate whether a URL is well formed and properly encoded. It also contains a configurable whitelist to restrict URLs to certain

protocols or certain hosts. Initially, the whitelist only checks for the `http`, `https`, and `ftp` protocols, but nothing else. Applications should define their own whitelist.

Cache Settings

The application has to take care that caching of data is disabled by setting appropriate HTTP headers on the server-side.

Static resources from SAPUI5 or from the application are not security relevant and are excluded from this rule, so they can safely be cached on the client.

User Management / Authentication

SAPUI5 does not provide any authorization or user management. An application, which implements such facilities based on SAPUI5 has to make sure that SSL/TLS is enabled to prevent cleartext passwords sent over the wire. Applications must not store any logon information on the client.

Local Storage

The local storage of browsers is not a secure storage, so while it can be used for static data, like enumerations, applications must not store any user or application data within the local storage.

Cross-Site Scripting

Cross-site scripting (XSS) is a widely known vulnerability most web sites have. This page does not provide general information about cross-site scripting but focuses on what you as an application developer using SAPUI5 can do to avoid these security issues.

To give a short info on XSS: It is about injecting script code into a web page, which is then executed in the context of the page and therefore not only can access any information which currently displayed on the screen but can either access session information contained in cookies, or send new requests to the server within the current session, or even try to exploit browser vulnerabilities to get full access to the machine the browser is running on.

Cross-site Scripting in SAPUI5-based Web Applications

AJAX frameworks in general are an interesting target for XSS exploits, as not only the HTML which is initially sent to the browser may contain vulnerabilities, but also the code which is used to visualize content on the client side may have bugs which can be exploited to get the JavaScript coding executed on the client side. In

addition to that, once a script has injected an AJAX application, it will be alive for a long time, as usually navigation will not reload the whole page which would also clean up any running JavaScript code, but stays within the same HTML document and uses a delta update mechanism to show new content.

It is important to understand that SAPUI5 is not involved in creating the HTML page which is sent to the client, so there is no way SAPUI5 can prevent XSS vulnerabilities which are contained in the HTML page itself. The application which is using the SAPUI5 rendering has to take care, according to the documentation of their server-side rendering framework, to properly escape user data, in a way that no JavaScript can be injected.

The SAPUI5 framework will take care of proper escaping for all content which is created and displayed on the screen using the controls provided by SAPUI5. There is no need for the application to HTML-escape user data, but the control API expects all data to be unescaped, so that it can be escaped as needed for the context it will be visualized.

HTML Sanitizer

SAPUI5 reuses the HTML4 sanitizer by Google by adapting it for the use of HTML5 coding. The Google sanitizer also supports CSS3 coding. In addition, the HTML5 sanitizer uses the URL whitelist which checks embedded URLs for correct formatting or against a given whitelist.

For adapting the sanitizer to support HTML5, the HTML attributes and elements have been reorganized according to the current HTML5 specification from W3C. All types and flags have been reviewed as accurately as possible and HTML4 elements that are no longer used in HTML5 have been removed. You can, however, still see them as comments. New or changed rules for HTML5 have been marked are "new" within the comments. The comments also state which attributes and elements are assigned to respective types and flags. All rules which were not 100% clear were analyzed in a way of similarity, so for example "audio" and "video" content behaves like images etc. URIEFFECTS state if a URL is loaded inplace within a tag where the actual document is in control of what type of content is loaded like "image" or if a new document is loaded like with "a href". LOADERTYPES state if content is loaded as sandboxed which means it is loaded within a specific surrounding player like with video content for example or if it is loaded freely without restrictions. Internally controls which accept arbitrary HTML content like the `sap.ui.richtexteditor.RichTextEditor` or the `sap.ui.core.HTML` use the HTML5 Sanitizer to sanitize the HTML code of their content and value to not infiltrate any dangerous coding. The option to sanitize the value can be enabled or disabled in the respective control properly via property: `RichTextEditor.sanitizeValue` or `HTML.sanitizeContent`. For the HTML control, it is disabled by default whereas for the RichTextEditor the sanitize option is enabled.

Related Information

Prevention of Cross-site Scripting [page 2213]

URL Whitelist Filtering

The SAPUI5 framework provides a client-side API to manage a white list for URLs. This whitelist can be used to validate arbitrary URLs if they are permitted or not.

Internal examples of how controls can use this feature are those controls which accept arbitrary HTML content like the `sap.ui.richtexteditor.RichTextEditor` and the `sap.ui.core.HTML`. These controls use the URL white list when a check (sanitization) is performed on the content. URLs inside their content are then automatically removed, except if they are listed in the URL whitelist. The option to sanitize the value can be enabled or disabled in the respective control properly via the `RichTextEditor.sanitizeValue` or the `HTML.sanitizeContent` property. For the HTML control it is disabled by default whereas for the RichTextEditor the sanitize option is enabled. When adding a path to the white list be aware to add a "/" at the start of the path if necessary, so "/index.epx" would be the correct entry instead of "index.epx". The last example below shows this.

Maintaining the URL Whitelist

The whitelist can be maintained with the following API:

- `jQuery.sap.addUrlWhitelist`
- `jQuery.sap.clearUrlWhitelist`
- `jQuery.sap.getUrlWhitelist`
- `jQuery.sap.removeUrlWhitelist`

Here is an example how valid URLs can be added to the white list:

```
// jQuery.sap.addUrlWhitelist(/* protocol */ undefined, /* host */ undefined, /*
port */ undefined, /* path */ undefined);
jQuery.sap.addUrlWhitelist(undefined, "www.sap.com");
jQuery.sap.addUrlWhitelist("https", "sdn.sap.com");
jQuery.sap.addUrlWhitelist(undefined, "sap.de", "1080");
```

Validating a URL

A URL can be validated by using the following API: `jQuerysapvalidateUrl`.

Here is an example how a given URL is validated against the before maintained white list:

```
jQuery.sap.validateUrl("http://www.sap.com"); // => true
jQuery.sap.validateUrl("http://sdn.sap.com"); // => false (wrong protocol)
jQuery.sap.validateUrl("https://sdn.sap.com"); // => true
jQuery.sap.validateUrl("ftp://sap.de:1080/anyftpfolder"); // => true
```

If no whitelist is maintained the URL validity check also basically checks the URL for being defined in a valid format.

Whitelist Service

SAPUI5 supports the configuration of a central whitelist service.

`frameOptions` uses the whitelist service to determine whether the application should run in the parent origin or not. The whitelist service call uses the parent origin as URI parameter (URL encoded) as follows:

```
GET url/to/whitelist/service?parentOrigin=https://parent.domain.com
```

The service responds to the request with a valid JSON:

```
{
    "version" : "1.0",
    "active"  : true | false,                    // defines if entry is active
(if not, framing will be allowed per default)
    "origin"  : "<same as passed to service>",
    "framing" : true | false                     // if active, describes if
framing should be allowed (see FrameOptions)
}
```

Related Information

Frame Options [page 1478]
Configuration Options and URL Parameters [page 703]

Frame Options

`frameOptions` is used to prevent security vulnerabilities like clickjacking. With the `frameOptions` configuration you define whether SAPUI5 is allowed to run embedded in a frame or only from trusted origins or not at all.

SAPUI5 provides the following configuration options for `frameOptions`:

Mode	Default	Description
allow	X	Allows to be embedded from all origins
deny		Denies to be embedded from all origins
trusted		Allows to be embedded from trusted origins according to the same-origin policiy and to be embedded to origins allowed by the whitelist service

With `frameOptionsConfig` the following additional configuration options can be set:

Parameter	Type	Default	Description
`callback`	`function(bSuccess)`		Function that is called with the success state **i Note** The function can be synchronously called from the SAPUI5 bootstrap script. The DOM (`document.body`) may not be accessible.
`timeout`	`number`	`10000`	After the delay, the page remains blocked and the provided callback is invoked (milliseconds)
`blockEvents`	`boolean`	`true`	Defines whether keyboard, mouse and touch events are blocked
`showBlockLayer`	`boolean`	`true`	Defines whether an invisible block layer is rendered to prevent interaction with the UI
`allowSameOrigin`	`boolean`	`true`	Defines whether same origin domains are allowed or not
`whitelist`	`string`		Contains the domain whitelist (comma-separated)

Example: `deny`

If the application is not intended to run in a frame, set `frameOptions` to `deny`:

```
<script id='sap-ui-bootstrap'
    src='resources/sap-ui-core.js'
    data-sap-ui-frameOptions='deny'>
</script>
```

Example: `trusted` with `callback`

To restrict the embedding to same-origin domains, set `frameOptions` to `trusted`. The `callback` in the following code sample is called with a boolean as success state and can be used to implement an application-specific behavior.

```
<script>
window["sap-ui-config"] = {
    frameOptions: 'trusted',
    frameOptionsConfig: {
        callback: function(bSuccess) {
            if (bSuccess) {
                alert("App is allowed to run!");
            } else {
                alert("App is not allowed to run!");
            }
        }
    }
};
</script>
<script id='sap-ui-bootstrap'
    src='resources/sap-ui-core.js'>
</script>
```

Example: Whitelist Service

To allow that the SAPUI5 application is embedded in cross-origin domains, configure a whitelist service. The whitelist service checks whether the application can run in the parent origin, or not.

```
<script>
window["sap-ui-config"] = {
    whitelistService: 'url/to/whitelist/service',
    frameOptions: 'trusted',
    frameOptionsConfig: {
        callback: function(bSuccess) {
            if (bSuccess) {
                alert("App is allowed to run!");
            } else {
                alert("App is not allowed to run!");
            }
        }
    }
};
</script>
<script id='sap-ui-bootstrap'
    src='resources/sap-ui-core.js'>
</script>
```

Example: Whitelist Service via `<meta>` Tag

Alternatively, a `<meta>` tag can be used to configure the `whitelistService` and set the `frameOptions` to `trusted`. This only applies if the `whitelistService` or `frameOptions` configuration is not set in the SAPUI5 configuration.

```
<meta name="sap.whitelistService" content="url/to/whitelist/service" />
<script  id='sap-ui-bootstrap'
    src='resources/sap-ui-core.js'>
</script>
```

Related Information

Whitelist Service [page 1478]
Configuration Options and URL Parameters [page 703]

Content Security Policy

Content Security Policy (CSP) adds an additional layer of security that enables the detection and mitigation of certain types of attacks including cross site scripting and data injection.

CSP restricts the sources from which the browser is allowed to load resources, such as scripts, fonts, and images:

- CSP mitigates and reports XSS attacks; CSP compatible browsers only execute scripts loaded in source files that are received from whitelisted sources.
- CSP also mitigates packet sniffing attacks by specifying the protocols that are allowed to be used in the web server, for example, specifying that content must only be loaded from HTTPS.

CSP is either enabled via a configuration in the web server to return the Content-Security-Policy HTTP header (preferred solution), or via the <meta> element in the meta tags of an HTML page.

For generic information about CSP, see https://www.w3.org/TR/CSP2/ .

For SAPUI5, we recommend that developers build their apps CSP-compliant, in particular regarding the loading of resources and the use of inline scripts. `eval()` is currently still required in SAPUI5 for synchronous loading. However, we recommend to load JavaScript resources asynchronously and this also avoids the use of `eval()`. For more information about asynchronous loading, see Modules and Dependencies [page 1094]. For more information about avoiding synchronous APIs which might lead to synchronous loading, see Legacy Factories Replacement [page 1124].

To build CSP-compliant applications without inline scripts, you must avoid the following when developing SAPUI5 apps:

- Script element with inlined source code
- Inline event handler
- `javascript`: URL

- `document.write()`, `createElement('script')`, and so on, if they are used to create inline scripts. Creating script references, such as `<script src="..."></script>` or non-script content with them is okay.

To be prepared for a CSP policy, which does not allow `eval()`, you must also avoid the following elements when developing SAPUI5 apps:

- `new Function()`
- `setTimeout(<non-fn>)`
  This will be ignored silently and not create a timer without `'unsafe-eval'`, that is, `<non-fn>` is never executed. `setTimeout(<fn>)` will work with and without `'unsafe-eval'`.
- `setInterval(<non-fn>)`
  This will be ignored silently and not create a repeated timer without `'unsafe-eval'`, that is, the `<non-fn>` is never executed. `setInterval(<fn>)` will work with and without the `'unsafe-eval'`.

To run in an environment in which CSP has been enabled, SAPUI5 requires the following directives:

- `script-src 'self' 'unsafe-eval' <source hosting UI5>;`
  SAPUI5 itself does not require `'unsafe-inline'`, but still requires `'unsafe-eval'` for synchronous loading of JavaScript resources. `'self'` is required for loading application resources. If SAPUI5 is not hosted with the application, an additional source entry (`<source hosting UI5>`) is required.
- `style-src 'self' 'unsafe-inline' <source hosting UI5>;`
  SAPUI5 requires `'unsafe-inline'` as it is used by many controls. In addition, `'self'` may be needed for loading styles from the application. If SAPUI5 is hosted with the application, `'self'` is required for loading the SAPUI5 styles. Otherwise, an additional source entry (`<source hosting UI5>`) is required. Similarly, the location of custom themes needs to be added.
- `font-src 'self' <source hosting UI5>;`
  `'self'` may be needed for loading fonts from the application. If SAPUI5 is hosted with the application, `'self'` is required for loading the SAPUI5 fonts, otherwise an additional source entry (`<source hosting UI5>`) is required. Similarly, the location of custom fonts needs to be added.
  Some specific SAPUI5 functionality may require specifying `data:` as source.
- `img-src 'self' <source hosting UI5>;`
  `'self'` may be needed for loading images from the application. If SAPUI5 is hosted with the application, `'self'` is required for loading the SAPUI5 images, otherwise an additional source entry (`<source hosting UI5>`) is required. Similarly, the location of custom themes needs to be added, if they contain images.
  If the backend provides additional links to images, their location needs to be added as well.
  Some specific SAPUI5 functionality may additionally require `data:` and/or `blob:`.
- `frame-src <source hosting UI5>;`
  For using the support assistant and/or the diagnostics tool, the location of the SAPUI5 framework (could be `'self'`) needs to be added as a source entry.
  Additional entries may be required depending on the integration, application, or test scenario.
  Some specific SAPUI5 functionality may additionally require `data:` and/or `blob:`.
- `worker-src <source hosting UI5>;`
  Some specific SAPUI5 functionality may require the source hosting SAPUI5 (could be `'self'`), `data:` and/or `blob:`.
- `child-src ;`
  For browsers not supporting `worker-src`, the corresponding entries need to be done here. This is the deprecated predecessor of `worker-src` and `frame-src`.

- `connect-src 'self' <source hosting UI5>;`
  `'self'` is required for loading application resources. If SAPUI5 is **not** hosted with the application, an additional source entry (`<source hosting UI5>`) is required.
  Some specific SAPUI5 functionality may require `wss:`.

To setup a most restrictive policy, setup CSP in report-only mode and start with a minimal policy. Monitor the reports to add missing sources. Finally switch CSP to enforcing the policy.

Right-to-Left Support

SAPUI5 application developers need to be aware of how applications behave when right-to-left (RTL) directionality is selected. Changing the directionality has a big impact on text-displaying controls, images and the alignment of the whole application.

The default text direction is left-to-right (LTR). This can be changed to right-to-left mode (RTL) by using one of the following configuration switches:

- URL parameter `sap-ui-rtl=true`
- Configuration object: `window["sap-ui-config"].rtl = true;`
- Script tag configuration: `data-sap-ui-rtl="true"`
- Setting an RTL language as default language for the application.

SAPUI5 then sets the overall page direction to RTL. All SAPUI5 content is then displayed in RTL mode. Self-written controls and content has to be tested separately. If you require manual styles, provide a style specifically for the RTL case by using `html[dir=rtl]`.

API Properties for Right-to-Left Support in Text-Displaying Controls

Languages with right-to-left (RTL) text directionality keep the default directionality of numeric values and texts in left-to-right (LTR) mode. To ensure correct handling, two API properties have been introduced - `textDirection` and `textAlign`.

Introduction

In Arabic, Hebrew and other languages that use the RTL text direction, when you see numerals or text from left-to-right languages (like symbols) on the UI, they are flipped to match the text direction. This common pitfall is visible when representing numerals (phone numbers, dates, currency values, etc.), which actually need to be displayed in LTR mode within the context of an RTL page.

Solution

Two new properites have been introduced to determine the directionality of the target content.

- `textDirection`
- `textAlign`

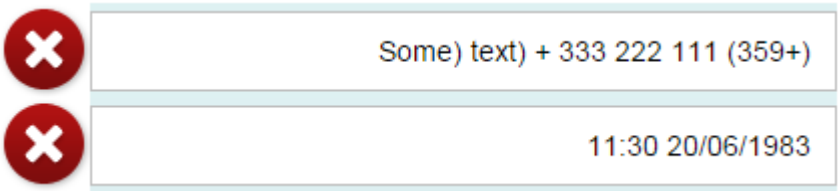
`textDirection` adds an HTML `dir` attribute to the part of the control that displays the target content.

`textAlign` and is used for reversing the text alignment. Adding the `textAlign` property is only needed if the control doesn't force the correct alignment.

The naming of the properties varies based on the actual use case of the control. For example, the `sap.m.DisplayListItem` control has `label` and `value` properties for text representation and the most common use case is to display numeric data in the `value` part of the control. The naming of the new property is `valueTextDirection` and since the control forces text alignment, the `valueTextAlign` property is not needed.

Examples

The examples below illustrate the default behavior of numeric data in an RTL page context – the individual parts of the text are mixed:



After setting the `textDirection` and `textAlign` properties of the control (inherited from `sap.m.InputBase`), the numeric data is displayed in LTR mode, despite the RTL page context. When in LTR mode, the default text alignment (**begin**) is kept, which differs from the page text alignment. Because of this, we should use `textAlign: end` as the control doesn't force the alignment of the text. Here is how this looks in an `sap.m.Input` control:

```
sap.m.Input ({
     value: "(+359) 111 222 333 + (Some) text",
     textDirection: sap.ui.core.TextDirection.LTR,
     textAlign: sap.ui.core.TextAlign.End
});
```

And here is how it looks on the screen:

This second example shows the behavior of a control that enforces correct text alignment - `sap.m.DisplayListItem`. In this case, you don't need to set the `textAlign` property.

```
new sap.m.DisplayListItem({
```

```
    label: "Phone Correct",
    value: "(+359) 111 222 333",
    valueTextDirection: sap.ui.core.TextDirection.LTR
});
```

And here is how it looks on the screen:

> → Tip
>
> To ensure that your application displays the data correctly, always test your application using RTL mode and real data.

Check the API Reference to see if your controls have these properties implemented.

Accessibility

In this guide we cover the most important accessibility aspects for application development, based on SAPUI5.

As an application developer, you need to be aware of accessibility in every step of the process. SAPUI5 controls have built-in accessibility features, but you need to take care when building and combining them into an application. Even though features like keyboard handling come out of the box, you need to pay attention to proper focus handling, so that all parts of the app are reachable with key commands. The following chapters showcase different aspects and offer tips how to test your app.

General Recommendations

When developing SAPUI5 applications, you need to pay attention to the correctness of the resulting HTML. Some vital accessibility features (screen reader and keyboard support) rely on a correct and meaningful structure of the application.

Rules and Guidelines

Don't change the HTML

Theming (CSS selectors), keyboard handling (tab order) and screen reader support are tightly coupled with the HTML structure of the generated pages. If you change the structure of the elements (for example, from custom JavaScript, HTML or CSS), this could break some or all of the accessibility aspects. In addition, it makes debugging the application more difficult.

Check the focus persistence

When opening or closing a dialog or navigating between pages, the focus should stay on the same control as it was on before opening or navigating. If the control no longer exists, the focus should be put on its parent (for example, if the control was inside an action sheet, set the focus on the button which opened the action sheet).

> **i** Note
>
> When the parent control cannot receive focus or is no longer available, the focus should go to a control nearby.

For more information, see sap.ui.core.Element/methods/focus API documentation in the Demo Kit.

Initial focus position

Within an application, the initial focus should be placed on the element that is most likely to be edited or interacted with first (for example, mandatory fields on a form).

When opening dialogs or new pages, the focus should be on the first focusable element inside the content area.

> **→** Tip
>
> On touch devices, if the first focusable element is a control that would open the soft keyboard, it is better to place the focus elsewhere - on the footer of the dialog or on a control that requires no keyboard input.
>
> A good practice is to set the focus on the first mandatory element.

Don't interfere with existing accessibility features

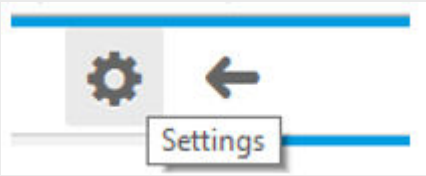
Overriding code, for example the keyboard tab order or SAPUI5 key handlers, will impact the correct handling and may break the accessibility of the whole application.

Make sure that each component has a unique ID

The SAPUI5 framework handles the creation of unique IDs automatically. If you pass IDs yourself, make sure that they are unique within the whole application. For more information, see the *Related Information* section.

Provide meaningful tooltips for icons, icon buttons and where otherwise appropriate

Provide a text alternative for non-text elements and the necessary additional information, in case the text space is insufficient. You should use a tooltip as a label for unlabeled elements only. Additionally, a `Placeholder` can be used as a label if it is not a formatting hint.

> **Example**
>


Tooltips for container controls

Container and layout controls such as `VerticalLayout` or `Grid` inherit the tooltip property from `sap.ui.core.Element`.

→ Tip

We do not recommend setting a tooltip for the whole layout control, as it will not be displayed. You can set the needed tooltips for the individual controls which the container holds.

→ Tip

If you are developing your own controls, follow the guidelines listed under *Related Information*.

Tips for Testing

Start the application and check each screen element. Check the following:

- Does it have a label or a tooltip (hover with the mouse)?
- Are the labels and tooltips (fully) visible, clear, accurate, and meaningful?
  - Is this also true for each input field or for a sequence of input fields? (For example: street and number)?
  - Is this also true for buttons, icons and images?
  - Is this also true for grouped information?
- Is the user informed about the required entries and input?

Related Information

Support for Unique IDs [page 814]
Keyboard Handling for SAPUI5 Controls for Developers [page 2237]
Screen Reader Support for SAPUI5 Controls [page 2244]
Theming [page 1254]

Text Size and Fonts

The size of the text and the font choice greatly impact the visibility and readability of an application. Additionally, your application also has to still be usable at high zoom levels.

Rules and Guidelines

As an application developer, you should always consider the fact that the application could be zoomed to 200%, and everything should still be visible. For that reason, any style that involves disabling zooming or setting fixed weight or height should be avoided. Properties that affect the zooming have to be set accordingly.

Here are some of the most common JavaScript properties that you should bear in mind:

- `user-scalable`
- `initial-scale`
- `maximum-scale`
- `minimum-scale`
- `width`
- `height`
- `target-densitydpi`

> ⚠ Caution
>
> Setting inappropriate values for these properties can completely disable the zooming of an application. Here is an example of such incorrect values:
>
> ```
> <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-
> scale=1,
>  minimum-scale=1, width=device-width, height=device-height,
>  target-densitydpi=device-dpi" />
> ```

Additionally, you can also make zoom adjustments or disable zoom completely in CSS. For example:

```
zoom {
   zoom: 150%;
}
```

Or

```
#elementId {
   -ms-transform: scale(2,3); /* Internet Explorer 9 */
   -webkit-transform: scale(2,3); /* Safari */
   transform: scale(2,3);
}
html {
   -ms-content-zoom-limit-min: 1;
   -ms-content-zoom-limit-max: 1;
}
html {
   touch-action: none;
}
```

Tips for Testing

Enlarge the application UI text by zooming up to 200%.

- Is each text element and each image enlarged to double width and double height?
- Is the text still readable (no overlaps)?
- Are text alternatives for non-text content magnified in the same way?
- Is the full content and functionality still available?
- Does the layout still look OK?

Colors and Theming

Theming is an important aspect for an SAPUI5 application. The different colors shown on the UI need to have a good contrast to each other in order to be easily distinguishable.

General Recommendations

Different people perceive and interpret colors in different ways, which is something you must take into account when developing your applications. High-contrast themes are already available for all SAPUI5 controls.

- Applications need to provide the user with an option to switch themes.
  - If the application runs in the SAP Fiori launchpad, this is covered automatically.
- There are two possible ways to change the theme:
  - With the URL parameter `sap-ui-theme`

    ```
    sap-ui-theme=sap_belize_hcb
    ```

    and

    ```
    sap-ui-theme=sap_belize_hcw
    ```

  - From the API using the core method `applyTheme`

    ```
    sap.ui.getCore().applyTheme("sap_belize_hcb");
    ```

    or

    ```
    sap.ui.getCore().applyTheme("sap_belize_hcw");
    ```

- You should avoid writing custom CSS. If you do need custom CSS for some reason, check to make sure everything is working fine on all available themes.
- If a new theme is created, the color contrast between the elements should be checked. People with visual impairments and people using the application under less than ideal circumstances (bad monitor, sunlight hitting the screen, window reflections) may not be able to read the text easily if the contrast levels are insufficient. Specialized tools can help you to measure the color contrast.

DOs and DON'Ts

DOs

- Use predefined CSS parameters in your CSS. You can find them in the following files within the SAPUI5 library:
  - `resources/sap/m/themes/base/library-parameters.json`
  - `resources/sap/m/themes/sap_hcb/library-parameters.json`
- Use `REM`/`EM` instead of `PX` as a unit.
- Use the flexible layout concept in CSS.
- Use media-queries when flexible layouts do not meet UX expectations.

- Don't forget the Retina display. You need to provide 2 versions for an icon (`icon.png` and `icon@2x.png`).

DON'Ts

- Define a custom color.
- Define a custom font family.
- Set fixed width/height (except for some exceptions like images).
- Define writing-modes (left-to-right or right-to-left), as this is handled by the SAPUI5 control itself.

Tips for Testing

Check the color contrasts.

- Take a screenshot, put it into an image tool and convert it to black and white
  - Are there screen elements, lines or texts that are disappearing?
  - Check the color contrast for elements that disappear.
- If time allows, check all contrasts of all elements.

Keyboard Handling

Keyboard handling enables users to access every UI element of the application with the keyboard and is therefore tightly connected to accessibility. Additionally, this aspect is coupled to the screen reader functionality.

General Recommendations

Accessibility of UI elements

Make sure that all available features of the application can be accessed by using only the keyboard - `TAB`, arrows, `ENTER`, and `SPACE`. The user should be able to activate the functionality of **all** active elements.

Tab order and reading order

The reading order of the page is very important for the application user experience. Those who use keyboard only should be able to navigate easily through every single element. You should always have in mind the fact that the page should have a logical reading order. This is especially important for those who use screen reader software, because in most cases they will follow exactly the tab order of the application and illogical tab orders can confuse them.

> ✤ Example
>
> When you have to select a country and city from select boxes, the country should be focused first and after that the city.

Tips for Testing

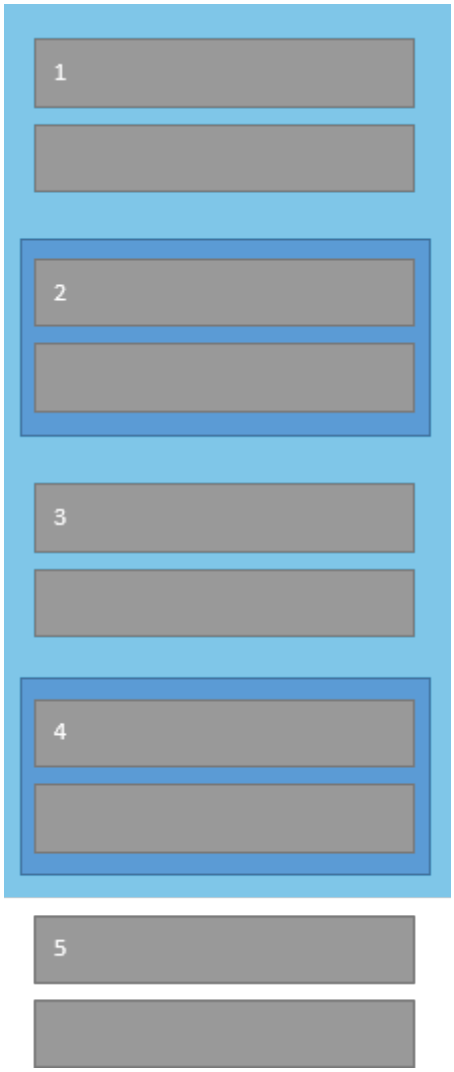
Start the application and put away your mouse.

- Can you reach every active screen element just by using the keyboard?
  - Is this true also for dynamically created texts or control elements?
- Can you navigate within control elements (for example, list, table, tabstrip) using arrow keys?
- Can you also navigate away from each UI element using the keyboard?
- Does the visible focus follow the exact keyboard commands? Is it always identifiable and in the visible area?
- Can you execute all actions? (Compare with what you can do with the mouse)

Fast Navigation

Adjacent controls within the tab chain can be grouped. Within such a group, F6 skips all controls of the group and moves the focus to the first control in the tab chain of the next group. Shift + F6 moves the focus to the first control of the previous group. Adjacent tab chain elements between groups are automatically handled as one group. For nested groups, the most concrete group is used.

The following image describes this behavior. Groups are highlighted in blue, elements in the tab-chain are grey.



The F6 navigation cycles. This means if the focus is within the last group in the group chain, the focus moves to the first control in the first group. This leads to an additional F6 chain, which allows fast navigation through applications. Larger controls like the `Table`, `Panel`, and `Form` provide their own groups by default. The application developer defines further groups.

As described, some larger controls or containers already define `F6` groups. If a group is defined on root level of a control or element, the group can be removed by using the `CustomData` mechanism.

Coding Example:

```
oControl.data("sap-ui-fastnavgroup", "false", true/*Write into DOM*/);
```

XML View Example:

```
<mvc:View
  xmlns:core="sap.ui.core"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">
  <Panel>
    <headerToolbar>
      <Toolbar>
        <Button icon="sap-icon://settings" />
```

```
      </Toolbar>
    </headerToolbar>
    <content>
      <Text text="Lorem ipsum dolor st amet..." />
    </content>
    <customData>
      <core:CustomData key="sap-ui-fastnavgroup" value="false"
writeToDom="true" />
    </customData>
  </Panel>
</mvc:View>
```

The same way it is possible to make a control or element to be an `F6` group. However, keep in mind that not all elements are represented in the DOM.

Coding Example:

```
oControl.data("sap-ui-fastnavgroup", "true", true/*Write into DOM*/);
```

XML View Example:

```
<mvc:View
  xmlns:core="sap.ui.core"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">
  <Panel>
    <headerToolbar>
      <Toolbar>
        <Button icon="sap-icon://settings" />
      </Toolbar>
    </headerToolbar>
    <content>
      <Text text="Lorem ipsum dolor st amet..." />
    </content>
    <customData>
      <core:CustomData key="sap-ui-fastnavgroup" value="true"
writeToDom="true" />
    </customData>
  </Panel>
</mvc:View>
```

Also, DOM elements that are not controlled by SAPUI5 controls can be grouped by setting `data-sap-ui-fastnavgroup="true"`.

Screen Reader Support

SAPUI5 application developers need to be aware of how the screen reader reads out the contents of the UI. Labels, headings, and descriptions help you describe the contents and visual elements of an application.

General Recommendations

The following rules and guidelines will help you avoid common pitfalls and show you best practices. You still need to be aware that there may be deviations between the interpretation of the markup by the different screen readers.

Generate valid HTML

The screen reader software gets the information about the page directly from the DOM. Therefore, if the DOM is invalid, the information presented to the user might be invalid as well. Ideally, if the DOM is correct, the screen reader software will interpret it correctly. When you need to create new controls or change the HTML structure of existing ones for some reason, you have to check the validity of the resulting HTML.

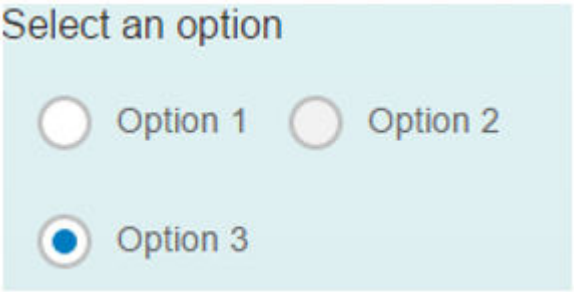
Use titles for complex components

Complex controls like pages, panels, and tables need a title to summarize the contents they hold. If you use the standalone `Title` control, make sure that it is associated with the element that needs the title.

Example

```
<mvc:View
      xmlns:mvc="sap.ui.core.mvc"
      xmlns="sap.m">
   <Title id="rbGroupTitle" text="Select an option" textAlign="Center"/>
   <RadioButtonGroup columns="2" selectedIndex="2"
ariaLabelledBy="rbGroupTitle">
      <buttons>
         <RadioButton id="RB2-1" text="Option 1"/>
         <RadioButton id="RB2-2" text="Option 2" editable="false"/>
         <RadioButton id="RB2-3" text="Option 3"/>
      </buttons>
   </RadioButtonGroup>
</mvc:View>
```

Result:



Use labels

Make sure that all edit boxes, search fields, and column headers have labels. If not, use the `Label` control and add one. Labels have to be connected to each control, for example by using one of the following:

- `labelFor="…"`
- `aria-label="…"`
- `aria-labelledby="…"`
- `placeholder="…"`
- `title="…"`

Titles in headings

For headings (table toolbar, page header, form toolbar, panel toolbar) the text should be in a `sap.m.Title` control (make sure that it is associated with the element that needs the title and, if not, add the reference using `aria-labelledby`).

Example

```
<mvc:View
      xmlns:mvc="sap.ui.core.mvc"
      xmlns="sap.m">
   <Panel expandable="true">
      <headerToolbar>
         <Toolbar height="3rem">
            <Title text="Header"/>
            <ToolbarSpacer/>
            <Button icon="sap-icon://settings"/>
            <Button icon="sap-icon://drop-down-list"/>
         </Toolbar>
      </headerToolbar>
      <content>
         <Text text="Lorem ipsum"/>
      </content>
   </Panel>
</mvc:View>
```

Labels and descriptions for input controls

When using the `Input` control, always provide a label (make sure it is connected with the input). You can also use the `description` property to add additional information. For the input, the description is usually used for showing the unit of measurement (for example. "EUR").

Example

```
<mvc:View
   xmlns:l="sap.ui.layout"
   xmlns:mvc="sap.ui.core.mvc"
   xmlns="sap.m">
   <l:VerticalLayout
      class="sapUiContentPadding"
      width="100%">
      <l:content>
         <Label text="Product price" labelFor="productPriceInput"/>
         <Input
            id="productPriceInput"
            value="220"
            description="EUR / 5 pieces"
            width="200px"
            fieldWidth="60px" />
      </l:content>
   </l:VerticalLayout>
</mvc:View>
```

Result:

Empty labels in drop-down lists

In some cases, you may need to provide an empty option in a drop-down field such as `Select`. This way you want to indicate that none of the items in the drop-down is applicable, or offer the empty option as a way to clear the selection. In this case, you should properly label the empty option with *(None)* and not leave it blank.

The labeling on the empty option will be read by the screen reader and the end user will be correctly informed about the semantics of the empty option.

> → Tip
>
> The empty label *(None)* should always be the first item in the drop-down.

List with info toolbar

When using the `List` control with visible non-active info toolbar with plain text content, you need to associate the `aria-labelledby` of the list with the text content of the `infoToolbar` aggregation.

> ✤ Example

```
...
<List ariaLabelledBy="textInInfoToolbarId">
                <infoToolbar>
                                <OverflowToolbar active="false">
                                                <Label
id="textInInfoToolbarId" text="Announce this text when the first list item is
focused" />
                                </OverflowToolbar>
                </infoToolbar>
                <items>
              ...
</List>
...
```

Provide additional information when there are changes in the screen

Make sure to provide additional information to the user, when changes in the screen are done, based on user interaction. You can use one of the following informative techniques:

- Include additional text description via `ariaDescribedBy/ariaLabelledBy` API for the control triggering the update (Button, Search Field, or other interactive control). For example: When search results will be placed in the area below, you can add a text describing how/where the users can locate the results.
- Include additional text description via `ariaDescribedBy/ariaLabelledBy` on the parent control level. For example: In the case of apps using the master-detail pattern, where the list is on the left side and results are presented on the right side, you should add additional text describing that upon selection in the list, the details will be loaded in the details panel.
- When something is opening on the screen, you can move the focus there, if your use case requires it. For example, for apps using the master-detail pattern when the user needs to browse the available items, the focus should stay in the master list area. And in cases when the user has selected an item from the master list and needs to perform an action, the focus should move to the details area.

  > i Note
  >
  > For controls that are toggling/opening regions and are still present after the toggling, then the focus should remain on them. But if the control is hidden afterwards – the focus needs to be moved, into the default focus position in the toggled/opened region.

Tips for Testing

Start a screen reader, start the application, walk through the application using the keyboard and listen. For example, you can also use *Virtual Cursor mode* or similar functions of your screen reader because some elements do not need to have the focus to be readable by screen readers.

- Is everything that you need to use the application read correctly?
  - Each element's role, name, state, label, tooltip, further information (attached errors, usage hints)?
  - Is this the case for interactive and semantic/non-semantic elements?
  - Actively check that all visible UI elements are read.
- Is it read correctly?
  - No duplicates?
  - No strange pronunciation, like reading English with German words or vice versa?
  - No nonsense, wrong values, another element's attributes?
- Are screen updates like application messages, dialogs (popups), and similar dynamic content read correctly and at the right time?

Control-Specific Behavior

Screen readers need to handle SAPUI5 controls with similar functionality in a similar way. Still there are differences, and application developers need to know them, in order to use the correct control for the desired task.

The first thing that should be read upon entry for any control has to be its ARIA role. What is read afterwards for each control is different, but similar types of controls will read similar elements.

Dialog Controls

SAPUI5 dialog controls can serve different purposes within the application and are therefore handled differently by the screen reader.

General Behavior

Dialog controls are used to interact with the user in two main ways. Popups, message boxes, and busy dialogs interrupt the user and require some additional interaction. Message toast and message strip are used to just display some status information without interrupting the user. Interrupting dialogs have a more complex structure, but all generally follow the same structure - a title, content area and some actions. The title should be the first thing read by the screen reader. It should explain the general purpose of the dialog (i.e. Data confirmation, Interruption). If the dialog contains initially focused elements, like action buttons, those should be read after the title announcement. The contents of the dialog should be read in their respective order.

Titles and Labels

The behavior of controls based on Dialog/Popup (`role="dialog"`) changes depending on their `aria-labelledby` attribute.

If the the control has an `aria-labelledby` attribute, the screen reader will announce the following elements:

- The provided label
- The role of the dialog
- The currently focused element

If the the control does NOT have an `aria-labelledby` attribute, the screen reader will announce the following elements:

- All elements in the dialog (regardless if interactive or not)
- The role of the dialog
- The currently focused element

The `title` property of a Popover/Dialog is used to display the title of the dialog. If the dialog has no visible title, but one is needed, it can be provided as a reference to another control in `aria-labelledby`. All other text that needs to be read before the title, can be added there as well.

i Note

A dialog without a title will be read completely when it appears. Users with screen readers will not be able to differentiate it from the rest of the application.

Example

Dialog with `ariaLabelledBy`

```
    var oDialog = new sap.m.Dialog({
                title: "Dialog Title",
                ariaLabelledBy: "textId",
                content: [
                    new sap.m.Text({id: "textId", text: "A sample text that
will be annoucned by JAWS after the title, when the dialog is opened."})
                ]
            });

            var btn = new sap.m.Button({
                text:'Hello World',
                press: function(){
                    oDialog.open();
                }
            });
            btn.placeAt('content');
```

More detailed behavior is described in the table below.

Detailed Behavior

In the following table you can see how the different dialog controls are read by the screen reader.

Table 71: Screen Reader Behavior in Dialog Controls

SAPUI5 Control	What is read by default
Dialog / Select Dialog	• Dialog title • Initially focused element
Busy Dialog	• Dialog header • Dialog message • Progress bar
Message Box	• Dialog title • Dialog message
Message Toast	• Message text when the toast appears
Message Popover / Popover	• No specific element needs to be read or focused.
Message Strip	• The complete content including text, icons and links • Message type • Associated headings (if any) **i Note** The *Close* button should not be read initially.

User Action Controls

SAPUI5 action controls are used for triggering interactions with the application. Therefore it is important that screen readers read the correct set of properties for them in order to insure their proper usage.

Table 72: Screen Reader Behavior in Input Controls

SAPUI5 Control	What is read by default
Button	• Button text • Custom types • Tooltip (if the button is only an icon)
Link	• Link text • Link label • Tooltip
Breadcrumbs	• In Virtual Cursor mode - upon entry announce the label *Breadcrumb Trail* • For each breadcrumb - text, label and tooltip • For separators - announce textual descriptions if icons are used

SAPUI5 Control	What is read by default
Menu Button	• Button text • Statement that button opens a menu • Custom types • Tooltip (if the button is only an icon)
Split Button	• Button text • Statement that button opens a menu • Custom split button types • Tooltip (if the button is only an icon)
Toggle Button	• Button text • Toggle state • Custom button types • Tooltip (if the button is only an icon)

User Input Controls

SAPUI5 input controls are used to get and store user data. The most common use of these controls is as part of a form.

Table 73: Screen Reader Behavior in Input Controls

SAPUI5 Control	What is read by default
Check Box	• Checkbox label • Checkbox state • Checkbox value • Tooltip
Combo Box	• Combobox label • Combobox placeholder text (if any) • Combobox state • Combobox value • Tooltip Other interactions that need to be announced: • Expanding\collapsing the combobox • Value changes that happen without expanding • When navigating the dropdown items: item text, item state, position in the list, total number of items

SAPUI5 Control	What is read by default
Date Picker	• Picker text • Picker placeholder text (if any) • Picker state • Picker value • Tooltip
Facet Filter	Light Variant (only one button): • Label for the filter • Default statement: "no filter selected" Simple Variant (several buttons): • Labels for the filters
Input	• Input state • Input value • Input label • Tooltip • Associated descriptions • Associated messages • Additional properties like `valueHelp`
Mask Input	Basic behavior is like `Input`. Additionally the screen reader should read: • Masked input characters/symbols • Symbols that have already been entered

Container Controls

SAPUI5 container controls do not directly interact with the application user. Still they play an important role in the application navigation and structure.

SAPUI5 Control	What is read by default
Bar/Toolbar	• Upon entry in Virtual Cursor mode - ARIA role • In Virtual Cursor mode - Type of the contained labels (header, subheader, footer)

SAPUI5 Control	What is read by default
Icon Tab Bar	• Element role (tapstrip or tabpanel) • Sub-element role (tab) • Textual descriptions for icon tabs • Actual position of the selected tab in the set • Total size of the tab set • Current state of the selected tab
Message Page	• In Virtual Cursor mode - all text on the page
Page	• Landmark roles for the substructure elements (header, content, footer)
Panel	• Button text • Toggle state • Custom button types • Tooltip (if the button is only an icon)
Object Header	• The Object Header title • In Virtual Cursor mode - all content of the region **i Note** The whole region should be available as a landmark

Display Controls

SAPUI5 display controls are used to indicate the progress of some action or to show visual elements like images and text.

Table 74: Screen Reader Behavior in Display Controls

SAPUI5 Control	What is read by default
Busy Indicator	• Duration - indeterminate
Image	• Textual description of the image content **i Note** Decorative images need not be read out. **i Note** Interactive icons, that are used as buttons, should be read as such (`role="button"`).

SAPUI5 Control	What is read by default
Label	• Label text
Numeric Content	• Description of the numeric value
Object Identifier	In Virtual Cursor mode: • Content
Object Number	In Virtual Cursor mode: • Content • Status
Object Status	In Virtual Cursor mode: • Content • Status • Textual description of the icons used
Progress Indicator	• Progress state • Current value • Minimal value • Maximal value • Value changes
Text	In Virtual Cursor mode: • Text content

List Controls

List controls are used to store entities

SAPUI5 Control	What is read by default
Breadcrumbs	• Upon entry in Virtual Cursor mode - ARIA role • In Virtual Cursor mode - Type of the contained labels (header, subheader, footer)
Icon Tab Bar	• Element role (tapstrip or tabpanel) • Sub-element role (tab) • Textual descriptions for icon tabs • Actual position of the selected tab in the set • Total size of the tab set • Current state of the selected tab
Message Page	• In Virtual Cursor mode - all text on the page

SAPUI5 Control	What is read by default
Page	• Landmark roles for the substructure elements (header, content, footer)
Panel	• Button text • Toggle state • Custom button types • Tooltip (if the button is only an icon)
Object Header	• The Object Header title • In Virtual Cursor mode - all content of the region **i Note** The whole region should be available as a landmark

Composite Controls

Composite SAPUI5 controls, like Wizard or Semantic Page, are read based on the specific behavior of their parts. Still these controls require additional ARIA labels to correctly separate the areas within the control. Here are some examples:

- Wizard Navigation Area - needs to be marked as a `navigation` landmark
- Dynamic Side Content - needs to be marked as a `complementary` landmark

When creating new composite controls, you need to make sure to properly assign your controls to regions and label these regions correctly.

sap.m.Page Custom Landmarks

The `sap.m.Page` control is used in a lot of applications. As a container control it has three distinct regions that hold content - header, content area and footer. There is also an additional role for the whole page. Each of these areas needs to have a clearly defined ARIA role that explains what the region does. The ARIA roles for all these regions can be set to custom values using the properties of `sap.m.PageAccessibleLandmarkInfo`. Possible values for these roles are stored in the `AccessibleLandmarkRole` enumeration. In the example below you can see how this is done in the `landmarkInfo` property of `sap.m.Page`.

```
<App>
    <Page title="Hello World" height="100%">
        <headerContent>
            <Title text="Just Page" />
        </headerContent>
        <subHeader>
            <Bar>
                <contentLeft>
                    <Text text="Text in content left bar subheader "/>
```

```
                </contentLeft>
            </Bar>
        </subHeader>
        <footer>
            <Bar>
                <contentLeft>
                    <Text text="Text in content left bar footer "/>
                </contentLeft>
            </Bar>
        </footer>
        <content>
            <Text text="Text in page content"/>
        </content>
        <landmarkInfo>

            <PageAccessibleLandmarkInfo
                rootRole="Region"
                rootLabel="Root Label"
                contentRole="Region"
                contentLabel="Content Label"
                footerRole="Region"
                footerLabel="Footer Label"
                headerRole="Region"
                headerLabel="Header Label"
                subHeaderRole="Region"
                subHeaderLabel="SubHeader Label" />
        </landmarkInfo>
    </Page>
</App>
```

Labeling and Tooltips

The following guidelines help you properly label your controls in order to have good accessibility.

General Considerations

Top 5 things to do for better screen reader support for labels

1. Label all elements and element containers correctly and completely.
2. Provide text alternatives for visual labels
   Use tooltips only in rare cases. They should not be used as a replacement for a label.
   Use the `alt` attribute for images
3. Describe controls and give additional information as part of the UI
   Use `ariaDescribedBy` where needed.
4. Identify regions correctly according to their purpose.
   Use containers with a correct meaning and Landmark roles.
5. Provide accessible alternatives and describe how to use them (for example in the documentation of the application).

Table 75: Rules and Guidelines for Labeling and Tooltips

Rules and Guidelines	Examples/Clarification
Non-decorative `sap.m.Image/sap.ui.core.Icon` should provide a meaningful alternative description in the `alt` property.	**Example** `<Image id="image_not_decorative" src="IMAGE_PATH" alt="This is an image showing an elephant" decorative=false>`
Interactive `sap.m.Image/sap.ui.core.Icon` (that has a press handler) should not be decorative.	**Example** `<Image src="IMAGE_PATH" alt="This is an image with a press handler" decorative=false press=onImagePress>`
Icon-only `sap.m.Button` should have a tooltip.	**Example** `<Button icon="sap-icon://action" press="onPress" alt="An action button" ariaLabelledBy="actionButtonLabel"/ >`
Labels should **not** have a tooltip.	This could lead to ambiguity.
Input elements should have labels.	Every input needs a label for its description and purpose. Even if the app doesn't include one, you can set one in `sap.ui.core.InvisibleText`. The placeholder text should not be used as a label.
Tables should have titles	Tables with hidden titles or in containers with titles (for example,single tables in tab strip panels) should be labeled with `sap.ui.core.InvisibleText` in combination with `arialabelledby`.
Button that has a text, should **not** have a tooltip.	**Example** `<Button text="Default" press="onPress" />`

Rules and Guidelines	Examples/Clarification

`Aria-labelledby` and `aria-describedby` associations should point to existing DOM elements.	Example
	`<Page title="Page">`
	`<content>`
	`<Button text="Home"`
	`ariaLabelledBy="invisibleId"/>`
	`<core:InvisibleText`
	`id="invisibleId" text="I am a`
	`hidden label"/>`
	`</content>`
	`</Page>`

Labels should be connected with the labelled elements via `labelfor`.	Example
	`<Label text="Name" labelFor="I1">`
	`<Input id="I1">`

> **i Note**
>
> If you want to enlarge the size of the standard tooltips, you need to change the system font size. Tooltips are rendered by the browser using native window API and thus their size cannot be influenced by the SAPUI5 framework.

The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure

The SAPUI5 ABAP repository is used to store SAPUI5 apps, components, and libraries. SAP uses it for delivering various types of SAPUI5 apps, for example SAP Fiori or High Performance Analytics (HPA) apps. The SAPUI5 ABAP repository can also be used by customers to store their own SAPUI5 apps and extension projects.

> **i Note**
>
> This chapter is only relevant if you're using the SAPUI5 ABAP repository and the ABAP back-end infrastructure.

The SAPUI5 ABAP repository is part of the SAPUI5 ABAP back-end infrastructure and is the umbrella term for the single SAPUI5 repository of each application. Technically, this infrastructure is based on the Business Server Page (BSP) repository and each SAPUI5 repository is represented by an individual BSP application.

> **⚠ Caution**
>
> For data integrity reasons, don't modify content of the SAPUI5 ABAP repository directly by editing the corresponding BSP applications in ABAP Workbench (transaction `SE80`). For more information, see Technical Remarks [page 1510].

The BSP runtime is not used at runtime and SAPUI5-specific request handlers are used instead. Therefore, the flow logic ABAP parts cannot be used as they are not executed at runtime.

The SAPUI5 text repository is part of the SAPUI5 ABAP repository. It's intended to be used as fallback mechanism if translation by properties files is not possible.

> **i Note**
>
> SAPUI5 distribution layer artifacts, such as control libraries, are not stored in the SAPUI5 ABAP repository, but in a separate repository (technically based on the MIME repository). The runtime access is realized by SAPUI5-specific request handlers.

The following design time tools use the SAPUI5 ABAP repository:

- SAP Web IDE
- Special SAPUI5 repository upload and download reports
- OData services

For more information, see Design Time Aspects [page 1511].

Further Features of the SAPUI5 ABAP Back-End Infrastructure

- SAPUI5 application index addressing the following:
  - Indexing content of the descriptor for applications, components, and libraries (for example, used by SAP Fiori apps)
  - Cache busting on the level of single and multiple application resources

  Fore more information, see SAPUI5 Application Index [page 1525].
- Cache busting on the level of single and multiple application resources
  Which cache busting mechanisms are used, and in which cases, depends on the SAPUI5 app.
  For more information, see Cache Behavior for Application Resources [page 1516] and Cache Buster for SAPUI5 Application Resources [page 1516].

Availability

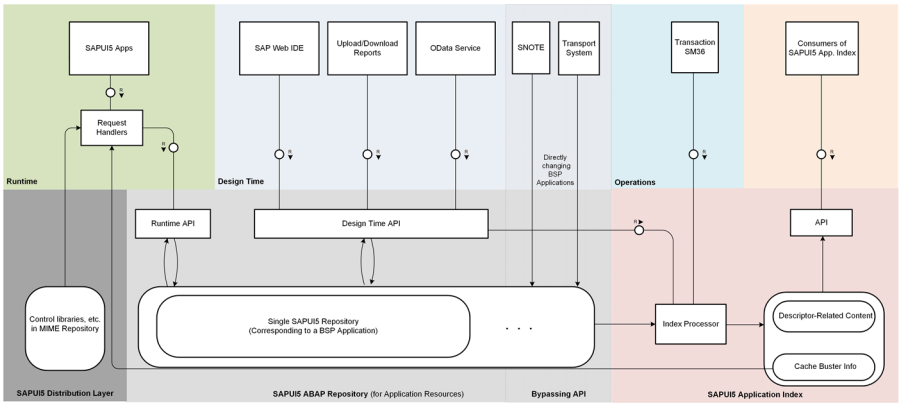
What?	Available with ...
SAPUI5 ABAP repository	SAP Business Suite systems from version 7.00 containing the user interface (UI) add-on for SAP NetWeaver, which contains the software component `UI_INFRA`
Team repository provider (available in Eclipse)	SAP Business Suite systems from version 7.31 containing the UI add-on for SAP NetWeaver, which contains the software components `UI_INFRA` and `UI5_731` **i Note** The team repository provider, which is part of the SAPUI5 tools for Eclipse, is no longer updated after SAPUI5 release 1.71. For more information, see SAPUI5 Tools for Eclipse – Now is the Time to Look for Alternatives.

What?	Available with ...
SAPUI5 repository upload and download reports (alternative for the team repository provider with similar functions)	SAP Business Suite system from version 7.00 For more information, see Using the SAPUI5 ABAP Repository Upload and Download Reports to Synchronize [page 1515].
SAPUI5 control libraries	SAP Business Suite systems from version 7.00 containing the UI add-on for SAP NetWeaver in the software component `UISAPUI5`

> **i Note**
>
> From SAP NetWeaver 7.4 SP1, all required SAPUI5 parts listed above are already part of software component `SAP_UI`.

Big Picture: How Does it All Work?

The following image shows an overview of the main tools, repositories, APIs, and parts of the ABAP back end.



- The **SAPUI5 ABAP repository** contains single SAPUI5 repositories (each represented by a BSP application). The SAPUI5 ABAP repository is used to store SAPUI5 application resources.
- At **design time**, you can access the SAPUI5 ABAP repository through the design time API using three different design time tools: SAP Web IDE, special SAPUI5 repository upload and download reports, and OData services.
- When applying SAP Notes with **transaction SNOTE** and using the **transport system** to transport your changes, for example to quality systems, the BSP applications that represent single SAPUI5 repositories are directly accessed (passing over the design time API).
- After carrying out content changes using the design time API, the **index processor** recalculates the **SAPUI5 application index**.

  > **i Note**
  >
  > This doesn't happen for changes carried out using transaction `SNOTE` or the transport system. To ensure periodic recalculations of the SAPUI5 application index for these changes too, use transaction

> `SM36` to schedule periodic runs of the index processor triggered by the report `/UI5/APP_INDEX_CALCULATE`.

- The SAPUI5 application index contains information related to the supported cache busting mechanisms and the content of the descriptor for SAPUI5 apps, components, and libraries. The content of the SAPUI5 application index can be **consumed by an API**, for example, by SAP Fiori launchpad. Cache busting-related information of the SAPUI5 application index can also be used by the request handlers (at runtime, for SAPUI5 apps that are using this mechanism).
- At runtime, the application resources needed by SAPUI5 apps are accessed through a specific **runtime API** from the SAPUI5 ABAP repository, whereas the SAPUI5 distribution layer artifacts (such as controls) are retrieved from the **MIME repository**. The application resources and the SAPUI5 distribution layer artifacts are served through SAPUI5-specific request handlers assigned to the respective ICF nodes.
- Another part of the SAPUI5 ABAP repository is the SAPUI5 text repository.

Technical Remarks

Here are more details about the SAPUI5 ABAP repository, the runtime handler, and content-specific functionality.

SAPUI5 ABAP Repository

The SAPUI5 ABAP repository uses BSP applications to store SAPUI5 apps, components, and libraries. To be precise, for text files it uses the page fragments and pages with flow logic of the BSP applications, and for binary files the MIME objects of the BSP applications.

As mentioned above, don't edit the BSP applications in transaction `SE80`. Instead use the appropriate development tools which use the SAPUI5 ABAP repository APIs to ensure the integrity of the data.

The SAPUI5 ABAP repository supports explorer-like file and folder structures. Therefore the BSP repository is enhanced accordingly to also support file and folder structures. This means that the sources of an app might look different in a BSP application compared to the corresponding single SAPUI5 repository when viewing it in SAP Web IDE or Eclipse:

- Path mapping in XML (`UI5RepositoryPathMapping.xml`)
- GUID-like or hash-like names
- Texts looking different to the version in Eclipse or at runtime:
  - Trailing spaces in text lines are escaped with ` ` in the respective BSP pages.
  - Lines longer than 254 characters are broken into multiple lines and get a plus (+) on position 255.
- A BSP application representing a single SAPUI5 repository uses the application class `/UI5/CL_UI5_BSP_APPLICATION`.

Runtime Handler

- For runtime access, each single SAPUI5 repository has an ICF node beneath node `/sap/bc/ui5_ui5`.
- Handler classes are assigned to node `/sap/bc/ui5_ui5` for the following purposes:
  - Accessing SAPUI5 ABAP repository content
  - SAPUI5 distribution layer including SAPUI5 core libraries

Content-Specific Functionality

In general, the SAPUI5 ABAP repository and runtime access to resources do not make assumptions regarding its content. However, there are the following exceptions:

- The `index.html` files are realized as pages with flow logic in the BSP application (whereas all other text files are realized as page fragments). In this way, they can be tested directly in transaction `SE80`. (For example, SAPUI5 application projects created with SAPUI5 tools for Eclipse have a file of this type.)
- There is a special logic for runtime handling that deals with the `WebContent` folder and `web.xml` that are used in dynamic web projects in Eclipse. (SAPUI5 application projects created with SAPUI5 tools for Eclipse are based on dynamic web projects.)
- There is a special logic for runtime handling of properties files (resource bundles) in the form of a server-side locale fallback. This means that if a properties file for a specific locale is requested and does not exist, the next matching properties file is returned.

Design Time Aspects

You can upload and download files from and to the SAPUI5 ABAP repository with the following tools:

- SAP Web IDE
- SAPUI5 repository upload and download reports
- OData service (upload only)

We recommend to use SAP Web IDE for developing complex apps with the latest innovations. Some advantages of SAP Web IDE:

- Uses Git for version control of the sources
- Supports a client-side build that generates minified files and packages preload bundles to optimize start-up performance
- Pushes the SAP Web IDE state to the ABAP system in a one-way, overwrite-all operation

Don't manually edit the BSP application representing a single SAPUI5 repository.

The SAPUI5 ABAP repository supports typical code pages for transferring of text files. However, there are some code pages that are not supported, for example code page `CP932` containing Japanese characters. If in doubt, we recommend that you encode the files transferred to the SAPUI5 ABAP repository in `UTF-8`.

Related Links

- SAP Web IDE

- Using the SAPUI5 ABAP Repository Upload and Download Reports to Synchronize [page 1515]
- Using an OData Service to Load Data to the SAPUI5 ABAP Repository [page 1513]

View and Change Content of the SAPUI5 ABAP Repository

You can create and change customer content in the SAPUI5 ABAP repository. You can also retrieve content from the SAPUI5 ABAP repository to view it.

> **i Note**
>
> Changes to content that is delivered by SAP in the SAPUI5 ABAP repository in customer systems are not supported. Extension concepts exist that allow custom enhancements of content that is delivered by SAP. This depends on the type of the content (see the documentation in question). For example, see the extensibility information for SAP Fiori for SAP Business Suite on SAP Help Portal at http://help.sap.com/fiori under *SAP Fiori Implementation Information* > *Extensibility Information*.

Virus Scan During Uploads to the SAPUI5 ABAP Repository

From SAP NetWeaver 7.0, SAP delivers the virus scan profile `/UI5/UI5_INFRA_APP/REP_DT_PUT` for ABAP with the user interface add-on for SAP NetWeaver. This virus scan profile is used to store files in the SAPUI5 ABAP repository.

Examples:

- Upload using SAPUI5 ABAP repository API `/UI5/CL_REP_DT`, method `/UI5/IF_UI5_REP_DT~PUT_FILE` from SAP NetWeaver 7.0
- SAPUI5 team repository provider in SAP NetWeaver 7.3 EHP1, or from SAP NetWeaver 7.40

Perform Static Checks on SAPUI5 Apps

As of SAP Fiori technology release 1911, you can perform static checks on SAPUI5 apps that you have created.

The static checks verify the following:

- Does the SAPUI5 app exist in the SAPUI5 ABAP repository?
- Is the SAPUI5 app active?
- Does the corresponding ICF node exist under `/sap/bc/ui5_ui5`?

These checks have been delivered via the Code Inspector. For more information, call transaction `SCI` and click on the ⓘ button. In the Performace Assistant window, click on the link in **For more information, see the extended help in Code Inspector.** The extended help describes how you can perform the above checks for your own SAPUI5 apps.

Using an OData Service to Load Data to the SAPUI5 ABAP Repository

You can use the OData service `/UI5/ABAP_REPOSITORY_SRV` to upload a SAPUI5 app, component, or library to the SAPUI5 ABAP repository.

If you use a repository, for example git, and a build server, for example Jenkins, for the central coordination of your SAPUI5 developments, you can use the OData service `/UI5/ABAP_REPOSITORY_SRV` to upload the respective files that are collected in a zip file to the SAPUI5 ABAP repository. The OData service uploads the zip file to the SAP NetWeaver AS ABAP into a BSP application that is created or updated during the upload. This BSP application represents the SAPUI5 ABAP repository. From there, the app, for example, can be used in the SAP Fiori launchpad. The OData service `/UI5/ABAP_REPOSITORY_SRV` uses the SAP Gateway service builder project `/UI5/ABAP_REPOSITORY`.

The OData service offers the basic entity `Repository` and supports `GET`, `CREATE`, `UPDATE`, and `DELETE` operations. On return, the HTTP status reports either success or errors which may have occurred during the operation. The response header or the response body contain additional information.

The base64-encoded zip archive that contains the app, component, or library files is provided in the `ZipArchive` property. The operations `CREATE` and `UPDATE` use the file provided in the property for the operation. You use the `GET` method of the OData service to retrieve a basic XML form that you can use for the `CREATE` and `UPDATE` operations. To remove a SAPUI5 ABAP repository, you use the `DELETE` method that, if successful, deleted the corresponding BSP application and its SICF service and updates the SAPUI5 application index.

URL Parameters

The following URL parameters are provided for the communication of mandatory or optional parameters for the operations that are not part of the `Repository` entity itself:

- `CodePage`: Contains the information about the code page of your text files, for example, `CodePage='UTF8'`
- `TestMode`: If set to `TRUE`, the operation runs as a test and no upload takes place.
- `TransportRequest`: Specifies an ABAP transport request

URL Parameters to Reduce the HTTP Response Header Size

If you want to reduce the HTTP response header size, for example because of the error message mentioned below, you can use the following URL parameters to the OData call:

- `CondenseMessagesInHttpResponseHeader=X`
  This limits the number of detail messages for the load operation to 6. Any additional messages are omitted.
- `DetailMessagesInHttpResponseHeaderUpTo=<number of detail messages>`
  Enter the number of detail messages to be listed in the HTTP response header.

If you upload a zip archive containing a SAPUI5 app, component, or library into the SAP NetWeaver AS ABAP for deployment or for delivery, a SAPUI5 ABAP repository is created or updated and the Business Add-In (BAdI) *SAPUI5 Repository Load* (`/UI5/BADI_REPOSITORY_LOAD`) is called. You need to implement this BAdI on the SAP NetWeaver AS ABAP and use it to check and adjust the parameters that control the OData service,

for example, if you want to determine or create an ABAP transport request automatically. For more information about the Business Add-In, see the BAdI documentation in the SAP system.

To further control the upload operation of the OData service, you can use the following text files in the archive:

- `'.Ui5RepositoryIgnore'`: Each line in this text file describes a file pattern that indicates which files shall be ignored during the upload. The line contains a substring of the file path or a regular expression starting with '^' and ending with '$'. This setting overwrites the build-in default.
- `'.Ui5RepositoryTextFiles'` and `'.Ui5RepositoryBinaryFiles'`: These text files are used to identify text and binary content in addition to the build-in default. If it is not clear whether a file is text or binary, a warning is issued in the log and the file is not uploaded.

For testing the OData service, you can use the SAP NetWeaver AS ABAP with the SAP Gateway client (transaction `/IWFND/GW_CLIENT`).

> **i Note**
>
> For operations on a SAPUI5 ABAP repository, you need the `S_DEVELOP` authorization.

Error Message `io.netty.handler.codec.TooLongFrameException: HTTP header is larger than 8192 bytes`

You get this message if you use the SAP Cloud Connector to call `/UI5/ABAP_REPOSITORY_SRV` and your SAP Cloud Connector Configuration doesn't allow HTTP response headers larger than 8 kB (which is the default setting). You have two options to solve this:

- Change the configuration of SAP Cloud Connector: Go to the installation directory of SAP Cloud Connector and open the XML file `<sccdir>\scc_config\scc_config.ini`. Change the value for the parameter `httpProtocolProcessorMaxResponseHeaderSize` from `8` to `30` (kB). Restart the SAP Cloud Connector.
- Use the URL parameters mentioned above to reduce the HTTP response header size.

For more information, see SAP Note 2875647.

Related Information

Activate and Maintain Services

SAP Note 2875647

Using the SAPUI5 ABAP Repository Upload and Download Reports to Synchronize

You can upload an SAPUI5 app to or download it from the SAPUI5 ABAP repository by using the SAPUI5 ABAP repository upload and download reports.

Single SAPUI5 App

To upload, download, or delete a single SAPUI5 app, use the report `/UI5/UI5_REPOSITORY_LOAD`. Enter the name of the SAPUI5 app and specify whether you want to update, download, or delete it. You can also specify whether or not the line endings are adjusted automatically during the upload. In contrast to the ABAP team repository provider, the report does not offer a built-in code merge.

Uploading SAPUI5 Apps From an Archive

Prerequisite: The files to be uploaded are located on an HTTP web server.

- To upload an SAPUI5 app from a zip or war archive, use the report `/UI5/UI5_REPOSITORY_LOAD_HTTP`.
- To upload multiple SAPUI5 apps from a zip or war archive at once, use the report `/UI5/UI5_REPOSITORY_LOAD_HTTPN`.

Enter the relevant parameters and specify whether or not the line endings are adjusted automatically during the upload. You can also provide the parameters in the file `.Ui5RepositoryUploadParameters` located in the archive. Each line represents a parameter. The format is `<parameter name> = <parameter value>`.

If you use the SAP Web IDE for developing apps, we recommend that you use the *Deploy to ABAP Repository* function in SAP Web IDE. For more information, see *Deploy Applications to the SAPUI5 ABAP Repository* in the SAP Web IDE Full-Stack documentation on SAP Help Portal underhttps://help.sap.com/viewer/product/SAP_Web_IDE/ → SAP Web IDE Full-Stack Developer Guide. If you use the `/UI5/UI5_REPOSITORY_LOAD_HTTP` and `/UI5/UI5_REPOSITORY_LOAD_HTTPN` reports to upload your project to the SAPUI5 ABAP repository instead, make sure that you upload only the content of the `webapp` or `dist` folder without the folder itself. This ensures that the `manifest.json` is in the root of the structure in the SAPUI5 ABAP repository and the app index can be loaded.

You can also specify whether the reports run in delta or in test mode:

- Delta mode: You only want to upload the files that are new or that have been modified.
- Test mode: You want to see a log file displaying what the report is doing.

> **i Note**
>
> If you use the reports, the SAPUI5 application index is updated automatically and any errors are displayed. For more information, see SAPUI5 Application Index [page 1525].

The functions of the reports are also available in the RFC-enabled function module `/UI5/REPO_LOAD_FROM_ZIP_URL`. It can be called remotely, for example, from Maven builds. For more information, see the documentation of the reports and of the function module `/UI5/REPO_LOAD_FROM_ZIP_URL`.

Runtime Aspects

The BSP runtime is not used. Instead there's an SAPUI5-specific handler to that gets the resources from the SAPUI5 ABAP repository. This handler is assigned to the corresponding ICF nodes.

Accessing Resources

In general, you access a resource in the SAPUI5 ABAP repository at runtime with the following URL:

`- <protocol>://<host name>:<port number>/sap/bc/ui5_ui5/<namespace>/<application name>/<resource name>`

Launching SAPUI5 Apps on an ABAP Server

You launch an SAPUI5 app located in the SAPUI5 ABAP repository by using its public URL in a browser.

Cache Behavior for Application Resources

By default, the application files are stored in the browser cache for one year to speed up the performance of an SAPUI5 app in a productive environment. To get the latest changes, you need to force your SAPUI5 start page to refresh, for example, with `CTRL` + `F5` on Windows systems. (If the refresh doesn't work, clear your browser cache.)

If you are in development mode and want to get the latest changes immediately without refreshing your SAPUI5 start page, you can add the URL parameter `sap-ui-xx-devmode` to the SAPUI5 start page to force the browser to check whether there's a newer version of the application files available.

Cache Buster for SAPUI5 Application Resources

To avoid the need for end users to clean up the browser cache after a software update on the server, you can activate the cache buster for the following:

- SAPUI5 application resources (see Application Cache Buster [page 1134])
- SAPUI5 core resources (see Cache Buster for SAPUI5 [page 1132])

To activate cache busting on the level of single application resources and cache busting for SAPUI5 core resources, change the script tag with the ID `sap-ui-bootstrap` in the start page of the underlying SAPUI5 app:

- Change the value of the `src` attribute pointing to the SAPUI5 core to `resources/sap-ui-cachebuster/sap-ui-core.js`.

- Add the attribute `data-sap-ui-appCacheBuster="./"`.

Example (snippet from the sample app `/UI5/SIMPLETEST`):

```
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <script src="resources/sap-ui-cachebuster/sap-ui-core.js"
      id="sap-ui-bootstrap"
      data-sap-ui-libs="sap.m"
      data-sap-ui-theme="sap_belize"
      data-sap-ui-appCachebuster="./">
    </script>
  …
</html>
```

For more information, see Application Cache Buster: Enhanced Concept [page 1136].

Both the ICM server cache on the ABAP front-end server as well as the browser cache are used to optimize the performance of your SAPUI5 app. The cache buster mechanisms ensure that all application and core resources are up to date at any time and are requested only if needed.

Further Technical Information

- For proper operation you need to schedule the report `/UI5/APP_INDEX_CALCULATE` (replacing the report `/UI5/UPDATE_CACHEBUSTER` used in the user interface add-on 1.0 for SAP NetWeaver) periodically. Then the recalculation and caching of meta data for SAPUI5 apps potentially needed after any system and language import for an SAPUI5 app are done automatically in the background.
- Under typical conditions, for example when using SAPUI5 apps or when developing with the SAPUI5 repository team provider or the SAPUI5 repository upload and download reports, the application cache buster and the cache buster of the SAPUI5 core work fully automatically.
- You can trigger a refresh of the cache buster and application meta data for a specific SAPUI5 repository (and the application or component inside) manually by calling the URL `<URL to SAPUI5 application or component>/do-update-meta-data` in the browser. See SAP Note 2187043.
- In addition to cache busting on the level of a single application resource mentioned above, the cache buster for SAPUI5 application resources supports also cache busting on the level of multiple and all application resources.
- The cache buster technology is used by SAP Fiori launchpad, and SAP Fiori and High Performance Analytic (HPA) apps.
- Fore more information, see SAP Notes 2075016 and 2085648.

Fallback: Translating Apps Using the SAPUI5 Text Repository

As a fallback mechanism for translating apps, you can use the SAPUI5 text repository for storing the target texts.

What's the recommended translation approach?

The recommended translation approach is to store your target texts in properties files (resource bundles), a `<identifier>_<locale>.properties` file for every language the app is translated to (<locale> containing

the language and an optional country code). For example, `i18n_en_US.properties` would contain the American English texts of the `i18n.properties` source file. For more information, see Resource Bundles [page 1272].

When can I use the SAPUI5 text repository?

You can use the SAPUI5 text repository as a fallback if you cannot use the recommended approach. Please only use it in this case.

What's the fallback all about?

When you use the SAPUI5 text repository, the source texts are stored in the default properties file `<identifier>.properties` (that is, there are not *n* properties files as with the recommended approach). The SAPUI5 text repository writes the texts that are stored in the default properties file to a language-dependent database table. As usual, the texts can then be translated using transaction `SE63` (usually in a separate system). The master language of the SAPUI5 ABAP repository is taken as the master language for the submitted texts. Once the translated texts are transported back to the respective systems, they can be accessed at runtime when texts in a certain language are requested.

What do I have to do to use the SAPUI5 text repository?

You have to set up the default properties file in a special way by adding the following key as the first line in the properties file with texts that you submit to the SAPUI5 ABAP repository:

```
# SAPUI5 TRANSLATION-KEY <GUID with 32 characters>
```

You can create the GUID with the ABAP function module `GUID_CREATE`, or you can create the complete line with the ABAP report `/UI5/TEXT_FILE_GEN_TRANS_KEY`. (A UUID as used in Java is also supported.)

> ⚠ Caution
>
> The GUID serves as a unique identifier for the properties file. Never change the GUID or copy it to other properties files.

The properties file has to fulfill the following rules:

- It doesn't contain any duplicate text elements.
- The text must be on one line and can have a maximum of 255 characters.
- A text type is assigned to every text element.

Accidental overwriting of texts in the SAPUI5 text repository with texts from other properties files is prevented. If a duplicate GUID is detected, the properties files cannot be synchronized. It's not possible to rename or relocate a properties file and to submit it once it's contained in the SAPUI5 text repository. Before you can rename or relocate a properties file, you have to remove the texts from the SAPUI5 text repository by using the report `/UI5/TEXT_ENTRIES_DELETE`.

> ⚠ Caution
>
> If you run this report, all translated texts get lost. Therefore, only use it for texts that are not yet translated.

After running the report, submit the renamed or relocated properties file again.

If you reassign the BSP application on the ABAP server (representing a single SAPUI5 repository) from a local (`$TMP`) package to a non-local package, you also need to resubmit the properties file. All text elements are generated again with the new creation time stamp.

> ℹ Note
>
> Since the master language of the SAPUI5 ABAP repository serves as the master language of the texts, it's important that you use the same language for the creation of the repository that was used for the properties files. When you use the SAPUI5 repository team provider and create the BSP application manually, you have to choose the correct language on the logon screen.

Text Classification

Texts in a properties file are simple value key pairs separated by an equal sign (=) or a comma (,). However, to enable proper translation for these texts, you have to classify the texts with additional information, at least with the text type. You must place the additional information in the line directly above the text element, beginning with the number sign (#).

The complete line must have one of the following patterns (text type is mandatory, maximum length and additional context information are optional):

- `#<TextType>`
- `#<TextType>,<MaximumLength>`
- `#<TextType>,<MaximumLength>:<AdditionalContextInformation>`
- `#<TextType>:<AdditionalContextInformation>`

Text Type (Mandatory)

We recommend that you assign a text type to each text. The text type indicates to which user interface element the text is related. You can use the following main text types:

- For short texts (less than 120 characters) :

Text Type	Description
`XACT`	Accessibility
`XBUT`	Button
`XCKL`	Checkbox
`XCOL`	Column header
`XFLD`	Label

Text Type	Description
XGRP	Group title
XLNK	Hyperlink
XLOG	Log entry
XLST	List box item
XMIT	Menu item
XMSG	Message
XRBL	Radio button
XSEL	Selection
XTIT	Table title
XTOL	Tooltip

- For long texts (more than 120 characters):

Text Type	Description
YINS	Instruction
YMSG	Message

- For text elements that are not supposed to be translated, use the text type NOTR.

Maximum Length (Optional)

You can provide the maximum text length for translation. It must be greater than the source text length and must never exceed 255.

To ensure that translators have enough space for the translated texts, set an appropriate maximum text length for translation according to the source text length:

Length of Source Text in Characters	How Is the Maximum Text Length for Translation Calculated?
< 8	Multiply by 5; minimum 12 characters
8 - 30	Multiply by 3
> 31	Multiply by 1.5

Additional Context Information (Optional)

You can also add a comment for the translator.

How Translated Texts Are Accessed at Runtime

At runtime, the translated texts are read directly from the table of the SAPUI5 text repository and sent to the browser.

- The browser sends a request for the relevant properties file with a language extension to the back end, for example `i18n_en_US.properties`.
- The handler class in the back end analyzes the name of the properties file and extracts the language key.
- All texts that fit to this language key are read, collected, and sent back to the browser. The following access sequence applies:
  1. If a text is not available in the requested language, the English text is retrieved.
  2. If the English text is not available, the text in the default language is retrieved.
  3. If the text in the default language is not available, the default properties file from the SAPUI5 ABAP repository is retrieved.

Information for Translators

The following information is relevant for translators.

- Texts are treated as ABAP short texts with translation object type `UI5T`. You can translate them in transaction SE63.
- The translation object name is a GUID, which is the key taken from the first line of the original properties file containing the text elements (`# SAPUI5 TRANSLATION-KEY <GUID>`, as mentioned above).
- The text key of each text element consists of the text type and an individual GUID, separated by a blank.
- The texts are stored in the following database tables which are stored with an SAP or customer namespace:

Database Table	What Does it Contain?
`/UI5/TREP_TEXT` (master table)	○ Text name ○ Unique text GUID ○ Text type ○ Additional context information ○ Translation object name GUID (from the properties file)
`/UI5/TREP_TEXT_T` (language-dependent table)	○ Source and translated text ○ Keys: text GUID as in table `/UI5/TREP_TEXT`, and language key
`/UI5/TREP_FILES`	○ Translation object name (GUID from the properties file) ○ Path information for the properties file

Placeholder Handling in Transaction SE63

Source texts with placeholders are transformed when displayed in transaction `SE63`.

Type	What is it?	How is it Displayed in Transaction SE63?
Placeholder	Number in braces Example: `{0}`	With an additional ampersand Example: `{&0}`
Escaped placeholder	Placeholder enclosed in apostrophes Example: `'{0}'`	Without apostrophes Example: `{0}`

When the texts are written back to the database, the placeholders and escaped placeholders are transformed back to the original state. If a text contains a placeholder, you need to double any apostrophe (') in the text. Otherwise the apostrophe doesn't appear on the user interface during runtime.

> **i** Note
>
> Translators don't need to take care of this, as transaction `SE63` automatically doubles apostrophes (') when writing the text to the database. Therefore, translators do not need to change placeholder characters or character sequences, such as `{&0}`, `{0}`, or `'{&0}'`. Double apostrophes ('') in the source text, however, are displayed only as one apostrophe (') in transaction `SE63`. Examples:
>
Source Text	How is it Displayed in Transaction SE63?
Mark''s placeholder is used {0} times.	Mark's placeholder is used {&0} times.
Note that '{0}' is an escaped placeholder.	Note that {0} is an escaped placeholder.

As a developer, you must ensure that the source text is formally correct, for example by using the notation `{0}` for placeholders and enclosing placeholders in double apostrophes. Bear in mind that `&0` is not a valid placeholder.

Securing the SAPUI5 ABAP Repository

Here's everything you need to know about securing the SAPUI5 ABAP repository when using the team repository provider and the repository upload and download reports, when executing apps from the repository, when tracking code changes or text changes, or when using the SAPUI5 application index REST API.

Using the Team Repository Provider or the SAPUI5 ABAP Repository Upload and Download Reports

Using the team repository provider, you can synchronize the application resources between the team provider REST API and the SAPUI5 ABAP repository. Alternatively, you can use the interactive ABAP reports `/UI5/`

`UI5_REPOSITORY_LOAD` or `/UI5/UI5_REPOSITORY_LOAD_HTTP`, which offer a similar functionality. Compared to the team repository provider, these interactive reports do not offer a built-in code merge. You can use a separate source code repository such as Git or Subversion (SVN).

Authorization Objects for Team Repository Provider

Authorization Object	What Is It for?
`S_DEVELOP`	Create, update, and delete applications in the SAPUI5 ABAP repository
`S_ICF_ADM`	Create the application-specific ICF node under `/sap/bc/ui5_ui5/`
`S_TRANSPORT`	Create a new transport request or new task
`S_CTS_ADMI`	Transport applications
`S_CTS_SADM`	Transport applications
`S_ADT_RES`	Communication between the team provider REST API (for example used in SAP Web IDE) and the ABAP backend using the team repository provider
`S_RFC`, Activity *16 (Execute)*, with `RFC_NAME=SADT_REST_RFC_ENDPOINT` and `RFC_TYPE=FUNC`	Communication between the team provider REST API (for example used in SAP Web IDE) and the ABAP backend

> **i Note**
>
> In addition to assigning these authorization objects, you also have to activate certain ICF services . For more information, see Configuring the ABAP Back-end for ABAP Development Tools

Delivered Virus Scan Profiles

When uploading files to the SAPUI5 ABAP repository, you can perform a virus scan. SAP delivers virus scan profile `/UI/UI5_INFRA_APP/REP_DT_PUT`, which is used to store files in theSAPUI5 ABAP repository. This profile is deactivated when delivered. To activate it, create at least one basis profile and save it as the default profile. You can then activate one of the delivered profiles.

By default, it links to a reference profile that is the default profile. For more information, search for **`ABAB-Specific Configuration`** in the documentation of your SAP NetWeaver version on the SAP Help Portal.

Executing SAPUI5 Applications from the SAPUI5 ABAP Repository

Using an ICF handler, you can execute SAPUI5 applications by retrieving their resources from the SAPUI5 ABAP repository.

Delivered ICF Nodes

For the execution of SAPUI5 applications from the SAPUI5 ABAP repository, SAP delivers ICF node `/sap/bc/ui5_ui5/`. This node contains sub nodes for each application.

> **i Note**
>
> Since all services delivered by SAP are inactive initially, please activate all required services.

For more information, search for **Activating and Deactivating ICF Services** in the documentation of your SAP NetWeaver version on the SAP Help Portal.

Authorization Objects

There are no specific authorization objects needed to execute SAPUI5 applications from the SAPUI5 ABAP repository.

As for ICF service nodes in general, authorization for specific ICF service nodes can be restricted. For more information, search for **Defining Service Data** in the documentation of your SAP NetWeaver version on the SAP Help Portal.

Tracking Coding Changes and Text Changes in the SAPUI5 ABAP Repository

You can track code changes by using the general ABAP version control of the corresponding resource file. A new version is created when a new transport is written.

You can track text changes by using *Table History* (transaction `SCU3`). The relevant tables for texts are `/UI5/TREP_TEXT` and `/UI5/TREP_TEXT_T` (for translated text). Bear in mind that you have to activate table logging in the system for this feature.

Using the SAPUI5 Application Index REST API

The SAPUI5 application index REST API can be executed from ABAP systems with an ICF handler to get the transitive dependencies of an app.

> **i Note**
>
> This API is not for public use. It's only used when packaging SAP Fiori apps with SAP Mobile Platform Hybrid SDK plugins.

Delivered ICF Nodes

For the execution of the SAPUI5 application index REST API, SAP delivers ICF node `/sap/bc/ui2/app_index`.

> **i Note**
>
> Since all services delivered by SAP are inactive initially, please activate all required services.

For more information about ICF services and security, search for **Activating and Deactivating ICF Services** and **RFC/ICF Security Guide** in the documentation of your SAP NetWeaver version on the SAP Help Portal.

For more information about the SAPUI5 application index, see SAPUI5 Application Index [page 1525].

SAPUI5 Application Index

This index provides an indexing and caching mechanism for information related to apps, components, and libraries in the SAPUI5 ABAP repository and related to components and libraries in the SAPUI5 distribution layer.

> ⚠ Caution
>
> The index is used by several different services such as the SAP Fiori launchpad and cache buster. This means you have to make sure that the index is updated using the calculation report whenever the content of the SAPUI5 ABAP repository has changed. For more information whether the report is executed automatically or you have to run it manually, see Calculation Report - Automatic Run vs. Manual Scheduling [page 1526].
>
> Also descriptor and component IDs used in single SAPUI5 repositories have to be unique and valid. Fore more information, see Component IDs - Are They Unique and Valid? [page 1528].

The index makes it possible to retrieve and find this information significantly faster than when carrying out the calculations each time it's requested. The index is also required, for example, for finding the paths to SAP Fiori libraries.

The index contains the following:

- Cache busting information on the level of single and multiple application resources
- Certain properties or attributes (for example, component ID, used library, or dependencies) stored in the descriptor for apps, components, and libraries
  For more information, see Descriptor for Applications, Components, and Libraries [page 734].

Also in this section:

How is the Index Calculated?

The index is calculated by the report *Calculation of SAPUI5 Application Index for SAPUI5 Repositories* (`/UI5/APP_INDEX_CALCULATE`). The index is empty initially and needs to be calculated from scratch.

The report `/UI5/APP_INDEX_CALCULATE` has to be executed in every system whenever the content of the SAPUI5 ABAP repository has changed. It's enough to run the report for one client per system. For more information whether the report is executed automatically or you have to run it manually, see Calculation Report - Automatic Run vs. Manual Scheduling [page 1526].

The report allows you to specify the basis for the calculation of the index. Here's what you can choose from:

- Full update of all SAPUI5 repositories and the distribution layer regardless of any expiration dates
  This mode is active when you use the provided variant `SAP&ALL`.

> **Note**
>
> The *Full Calculation* option should only be used in exceptional cases. This will calculate the index of the entire SAPUI5 ABAP repository, even for content that hasn't changed.

- Calculation for those repositories and the distribution layer for which either the expiration period you specified (in hours and minutes) has expired or where a transport changed the content of the repositories and the distribution layer since the last update (delta mode)
  This is the recommended mode. The default value for the expiration period is 24 hours.
- A single repository
- The SAPUI5 distribution layer only

> **Note**
>
> The report `/UI5/APP_INDEX_CALCULATE` replaces the report `/UI5/UPDATE_CACHEBUSTER` used in the user interface add-on 1.0 for SAP NetWeaver.

Calculation Report - Automatic Run vs. Manual Scheduling

Changes to the content of the SAPUI5 ABAP repository require the index to be updated using the calculation report `/UI5/APP_INDEX_CALCULATE`. Depending on how the content of the repository is changed, the report is executed automatically or you have to run it manually.

Changes to Content in the Repository

This scenario usually happens in development systems. The content is changed, for example, by uploads from development tools like SAP Web IDE, the available SAPUI5 repository upload and download reports, the implementation of an SAP Note, or manual changes using transaction `SE80` (the latter is not supported and therefore not recommended at all).

The execution of the report to update the index is in most cases triggered automatically. Exceptions: implementation of SAP Notes, support package updates, release upgrades, changes to texts in the text repository with ABAP translation tools, and manual changes using transaction `SE80` (not supported and therefore not recommended). In these cases, you have to trigger an update for the applications in question manually or schedule a calculation of the index with a reasonable time interval. Here's an overview:

Type of Change to the SAPUI5 ABAP Repository	Manual Execution of the Calculation Report Required?
Deployment with SAP Web IDE	No, the report is executed automatically.
Upload with report `/UI5/UI5_REPOSITORY_LOAD`	
Installation of a new version of the SAPUI5 distribution layer	

Type of Change to the SAPUI5 ABAP Repository	Manual Execution of the Calculation Report Required?
Implementation of an SAP Note containing changes for an SAPUI5 app	Yes, run the report manually to update the index for the app or distribution layer in question.
Support package updates and release upgrades	
Manual changes using transaction `SE80` (not supported and therefore not recommended)	

> **i Note**
>
> If you run the report manually, it is strongly recommended to use the *Depending on Expiry Period of Transport Requests* option with reasonable values. This option calculates the SAPUI5 application index for affected content only. A full calculation should only be used in exceptional cases.

Import of Content to the Repository

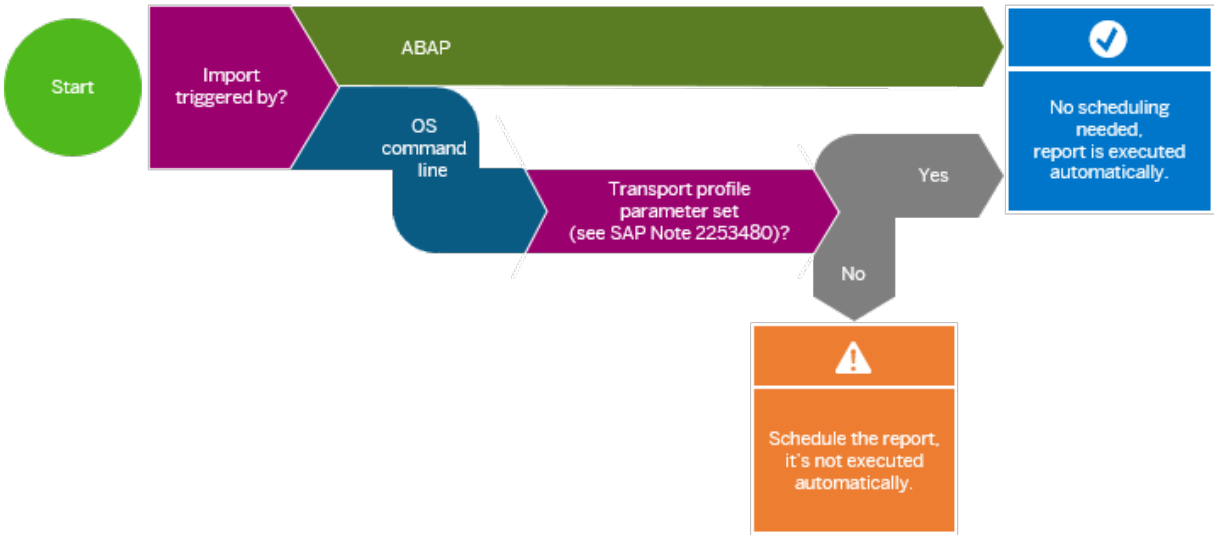
This scenario usually happens in test and production systems. The system automatically updates the index after transports which have been imported under certain conditions, for example, which version of the user interface add-on for SAP NetWeaver is installed and whether the Business Add-In (BAdI) CTS_IMPORT_FEEDBACK is called after an import. For more information, see SAP Note 2253480.

If you are unsure whether the BAdI is called, you can verify this in the import log of a transport. The BAdI is called if the import log contains a *Feedback after import or export* entry (there might be one after import and one after export entry) and the after import entry contains the following logs:

- *Start: Executing method FEEDBACK_AFTER_IMPORT for business add-on CTS_IMPORT_FEEDBACK*
- *Scheduled index update after finished import of transports*

You can access the job `/UI5/APP_IDX_UPD_AFTER_IMPORT` that is automatically scheduled by the BAdI in transaction `SM37`.

Here's how you can decide whether you need to schedule the report after transports:



If the import is triggered by ABAP, report `/UI5/APP_INDEX_CALCULATE` is executed automatically. If the import is triggered by OS command line and the transport profile parameter `FEEDBACK_IMPORT` is set (see SAP Note 2253480), the report is also executed automatically. If the transport profile parameter is not set, the report is not executed automatically.

If you have to schedule the report, there are three options for this:

- With a periodic time interval
  This is the easiest way. However, it has the small disadvantage that some time might elapse between the import of a transport and the start of the update depending on the interval you choose.
- Starting after event `SAP_IMPORT_STO`
  This triggers the report automatically after all transports of the import queue have been imported. However it doesn't trigger the report in cases where you perform a special import of a single transport.
- Schedule the report in both ways
  This is the most secure and fast way to ensure the index is up-to-date. The report ensures that no problems occur in cases where multiple executions are triggered in parallel.

Component IDs - Are They Unique and Valid?

Component IDs used in single repositories have to be unique and valid. Using the ABAP Test Cockpit (ATC), you can check if they are.

Unique means that the same component ID must not be contained in more than one single repository. Valid means that the component ID fulfills the following rules:

- Consists only of alphanumeric characters
- Contains only lowercase letters in all segments except the last segment; the last segment may contain camel case letters
- Does not begin with a number
- Does not contain special characters
- Contains a dot (.) as a separator of the namespace

- Is no longer than 70 characters; each individual segment is no longer than 40 characters (separated by a dot)

To check whether a component ID is unique and valid, you can use the ATC check `UI5 Component Consistency Check` (`UI5_COMP`). To check a component manually, select it in ABAP Workbench (transaction `SE80`) and choose *Check* > *ABAP Test Cockpit with...* from the context menu. Choose the *Checks* tab and under *Functionality* select `UI5 Component Consistency Check`. Choose *Execute Checks* (`F8`).

> **i Note**
>
> We recommend that your administrator defines the `UI5_COMP` check as a default check for all transports to the SAPUI5 ABAP repository.

The `UI5_COMP` check also indicates any errors that occurred when the manifest.json file was parsed.

Calculation Issues

Any issues during the calculation of the index are written as messages to the application log.

Message Type	Classification	What's the Issue?
Error	Very high	Exception because manifest.json file is not valid or cache buster token could not be calculated
Error	High	Component ID occurs or is used more than once
Warning	Medium	Component ID is potentially not valid

To access the application log, choose *View ... Logs* in the report `/UI5/APP_INDEX_CALCULATE`. The application log is also persisted in client 000 by default and can be analyzed using transaction `SLG1`.

Technically, the application log is referenced by the object `/UI5/APPIDX`. This object has the following subobjects:

Subobject	What Does It Contain?
GENERAL	General information like time and duration of the last calculation of the index, number of single repositories, and number of updated table entries
UI5REP	Error messages that occurred when a single repository was processed The external ID is the name of the single repository.

Subobject	What Does It Contain?
UI5COMP	Component-related messages from a consistency check carried out at the end of the calculation of the index The consistency check includes the following: • Whether the same component ID is contained in more than one repository • Whether the component ID is valid • Whether a reuse component defined in the dependencies is not contained in the application index The external ID is the component ID.

The application log contains only entries of the last run of the report `/UI5/APP_INDEX_CALCULATE` and any older entries are removed automatically. For more information, search for **`Analyze Logs`** in the documentation for your SAP NetWeaver version on the SAP Help Portal.

Monitoring

To monitor the execution of the report `/UI5/APP_INDEX_CALCULATE` and the calculation results, you can use Computing Center Management System (CCMS).

Monitor What?	How?
Execution of the report	Set up alerts for the report by using the data collection method `CCMS_BATCH_MONITORING` and the analysis method `CCMS_BP_MON_ANALYZE`. **i Note** This monitoring refers only to the execution of the job itself. It does not detect any application log entries with problems created by the report. For more information, search for **`Monitoring Jobs with the Alert Monitor`** and **`Setting Up the Monitoring of Jobs with the Alert Monitor`** in the documentation for your SAP NetWeaver version on the SAP Help Portal.

Monitor What?	How?
Calculation results	Build CCMS methods that evaluate the application logs for the object `/UI5/APPIDX` and the subobjects `GENERAL`, `UI5REP`, and `UI5COMP`. For more information, search for **`Creating a Data Supplier for the CCMS Alert Monitor`** and **`Application Log (BC-SRV-BAL)`** in the documentation for your SAP NetWeaver version on the SAP Help Portal.

Creating a Login Screen

Here's how you configure a login screen for SAP Fiori launchpad.

For more information, search for **`Login Screen for the Launchpad`** in the documentation for your SAP NetWeaver version on the SAP Help Portal. Replace your namespace and your app name in step 3 of this topic:

Navigate to default_host > sap > bc > ui5_ui5 > <your namespace> and double-click <your app name>

Browser Debugging for ABAP Developers

To debug in SAPUI5, use your browser's debugging tool.

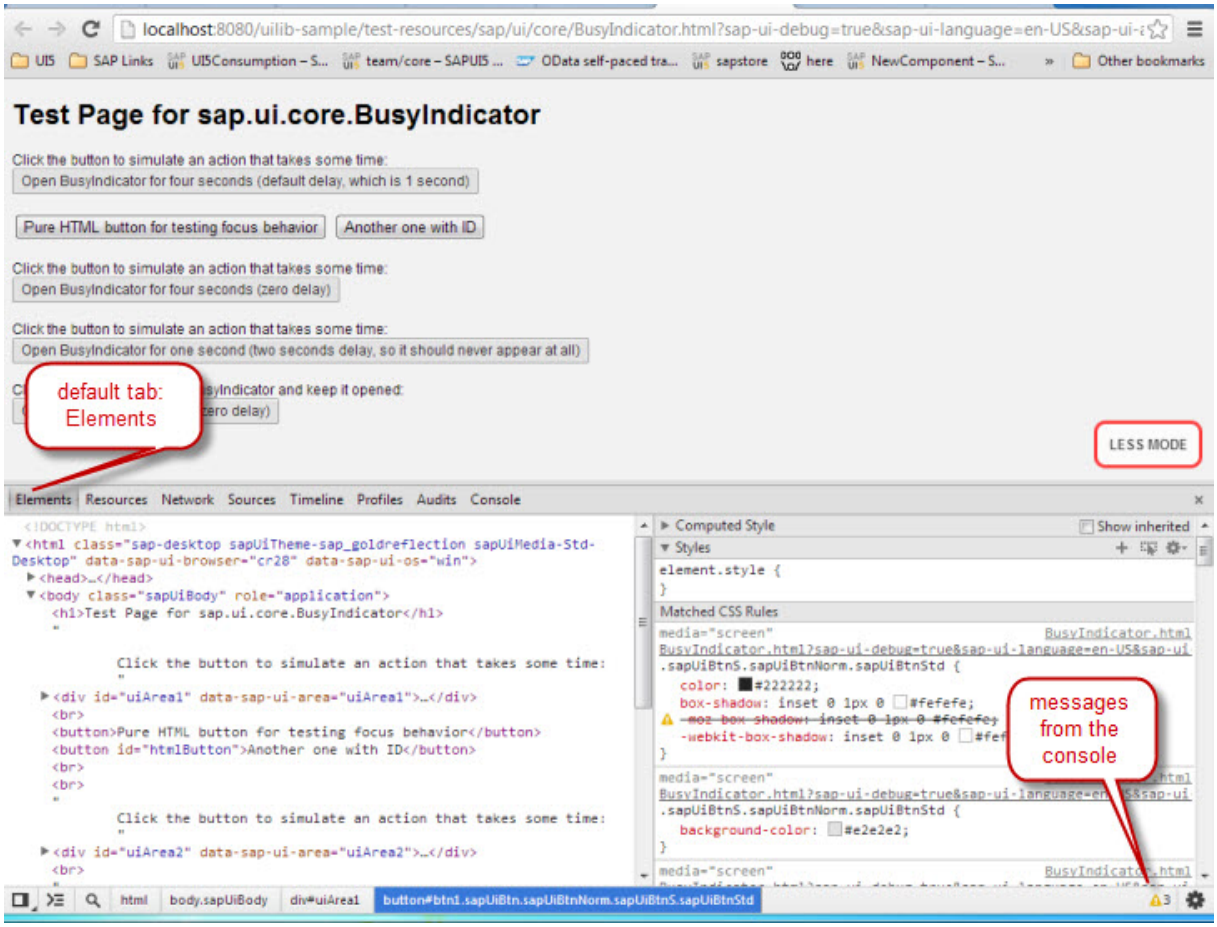
When you debug code in SAPUI5, keep in mind that you can **not** debug SAPUI5 in your IDE. If you use SAP Web IDE, for example, a breakpoint set in SAP Web IDE does **not** stop your script when it is executed in your browser, unless you use the `debugger;` statement explicitly in your code. The browser does not know about your IDE and does not communicate directly with it.

ABAP Debugger vs. Browser Debugger

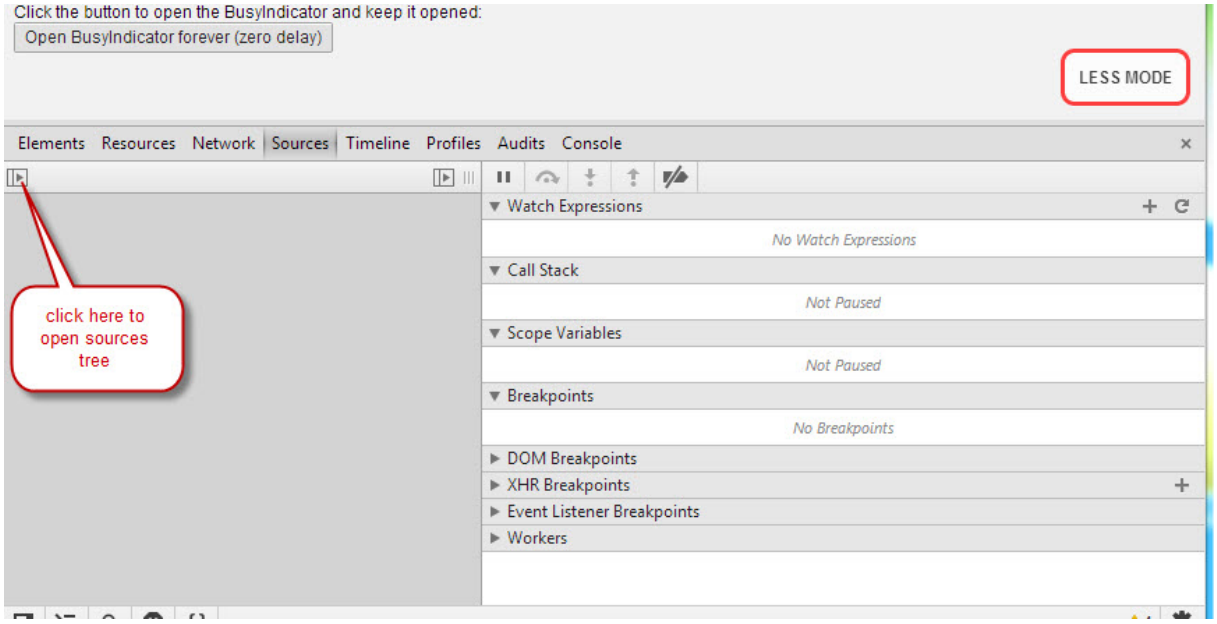
This section explains how you use the debugging tool of a Chrome browser. Keep in mind that you have to test your application on all browsers that are officially supported by SAP, because the implementation differs depending on the browser used, seeChoose your Browser [page 20]. To start the debugger, use the browser menu or choose **`F12`** (valid for most browsers).

The following explanations assume that your application is up and running on your web server, either a local Tomcat, or a remote server.

In a first step, locate the lines of code you would like to inspect and set breakpoints. The following figure shows an application that is opened in the Chrome debugger. The default tab *Elements* is opened, and a small bell icon with a number located at the right border of the footer indicates the number of messages from the console.

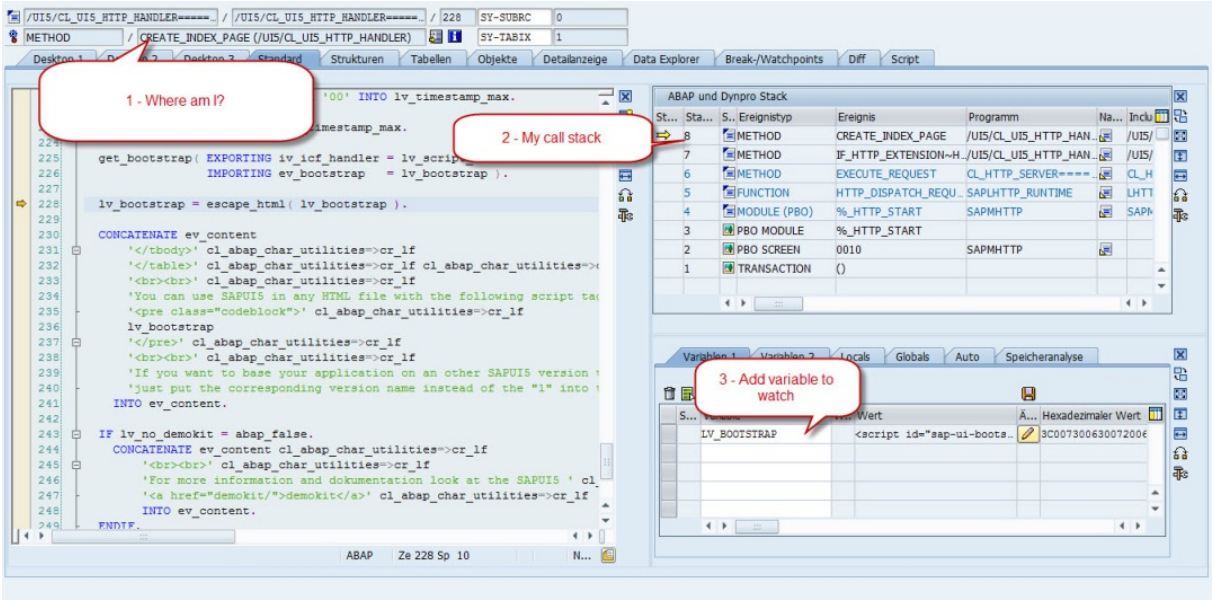


On the *Elements* tab, the HTML elements of the DOM are displayed in a tree structure. To see the JavaScript code within the application and to set a breakpoint there, open the *Sources* tab. From there, you can open any source files that is included. When you open the tools the first time, you ususally have to click the arrow icon on the left hand side of the *Sources* tab (as indicated in the following figure) to open the sources tree.

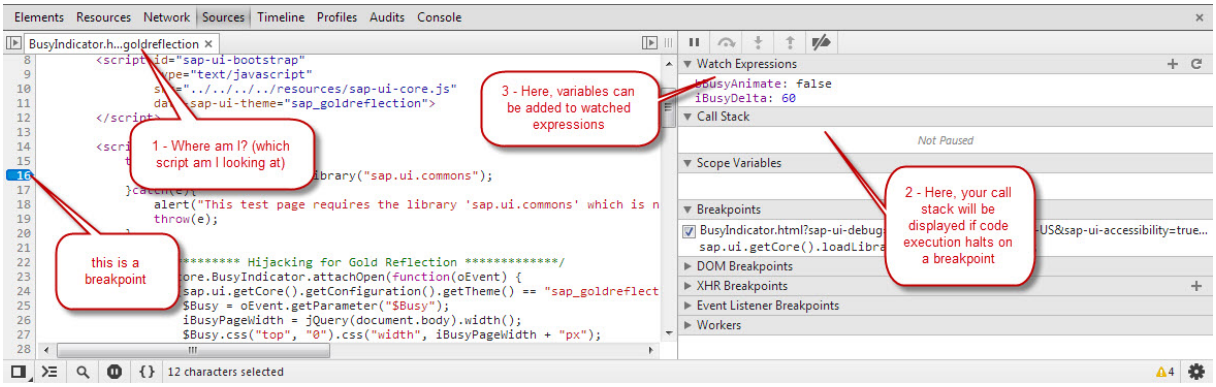


To see the actual content of the HTML page and to set a breakpoint, open the HTML page from the sources tree. This is similar to the ABAP debugger when you execute and debug an application from the workbench. The following figures show the ABAP workbench debugger and the Chrome debugger.

The following figure shows the ABAP workbench debugger. The bubbles indicate the opened application and its location (1), the call stack (2), and the tab where you enter the variables you want to watch (3).



The next figure shows the Chrome debugger. Here, the bubbles indicate the script you are looking at (1), the watch expressions where you can add the variables you want to watch (3), the call stack that indicates if the code execution stops on a breakpoint (2), and the breakpoint (4). The call stack is only visible when the code execution is on hold.



> **Note**
>
> JavaScript does not support a forward navigation, meaning that you can not jump to a method by double clicking. Instead, you either have to jump to the method during execution, or you open the file containing the method.

If you are not sure in which file exactly a piece of code is located, Firefox offers you an option to search through several files included in your page. Chrome, however, does **not** support this option.

Setting Breakpoints

The browser debugger supports several options for setting breakpoints. You can, for example, click once on the line number where you would like to break. To remove the breakpoint, click the respective line again. To temporarily disable or edit the breakpoint, right click on an existing breakpoint. To set a conditional breakpoint, right click on a line without breakpoint. You can also set breakpoints on a certain event or event listeners, see the option in the lower right screen area of the figure above. For more information, see the tutorials for your respective browser that are available in the internet.

> **i** Note
>
> In most cases after setting a breakpoint, you have to reload the page to execute the code again and to make it stop at the respective line. If you use Internet Explorer, choose *Start Debugging* in the developer tools to activate your breakpoint.

Adding Variables to Watch

To add a variable to the list of watched variables, open the context menu for the variable in the code line and choose *Add to watch*. Another option is to choose the + button at the top of the watch list to add a new line, in which you can then enter the name of the variable you want to watch.

Modifying Variables

If you want to modify a variable to find out if the code works correctly with a different value, open the console, for example by choosing **ESC** in the debugging tool and enter the new value manually directly in the JavaScript code. To confirm the change, choose **ENTER** in Chrome and *Execute* in Firefox.

Stepping Through Executed Code

In ABAP, a yellow arrow indicates the line of code that is currently being executed. In Chrome, the arrow is red and the code line is highlighted. The following table gives an overview of the function keys for the ABAP workbench and Java Script:

Function	ABAP	JavaScript
Step-by-step execution, also stepping into functions and loops	**F5**	**F11**
Step-by-step execution, stepping over functions	**F6**	**F10**

Function	ABAP	JavaScript
Skipping the rest of the current function and stepping out to the last cursor position	`F7`	`SHIFT+F11`
Resume execution	`F8`	`F8` (Internet Explorer: `F5`)

Developing Apps with SAP Fiori Elements

This section contains information about developing SAP Fiori apps using SAP Fiori elements.

SAP Fiori elements provide designs for UI patterns and predefined templates for common application use cases. App developers can use SAP Fiori elements to create SAP Fiori applications based on OData services and annotations that don't need JavaScript UI coding. The resulting app uses predefined views and controllers that are provided centrally. This means no application-specific view instances are required. The SAPUI5 runtime interprets metadata and annotations of the underlying OData service and uses the corresponding views for the SAP Fiori app at startup.

Why use SAP Fiori elements?

- High **development efficiency** to cover what 80% of all apps need
  You do not need to build the UI over and over again. Just reuse the common features required by most applications. They are provided by the SAP Fiori elements floorplans.
- **Design consistency**
  Predefined floorplans, views, and controllers ensure UI consistency within and across similar apps. Apps created using SAP Fiori elements are kept up-to-date as they are automatically adapted to the most recent design guidelines.
- **Decoupling of UI and business logic**
  The metadata-driven development model uses semantic annotations and significantly reduces the amount of front-end code. Developers can focus on the business logic.

> **i Note**
>
> The initial effort for creating an app using SAP Fiori elements might be higher than creating a freestyle SAP Fiori app. However, you will be richly rewarded for this effort after you've created more apps this way because your apps will benefit from using the framework and the included features, as described below.

You can create apps using the following SAP Fiori elements floorplans:

- List Report and Object Page [page 1622]
  SAP Fiori elements contain predefined templates for list reports and object pages. A list report lets users filter, view, and work with items (objects) organized in list (table) format. The list report is typically used in conjunction with an object page. This object page lets users work with objects, providing functions for viewing, editing, and creating objects.
- Worklist [page 1866]

A worklist displays a collection of items to be processed by the user. There is no need for sophisticated filtering. Working through the item list usually involves reviewing details of the list items and taking action. In most cases, the user has to either complete a work item or delegate it.

- Overview Pages [page 1930]
  An overview page is a data-driven SAP Fiori app for organizing large amounts of information. Information is visualized in a card format in an attractive and efficient way. Different cards are used for different types of content. The user-friendly experience makes viewing, filtering, and acting on data quick and easy. While presenting the big picture, business users can focus on the most important tasks enabling faster decision making as well as immediate action.
- Analytical List Page [page 1868]
  Analytical list page is a SAP Fiori elements application for detailed analytics. It lets you analyze data from different perspectives, to investigate a root cause, and to act on transactional content. You can identify relevant areas within data sets or significant single instances using data visualization and business intelligence. All this can be done seamlessly on one page.

Getting Started

To find out if and when to use SAP Fiori elements, read this guide. It is aimed at designers, product managers, developers, or anyone involved with application development, from inception to execution: When to use SAP Fiori elements: Usage guide.

You can find even more information on YouTube at Getting Started with SAP Fiori elements .

Features

SAP Fiori elements provide the following default floorplan features:

- *Edit* mode control, switching between display and edit, and submitting changes
- Handling of draft documents (draft saving is available)
- Message handling, including message lifecycle and message display
- Multi-device support
- Support of value help
- Storing and restoring of application states, for example, saving filter fields with their content in the list report
- Back navigation that takes the history into account
- Busy handling and prevention of accidental double-clicks

The following features come with every SAP Fiori app:

- SAP Fiori launchpad integration
- Reuse functions that do not require specific programming

Tool Support

You can use SAP Web IDE to create and maintain SAP Fiori element apps. This includes the following tools:

- Wizard for the initial creation of an app

- Annotation modeler for maintaining annotations
- Wizard to support the creation of app extensions

System Requirements

- SAP Web IDE 1.17 or higher (for more information see the Annotation Modeler in the SAP Web IDE documentation)
- The development of new transactional apps with draft capabilities requires ABAP Application Server as of SAP NetWeaver AS for ABAP 7.51 innovation package SP02 or higher. Draft capabilities are not available with SAP NetWeaver AS for ABAP 7.50. For more information, see Draft Handling [page 1631].
- If you want to use annotations in CDS: SAP NetWeaver 7.5 SP01 or higher.

> **i Note**
>
> We recommend that you download the most up-to-date version to ensure that you have the latest features.

More Information

SAP Fiori Design Guidelines

SAP Fiori Elements Feature Map

This topic lists UI elements, controls, and features that are supported by SAP Fiori elements.

You can search, filter, and sort content in the table below. Use your mouse to hover over the links in the *Help* column to display a short description of the feature.

> **i Note**
>
> This table provides a basic overview of UI elements, controls, and features in SAP Fiori elements. More information and implementation guidance are available in the detailed documentation for the SAP Fiori elements floorplans:
>
> - List Report and Object Page [page 1622], List Report Elements [page 1623] and Object Page Elements [page 1625]
> - Analytical List Page [page 1868]
> - Overview Pages [page 1930]
> - Worklist [page 1866]
>
> Generic information that applies to all floorplans can be found under How To Use SAP Fiori Elements [page 1550].
>
> For an overview of all available SAPUI5 versions and their maintenance status, see SAPUI5 Versions Maintenance Status.

Extending SAP Fiori Elements-Based Apps

You can use extension points to extend your generated app during the creation process. For more information, see the following resources:

- Cross-floorplan extension points and general information: Extending Generated Apps Using App Extensions [page 1585]
- Extending List Reports and Object Pages Using App Extensions [page 1799]
- Configuring Analytical List Page App Extensions [page 1915]
- Configuring Overview Page App Extensions [page 2024]

Table 76: Feature List

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Analytical card	Analytical list page Overview page	Analytical Card	Creating Key Performance Indicator Tags [page 1873] Analytical Cards [page 1976]
Analytical table	Analytical list page List report Object page	Analytical Table (ALV)	Configuring Tables [page 1735] Table Cards [page 1937] Table-Only View [page 1902]
Area micro chart	Analytical list page List report Object page Overview page	Area Micro Chart	Adding a Smart Micro Chart to a Table [page 1749] List Cards [page 1953]
Avatar	List report Object page	Avatar	Using Images, Initials, and Icons [page 1618]
Breadcrumb	Analytical list page Object page Overview page	Breadcrumb	Object Page Classic Header [page 2490]
Bullet chart	Analytical list page Object page	Bullet Chart	Smart Chart Facet [page 1705] Chart-Only View [page 1910]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Bullet micro chart	Analytical list page List report Object page Overview page	Bullet Micro Chart	Adding a Smart Micro Chart to a Table [page 1749] Smart Micro Chart Facet [page 1673] Smart Micro Chart [page 2424]
Busy indicator	Analytical list page List report Object page Overview page	Busy Indicator	
Button/ Action	Analytical list page List report Object page Overview page	Button	Actions [page 1605]
Card filter	Overview page		Configuring Card Filters [page 2000]
Chart/ Smart chart	Analytical list page List report Object page Overview page	Chart Smart Chart	Chart-Only View [page 1910] Smart Chart Facet [page 1705] Chart Cards Used in Overview Pages [page 1979]
Checkbox	Analytical list page List report Object page Overview page	Checkbox	Table-Only View [page 1902] Enabling Multiple Selection in Tables [page 1741]
Coloring cards based on threshold values	Overview page		Coloring Cards Based on Threshold Values [page 2007]
Column chart	Analytical list page Overview page	Column Chart	Chart-Only View [page 1910] Chart Cards Used in Overview Pages [page 1979]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Combo box	Analytical list page List report Object page Overview page	Combo Box	Value Help as a Dropdown List [page 1617] Configuring View Switch [page 2003]
Contact quick view	List report Object page		Adding a Contact Quick View to a Table [page 1756]
Copy and paste from Microsoft Excel to editable tables	Object page		Copying and Pasting from Microsoft Excel to Editable Tables [page 1774]
Cumulation (Waterfall chart)	Analytical list page Object page	Cumulation (Waterfall Chart)	Chart-Only View [page 1910] Smart Chart Facet [page 1705]
Currency	Analytical list page List report Object page Overview page	Currency	
Custom card	Overview page		Custom Cards [page 2028]
Date/time picker	Analytical list page List report Object page Overview page	Date/Time Picker	Smart Field [page 2410]
Date picker	Analytical list page List report Object page Overview page	Date Picker	Smart Filter Bar [page 2413] Smart Field [page 2410]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Date range selection	Analytical list page List report Object page Overview page	Date Range Selection	
Default sort order in a table	List report Object page		Defining the Default Sort Order [page 1737]
Dialog box	Analytical list page List report Object page	Dialog	Adapting Texts in the Delete Dialog Box (List Report) [page 1843] Adapting Texts in the Delete Dialog Box (Object Page Header) [page 1846] Adapting Texts in the Delete Dialog Box (Object Page with Nested Smart Table) [page 1848] Adding Action-Specific Messages to Confirmation Dialog Boxes [page 1782] Adapting Text for Confirmation Dialog Box When Deleting Lines in a Table [page 1776] Configuring a Confirmation Popup for Messages [page 1718]
Draft handling	List report Object page	Draft Handling	Draft Handling [page 1631]
Dynamic page layout	Analytical list page List report Overview page	Dynamic Page Layout	Descriptor Configuration [page 1931]
Editing status	List report Object page		Editing Status [page 1630] Disabling the Editing Status Filter [page 1660] Displaying the Editing Status [page 1763]
Filter bar	Analytical list page List report Overview page	Filter Bar	Compact Filter Setup [page 1884] Defining ValueList Annotation [page 1896] Adapting the Smart Filter Bar [page 1661] Disabling the Editing Status Filter [page 1660] Adding Custom Fields to the Smart Filter Bar [page 1839] Configuring the Global Filter [page 1934] Custom Filters [page 2030]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Flexible column layout	Analytical list page List report Object page	Flexible Column Layout (Layout + SAP Fiori Elements)	Enabling the Flexible Column Layout [page 1611]
Footer toolbar	Analytical list page List report Object page Overview page	Footer Toolbar	Quick View Cards [page 1974]
Form	List report Object page	Form	Defining and Adapting Sections [page 1698]
Formatting numeric values	Overview page		Formatting Numeric Values [page 2006]
Generic tag	Analytical list page	Generic Tag	Creating Key Performance Indicator Tags [page 1873]
Grid table	Analytical list page List report Object page	Grid Table	Table-Only View [page 1902] Setting the Table Type [page 1735]
Header toolbar	Analytical list page List report Object page Overview page	Header Toolbar	Actions [page 1605] Enabling Actions in Object Page Header [page 1693] Adding Custom Actions Using Extension Points [page 1831] Custom Actions [page 2025] Defining Custom Actions [page 1923]
Highlighting line items based on criticality	List report Object page		Highlighting Line Items Based on Criticality [page 1745]
Icon tab bar	Analytical list page List report Object page	Icon Tab Bar	Defining and Adapting Sections [page 1698] Multiple Views on List Report Tables [page 1645]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Image	Analytical list page List report Object page Overview page	Image	Displaying Images in Tables [page 1763] Setting up the Object Page Header [page 1665] List Cards [page 1953] Link List Cards [page 1963] Using Images, Initials, and Icons [page 1618]
Inline creation of table entries	Object page		Enabling Inline Creation of Table Entries on Object Page [page 1769]
Input field	Analytical list page List report Object page Overview page	Input Field	Adapting the Smart Filter Bar [page 1661] Defining and Adapting Sections [page 1698]
Interactive chart	Analytical list page	Interactive Chart	Visual Filter Setup [page 1885]
Label	List report Object page Overview page	Label	Defining and Adapting Sections [page 1698]
Line chart	Analytical list page Object page	Line Chart	Chart-Only View [page 1910] Smart Chart Facet [page 1705]
Line micro chart	Analytical list page List report Object page Overview page	Line Micro Chart	Adding a Smart Micro Chart to a Table [page 1749] Smart Micro Chart Facet [page 1673] List Cards [page 1953]
Link	Analytical list page List report Object page Overview page	Link	Adding a Contact Quick View to a Table [page 1756] Configuring the List Area [page 1957]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Link list card	Overview page	Link List Card	Link List Cards [page 1963]
List card	Overview page	List Cards	List Cards [page 1953]
Manage cards	Overview page		Customizing Overview Pages Using Runtime Capabilities [page 2038]
Menu button	List report Object page	Menu Button	
Message box	Analytical list page List report Object page Overview page	Message Box	Adding Confirmation Popovers for Actions [page 1782]
Message page	Analytical list page List report Object page Overview page	Message Page	
Message popover	List report Object page	Message Popover	Using Messages [page 1610]
Message toast	Analytical list page List report Object page Overview page	Message Toast	
Micro chart/ Smart micro chart	Analytical list page List report Object page Overview page	Micro Chart	Adding a Smart Micro Chart to a Table [page 1749] Smart Micro Chart Facet [page 1673] List Cards [page 1953] Smart Micro Chart [page 2424]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Multi-combo box	Analytical list page List report Object page	Multi-Combo Box	
Multi-input field	Analytical list page List report Object page Overview page	Multi-Input Field	Using the Smart MultiInput Control on the Object Page [page 1720]
Multiple selection of lines in tables	List report Object page		Enabling Multiple Selection in Tables [page 1741]
Multiple views on list report tables	List report		Multiple Views on List Report Tables [page 1645]
Navigation	Analytical list page List report Object page Overview page	Navigation	Configuring Navigation [page 1563]
P13n dialog	Analytical list page List report Object page	P13n Dialog	
Popover	Analytical list page List report Object page Overview page	Popover	Value Help as a Dropdown List [page 1617]
Prefilling fields when creating a new entity	List report Object page		Prefilling Fields When Creating a New Entity [page 1783]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Progress indicator	Analytical list page List report Object page	Progress Indicator	Adding a Progress Indicator to a Table [page 1747] Progress Indicator Facet [page 1686]
Quick view	Overview page	Quick View	Quick View Cards [page 1974]
Quick views for smart link navigation	List report Object page		Enabling Quick Views for Smart Link Navigation [page 1567]
Radial micro chart	Analytical list page List report Object page Overview page	Radial Micro Chart	Adding a Smart Micro Chart to a Table [page 1749] Smart Micro Chart Facet [page 1673] List Cards [page 1953] Smart Micro Chart [page 2424]
Rating indicator	Analytical list page List report Object page	Rating Indicator	Adding a Rating Indicator to a Table [page 1746]
Related apps button	Object page		Enabling the Related Apps Button [page 1697]
Resizing cards	Overview page		Customizing Overview Pages Using Runtime Capabilities [page 2038]
Responsive table	Analytical list page List report Object page Overview page	Responsive Table	Table-Only View [page 1902] Setting the Table Type [page 1735] Table Cards [page 1937]
Reuse components	Object page		Including Reuse Components on an Object Page [page 1721]
Side effects	List report Object page		Side Effects [page 1785]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Search	Analytical list page List report Object page Overview page	Search	Enabling the Search Function [page 1662]
Segmented buttons	Object page	Segmented Buttons	Adding Segmented Buttons to a Table Toolbar [page 1766]
Smart filter bar	Analytical list page List report Overview page	Filter Bar	Compact Filter Setup [page 1884] Defining ValueList Annotation [page 1896] Adapting the Smart Filter Bar [page 1661] Disabling the Editing Status Filter [page 1660] Adding Custom Fields to the Smart Filter Bar [page 1839] Configuring the Global Filter [page 1934] Custom Filters [page 2030]
Smart link	Analytical list page List report Object page Overview page	Smart Link	Adding a Contact Quick View to a Table [page 1756] Enabling Quick Views for Smart Link Navigation [page 1567]
Smart table	Analytical list page List report Object page Overview page	Smart Table	Table-Only View [page 1902] Setting the Table Type [page 1735] Table Cards [page 1937]
Sorting on cards	Overview page		Configuring Sort Properties [page 2001]
Stack card	Overview page	Stack Card	Stack Cards [page 1970]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Stacked bar micro chart	Analytical list page List report Object page Overview page	Stacked Bar Micro Chart	Adding a Smart Micro Chart to a Table [page 1749] Smart Micro Chart Facet [page 1673] List Cards [page 1953] Smart Micro Chart [page 2424]
Status colors and icons	List report Object page		Status Colors and Icons [page 1784]
Table card	Overview page		Table Cards [page 1937]
Table personalization	Analytical list page List report Object page Overview page	Table Personalization (Overview)	
Tables	Analytical list page List report Object page Overview page	Tree Table Responsive Table Grid Table Analytical Table (ALV)	Configuring Tables [page 1735] Table-Only View [page 1902] Setting the Table Type [page 1735] Table Cards [page 1937]
Table toolbar	Analytical list page List report Object page	Table Toolbar	Adding Segmented Buttons to a Table Toolbar [page 1766] Adaptation Extension Example: Adding a Button to the Table Toolbar in the List Report [page 1856]
Text	Analytical list page List report Object page Overview page	Text	Plain Text Facet [page 1669]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Text area	Analytical list page List report Object page Overview page	Text Area	
Title	Analytical list page List report Object page Overview page	Title	Adding Titles to Object Page Tables [page 1765] Adapting the Object Page Title and Subtitle [page 1667] Changing Default Titles for Unnamed Objects [page 1792] Configuring the Table Card Header Area (Optional) [page 1949]
Token	Analytical list page List report Object page Overview page	Token	
Tree table	Analytical list page List report Object page	Tree Table	Setting the Table Type [page 1735] Example: Adding Columns to a Tree Table in the List Report [page 1829]
Unit of measure on cards	Overview page		Setting Units of Measure [page 2004]
Value help	Analytical list page List report Object page Overview page	Value Help Dialog	Defining ValueList Annotation [page 1896] Value Help as a Dropdown List [page 1617]
Variant management	Analytical list page List report Object page Overview page	Variant Management	Managing Variants [page 1616] Descriptor Configuration [page 1869] Enabling Variant Management [page 1637] Creating a List Report Without Variant Management [page 1639] Descriptor Configuration [page 1931]

Controls, UI Elements, Features	Supported Floorplans	UI Design Guidelines	Help
Visual filter bar	Analytical list page	Visual Filter Bar	Visual Filter Setup [page 1885]

How To Use SAP Fiori Elements

Creating an app with SAP Fiori elements generally consists of the following steps:

- Prepare OData services
  You access the required back-end system information in your app using OData services.
  For more information, see Preparing OData Services [page 1550].
- Prepare UI annotations
  You use annotations to enable or modify certain default features and functionality.
  For more information, see Working With UI Annotations [page 1551].
- Build UI applications
  You create your project using the wizard in the SAP Web IDE.
  For more information including details about mandatory and optional post-generation tasks, see Building an App Using SAP Web IDE [page 1553].
- Extend SAP Fiori elements-based apps
  As an optional step, you can extend your app if needed, that is, in cases in which the manifest settings or annotations do not allow you to achieve the desired app behavior.
  For more information, see Extending SAP Fiori Elements-Based Apps [page 1585].

Preparing OData Services

You access the back-end system information using OData services. SAP Fiori elements support OData version 2 with vocabulary-based annotations.

The qualities that your service requires depend on the template you are using and how you want to use it, for example, whether it is read-only or editable. If it is editable, your service must support create, read, update, and delete (CRUD) operations and draft document handling.

Implement additional actions that must be supported as part of your OData service. If the data is represented in a list page, `$count` must be supported as well as `$filter` (for all filterable properties).

Creating a New OData Service

The easiest way to create a new OData service in an ABAP back-end system is to establish a Core Data Services (CDS) consumption view and generate the OData service from this view (data-source driven service creation). Combined with Service Adaptation Description Language) (SADL) this provides the following advantages:

- CRUD request handling is covered by a Business Object Processing Framework (BOPF) model that can be implicitly generated and controlled by the following annotations:

```
@ObjectModel.writeEnabled: true
        @ObjectModel.writeDraftPersistance: '<BO name>'
```

- Locking is handled by BOPF (implicit lock with modification requests)
- Paging is implicitly handled by SADL
- Draft document handling is implicitly covered if `@ObjectModel.writeDraftPersistance` is specified. However, you must also specify annotations, as defined in the draft specification.

For more information, see the SAP - ABAP Programming Model for SAP Fiori.

As an alternative, SADL support is also available in the SAP Gateway Service Builder (transaction `SEGW`).

This option is described in detail in Generating an OData Service Based on a Referenced Data Source.

More Information

- For more information about the BOPF model, see SAP - BOPF Developer Guide.
- For more information about ABAP CDS development, see UI Annotations.

Working With UI Annotations

Vocabularies and annotations allow you to extend OData services by adding information about how to interpret the OData service and its data. For a general introduction to vocabularies and annotations, see the following links:

- Vocabulary Based Annotations
- Vocabularies

On the SAP Gateway front-end server, you can find SAP-specific vocabularies in the SAP Gateway Service Builder (transaction `SEGW`) under *Extras* > *Vocabulary Repository*.

The following types of vocabulary-based annotations are available:

- In-place: These are part of the service's metadata document.
- Ex-place: These are composed of an Annotation Provider Class (APC) outside the metadata document. The APC is bound to the service using a registration in transaction `/IWBEP/REG_VOCAN`.

These annotations are available using a query to the SAP Gateway catalog service, `/sap/opu/odata/IWFND/CATALOGSERVICE;v=2/`. Entity Set: ‚Annotations‘.

Reusability

If the OData service is editable instead of read-only, add annotation elements as needed to activate or control the draft infrastructure. Existing read-only OData services that are annotated, such as fact sheets, can be reused for the list report and object page templates under these circumstances:

- The annotations have to be stored in the back-end system. They also need to have the life cycle of the data model in the back-end system. Note that facet texts need to be defined in a front-end file after generation.
- The OData service models (entity sets and entity types) are derived from CDS views. The CDS entities are the primary artifacts for the data model. The transactional runtime model (based on the BOPF) is generated based on annotations.
- The UI annotations are attached to CDS views using the tags as defined by the SAP - ABAP Programming Model for SAP Fiori.
- In the front end, CDS UI annotations are exposed generically and dynamically through SAP Gateway APIs (APC) as OData Version 4.0 annotation documents that can be addressed separately.
- Although you can use any annotation source, we recommend using CDS annotations in metadata extensions and exposing them using SAP Gateway and SADL. The exposure generates OData annotations (XML format) from the CDS annotations. The following sections explain which OData annotation controls each UI element.
  For more information, search for *SAP-ABAP Programming Model for SAP Fiori* and for *CDS Annotations* in the documentation for your SAP NetWeaver version on SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

Actions

General actions are available for draft-enabled documents (edit, save, cancel/discard). You can also define additional actions using annotations.

The draft-handling actions are handled by the Business Object Processing Framework (BOPF). However, you must back up the application-specific actions using an implementation in the OData service. For general information about actions and how to set them up, see Actions [page 1605].

Each action corresponds to an OData function import.

Field Control

> **i Note**
>
> This is relevant only for list report and object pages, worklists, and analytical list pages.
>
> Field controls are omitted from the list report page since it is a read-only page. In a list report page, field controls are conisdered only if there are custom actions that reference field controls.

You can use field controls to display a UI field as mandatory or read-only, and to hide the field. Field control information is partly static information and valid for all business document instances. However, most use cases

are dynamic and reflect the state of the UI application, business document, or user context and must be controlled by the business logic.

As business logic is implemented in the OData service, the OData service also has to provide the relevant field control information. This is valid for the static information the service metadata contains, as well as for dynamic information that is part of the entity data. There is a specification for an SAP extension of the OData protocol based on annotations for OData Version 2.0 that also covers field control. For more information, see SAP Annotations for OData Version 2.0.

The controls interpret and apply field control information automatically provided by the OData service. When using a smart field, binding the control's `value-Property` to a property in the OData model to achieve field control as defined by the OData service is sufficient. Field control of the OData service may be overruled by setting more restrictive properties for the smart field.

Sample Code

Snippet of XML-View definition

```
...
<!-- Field-control as defined in OData service -->
<SmartField value="{Description}" />

<!-- Overrule field-control of OData service - read-only here -->
<SmartField value="{Name}" editable="false" />
...
```

Building an App Using SAP Web IDE

You use SAP Web IDE to build UI applications using the SAP Fiori elements.

This video shows the step-by-step procedure for building an app. In this example, a list report and object page are created: .

For more information about creating apps, see the following detailed procedures:

1. Creating a Project [page 1554]
2. Checking Folder Structure and Project Artifacts [page 1555]
3. Replacing Standard UI Texts [page 1557]
4. Adding Cards to an Overview Page [page 1561] (only relevant for overview page)
5. Further Post-Generation Steps [page 1562]
6. Using the Extension Wizard [page 1562]

For more information about SAP Web IDE, see Getting Started with SAP Web IDE.

Creating a Project

You use SAP Web IDE to create an SAP Fiori app using SAP Fiori elements.

As an app developer, you must define a configuration in the SAP Web IDE. The main aspects are the destinations to the back-end metadata, navigation between pages, and page design (as pages may contain several templates).

The following figure shows the application in SAP Web IDE that starts the wizard for creating a new project:

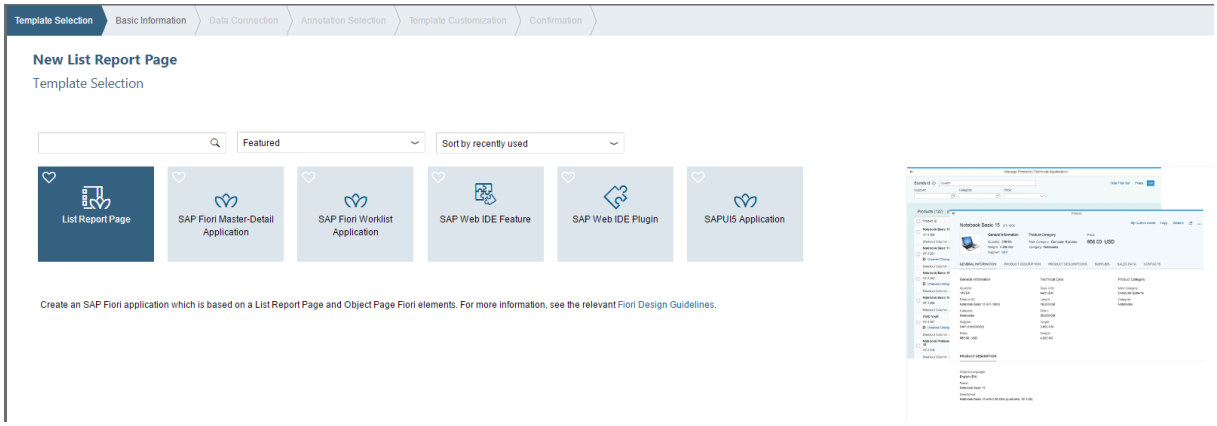


Figure 250: Template Selection Screen in SAP Web IDE

Prerequisites

- You have created an OData service in your ABAP back-end system. For more information, see Preparing OData Services [page 1550].
- You have created annotation files, if required.
- You have access to SAP Web IDE 1.17 or higher. For information about how to access SAP Web IDE, see App Development Using SAP Web IDE [page 44].
- You have an aggregate based entity sets for creating an analytical list page application.

Procedure

1. In the SAP Web IDE, go to the *File* menu, then choose *New* > *Project from Template*. The system starts the wizard for new projects.
2. Follow the guided procedure:

Step	Action
1. Template Selection	Select one of the SAP Fiori element floorplans and choose *Next*.

Step	Action
2. Basic Information	Enter the data that is relevant for your floorplan. Choose *Next*.
3. Data Connection	1. Choose *Service Catalog* and select the required data source from the list. 2. Choose a service and then choose *Next*.
4. Annotation Selection	Select the required annotation file and then choose *Next*.
5. Template Customization	1. Enter the data that is relevant for your floorplan. 2. Choose *Next* and then *Finish*.

3. Open your project (already selected in the project list).
4. Open the *webapp* folder.
5. Select *Component.js* and choose *Run*.

If you get a message that variants can't be loaded, choose *OK* to continue.

More Information

For more information about deploying new applications from SAP Web IDE to different servers, see Deploying Applications.

Checking Folder Structure and Project Artifacts

Once you have applied the template, the generated application is ready to run.

The new app or component reuses the views and controllers from `sap.suite.ui.generic.template`. You can find the destinations in the `neo-app.json` file. The resource links and route definitions for navigation are in the app descriptor file (`manifest.json`).

The following artifacts are generated:

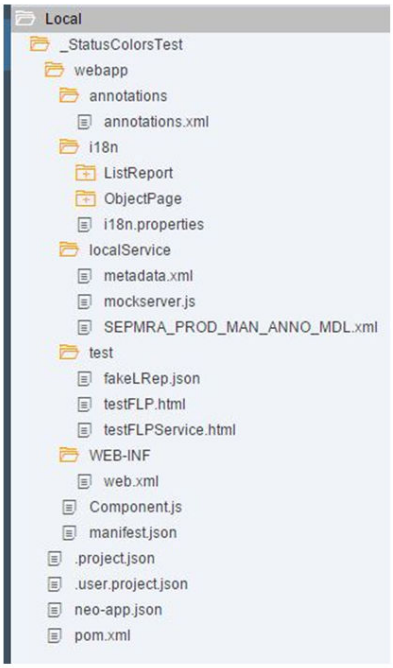


Figure 251: Folder Structure for List Report and Object Page

Component.js

After you generate the application, the SAPUI5 component `Component.js` that represents the application, can be executed. This component links to the manifest where further information can be found at runtime. This informations includes the OData resource links or the reference to the template.

> ⚠ Caution
>
> Do not edit or change this file.

The generated app uses the reuse component controller by referencing a template that uses transactional processing including draft-save. Similarly, the generated app uses the generic view in the template's folder, which is based on the XML templating [page 1018] approach.

Manifest.json

You can find the service and annotation resource links that you have entered in the configuration wizard in the app descriptor (`manifest.json` file).

Here, the annotations are read using the catalog service of SAP Gateway. This is the standard method when using the auto-exposure feature of the application infrastructure.

The local resources `localService/metadata.xml` and `localService/SMART_PROD_MAN_ANNO_MDL.xml` are generated for local tests that want to simulate back-end access.

Besides the annotations that are retrieved from the back-end system, the local resource `annotations.xml` plays a role at runtime, as this file contains the facet descriptions for the object page. This is where you have to maintain the labels for the different facets.

We recommend that you use i18n properties, for example, to maintain the texts in the related `i18n` resource file. For more information, see Replacing Standard UI Texts [page 1557] and Maintaining Section Texts for the Object Page [page 1559].

> **i Note**
>
> You can use the annotation modeler to maintain `UI.facets`. For more information, see Annotation Modeler.

Neo-app.json

You can find the routing information in the `neoapp.json` file that is based on the destination you have chosen:

```
{
  "welcomeFile": "index.html",
  "routes": [
    {
      "path": "/sap/opu/odata",
      "target": {
        "type": "destination",
        "name": "<DestinationName>",
        "entryPath": "/sap/opu/odata"
      },
      "description": "<YourDescription>"
    },
    {
      "path": "/sap/bc/lrep",
      "target": {
        "type": "destination",
        "name": "<DestinationName>",
        "entryPath": "/sap/bc/lrep"
      },
      "description": "<DestinationName>"
    },
```

You can adapt the destinations in order to address a different back-end system. This option is also available under ▶ *Run Configurations* ▶ *Advanced Settings* ◼.

Replacing Standard UI Texts

If required, you can replace standard UI texts for apps that you have created with SAP Fiori elements.

Standard texts are available in the generic framework (for example, the button texts for draft concepts) and belong to the template components (for example, list report and object page). The following sections describe how you replace texts in your generated apps.

When you have created your specific application component, for example, in the SAP Web IDE, standard texts are available from a specific template component (`sap.suite.ui.generic.template`) and from the generic template component.

Standard texts can be overwritten by application-specific texts. Texts from the *Generic Application Component* cannot be replaced.

Perform the following steps to replace the standard UI texts:

1. Find the resource file for your application.
2. Get the standard UI texts for your application.
3. Copy and paste the standard UI texts to your i18n property file and adapt them as needed.

How to Find the Resource File of Your Application Component

The SAP Web IDE automatically generates the following folders and files when you create an app with SAP Fiori elements:

- <root-folder>
- |—i18n
- |—<shortened template component name>, for example, List Report and Object Page
- |—<entitySet>
- |—i18n.properties

> **i Note**
>
> This file contains instructions on how to find the standard UI texts for your application. You need them for the step How to Replace the Standard UI Texts with Application-Specific Texts [page 1559].

The folder path to the resource model appears as shown below. Since the `manifest.json` file also refers to the title and description of the app, there is a general `i18n.properties` file on the top level:

- `i18n.properties`
- `i18n/ListReport/<entitySet>/POHeaders/i18n.properties`
- `i18n/ObjectPage/<entitySet>/i18n.properties`
- `i18n/ObjectPage/<subEntitySet>/i18n.properties`

> **i Note**
>
> For object pages, the number of i18n files corresponds to the number of object pages defined in the app.

The app descriptor (`manifest.json` file) of a purchase order application, for example, specifies the SAP UI5 models.

Example:

```
"sap.ui5": {

        ...
    "models": {
        "i18n": {
            "type": "sap.ui.model.resource.ResourceModel",
            "uri": "i18n/i18n.properties"
      },

        "i18n|sap.suite.ui.generic.template.ListReport|POHeaders": {
```

```
            "type": "sap.ui.model.resource.ResourceModel",
            "uri": "i18n/ListReport/POHeaders/i18n.properties"
              },
        "i18n|sap.suite.ui.generic.template.ObjectPage|POHeaders": {
                    "type": "sap.ui.model.resource.ResourceModel",
                    "uri": "i18n/ObjectPage/POHeaders/i18n.properties"
              },
        "i18n|sap.suite.ui.generic.template.ObjectPage|POItems": {
                    "type": "sap.ui.model.resource.ResourceModel",
                    "uri": "i18n/ObjectPage/POItems/i18n.properties"
```

The URL reflects the folder path to the resource model. The model's name, `i18n|sap.suite.ui.generic.template.ObjectPage|POHeaders`, is separated by lines used to identify the specific template component and entity set for which the resource model can be enhanced by editing the `i18n.properties` file.

How to Replace the Standard UI Texts with Application-Specific Texts

To replace the standard UI texts, perform the following steps:

1. Go to the final block of the standard UI texts that starts as follows: `#---Final block: texts to be redefined by the application -------`
2. Copy this block to the corresponding i18n property file of your app under `webapp/i18n`.
   Consider the detailed instructions that you may find as comments in the original property file: For example, a text might be relevant only for the root object, or it might also be relevant for detail pages of subitems. In the latter case, if you have defined multiple object pages in your app, you have to copy and adapt each of them.
3. After copying the blocks to the relevant files, adjust the texts as described in the comment. For example, replace the generic text *object* by your entity type name.

> **i Note**
>
> To save translation costs, do not copy and redefine more texts than needed.

Maintaining Section Texts for the Object Page

If you have created sections in the local annotations file, you need to use the `i18n properties` file to edit the texts in the related `i18n` resource file.

> **i Note**
>
> This step is necessary only if you do not maintain sections as CDS annotations, for example, because your backend system does not allow you to do so.
>
> If your back-end system is based on SAP NetWeaver 7.52 or higher (SAP S/4HANA) or on SAP NetWeaver 7.68 or higher (SAP S/4HANA Cloud) respectively, you can maintain sections as CDS annotations.

Sections or facets are stored in the local annotations file. You have to add labels manually after the app has been generated. They should refer to the `i18n.properties` file.

Figure 252: Object Page: Sections = com.sap.vocabularies.UI.v1.Facets

You define the labels in the `i18n.properties` file:

```
…
#XTIT: Facet Label
@GeneralInformation=General Information
#XTIT: Facet Label
@TechnicalData=Technical Data
#XTIT: Facet Label
@ProductCategory=Product Category
#XTIT: Facet Label
@ProductDescription=Product Description
#XTIT: Facet Label
@ProductDescriptions=Product Descriptions
#XTIT: Facet Label
@Supplier=Supplier
#XTIT: Facet Label
@Contacts=Contacts
…
```

For more information about resources or `i18n.properties` files, see Replacing Standard UI Texts [page 1557].

Maintaining Standard Text for Smart Tables on the Object Page

In a smart table, if the system does not find any entries when using the smart filter, a standard UI text that you can adapt in the i18n file of your app is displayed.

If you want to display your own text instead of the standard text, use the `NOITEMS_SMARTTABLE` key. If there are multiple tables on the object page, you can provide separate texts for each table using the following key: `NOITEMS_SMARTTABLE|<EntitySetName>|<SmartTableId>`.

> **i Note**
>
> This text is not in the i18n file by default. You have to insert it if you want to change the standard text.

Adding Cards to an Overview Page

Add cards to populate the overview page that you created.

Procedure

1. On the *Development* tab, select the overview page project that you created, and choose *File* > *New* > *Card*.

   > **i Note**
   >
   > If you have created a Multi-Target Application, select the Multi-Target Application project that you created and choose *File* > *New* > *Card*.

2. Select an existing data source or create a new data source for the card.
3. Select one of the following card types:

Option	Description
List Card	Displays an array of items in a vertical list. A number of list types is available.
Link List Card	Displays an array of items in a vertical list with title, picture, icon, or subtitle.
Table Card	Displays items in a table with three-columns.
Stack Card	A collection of single-object cards. When opened, users can perform actions on the individual items in the stack.
Analytic Chart Card	This type of chart card show data in a variety of formats. For example, they can be cards that display data in a series of data points connected by straight lines that use bubbles to visualize the data dimension, or in columns or stacked columns to help view multiple measures or dimensions.

   > **i Note**
   >
   > Overview Page lets you configure view switch and KPI header section for List, Table, and Analytic Chart cards. Selecting the checkbox:
   >
   > - *Select to enable view switch for this card* lets you configure multiple views, apply different filtering, and sorting options in the card.
   > - *Select to add KPI header for this card* lets you configure KPI header information in the card. If you are using an SAP Smart Business Modeler Apps for SAP S4HANA, click *Next* and choose *Select KPI Annotation* step to configure KPI information.

   For more information about the different card types, see Cards Used in Overview Pages.

4. Different card types require different configuration details. Fill in the required details for the selected card type.

5. Choose *Finish* to complete the wizard.
6. Build and run your application.
    - Open your project (already selected in project list).
    - Open the *webapp* folder.
    - Select *Component.js* and choose *Run*.

Further Post-Generation Steps

To enable or modify certain default features for apps based on SAP Fiori elements, you can modify the `manifest.json` file and adapt the annotations, after you have generated the app.

Apart from this, you can also extend generated apps using the extension wizard in SAP WebIDE, or manually. For more information, see Using the Extension Wizard [page 1562].

To extend generated apps manually, see Extending SAP Fiori Elements-Based Apps [page 1585].

You have several options with which to manually enhance the generated app. They are described in the floorplan-specific configuration sections:

- List Report and Object Page [page 1622]
- Worklist [page 1866]
- Analytical List Page [page 1868]
- Overview Pages [page 1930]

Using the Extension Wizard

For list reports, object pages, and analytical list pages, you can use the extension wizard in the SAP Web IDE to create app extensions.

Context

For information about the extensions you can add using the extension wizard and how to manually extend apps, see Extending SAP Fiori Elements-Based Apps [page 1585].

Procedure

1. In the SAP Web IDE, select your generated app, and choose *File* > *New* > *Extension*. The system starts the extension wizard.
2. Select your SAP Fiori elements template and follow the guided procedure.

Configuring Navigation

SAP Fiori elements control the navigation within an app (internal navigation) and the navigation to and from an app (external navigation).

General navigation aspects are listed in the Navigation section of the SAP Fiori Design Guidelines. For information about navigation options for the overview page, see Configuring Card Navigation [page 1998].

With SAP Fiori elements, the following navigation options are available and can be configured:

- Internal navigation
  - Standard navigation within an app
  - Navigation after executing a function

  For more information, see Configuring Internal Navigation [page 1581].
- External navigation

  A SAP Fiori elements app can be the app from which the navigation is triggered (outbound) or the target of the navigation (inbound). Of course, both can also be the case in the same navigation.
  - Outbound navigation
    - Using a URL
    - Using a semantic object (intent-based navigation)
  - Inbound navigation

  For more information, see Configuring External Navigation [page 1563].

Configuring External Navigation

This section describes the configuration options for navigating from an app (outbound) and navigating to an app (inbound).

Outbound Navigation

You can either specify a URL or associate a semantic object (= intent-based navigation) for external navigation targets.

Using a URL

You have two annotation options: You can either specify the absolute URL explicitly, or you can use a path reference to a property using the annotation `DataFieldWithUrl` as follows:

> **i Note**
>
> This option is supported only by the list report and the object page.

Example `DataFieldWithURL`:

```
<Record Type="UI.DataFieldWithUrl">
    <PropertyValue Property="Value" Path="URL"/>
    <PropertyValue Property="Url" Path=""/>
```

```
</Record>
```

An `m.Link` control is rendered for the property on the list report or object page (if it's in *Display* mode).

Navigation to a Semantic Object (Intent-Based Navigation)

If you associate a semantic object annotation with any property, this establishes Intent-Based Navigation.

An intent is a mechanism that lets users perform actions on semantic objects (such as navigating to a sales order or displaying a fact sheet), without having to worry about the UI technology or technical implementation of the navigation target. Intent-based navigation is necessary in the following cases:

- Depending on the user's role, a different application or view of an application must be displayed.
- You want to define an ambiguous navigation target. This means that, at runtime, a list of potential targets is suggested to the user.

This is an example of a cost center as a semantic object:

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="ZFAR_CUSTOMER_LINE_ITEMS2_SRV.Item/CostCenter">
    <Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"
String="CostCenter"/>
</Annotations>
```

These are your options for intent-based navigation:

- Using a smart link control
  To render a field as a smart link control, you must associate a semantic object annotation with the property. Note that a smart link control is only rendered on the list report or object page if it's in display mode.

  Sample Code

  ```
  <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
  Target="ZFAR_CUSTOMER_LINE_ITEMS2_SRV.Item/CostCenter">
      <Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"
  String="CostCenter"/>
  </Annotations>
  ```

  When a user chooses the link, and only one navigation target is found, direct navigation to the target is triggered. If more than one target is found, the system displays a popover containing some text and links to the targets for the user to choose from. You can enhance the content of this popover and display a quick view containing more information about the navigation target. For more information, see Enabling Quick Views for Smart Link Navigation [page 1567].
- Using a button
  To provide a button for navigation, you annotate a property as a `DataFieldForIntentBasedNavigation`.

  Sample Code

  ```
  <Record Type="UI.DataFieldForIntentBasedNavigation">
      <PropertyValue Property="Label" String="My Button for navigation"/>
      <PropertyValue Property="SemanticObject" String="MySemanticObject"/>
      <PropertyValue Property="Action" String="manage"/>
  </Record>
  ```
- Using a link

To provide a link for navigation, you annotate a property as a `DataFieldWithIntentBasedNavigation`. You can use this type of link in tables and forms, that is, a `DataFieldWithIntentBasedNavigation` can be added to `LineItem` or `FieldGroup` annotations. The link text is set according to the "Value" property (in the example below this is the value of `SomePath`). Note that `sap:unit` annotations are currently not evaluated in this context.

Sample Code

```
<Record Type="UI.DataFieldWithIntentBasedNavigation">
    <PropertyValue Property="Label" String="My Link for navigation" />
    <PropertyValue Property="Value" Path="SomePath" />
    <PropertyValue Property="SemanticObject" String="MySemanticObject"/>
    <PropertyValue Property="Action" String="manage"/>
</Record>
```

For examples of how to use the `DataFieldWithIntentBasedNavigation` annotation, see Form Facet [page 1690] and Adding Line Item Actions in Tables [page 1743].

You can replace standard internal navigation with external navigation by using intent-based navigation. For more information, see Changing Navigation to Object Page [page 1583].

Navigation to the App (Inbound Navigation)

Navigation to the app uses deep linking. For more information, see Navigation in the SAP Fiori Design Guidelines and go to the Deep Links section.

For the SAP Fiori launchpad, the configuration steps for Intent-Based Navigation are also relevant since it's the same mechanism.

Note

You need the SAP Fiori launchpad for this type of navigation. For a stand-alone app, you need to change the links in the annotations as required.

SAP Fiori elements automatically process inbound navigation. Since SAP Fiori elements apps are capable of all modes (create, display, edit), you should define "manage" as the action for inbound navigation . If the parameters provided are specific enough to define an instance, the app navigates directly to the object page. Otherwise, the list report is shown with filters set according to the provided parameters.

Enter the `preferredMode` parameter to specify the mode in which the object page should be opened:

PreferredMode Parameter	Results in the following mode
display	Object page opens in display mode unless the user is already working on a draft. In this case, the draft is opened in edit mode.
edit	A draft is created if one doesn't exist yet. If an outdated draft by another user exists ("unsaved changes"), the user can decide to cancel this draft and create their own, or to keep the other user's draft and open it in display mode. If another user's draft exists and is not outdated, the page is opened in display mode.

PreferredMode Parameter	Results in the following mode
create	A new document is created. You can use URL parameters to prefill specific values. Example: To set the value 01 for the *DefectCategory* field, enter the URL ...`#Defect-displayWorklist?preferredMode=create&DefectCategory=01`. **i Note** The target application must specify in its manifest.json which parameters are to be used from the incoming URL. In the following example, only the `Supplier` parameter is used. `"sap.ui.generic.app": { "_version": "1.2.0", "settings": { ... "inboundParameters": { "DefectCategory": { "useForCreate": true } } }, "pages": [ { ...`
mode	If the specified mode isn't suitable for the current draft state, there is no silent fallback. However, the user gets an error message before the object page is opened in the potentially available mode.

i Note

There is no double navigation. If no object page is defined in the target app's manifest or the (usually internal) navigation to the object page is overridden by external navigation, the list report is shown.

Inbound Navigation to Subobject Pages

You can configure inbound navigation to any subobject page belonging to an app by using deep linking.

To do so, make the following settings:

1. In the target application's manifest, for the corresponding subobject page, set `allowDeepLinking: true`. The example below shows the subobject page section of the manifest.json:

```
"pages": [{
                                "navigationProperty": "to_ProductText",
                                "entitySet": "STTA_C_MP_ProductText",
                                "component": {
                                        "name":
"sap.suite.ui.generic.template.ObjectPage",
                                        "settings": {
                                                "allowDeepLinking": true
                                                }
                                }
                            },
```

```
                              {
```

If multiple pages on the same level have this property set to `true`, the entry that comes first in the manifest is used.
Navigation to any level is possible by setting `allowDeepLinking: true` for each level. Note that each level needs to have the setting `allowDeepLinking: true`. In the following example, the navigation goes to subobject level 3:

```
Object Page1: {
               SubObjectPage1:{
                              allowDeepLinking:true
SubObjectPage2:{
               allowDeepLinking:true
SubObjectPage3:{
               allowDeepLinking:true
}
}
}
}
```

2. Provide the URL parameters for navigation, as described under https://help.sap.com/viewer/cc1c7615ee2f4a699a9272453379006c/7.5.5/en-US/bd8ae3d327ab4541bcce8e7353c046fc.html.

> **i Note**
>
> - Navigation to any subobject page is only possible using semantic keys. The technical key is used only for the root object page if the semantic key is not available.
> - If the relevant keys (semantic or technical keys) are used for the root object page, along with its values in startup parameters, deep linking to the first-level object page is automatic.

Enabling Quick Views for Smart Link Navigation

You can enrich the popovers for smart link navigation with additional information to display quick views.

Context

You can display information about the navigation target already on the source entity. This information - the quick view - is stored in the association end type. To enable the quick views, you have to annotate `com.sap.vocabularies.UI.v1.QuickViewFacets` at the association end type of the property that has been annotated as a semantic object. If you annotate `QuickViewFacets` for the popover, a new title area and additional information, such as, a field group, are displayed according to the `QuickViewFacets`.

> **i Note**
>
> `QuickViewFacets` can only be annotated for those `EntityTypes` that are in the same service. Only these are referenced with referential constraints in the metadata document.

This video shows the step-by-step procedure for enabling quick views for smart link navigation:

To do so, perform the following steps:

Procedure

1. Identify the property that has been annotated as a semantic object.

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType/Supplier">
   <Annotation Term="Common.SemanticObject" String="EPMProduct"/>
</Annotations>
```

2. In the metadata document, you can find the reference to the association end type. Check for a referential constraint that includes the identified property as `Dependent`. For the `Supplier` property in the entity type STTA_C_MP_ProductType, which has a set of navigation properties, only `to_Supplier` includes the `Supplier` property as `Dependent`.

```
<Association Name="assoc_2CCAF987BA334B3BD1DF2404F50BC9C5" sap:content-
version="1">
    <End Type="STTA_PROD_MAN.STTA_C_MP_ProductType" Multiplicity="1"
Role="FromRole_assoc_2CCAF987BA334B3BD1DF2404F50BC9C5"/>
    <End Type="STTA_PROD_MAN.STTA_C_MP_SupplierType" Multiplicity="0..1"
Role="ToRole_assoc_2CCAF987BA334B3BD1DF2404F50BC9C5"/>
    <ReferentialConstraint>
        <Principal Role="ToRole_assoc_2CCAF987BA334B3BD1DF2404F50BC9C5">
            <PropertyRef Name="Supplier"/>
        </Principal>
        <Dependent Role="FromRole_assoc_2CCAF987BA334B3BD1DF2404F50BC9C5">
            <PropertyRef Name="Supplier"/>
        </Dependent>
    </ReferentialConstraint>
</Association>
```

3. Annotate `UI.QuickViewFacets` under the association end type of the `Dependent` property as follows:

```
<!-- QuickViewFacets annotation for Supplier-->
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_SupplierType">
    <Annotation Term="UI.QuickViewFacets">
        <Collection>
            <Record Type="UI.ReferenceFacet">
                <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#SupplierQuickViewPOC_FieldGroup_1" />
            </Record>
        </Collection>
    </Annotation>
    <Annotation Term="UI.FieldGroup"
Qualifier="SupplierQuickViewPOC_FieldGroup_1">
        <Record>
            <PropertyValue Property="Data">
                <Collection>
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Label" String="Company
Name" />
                        <PropertyValue Property="Value" Path="CompanyName"/>
                    </Record>
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Label" String="Supplier" />
                        <PropertyValue Property="Value" Path="Supplier"/>
                    </Record>
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Label" String="Email
Address" />
```

```
                        <PropertyValue Property="Value" Path="EmailAddress"/>
                    </Record>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

4. Ensure that you have defined `FilterFacets`.

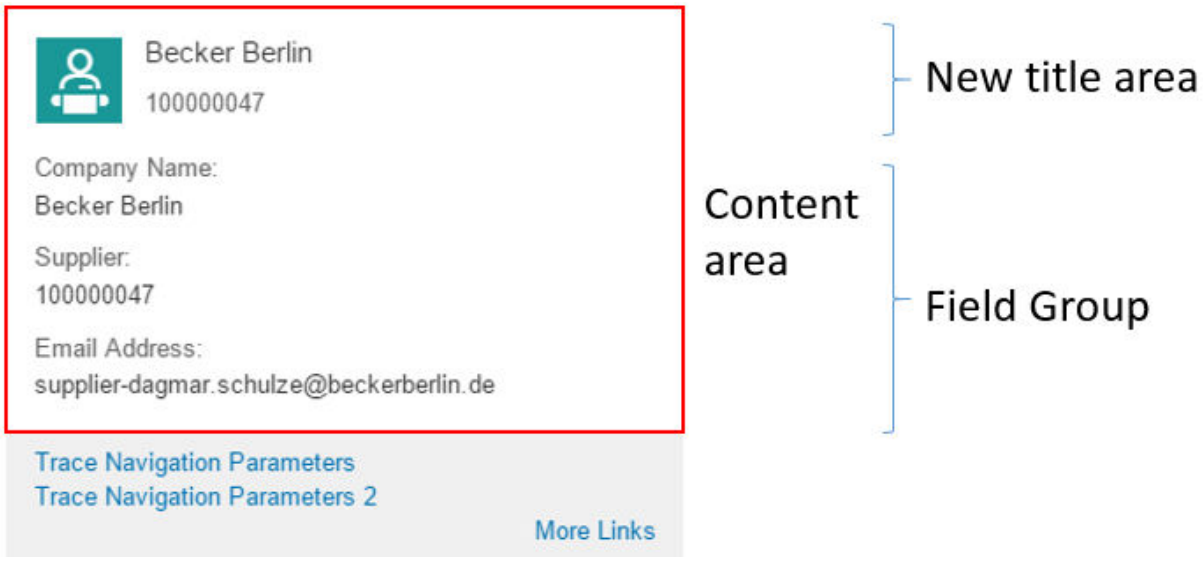
   If you do not define `FilterFacets`, all field groups are displayed in the smart filter bar.

   Below, find an example of a `FilterFacets` annotation on the target entity type:

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_SupplierType">
   <Annotation Term="UI.FilterFacets">
      <Collection>
            <Record Type="UI.ReferenceFacet">
                <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#MyFilterGroup"/>
            </Record>
      </Collection>
   </Annotation>
</Annotations>
```

Results

A quick view for smart link navigation is generated and can look like this:



For more information about the system behavior and configuration options, see Configuring Quick Views for Smart Link Navigation [page 1570].

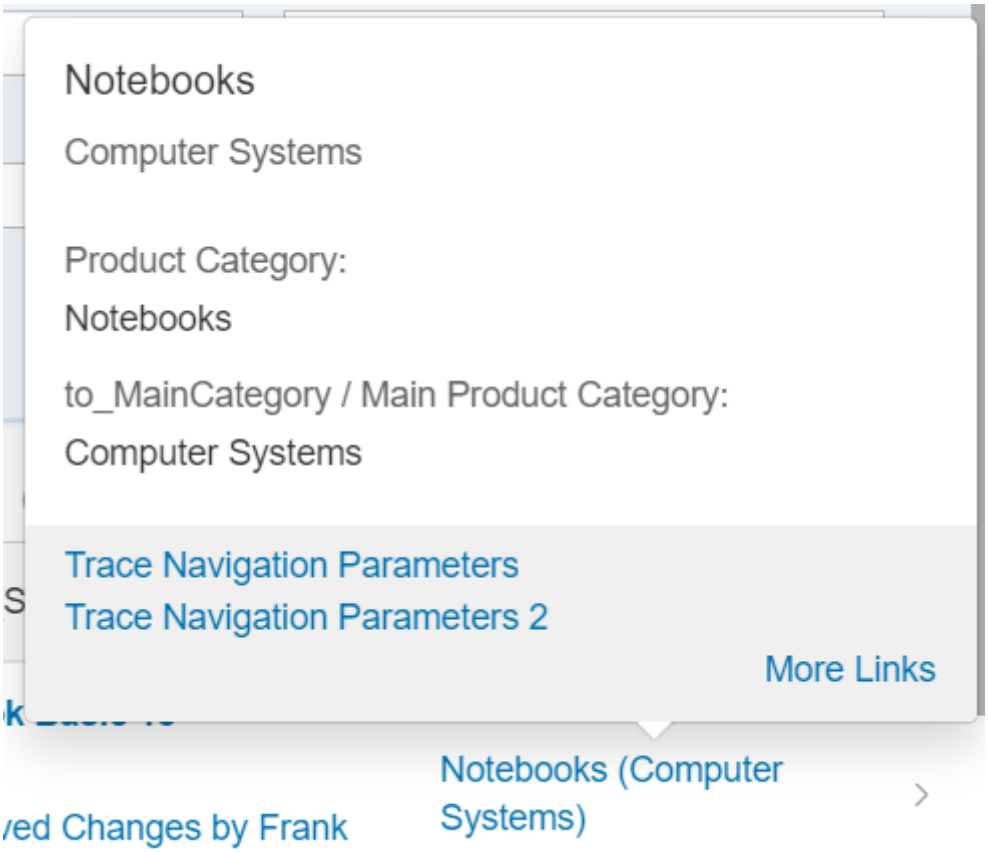
Configuring Quick Views for Smart Link Navigation

You can configure the content area of the quick views to display specific data.

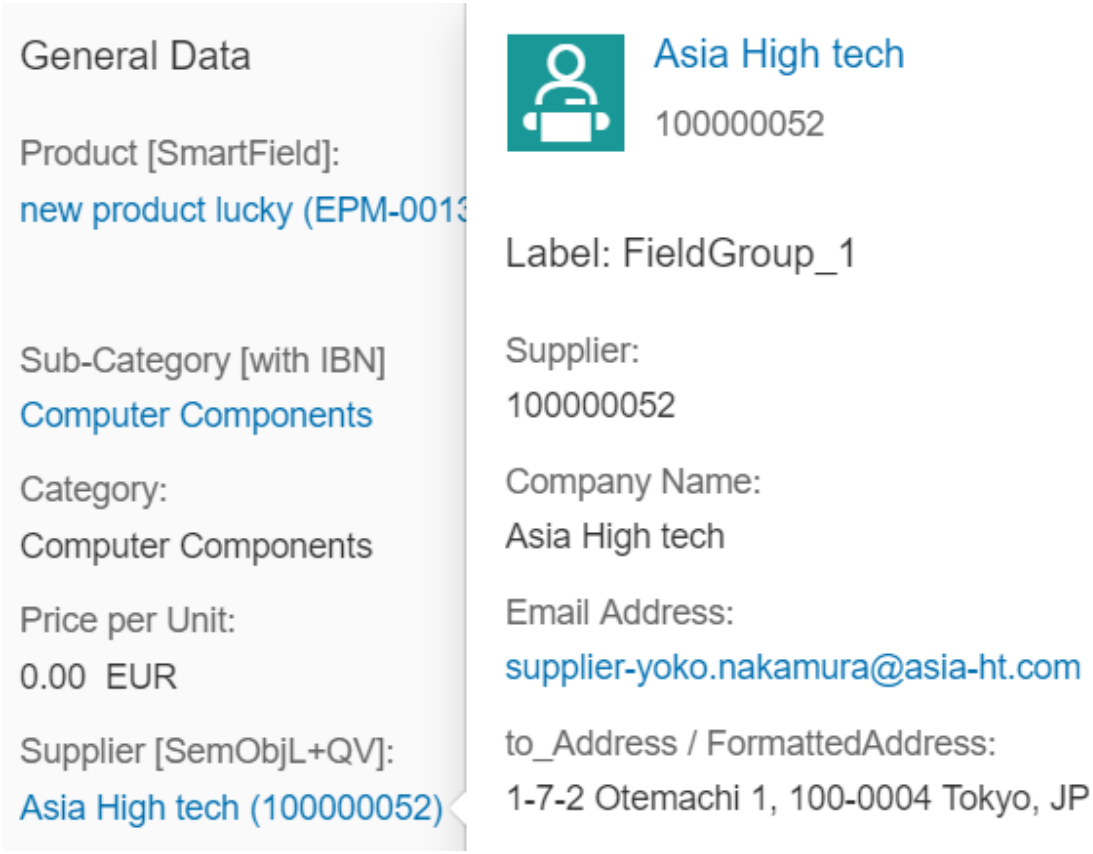
The content area, consisting of a title and additional information, for example, a field group, has a default behavior and can be adapted to your needs.

Title Area

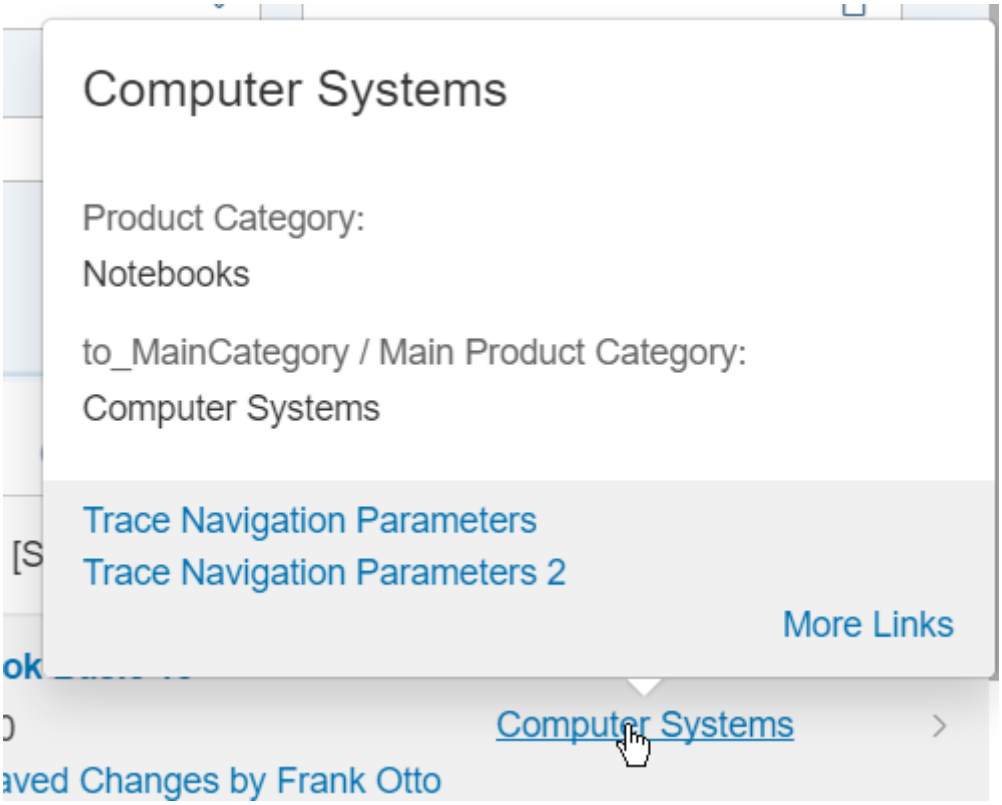
- Images
  - To display an image, annotate `HeaderInfo.ImageUrl` or `HeaderInfo.TypeImageUrl`. If you don't, no image is displayed.
  - If you annotate `HeaderInfo.ImageUrl` and `HeaderInfo.TypeImageUrl`, `ImageUrl` is evaluated first, and `TypeImageUrl` second. The `ImageUrl/TypeImageUrl` string and path including navigation properties are evaluated.
- Title
  - Enter the title according to the `TextArrangement` annotation. See the figure below: `TextArrangementType/TextLast`. Note that *Computer Systems* is declared as `TextLast` here.



- If a main navigation has been defined, the title is displayed as a link. In the example below, see the *Asia High tech* link:



- Description
  - The description is always displayed beneath the title and must be filled according to the `TextArrangement` annotation.
  - If the description is not filled, the title size is increased automatically and the description field remains empty, as shown below (`TextArrangementType/TextOnly`).

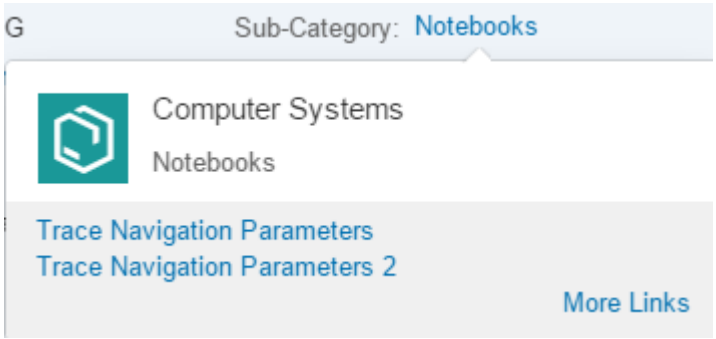


Content Area

The content area can contain field groups, contacts, and Data.Points.

Field Groups

- You can include any number of field groups or none at all. The example below shows a quick view with no reference facet, however, a header image included:

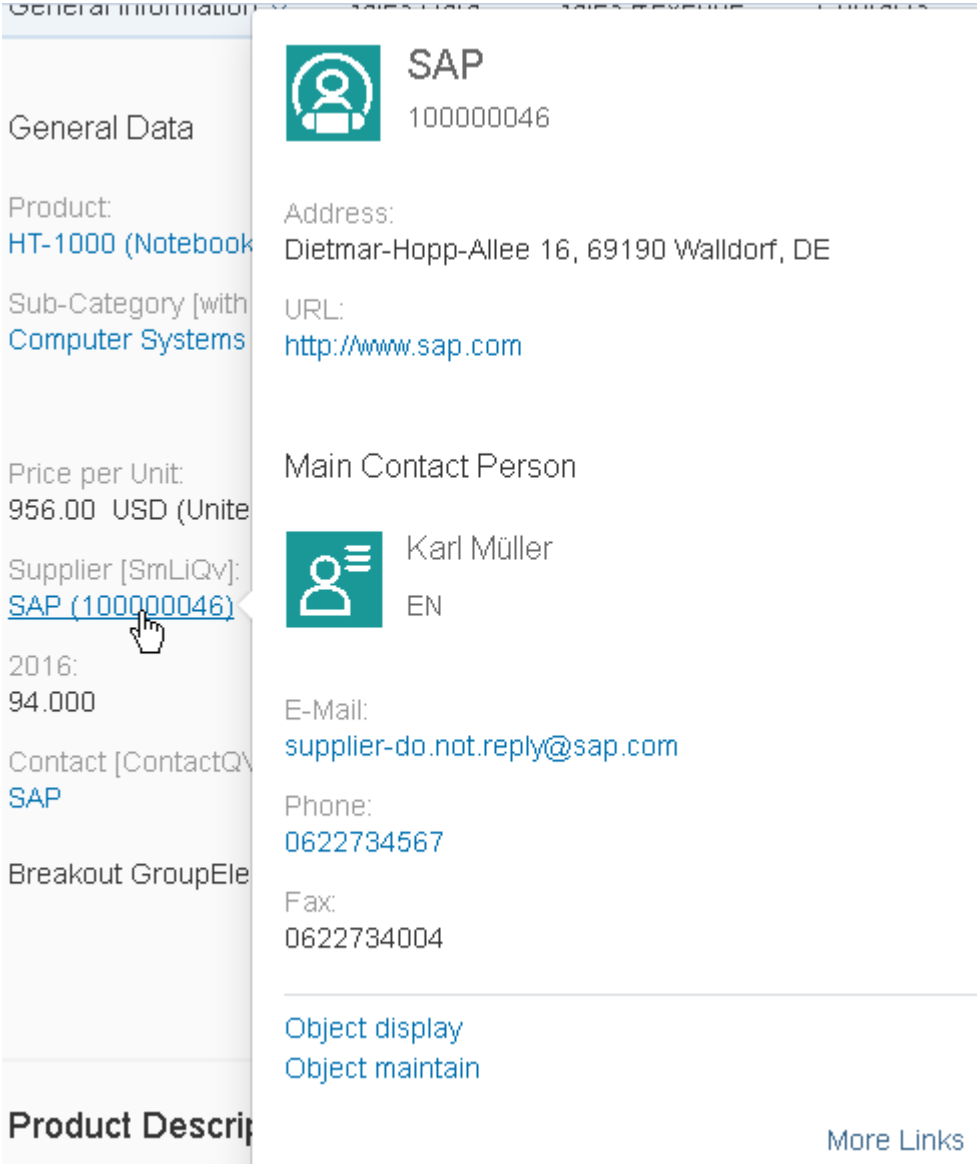


- A field group can have a label. It is taken from the from within the `<Record Type="UI.ReferenceFacet">`.
- For fields, the path including navigation properties is evaluated.
- Fields in the field group are smart fields. They support annotations such as: `IsEmailAddress`, `IsUrl`, and `IsPhoneNumber`. Note that any links that would create a popover on the quick view are ignored by the system.

- There are different types of content for field groups:
  - Interpreted by `SmartField`: `DataField` including criticality, `DataFieldWithUrl`
  - Interpreted by SAP Fiori elements: `DataFieldWithIntentBasedNavigation`

Contacts

You can display any number of contacts or none at all. See the example below:



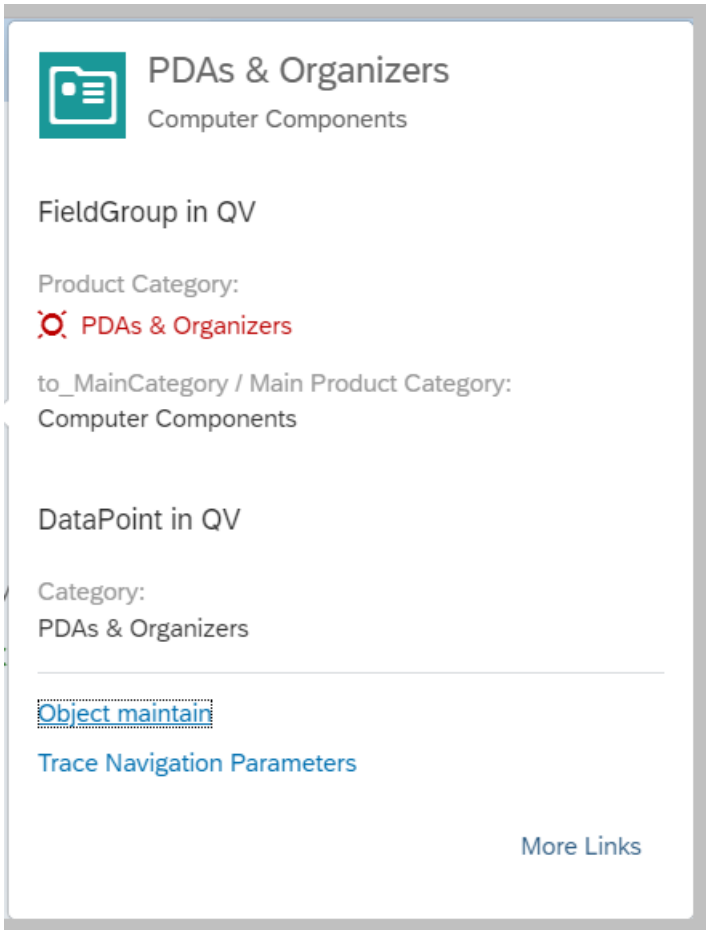
The following applies:

- You can place the contact anywhere. It is specified by the position of the reference facet in the collection.
- If the picture, title, and description belonging to a contact (contact title area) correspond with the content of the title area , the contact title area is not displayed.
- The reference facet must point to a `com.sap.vocabularies.Communication.v1.Contact`.

DataPoints

- You can place an existing DataPoint in your annotation.

- You can place the DataPoint anywhere. It is specified by the position of the reference facet in the collection.
- A DataPoint can have a label. It is taken from within the `<Record Type="UI.ReferenceFacet">`.



The sample code shows a quick view facet containing field group, contact and DataPoint:

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_SupplierType">
   <Annotation Term="UI.QuickViewFacets">
      <Collection>
         <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#SupplierQuickViewPOC_FieldGroup_1" />
         </Record>
         <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="Main Contact Person" />
            <PropertyValue Property="Target"
AnnotationPath="@Communication.Contact#KeyAccount"/>
         </Record>
         <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="DataPoint in QV"/>
            <PropertyValue Property="Target"
AnnotationPath="@UI.DataPoint#Product"/>
         </Record>
      </Collection>
   </Annotation>
</Annotations>
```

Quick Views for Smart Link Navigation: Further Configuration Examples

You have various options for configuring the quick view. This documentation provides some examples.

Example 1

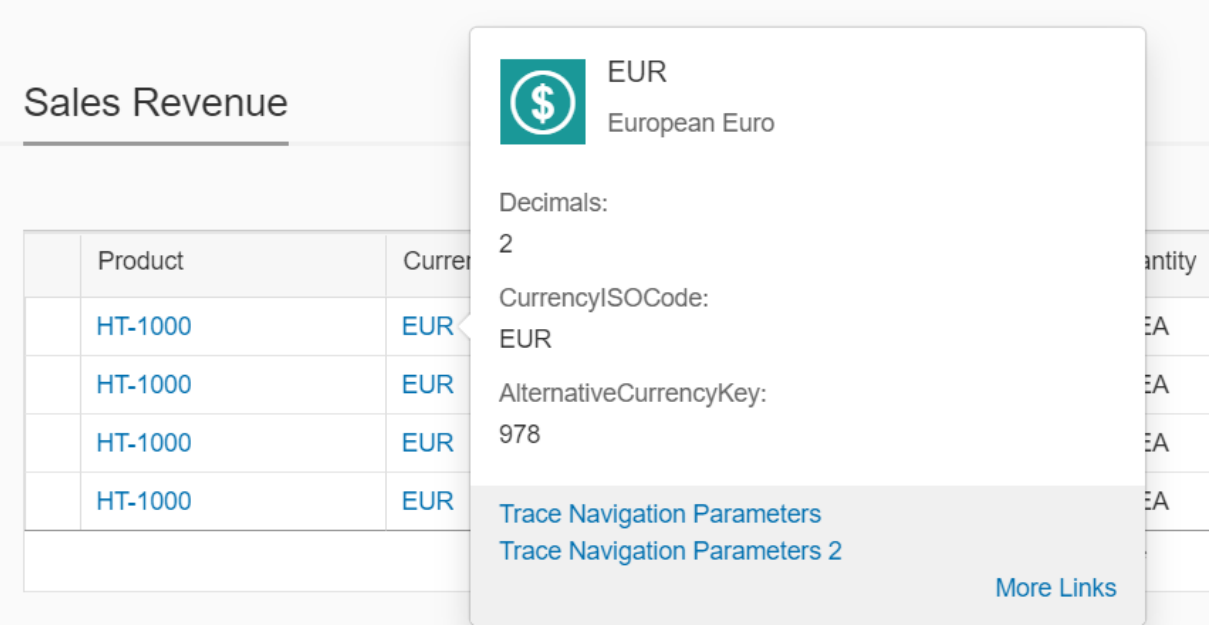

Asia High tech
100000052
Label: FieldGroup_1
Supplier:
100000052
Company Name:
Asia High tech
Email Address:
supplier-yoko.nakamura@asia-ht.com
to_Address / FormattedAddress:
1-7-2 Otemachi 1, 100-0004 Tokyo, JP
Label: FieldGroup_2
Phone Number:
9078563412
URL:
http://www.asia-ht.com
to_Address / AddressValidityStartDate:
Jan 1, 2000
Trace Navigation Parameters
Trace Navigation Parameters 2
More Links

```
<!-- QuickViewFacets annotation for Supplier-->
            <Annotations Target="STTA_PROD_MAN.STTA_C_MP_SupplierType/
EmailAddress" xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation
Term="com.sap.vocabularies.Communication.v1.IsEmailAddress" Bool="true"/>
            </Annotations>
            <Annotations Target="STTA_PROD_MAN.STTA_C_MP_SupplierType/URL"
xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation Term="Org.OData.Core.V1.IsUrl" Bool="true"/>
            </Annotations>
            <Annotations Target="STTA_PROD_MAN.STTA_C_MP_SupplierType/
PhoneNumber" xmlns="http://docs.oasis-open.org/odata/ns/edm">
                <Annotation
Term="com.sap.vocabularies.Communication.v1.IsPhoneNumber" Bool="true"/>
            </Annotations>
            <Annotations
Target="STTA_PROD_MAN.STTA_C_MP_SupplierType">
                <Annotation Term="UI.QuickViewFacets">
                    <Collection>
                        <Record Type="UI.ReferenceFacet">
                            <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#SupplierQuickViewPOC_FieldGroup_1" />
                        </Record>
                        <Record Type="UI.ReferenceFacet">
                            <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#SupplierQuickViewPOC_FieldGroup_2" />
                        </Record>
                    </Collection>
                </Annotation>
                <Annotation Term="UI.FieldGroup"
Qualifier="SupplierQuickViewPOC_FieldGroup_1">
                    <Record>
                        <PropertyValue Property="Label" String="The first field
group label" />
                        <PropertyValue Property="Data">
                            <Collection>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="Supplier" />
                                    <PropertyValue Property="Value"
Path="Supplier"/>
                                </Record>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="Company Name" />
                                    <PropertyValue Property="Value"
Path="CompanyName"/>
                                </Record>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="Email Address" />
                                    <PropertyValue Property="Value"
Path="EmailAddress"/>
                                </Record>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="to_Address / FormattedAddress" />
                                    <PropertyValue Property="Value"
Path="to_Address/FormattedAddress"/>
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
                <Annotation Term="UI.FieldGroup"
Qualifier="SupplierQuickViewPOC_FieldGroup_2">
                    <Record>
```

```
                        <PropertyValue Property="Label" String="The second field
group label" />
                        <PropertyValue Property="Data">
                            <Collection>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="Phone Number" />
                                    <PropertyValue Property="Value"
Path="PhoneNumber"/>
                                </Record>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="URL" />
                                    <PropertyValue Property="Value" Path="URL"/>
                                </Record>
                                <Record Type="UI.DataField">
                                    <PropertyValue Property="Label"
String="to_Address / AddressValidityStartDate" />
                                    <PropertyValue Property="Value"
Path="to_Address/AddressValidityStartDate"/>
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
            </Annotations>
```

Example 2: Quick View in Object Page Table

> **i Note**
>
> Not all columns that are supported in a table support the display of a quick view.



For the currency, no referential constraint is defined by the service. This is why you need to make the following entry:

```
<Association Name="assoc_6D52161C1362D99A31996E5BB23202E8" sap:content-
version="1">
    <End Type="STTA_PROD_MAN.STTA_C_MP_ProductSalesDataType" Multiplicity="1"
Role="FromRole_assoc_6D52161C1362D99A31996E5BB23202E8"/>
    <End Type="STTA_PROD_MAN.I_CurrencyType" Multiplicity="0..1"
Role="ToRole_assoc_6D52161C1362D99A31996E5BB23202E8"/>
    <ReferentialConstraint>
        <Principal Role="FromRole_assoc_6D52161C1362D99A31996E5BB23202E8">
            <PropertyRef Name="Currency"/>
        </Principal>
        <Dependent Role="ToRole_assoc_6D52161C1362D99A31996E5BB23202E8">
            <PropertyRef Name="Currency"/>
        </Dependent>
    </ReferentialConstraint>
</Association>
```

Quick view facets annotation:

```
<!-- QuickViewFacets annotation for Currency-->
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductSalesDataType/Currency">
    <Annotation Term="Common.SemanticObject" String="EPMProduct"/>
</Annotations>
<Annotations Target="STTA_PROD_MAN.I_CurrencyType">
    <Annotation Term="UI.QuickViewFacets">
        <Collection>
            <Record Type="UI.ReferenceFacet">
                <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#CurrencyQuickViewPOC_FieldGroup_1" />
            </Record>
        </Collection>
    </Annotation>
    <Annotation Term="UI.FieldGroup"
Qualifier="CurrencyQuickViewPOC_FieldGroup_1">
        <Record>
            <PropertyValue Property="Data">
                <Collection>
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Label" String="Decimals" />
                        <PropertyValue Property="Value" Path="Decimals"/>
                    </Record>
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Label"
String="CurrencyISOCode" />
                        <PropertyValue Property="Value" Path="CurrencyISOCode"/>
                    </Record>
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Label"
String="AlternativeCurrencyKey" />
                        <PropertyValue Property="Value"
Path="AlternativeCurrencyKey"/>
                    </Record>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

Passing Variant IDs as URL Parameters

You can pass the variant ID as part of the navigation context (or as a URL parameter) while navigating from an application to the analytical list page or to the list report, or vice versa.

You can choose to pass a page variant or a control variant using these parameters:

- `sap-ui-fe-variant-id`: Page variant ID you want to set

  > Example
  >
  > https://abc.com/ui#SalesOrder-analyze_deliv_perf?sap-ui-fe-variant-id=**myVariantId**

- `sap-ui-fe-filterbar-variant-id`: Parameter for the filter bar control variant

  > Example
  >
  > https://abc.com/ui#SalesOrder-analyze_deliv_perf?sap-ui-fe-filterbar-variant-id=**myFilterbarId**

- `sap-ui-fe-chart-variant-id`: Parameter for the chart control variant

  > Example
  >
  > https://abc.com/ui#SalesOrder-analyze_deliv_perf?sap-ui-fe-chart-variant-id=**myChartId**

- `sap-ui-fe-table-variant-id`: Parameter for the table control variant

  > Example
  >
  > https://abc.com/ui#SalesOrder-analyze_deliv_perf?sap-ui-fe-table-variant-id=**myTableId**

  When both the chart variant and table variant are passed:

  > Example
  >
  > https:/abc.com/ui#SalesOrder-analyze_deliv_perf?sap-ui-fe-chart-variant-id=**myChartId**&sap-ui-fe-table-variant-id=**myTableId**

> Sample Code

```
...
"crossNavigation": {
  "inbounds": {
...
 },
"outbounds": {
  "EPMProductManage": {
    "parameters": {
    "sap-ui-fe-variant-id":"id_1542011587281_980_page"
    }
  }
}
}
```

Note

- If you add both a page variant and a control variant to a URL:
  - The page variant-based analytical list page or list report ignores the control variant ID and applies the page level variant
  - The control variant-based analytical list page or list report applies the control variant ID. If the control variant ID is missing, the page variant ID applies to the control
  - If the variant ID passed is invalid, the default or standard variant is considered.
- Adding the variant ID to the URL overrides the user's default variant ID

Related Information

Managing Variants [page 1616]

Configuring Internal Navigation

SAP Fiori elements control the navigation within an app (internal navigation). This section describes the configuration options that you have.

Standard Navigation Within an App

In the manifest.json, you define which pages are available in the app. At the top level, a "pages" map is defined. This map should have only one point of entry. This is the main point of entry to the app and should always be a list report. Each page can have a "pages" map containing all subpages of the given page. Only one subpage is allowed in the list report. This should be an object page for the same entity set. An object page can have several subpages.

You can control whether it is possible to navigate to a detail page. It simply depends on whether you keep the predefined definition of a subpage:

Sample Code

```
"sap.ui.generic.app": {
  "pages": {
    "MyListReport": {
      "entitySet": "MyEntitySet",
      "component": {
        "name": "sap.suite.ui.generic.template.ListReport",
        "list": true
      },
      // Navigation to detail page: eliminate this block if no navigation is
needed
      "pages": {
        "MyObjectPageOnFirstLevel":          {
          "entitySet": "MyEntitySet",
          "component": {
            "name": "sap.suite.ui.generic.template.ObjectPage"
```

```
            },
            "pages": {
                                        "MyFirstObjectPageOnSecondLevel": {
                "navigationProperty": "to_MyOtherEntitySet",
                "entitySet": "MyOtherEntitySet",
                "component": {
                  "name": "sap.suite.ui.generic.template.ObjectPage"
                },
              },

  "MySecondObjectPageOnSecondLevel": {
                "navigationProperty": "to_MyOtherEntitySetAlternateNavigation",
                "entitySet": "MyOtherEntitySet",
                "component": {
                  "name": "sap.suite.ui.generic.template.ObjectPage"
                },
              },

  "MyThirdObjectPageOnSecondLevel": {
                "navigationProperty": "to_MyThirdEntitySet",
                "entitySet": "MyThirdEntitySet",
                "component": {
                  "name": "sap.suite.ui.generic.template.ObjectPage"
                }
              }
            }
          }
        }
      }
    }
  }
```

The same holds true for the navigation to a second object page. This is possible only when the definition is kept in the manifest. If you want to have multiple subpages on the same level, you need to have multiple definitions. The subpages are identified by the corresponding entity set and the navigation property.

A chevron indicates the navigation options. The user can navigate by clicking on the line.

> **i** Note
>
> In a non-draft app, if the user is in edit mode on an object page and has made changes before the navigation has been executed, the system displays a message indicating that the changes will be lost if the user navigates without saving first.

Navigation Between Entities of an App

You can link entities within an app. This allows users to navigate between the entities within the application. You can use this app-internal linking in the object header, in sections, and in tables. For example, within a sales order app, you can link from a sales order to another sales order, from a sales order item to the sales order header, or from a sales order schedule line to a schedule line of another sales order.

> **i** Note
>
> This feature is available only on the object page.

The following example shows how to use the `DataFieldWithNavigationPath` annotation to link entities:

```
<Annotation Term="UI.FieldGroup" Qualifier="NavExample">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataFieldWithNavigationPath">
                    <PropertyValue Property="Label" String="Ref. Sales Order" />
                    <PropertyValue Property="Value" Path="RefSalesOrderID" />
                    <PropertyValue Property="Target"
NavigationPropertyPath="to_SalesOrder" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Navigation after Executing a Function

Navigation may also be required after running a regular `FunctionImport` that is not tagged as a `DataFieldForIntentBasedNavigation`. The behavior after the execution of the function is controlled by the application as follows: If the `FunctionImport` returns `multiplicity 1` and `ReturnType=EntityType + EntitySet`, then the SAP Fiori element navigates to the instance that is returned by the function import.

Changing Navigation to Object Page

Navigation from the list report view to the object page in the same app is enabled by default. If required, you can disable this navigation or replace it with navigation to another app (external navigation).

Disable Navigation

You can use the `manifest.json` file to control whether it is possible to navigate to a detail page by keeping the predefined definition of a subpage. If you wish to disable navigation, follow the instructions in the example below to remove the appropriate code.

Example with Navigation

```
"sap.ui.generic.app": {
            "pages":[{
                "entitySet": "Zfarvd_Bs_Hd_Bo",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true,
                    "settings": {
                        "gridTable": false,
                        "hideTableVariantManagement": false
                    }
                },
//Navigation to detail page: eliminate this block if no navigation is needed
                "pages": [{
```

```
                    "entitySet": "Zfarvd_Bs_Hd_Bo",
                    "component": {
                        "name": "sap.suite.ui.generic.template.ObjectPage"
                    },
```

Enable External Navigation

You can define an external navigation target using intent-based navigation in the `manifest.json` file by modifying the `navigation` > `display` entry. This allows you to overwrite existing internal navigation with external navigation, for example, from a line item in the list report to an object page in a different app, or from an object page to a subpage in another app. The navigation can only be set up from a line in a responsive table, or from the *Show Details* link in a grid table.

In the example below, the standard navigation from the list report to the object page has been replaced with navigation to an object page in another app.

```
    "sap.app": {
        "_version": "1.2.0",
        ...
        "crossNavigation": {
            "inbounds": {},
            "outbounds": {
                "ExampleNavigationTarget":
                {
                    "semanticObject": "EPMProduct",
                    "action": "manage_st"
                }
            }
        }
    ...
...
    "sap.ui.generic.app": {
        "_version": "1.2.0",
        "pages": [{
            "entitySet": "STTA_C_MP_Product",
            "component": {
                "name": "sap.suite.ui.generic.template.ListReport",
                "list": true
            },
            "pages": [{
                "entitySet": "STTA_C_MP_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ObjectPage"
                },
// Navigation to an external target instead of a detail page: Add this block to
set up external navigation.
                "navigation": {
                    "display": {
                        "path": "sap.apps.crossNavigation.outbounds",
                        "target": "ExampleNavigationTarget"
```

> **i Note**
>
> - The example above applies to `sap.ui.generic.app->_version` 1.2.0.
> - The path and target you specify for external navigation must point to an existing `outbounds` entry.

For more information about the `crossNavigation` attribute in the `sap.app` namespace, see Descriptor for Applications, Components, and Libraries [page 734]. In *Table 2: Attributes in the sap.app namespace*, go to the `crossNavigation` attribute and see the description for `outbounds`.

Extending SAP Fiori Elements-Based Apps

You can extend your application if needed.

There are two options to extend your app:

- App extensions: Are made by developers during the creation of an SAP Fiori elements-based app, using framework extension points provided by SAP Fiori elements. The extensions are made, for example, using manifest changes or SAPUI5 extension points, depending on the floorplan. For more information, see Extending Generated Apps Using App Extensions [page 1585].
- Adaptation extensions: Let customers and partners introduce their own functionality to an existing app, as part of an adaptation project, in a consistent and upgrade-safe manner. For more information, see Extending Delivered Apps Using Adaptation Extensions [page 1596].

  > **i Note**
  >
  > This option is possible only for list report, object page, and analytical list page.

Extending Generated Apps Using App Extensions

You can use framework extension points to extend your generated app during the creation process.

The extensions are made, for example, using manifest changes or SAPUI5 extension points, depending on the floorplan.

> **i Note**
>
> Make sure you have read the following information: Read Before Extending a Generated App [page 1586].
>
> > **⚠ Caution**
> >
> > Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
> >
> > After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

For list reports, object pages, and analytical list pages, you can either use the extension wizard in the SAP Web IDE to create app extensions or you can create them manually.

You can use SAP WebIDE to add the following app extensions:

- List report
  - Filter
  - Action
  - Column
- Object page
  - Action
  - Facet
  - Column
  - Header
  - Form
- Analytical list page
  - Filter
  - Action
  - Column

Related Information

Extending List Reports and Object Pages Using App Extensions [page 1799]
Configuring Analytical List Page App Extensions [page 1915]
Configuring Overview Page App Extensions [page 2024]
Extending Apps [page 2143]

Read Before Extending a Generated App

Before you start creating an extension for your app, make sure you have read the following information.

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Take into account that:

- Implementing an extension in your app means that the coding lies withing the application's responsibility.
- For the extension of any system logic or functions that are related to existing controls or components provided by SAP Fiori elements, always use the extensionAPI. It is the official interface between the actual extension and the functions provided by SAP Fiori elements. SAP guarantees implemented system

behavior, functions, and compatibility only if the official interface is used correctly. This is the prerequisite for receiving the necessary support and quality assurance.

- If you interact only with controls that were generated within your extension, you don't need the Extension.API.
- Create extensions based only on the use cases described in this documentation.

Using the Standard SAPUI5 API

In extension coding, you can use the standard SAPUI5 programming API. However, you should do so with care.

After you have defined a view extension, you can access and modify the properties of all UI elements defined within these extensions (such as changing the visibility). You can access the elements you have created by their ID. However, you must not access any UI elements that are not defined within your view extensions.

> ⚠ Caution
>
> If you do not adhere to this guideline, your app may not work with future SAP UI5 versions because SAP Fiori elements might exchange controls for new ones that have a different API.

Make sure you also do not use the following:

- Services provided by the namespace `sap.ui.generic.app`, since these services are intended for use only by freestyle-apps or within the generic list report and object page and analytical list page template implementation.
- Services provided directly by the namespace `sap.ui.generic.template`. Unwanted side effects may occur if two layers (template coding and extension coding) access these services at the same time.

Notes on Models

Several models (instances of `sap.ui.model.Model`) are attached to the list report and object page, and analytical list page template artifacts.

- **OData Model**
  The most prominent is the default model. This is the OData model specified in the `manifest.json` file.
  You can use this model for data-binding in your own view extensions.
  Access the model (through the standard SAPUI5 API methods) with care, since side effects may interfere with the template coding that also uses this model.
  This applies in particular to function imports. Therefore, use method `invokeActions` of the extension API to call function imports.
- **UI Model**
  Each view has its own model attached that has the name `ui`. This model can be used in view and controller extensions for read purposes.
  This model contains the following properties, all of which have Boolean values:
  - `enabled`: Indicates whether active UI elements (such as buttons) should be enabled.
  - `editable`: Indicates whether input fields or similar UI elements should be in an editable state.
  - `createMode`: Indicates whether the UI displays an entity that is about to be created (no active version exists yet).

Note that it is also possible to register changes to these properties. However, the logic that determines at which point in time these properties are set and reset can still be changed.

> ⚠ Caution
>
> It is strictly forbidden to perform any change operations on the properties of the UI model.

- **SAP Fiori Elements Private Model**
  Additional model that is attached to each view that contains properties used for internal purposes within the templates.

  > ⚠ Caution
  >
  > It is strictly forbidden to access this model in any way. Do not access any model other than the default model and the `ui` model unless you have attached it to the `ManagedObject` yourself.

- **Application-Specific Models**
  You may want to define your own JSON model and attach it to UI elements. You can do this easily if the model is attached to a UI element that exists only within the scope of an extension. However, use models that are attached to a higher level (for example, to the whole view) only if absolutely necessary. In this case, you should use a name containing your own namespace to clearly separate this model from models defined by other parts of the framework.

Using the ExtensionAPI

This API consists of several elements that are described below. It can be used for the analytical list page, list report, and object page.

API Methods

When coding the implementation of an extension hook or an event handler used in a view extension, you can use the public methods of `sap.ui.core.mvc.Controller`. For information about using standard SAPUI5 programming APIs, see the relevant section above.

You can also access a service provided by the template framework. From the controller, you can access these services through `<YourController>.extensionAPI`.

This gets you an object that is specific to the template you are currently enhancing, as shown in the examples below:

Template	Instance
List Report	`sap.suite.ui.generic.template.ListReport.extensionAPI.ExtensionAPI`
Object Page	`sap.suite.ui.generic.template.ObjectPage.extensionAPI.ExtensionAPI`

Template	Instance
Analytical List Page	`sap.suite.ui.generic.template.AnalyticalListPage.extensionAPI.ExtensionAPI`

Note

Do not rely on the names of these classes in your coding, as they can still be changed. However, the set of methods provided by these objects will only be extended in a compatible way.

For more information, see ExtensionAPI, ExtensionAPI for list report extensions and ExtensionAPI for object page extensions.

Any other methods or properties of the controller (in particular any components whose names start with `'_'`) should be considered private and therefore not be used.

Caution

Do not create any instances of classes in the namespace `sap.suite.ui.generic.template`. This namespace is not intended for public use.

Sample Code

```
/*
 * Assumed use case: When the price is changed to a critical value (more than
1000), an email should be generated and sent.
 * This should not happen for changes to the draft but only after activation
has been successfully processed in the
 * back-end system.
 */

 (function() {
    "use strict";

    function onAfterActivate(oEvent) {
        /*
        * AfterActivate event is raised at the end of front-end processing
for activation. The object handed into the
        * handler contains a promise that is resolved after a successful
response from the back-end system.
        */
        oEvent.activationPromise.then(function(oResponse) {
            if (oResponse.data.Price > 1000) {
            sap.m.URLHelper.triggerEmail(null, "critical price change",
"changed price of " + oResponse.data.Product_Text
            + " to " + oResponse.data.Price + " " + oResponse.data.Currency);
            }
        });
    }
```

Using the SecuredExecution Method

The API for developers of extensions for SAP Fiori elements provides the `securedExecution` method that can be used for various purposes.

Use the `securedExecution` method whenever one of the following operations should be performed by extension coding:

- An asynchronous operation
- An operation that needs to be synchronized with other operations that are potentially triggered by the user
- An operation that could result in losing the data entered by the user. Note that this is only relevant in non-draft scenarios.
- Displaying custom messages to the user, as described in Adding Custom Messages [page 1590]
- Changing the title of the message popup after a quick action has been performed by the system. For examples, see Adding Custom Messages [page 1590]. For more information, see the API Reference for securedExecution.

The function that is to be executed must be encapsulated in a `fnFunction` function. In most use cases, the operation performed by this method is executed asynchronously. Therefore, `fnFunction` is to return a `Promise` that indicates that the operation has finished.

> **i Note**
>
> `fnFunction` must not perform any user interactions.

Depending on the state of the user interaction, `fnFunction` may or may not be executed. For example, `fnFunction` is not executed if a user has entered data before the operation starts. The user cancels the operation in the dataloss confirmation dialog and would lose the entered data. Note that the `securedExecution` method also returns a `Promise`. This `Promise` is rejected if `fnFunction` is not executed for any reason. If `fnFunction` is executed, the `Promise` returned by `securedExecution` behaves like the `Promise` returned by `fnFunction`.

As a consequence, if you need data to be returned by `fnFunction`, you should pass this data to the `Promise` returned by `fnFunction`.

> **i Note**
>
> In most cases, it is not necessary to return data from within `fnFunction`, since you can also perform the evaluation in `fnFunction`. However, if the code that handles this data is used for a user interaction, this is not possible due to the restriction explained above.

For more information, see the API Reference for securedExecution.

Adding Custom Messages

You can use the `securedExecution` method from the API for developers of template extensions to add and display custom messages.

This concept for message handling is based on the SAPUI5 MessageManager. For more information, see the API Reference for MessageManager.

When a busy session starts, all transient messages that are still in the MessageManager are removed automatically since the system assumes that they belong to previous user interaction.

When the busy session ends, all transient messages that have been collected by the MessageManager are automatically displayed to the user in a well-defined way, based on the severity of the message. This applies to (transient) messages that have been pushed into the MessageManager automatically (for example, because they come from the OData model). This also applies to transient messages that are pushed to the MessageManager explicitly by way of application coding. See the sample coding below for more details:

Sample Code

```
onCustomButtonPressed: function(oEvent){
  var oSource = oEvent.getSource();
  var oModel = oSource.getModel();
  var sBindingPath = ...; // binding path to retrieve some information
  var fnFunction = function(){
    return new Promise(function(fnResolve, fnReject){
      oModel.read(sBindingPath, {
        success: function(oResponse){
          var oMessage = new sap.ui.core.message.Message({
            message: "We have received the following response: " + oResponse,
            persistent: true, // make message transient
            type: sap.ui.core.MessageType.Success
          });
          var oMessageManager = sap.ui.getCore().getMessageManager();
          oMessageManager.addMessages(oMessage);
          fnResolve();
        },
        error: fnReject
    });
  };
  var mParameters = {
        "sActionLabel":  oEvent.getSource().getText() // or "Your custom text"
   };
  this.extensionAPI.securedExecution(fnFunction, mParameters);
}
```

You can use this option for sending messages without performing an asynchronous operation. If you do, the busy session is stopped immediately after the start. The only visible consequence of the busy session is the display of all transient messages that have been pushed to the MessageManager within this session.

Note

The sample code above shows that a message is marked as transient, by setting the attribute `persistent` to `true`. This attribute describes the lifetime of this message from the perspective of the SAP UI5 MessageManager. The lifecycle of transient messages is not relevant to the MessageManager. This means that they are persisted until another agent deletes them from the MessageManager. The SAP Fiori elements framework triggers the deletions, which effectively limits the lifetime of these messages.

The lifecycle of messages with the attribute `persistent` : `false` is controlled by the MessageManager. This kind of message is automatically removed when the entity this message has been bound to is reloaded from the backend.

The `sActionLabel` parameter is used to show a custom title for the message popup which is displayed if multiple messages come from the backend. The default title is *Messages*. For a single transient info message, `sActionLabel` has no effect, since a message toast is shown.

For more information on the `securedExecution` method, see Using the SecuredExecution Method [page 1590] and the API Reference for securedExecution.

For general information on messages, see Using Messages [page 1610].

Adapting Transient Messages that Come from the Backend

You can use an extension point to adapt transient messages that come from the backend system for the list report and object page as well as for the analytical list page.

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

The extension point (`adaptTransientMessageExtension`) is hit always before the transient message is displayed. If your app uses the flexible column layout, the extension point of the list report, the extension point of the object page, and the analytical list page extension point are hit. Note that you have to ensure that the message model has been adapted for list report, object page, and analytical list page.

Perform these steps:

1. Register your extension in the manifest.json, as follows:

```
"extends": {
   "extensions": {
      ...
      "sap.ui.controllerExtensions": {
         ...
         "sap.suite.ui.generic.template.ListReport.view.ListReport": {
            ...
            "controllerName": "STTA_MP.ext.controller.ListReportExtension",
            ...
         }
      }
      ...
```

2. Implement your controller extension.
   You have to implement the `adaptTransientMessageExtension` function within the list report controller extension, the object page controller extension, or the analytical list page extension, respectively.

```
adaptTransientMessageExtension:function() {

if(sap.ui.getCore().getMessageManager().getMessageModel().oData.length) {
                var msgText = "This message has beed added through List
Extension" ;
                var consolidatedMessage = new sap.ui.core.message.Message({
                    message: msgText,
                    type: sap.ui.core.MessageType.Information,
```

```
                    target: '',
                    persistent: true
                });

sap.ui.getCore().getMessageManager().addMessages(consolidatedMessage);
        }
}
```

> **Note**
>
> The extension point is only available for transient messages. Do not make any changes to state messages from the message model (`sap.ui.getCore().getMessageManager().getMessageModel()`).
>
> Do not alter the target of any message from the message model as this may cause the message model services to stop working.

Extending the Bookmark Function to Save Static Tiles to the SAP Fiori Launchpad

You can extend the standard bookmark function by adding an extension point to the list report or analytical list page controller extension. Static tiles are then saved to the SAP Fiori launchpad instead of dynamic tiles.

Users can use the bookmark function via the *Share* button to make list reports, object pages, or analytical list pages directly accessible from the SAP Fiori launchpad while preserving all filter values they have set before. The bookmark is added as a tile to the SAP Fiori launchpad.

For the list report and the analytical list page, per default, a dynamic tile is created. For the object page, a static tile is created. Dynamic tiles fetch data from a web service which may result in the following issues:

- They can access a different service URL and thus retrieve a wrong value
- They might fetch data that does not correctly represent the tile's purpose
- They might be inconsistent if the original tile is static

If you want to enforce the creation of a static tile, add the extension point `onSaveAsTileExtension` to the list report or analytical list page controller extension.

> **Note**
>
> This extension point can only be used for tiles that are added to the launchpad using the bookmark function.

As a prerequisite, you have already added the controller extension to the manifest.json of your app.

Overwrite the value in the `serviceUrl` field of the relevant `oShareInfo` object that can either be an empty string to enforce usage of a static tile, or any other service URL string, as follows:

> **Sample Code**

```
sap.ui.controller("ListReportExtension", {
   onSaveAsTileExtension: function(oShareInfo) {
      oShareInfo.serviceUrl = ""; // Force static tile
   }
});
```

Modifying Startup Parameters Using an Extension

You can modify startup parameters using an extension method.

You can use the extension method `modifyStartupExtension` to:

- Modify selection variants: You can modify the filter context while navigating from the SAP Fiori launchpad or from another application to the list report, the overview page, or the analytical list page.
  - List report
    A source app may provide parameters which need to be modified so that they can be applied to the SmartFilterBar in the target app. For example, the source app provides the parameters `FiscalYear` and `FiscalPeriod`, but the target app only understands the `FiscalYearPeriod` parameter. This means that the two source app parameters need to be combined into one parameter, `FiscalYearPeriod`, in the target app before the paramaters can be applied to the SmartFilterBar.
    In some cases, parameters need to be added, deleted, or renamed.
  - Analytical list page and overview page
    The filter context is passed to the application using the standard `SelectionVariant` annotation format.
    In addition, the filter context may contain:
    - Values from a default variant (such as DisplayCurrency)
    - SAP Fiori launchpad user default values
    - CDS default values that come from `Common.FilterDefaultValue`

    > **i Note**
    >
    > The analytical list page ensures that the `SelectionVariant` passed to the application via the extension is filled with the filter context that would otherwise be set to the filter bar. This filter context can have different values based on the scenario:
    > - External navigation to the analytical list page: The `SelectionVariant` will have the navigation context passed by the source application. It could have the `DisplayCurrency` value set in the SAP Fiori launchpad user default settings.
    > - Navigation to the analytical list page via SAP Fiori launchpad tile: If a default variant is maintained, the `SelectionVariant` has values from the default variant. If not, it has the values from the SAP Fiori launchpad user default values. If these values are also missing, the `SelectionVariant` has the CDS defaults that come from the `Common.FilterDefaultValue`.
- Dynamically choosing a particular tab when starting a list report with multiple views and multiple tables
  This can be relevant, for example, when launching an app from the SAP Fiori launchpad or during external navigation to a list report with multiple views and multible tables. For example, when navigating from an overview page, depending on the card clicked, a particular tab should be selected in the list report.

The object `oStartupObject` passed in this method has the following properties:

- `selectionVariant`: Contains the selection variant object that is passed from source app. You can modify this object in the target app.
- `urlParameters`: Is used to decide which tab is to be loaded dynamically. The data in `urlParameters` is used only as a deciding factor for dynamically selecting a tab (relevant for list report only).
  For example, if `urlParameters` contains the sales order status "paid", the system chooses the tab that contains the sales order status "paid" in the multiple views application.

- `selectedQuickVariantSelectionKey`: Optional string that is the key provided while creating the tabs in the manifest. By setting this value, the default tab is set (relevant for list report only).

> **i Note**
>
> The call to the extension point is performed only if there's initial navigation to the analytical list page. It is not triggered at other times (for example, when the user changes variants or when the user makes changes to selections in the filter bar, or when navigating back/ refreshing an analytical list page app that has an iappState.

Code Samples

List Report

To pass the filter context during navigation, overwrite the `modifyStartupExtension` extension method in the list report.

> **Sample Code**

```
modifyStartupExtension: function(oStartupObject) {
        oSelectionVariant = oStartupObject.selectionVariant;
        if (oSelectionVariant) {
        oSelectionVariant.removeSelectOption("TaxAmount");
        oSelectionVariant.addSelectOption("Product", "I", "EQ",
"EPM-2365436");
        }
        oStartupObject.selectedQuickVariantSelectionKey = "_tab2";
    }
```

Analytical List Page

To pass the filter context during navigation, define the `oStartupObject` object in your application extension.controller.js file as shown here:

> **Sample Code**

```
modifyStartupExtension: function (oStartupObject) {
    var oSelectionVariant = oStartupObject.selectionVariant;
    if (oSelectionVariant.getSelectOption("CustomerCountry")
&oSelectionVariant.getSelectOption("CustomerCountry")["0"].Low === "AR") {
        oSelectionVariant.addSelectOption("WBSElement", "I", "EQ", "BLUE
PRINT VALIDATION");
        oSelectionVariant.addSelectOption("CompanyCode", "I", "EQ", "EASI");
    }
}
```

Overview Page

To pass the filter context during navigation, define the `oCustomSelectionVariant` object in your application extension.controller.js file as shown here:

> **Sample Code**

```
modifyStartupExtension: function (oCustomSelectionVariant) {
```

```
    oCustomSelectionVariant.addSelectOption("SupplierName", "I", "EQ",
"Talpa");
}
```

Extending Delivered Apps Using Adaptation Extensions

You can extend delivered apps based on SAP Fiori elements by using the SAPUI5 Visual Editor.

Context

You can implement extensibility functions as part of a UI adaptation project by using the SAPUI5 Visual Editor. The adaptation project references the applications delivered by SAP as base or parent applications.

> **i** Note
>
> This procedure is relevant only for list reports, object pages, overview pages, and analytical list pages.
>
> The flexible column layout is not supported by the SAPUI5 Visual Editor.

You can use adaptation extensions for these extensibility points:

- List report and analytical list page
  - Add additional fields to the smart filter bar
  - Add additional columns to tables
  - Add additional table toolbar buttons and extension controller logic
  - Override extensibility functions
    - `onInitSmartFilterBar`
    - `provideExtensionAppStateData`
    - `restoreExtensionAppStateData`
    - `ensureFieldsForSelect`
    - `addFilters`

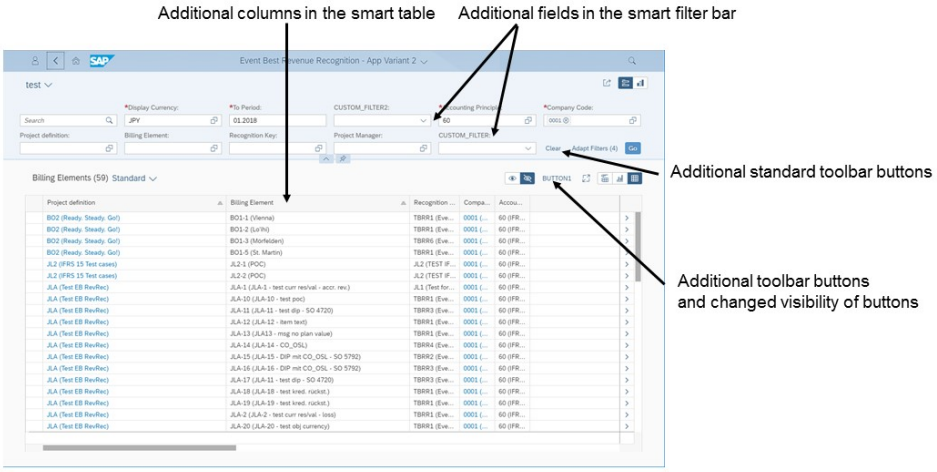


Figure 253: Adaptation Extension Options in the List Report

- Object page
  - Global actions
  - Additional sections
  - Add additional fields to the header facet
  - Add additional fields and field groups to forms
  - Override extensibility functions
    - `provideExtensionStateData`
    - `restoreExtensionStateData`
    - `ensureFieldsForSelect`
    - `addFilters`

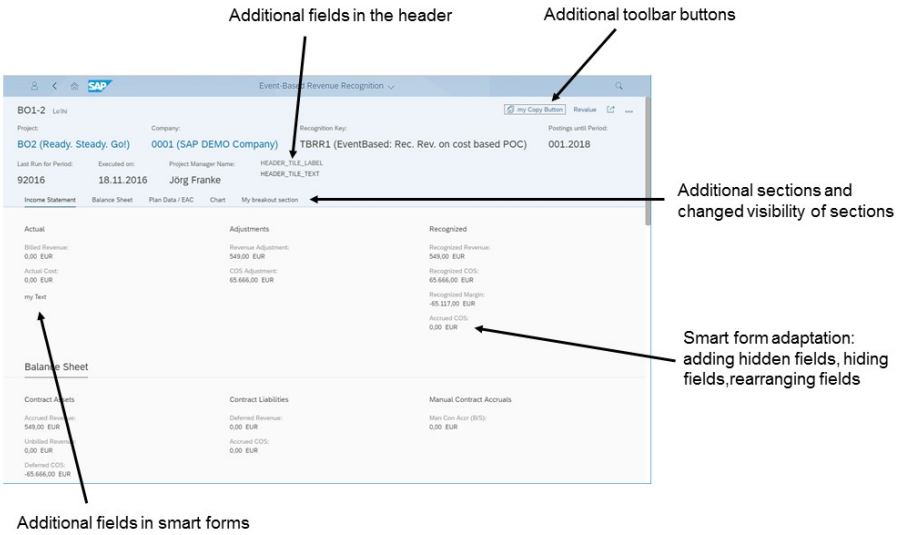


Figure 254: Adaptation Extension Options on the Object Page

- Overview page
  - Add additional fields to the smart filter bar
  - Add global actions to filter bar
  - Add additional extensions controller logic
  - Add cards
  - Clone cards
  - Edit cards
  - Override extensibility functions
    - `provideExtensionAppStateData`
    - `restoreExtensionAppStateData`
    - `addFilters`
    - `provideStartupExtension`
    - `provideExtensionNavigation`
    - `provideCustomActionPress`
    - `provideCustomParameter`

In SAP Web IDE, here's what you do:

Procedure

1. Choose *File* > *New* > *Adaptation Project*.
2. Enter the project name and the application title and choose *Next*.
3. Select the system and the base application. Make sure you deselect *Enable Safe Mode* and choose *Finish*.

   The system generates the adaptation project in your workspace.

4. Right-click your adaptation project and choose *SAPUI5 Visual Editor*.

   The system starts the editor. You can make your adaptations on the *Edit* tab page. For an example, see Adaptation Extension Example: Adding a Button to the Table Toolbar in the List Report [page 1856].

   For information about adapting the UI in the SAP Web IDE, choose *Help* > *Documentation* > *Developing* > *Developing Web Applications* > *SAPUI5 Visual Editor*.

Extending Apps Using a Canvas Page

You can embed content into a canvas page within an app.

A canvas page is an empty custom page in a SAP Fiori elements-based app. The content needs to be provided by an embedded implementing component.

The corresponding section in the manifest looks like this:

Sample Code

```
...
 "sap.ui.generic.app": {
    ...
        "pages": {
            "thePageName": {
                "component": {
                    "name": "sap.suite.ui.generic.template.Canvas",
                    "settings": {
                    }
                },
                "implementingComponent": {
                    "componentName": "theImplementingComponentQualifiedName",
                        "settings": {
                        ...
                    }
                }
            }
        },
        ...
}
...
```

You can use the `componentUsage` property instead of `componentName` to faciliate a component usage that has been defined in the corresponding section of the manifest.

There are two settings sections:

- The settings section placed in the `component` section contains properties which are specific to the canvas component.
- The settings section placed in the `implementingComponent` section can be used to define a data binding between public properties of the embedding component and context information. This corresponds to embedding a reuse component on an object page. See Including Reuse Components on an Object Page [page 1721].

For example, the following snippet allows the implementing component to use the addFooterBarToPage method to add the standard footer bar to the canvas page:

Sample Code

```
...
"sap.ui.generic.app": {
    ...
        "pages": {
            "thePageName": {
                "component": {
                    "name": "sap.suite.ui.generic.template.Canvas",
                    "settings": {
                        "requiredControls": {
                            "footerBar": true
                        }
                    }
                },
                "implementingComponent": {
                    "componentName": "theImplementingComponentQualifiedName",
                            "settings": {
                        ...
                    }
                }
            }
        },
        ...
}
...
```

If the canvas page is to include flexible column action buttons (via getFlexibleColumnLayoutActionButtons), replace the `footerBar` setting by `flexibleColumnLayoutActions` in the snippet above. You can also use both controls.

Navigating to a Canvas Page

To implement navigation to a canvas page, you need to add an additional section to the manifest:

Sample Code

```
...
"sap.ui.generic.app": {
    ...
        "pages": {
            "thePageName": {
                "component": {
                    "name": "sap.suite.ui.generic.template.Canvas",
                    "settings": {
                        "requiredControls": {
                            "footerBar": true
                        }
                    }
                },
                "implementingComponent": {
                    "componentName": "theImplementingComponentQualifiedName",
                            "settings": {
                        ...
                    }
                },
```

```
                "routingSpec": {
                    "noOData": true,
                    "noKey": true,
                    "routeName": "theRouteName"
                }
            }
        },
        ...
}
...
```

`theRouteName` is used to build a route name within the app. Therfore, the same name must not be used twice within an app. Additionally, the name of this route must not be identical with the name of any entity set or any navigation property within the OData service on which the app is based. Below, finde an exception from this rule.

The `routingSpec` as defined above enables navigation to the canvas page. You must use the extension API to trigger navigation, as shown in the following code snippet. Place this piece of code in an appropriate event handler (for example, in the press-handler of a button) in the controller extension of the source page of the navigation.

Sample Code

```
...
var oApi = this.extensionAPI;
var oNavigationController = oApi.getNavigationController();
oNavigationController.navigateInternal("", {     routeName:
"theRouteName"   });
...
```

Note

The instance of the extension API used for this purpose must belong to the parent page of the canvas page the navigation leads to.

For example, the corresponding structure in the manifest might look like this:

Sample Code

```
...
"sap.ui.generic.app": {
...
    "pages": {
        "theParentPageName": {
            "entitySet": "theMainEntitySet",
            "component": {
                "name": "sap.suite.ui.generic.template.ObjectPage",
                "settings": {
                    ...
                }
            }
            "pages": {
                "thePageName": {
                    "component": {
                        "name":
"sap.suite.ui.generic.template.Canvas",
                        "settings": {
                            ...
                        }
                    },
```

```
                    "implementingComponent": {
                        "componentName":
"theImplementingComponentQualifiedName",
                                "settings": {
                                    ...
                        }
                    },
                    "routingSpec": {
                        "noOData": true,
                        "noKey": true,
                        "routeName": "theRouteName"
                    }
                },
                ...
            }
        },
        ...
    }
...
}
```

In this case, only the the extension API that belongs to the `theParentPageName` page can be used to navigate to the `thePageName` canvas page. Note that the context of the parent page is passed to the canvas page in this scenario. In this example, this means that controls that are embedded in the `theImplementingComponentQualifiedName` component can be directly bound to properties of `theMainEntitySet`. This context is also passed to the standard lifecycle methods `stStart`and `stRefresh` of the `theImplementingComponentQualifiedName` component.

Navigating to a Canvas Page with an Additional Key

The navigation techniques described above enable you to add a canvas page that shows additional information for the same object as its parent page.

If you want the canvas page to display information that depends on what a user has chosen on the parent page, you need to set the `noKey` parameter in the `routingSpec` to false. The manifest of the app then looks like this:

Sample Code

```
...
"sap.ui.generic.app": {
...
    "pages": {
        "theParentPageName": {
            "entitySet": "theMainEntitySet",
            "component": {
                "name": "sap.suite.ui.generic.template.ObjectPage",
                "settings": {
                    ...
                }
            }
            "pages": {
                "thePageName": {
                    "component": {
                        "name":
"sap.suite.ui.generic.template.Canvas",
                        "settings": {
                            ...
                        }
                    },
                    "implementingComponent": {
                        "componentName":
"theImplementingComponentQualifiedName",
                                "settings": {
```

```
                            ...
                        }
                    },
                    "routingSpec": {
                        "noODData": true,
                        "noKey": false,
                        "routeName": "theRouteName"
                    }
                },
                ...
            }
        },
        ...
    }
...
}
```

In this case, the information about the users' choice can be passed as a parameter in the navigation.

You can place the code in a suitable event handler, as described above. Alternatively, if the canvas page is the target of the standard navigation provided in one of the tables displayed on the source page, the `onListNavigationExtension` (list report, object page) function should be used to place this code.

In this alternative scenario, choose the value of the `routeName` manifest property carefully, based on the following decision options:

- The standard object page for the corresponding entity is still in place. This means that the standard navigation is only redirected to the canvas page on a case-by-case basis. In this case, the normal rules for choosing `theRouteName` apply.
- The standard object page for the corresponding entity set is not there. In this case, choose the name of the corresponding entity set as the `theRouteName`. This overrules the guideline that `theRouteName` should be different from all entity set names. If there is more than one canvas page (and the correct one is chosen on a case-by-case basis), only one value of the `routeName` property should be set to the name of the entity set.

In all scenarios, the code for executing the navigation should look like this:

Sample Code

```
...
var oApi = this.extensionAPI;
var oNavigationController = oApi.getNavigationController();
var sUsersChoice = ... // do whatever is necessary to determine the users
choice
oNavigationController.navigateInternal(sUsersChoice, {     routeName:
"theRouteName"   });
...
```

Note that only one string can be passed to the canvas page in this fashion. If the information to be passed to the canvas page is complex, the application needs to encode this information in one string.

The following piece of code shows how the information about the user's choice can be evaluated in the implementing component of the canvas page:

Sample Code

```
...
  stRefresh: function(oModel, oBindingContext, oExtensionAPI) {
    var oNavigationController = oExtensionAPI.getNavigationController();
```

```
    var aKeys = oNavigationController.getCurrentKeys();
    var sUserChoice = aKeys[aKeys.length - 1];
    var oComponentModel = this.getComponentModel();
    oComponentModel.setProperty("/UsersChoice", sUserChoice);
}
...
```

Related Information

Developing Reuse Components [page 1726]

Adapting the UI

You can extend and customize your SAP Fiori application using the SAPUI5 Visual Editor in the SAP Web IDE.

For information about the features that you can adapt, see:

- Adapting the UI: List Report and Object Page [page 1860]
- Creating a Binding Change [page 1864]
- Adapting the UI: Analytical List Page [page 1925]

> **i Note**
>
> Adapt the UI only for the use cases described here. Otherwise, issues regarding consistency, compatibility, or other problems may occur immediately or in future releases.

For information about adapting the UI, in the SAP Web IDE, choose *Help* > *Documentation* > *Developing* > *Developing Web Applications* > *SAPUI5 Visual Editor*.

General Concepts and Configuration

Related Information

Actions [page 1605]
Using Messages [page 1610]
Enabling the Flexible Column Layout [page 1611]
Adapting the Application Header [page 1615]
Managing Variants [page 1616]
Responsiveness Options: Example [page 1617]
Value Help as a Dropdown List [page 1617]

Using Images, Initials, and Icons [page 1618]
Keyboard Shortcuts [page 1619]
Initial Expansion Level for Tables in List Reports & Analytical List Pages [page 1620]

Actions

You can use generic actions provided by SAP Fiori elements, and you can implement application-specific actions using annotations or extension points.

Actions either trigger an interaction with the back end, calling an OData service, or they trigger navigation. Depending on where you want to place an action, and how you want to use it, specific attributes, prerequisites, and guidelines apply. These are described below and in the floorplan-specific sections. For details about navigation, see Configuring Navigation [page 1563].

> **i Note**
>
> The overview page only supports micro actions, for example, actions in the quick view cards that open when you click the right-hand side of the stack card. For more information, see Quick View Cards [page 1974] and Custom Actions [page 2025].

Overview

	Annotation-based or custom action?	Context-dependent or independent?	Floorplan	Link
Global action	Custom action	Context-independent	• List report • Object page • Overview page • Analytical list page	Adding Custom Actions Using Extension Points [page 1831] (List report and object page) Custom Actions [page 2025] (Overview page) Defining Custom Actions [page 1923] (Analytical list page)

	Annotation-based or custom action?	Context-dependent or independent?	Floorplan	Link
Actions in table toolbar	• Annotation-based • Custom action	• Context-dependent • Context-independent	• List report • Object page • Analytical list page	Actions in the List Report [page 1641] Enabling Actions in the List Report [page 1642] Enabling Action Buttons in Tables on the Object Page [page 1771] Table-Only View [page 1902] (Analytical list page) Defining Custom Actions [page 1923] (Analytical list page)
Actions on the object page	• Annotation-based • Custom action	• Context-dependent • Context-independent	Object page	Displaying Actions on the Object Page [page 1664] Enabling Actions in Object Page Header [page 1693] Enabling Action Buttons in Tables on the Object Page [page 1771]
Actions in forms in sections on the object page	• Annotation-based • Custom action	Context-independent	Object page	Adding Action Buttons to Forms in Sections [page 1710] Adding Custom Actions Using Extension Points [page 1831]
Actions in smart chart toolbar	Annotation-based	Context-independent	Object page	Defining Actions in Smart Chart Toolbar [page 1707]
Actions in chart toolbar	Using manifest settings	Context-dependent	Analytical list page	Defining Custom Actions [page 1923] Chart-Only View [page 1910]

	Annotation-based or custom action?	Context-dependent or independent?	Floorplan	Link
Line item actions	Annotation-based	Context-dependent	• List report • Object page • Analytical list page	Adding Line Item Actions in Tables [page 1743] (List report, object page) Table-Only View [page 1902] (Analytical list page)
Determining actions	• Annotation-based • Custom action	• Context-dependent • Context-independent	• List report • Object page • Analytical list page	Adding Determining Actions [page 1778] (List report, object page) Adding Custom Actions Using Extension Points [page 1831] (List report, object page) Table-Only View [page 1902] (Analytical list page)
Actions in quick view cards	Annotation-based	Context-dependent	Overview page	Quick View Cards [page 1974]

Generic and Application-Specific Actions

Generic Actions

In the list report and on the object page, the standard actions *Create* (+ button), *Delete*, *Edit*, as well as actions triggering external navigation, are provided by SAP Fiori elements. You can enable or disable these actions. For more information, see Enabling Actions in the List Report [page 1642], Enabling Action Buttons in Tables on the Object Page [page 1771], and Enabling Actions in Object Page Header [page 1693].

App-Specific Actions

You can implement your own actions using annotations or extension points. These types of actions are available:

- Annotation-based actions
  - Actions that require user confirmation, for example, those for critical actions that have severe consequences. The system opens a dialog in which the user has to confirm the action.
  - Actions that require additional user input, for example, an approval comment. The system opens a dialog with one or more entry elements in which the user enters the required data. The system can pre-fill data, if applicable.

- Actions that require none of the above. The system triggers the action.

  Example: Adding Line Item Actions in Tables [page 1743]
- Actions using manifest settings

  For the analytical list page, you can define actions in the chart toolbar. For more information, see Chart-Only View [page 1910] and Defining Custom Actions [page 1923].
- Custom actions

  If your use case requires actions that cannot be implemented using annotations, you can use extension points to add actions to your app. For more information, see Adding Custom Actions Using Extension Points [page 1831], .

Actions by Place on the UI

Global Actions

Global actions can be implemented by using an extension point. They refer to the whole page, for example, *Display Log*. They are placed at the top of the page.

For more information, see:

- Adding Custom Actions Using Extension Points [page 1831] (List report and object page)
- Custom Actions [page 2025] (Overview page)
- Defining Custom Actions [page 1923] (Analytical list page)

Line Item Actions

Line item actions in tables are annotation-based. They are displayed in a column (specified in the annotation) in the individual line item. When the user triggers the action, it affects only the individual line item.

For more information, see Adding Line Item Actions in Tables [page 1743] (list report and object page) and Table-Only View [page 1902] (Analytical list page).

Actions in the Table Toolbar

You can display actions in the toolbar to allow users to perform an action for one or more lines in the table. For more information, see:

- Enabling Actions in the List Report [page 1642]
- Enabling Action Buttons in Tables on the Object Page [page 1771]
- Table-Only View [page 1902] (Analytical list page)
- Defining Custom Actions [page 1923] (Analytical list page)

Determining Actions

Determining actions are placed in the footer of the app. They are used to trigger actions directly using the context of the table in the list report, or the context of the page in the object page. See also Adding Determining Actions [page 1778] (list report, object page) and Table-Only View [page 1902] (analytical list page).

Floorplan-Specific Actions

Depending on the floorplan, you have various ways to define, enable, and display actions. For more information, see Actions in the List Report [page 1641] and Displaying Actions on the Object Page [page 1664].

Context-Independent and Context-Dependent Actions

Both actions calling OData FunctionImports (`UI.DataFieldForAction`) and actions for external navigation (`UI.DataFieldForIntentBasedNavigation`) can be either context-independent or context-dependent. For context-dependent actions, users have to select a line item in a table. Only then are the buttons that visualize these actions enabled. However, they are always visible. For context-independent actions, users do not have to select a line item in a table. Buttons visualizing context-independent actions are always enabled.

Actions calling OData FunctionImports

Actions calling OData FunctionImports can be added via a `UI.DataFieldForAction` annotation.

Sample Code

```
<Record Type="UI.DataFieldForAction">
  <PropertyValue Property="Label" String="My Action"/>
  <PropertyValue Property="Action"
String="STTA_SALES_ORDER_WD_20_SRV.STTA_SALES_ORDER_WD_20_SRV_Entities/
C_STTA_SalesOrder_WD_20My_FunctionImport"/>
  <PropertyValue Property="InvocationGrouping"
EnumMember="UI.OperationGroupingType/Isolated"/>
</Record>
```

Context-dependent FunctionImports provide a `sap:action-for` annotation defining the entity type for the required context.

Sample Code

```
<FunctionImport Name="C_STTA_SalesOrder_WD_20Setdisabledstatus"
ReturnType="STTA_SALES_ORDER_WD_20_SRV.C_STTA_SalesOrder_WD_20Type"
EntitySet="C_STTA_SalesOrder_WD_20" m:HttpMethod="POST" sap:action-
for="STTA_SALES_ORDER_WD_20_SRV.C_STTA_SalesOrder_WD_20Type"
sap:applicable-path="Setdisabledstatus_ac">
```

Context-independent FunctionImports do not provide a `sap:action-for` annotation.

Sample Code

s

```
<FunctionImport Name="C_STTA_SalesOrder_WD_20Create_simple"
ReturnType="STTA_SALES_ORDER_WD_20_SRV.C_STTA_SalesOrder_WD_20Type"
EntitySet="C_STTA_SalesOrder_WD_20" m:HttpMethod="POST" />
```

Context-independent actions calling OData FunctionImports can be placed in the table and smart chart toolbars of the list report and the object page, as determining actions in the list report or in the object page header.

Note

- If you use context-free actions, you need to label them in a way that makes it clear to the app user that the action is context-free.
- SAP Fiori elements currently only supports context-independent and context-dependent actions without input parameters.

- You can also call function imports without input parameters using multi-selection in tables. However, this is not possible if function imports have input parameters.

Context-independent actions for external navigation

The following coding sample shows the annotations for a context -independent action for external navigation (`UI.DataFieldForIntentBasedNavigation`) (Property="`RequiresContext`" Bool="`false`"):

Sample Code

```
<Record Type="UI.DataFieldForIntentBasedNavigation" >
            <PropertyValue Property="Label" String="Navigation Tester with
RequiresContext"/>
            <PropertyValue Property="SemanticObject" String="Object"/>
            <PropertyValue Property="Action" String="Action"/>
            <PropertyValue Property="RequiresContext" Boolean="false"/>
       </Record>
```

Context-dependent actions for external navigation

The following coding sample shows the annotations for a context-dependent action for external navigation (`UI.DataFieldForIntentBasedNavigation`).

Sample Code

```
<Record Type="UI.DataFieldForIntentBasedNavigation" >
            <PropertyValue Property="Label" String="Navigation Tester with
RequiresContext"/>
            <PropertyValue Property="SemanticObject" String="Object"/>
            <PropertyValue Property="Action" String="Action"/>
            <PropertyValue Property="RequiresContext" Boolean="true"/>
       </Record>
```

Related Information

Adding Confirmation Popovers for Actions [page 1782]
Using Action Control for Context-Dependent Actions [page 1778]
Adding Action-Specific Messages to Confirmation Dialog Boxes [page 1782]
Configuring Navigation [page 1563]

Using Messages

Two types of messages are used in SAP Fiori elements (List report / object page and analytical list page only):

- Transient messages
  Messages that refer to the current function, for example, "Document can't be printed as printer is not available".

The messages are only valid, and can be displayed, for a short period of time after the user has executed an action or an operation. They are not saved to the database.

- State messages
  Messages that refer to the state of an instance, for example, "Amount must not be negative".
  These messages are displayed on the UI until the state of the related instance has been corrected, for example, by changing the amount attribute to a positive value. The messages are displayed to the user and also persisted in the back-end system.

By default, messages are handled as follows:

- General
  - After executing an action, all related transient messages are displayed in a dialog box. All new state messages are shown in a message popover, together with the existing state messages.
  - Draft operations are handled like actions, for example, activate, prepare, or validate operations.

    > **i Note**
    >
    > Avoid raising transient messages during prepare and validate operations. The user would be able to see them in a dialog box and this UI behavior is not needed in edit mode.

  - If only one transient success message is raised, this is shown in a toast message. In all other cases, the messages are shown in a dialog box.
- List report: Only transient messages are displayed, in a dialog box after an action has been executed
- Object page:
  - In display mode, only transient messages are displayed after an action has been executed.
  - In edit mode, all state messages related to the object are displayed in a popover message. The user can open the message for the related object, for example, the displayed sales order. Messages for the items are not displayed. The user has to navigate to the item. The popover also displays client validation messages.

Enabling the Flexible Column Layout

The flexible column layout allows users to see more details on a page, and to expand and collapse the screen areas, depending on their requirements.

The flexible column layout offers different layouts with up to three columns. Depending on which panel the user is focused on, it can be expanded. The user can also switch between different layouts and enable full-screen mode.

> **i Note**
>
> - For the overview page, this layout is not relevant.
> - The analytical list page only supports the `TwoColumnsBeginExpanded` layout. For more information, see also 2409984.
> - The flexible column layout can only be used in draft scenarios. In display-only, non-draft apps, you can also use the flexible column layout.

To enable the flexible column layout in an app, create an entry in manifest.json, as follows:

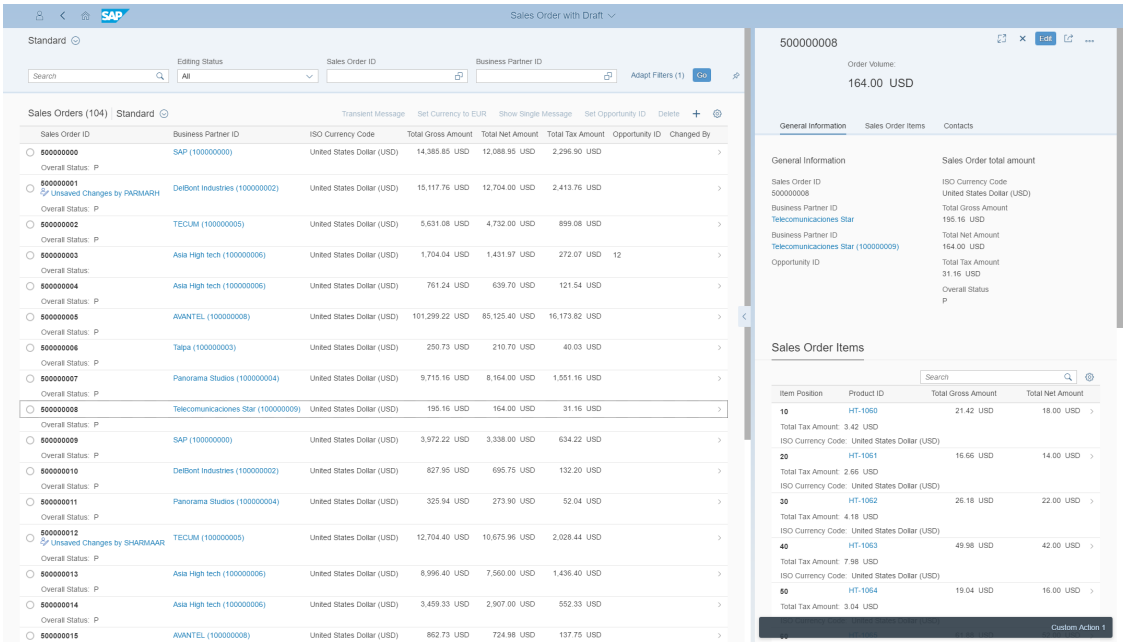
```
"sap.ui.generic.app": {
```

```
        "_version": "1.1.0",
        "settings": {
            "flexibleColumnLayout": {
                "defaultTwoColumnLayoutType": "TwoColumnsMidExpanded",
                "defaultThreeColumnLayoutType": "ThreeColumnsEndExpanded"
            }
        },
        "pages": [...
```

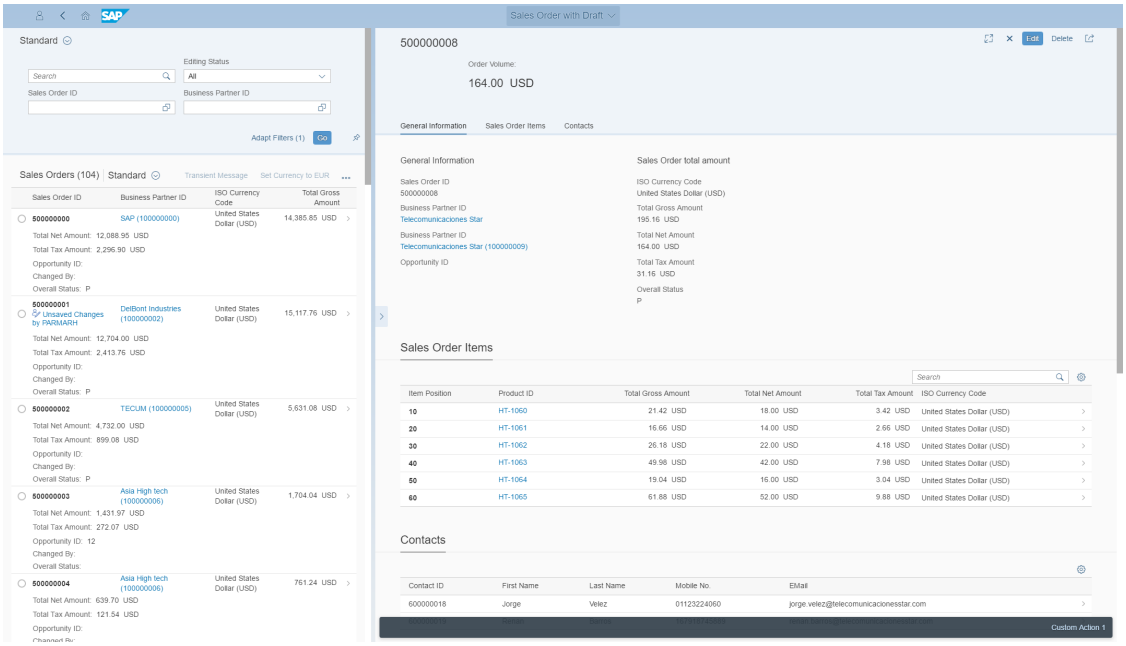
> **i Note**
>
> For optimum readability, you can set the `PopinLayout` property to `Block`, `GridLarge`, or `GridSmall`. For more information, see Adapting the UI: List Report and Object Page [page 1860].

Use the following attributes to create the column layout you want:

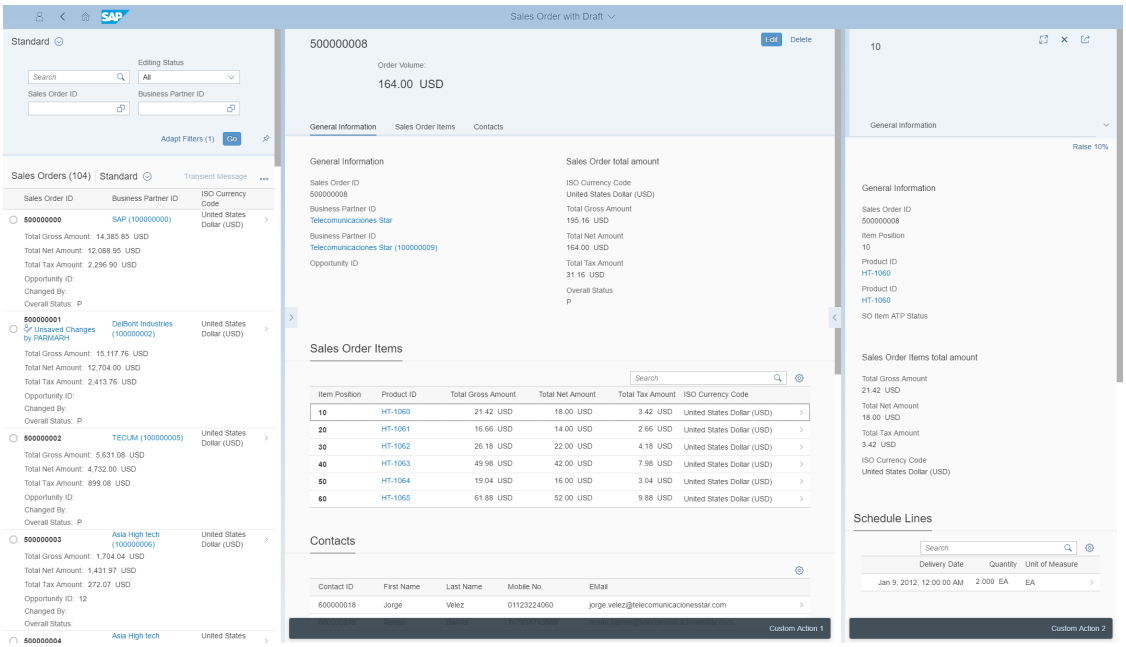
- `defaultTwoColumnLayoutType`: 2-column layout with these options:
  - `TwoColumnsBeginExpanded`



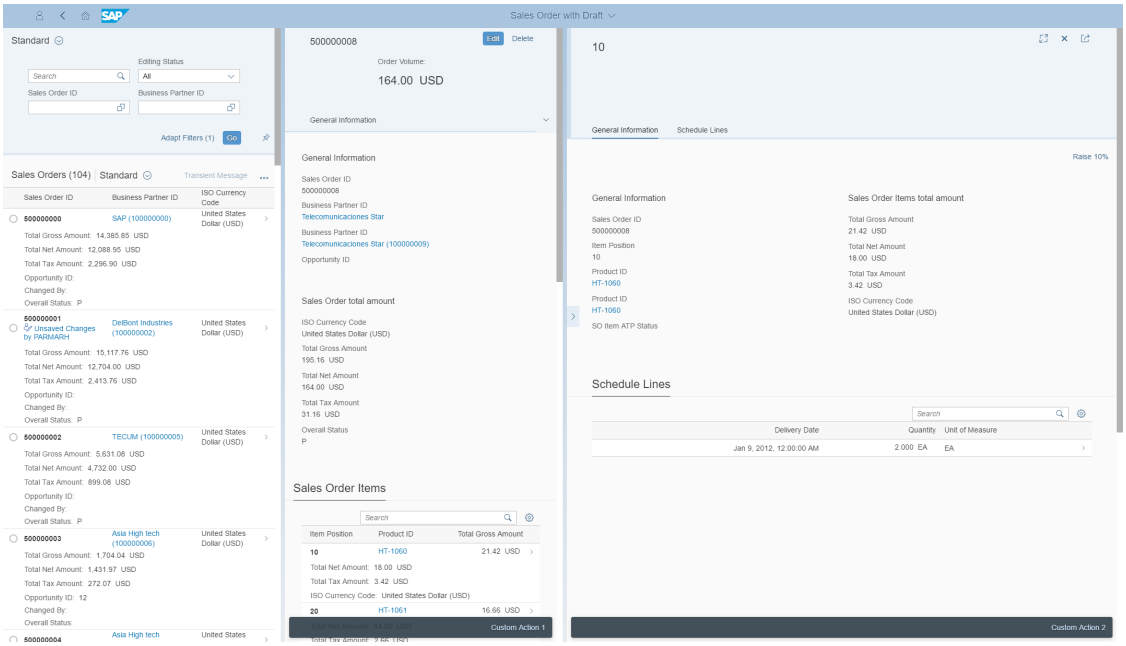
  - `TwoColumnsMidExpanded`



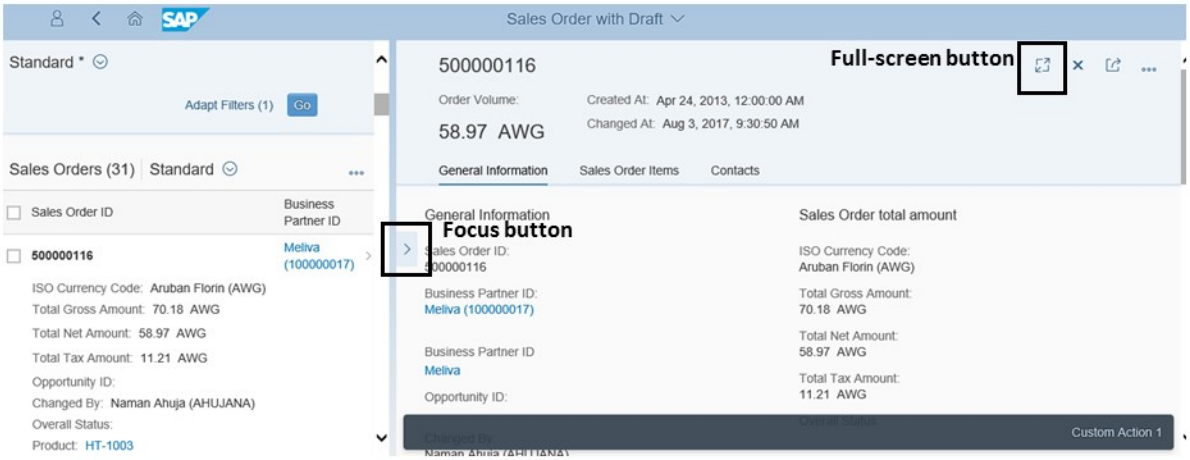
- `defaultThreeColumnLayoutType`: 3-column layout with these options:
  - `ThreeColumnsMidExpanded`



  - `ThreeColumnsEndExpanded`



Users can expand and collapse the columns using the focus buttons. They can change to full-screen mode by choosing the full-screen button.



Defining a Default Layout

For each page configured in the manifest you can define a default layout that is used when the page is opened. You can use the `defaultLayoutType` property to do so. For example, you can use the `MidColumnFullScreen` property value to open a page in full-screen mode. This overrides the layout which has been defined in the global `flexibleColumnLayout` settings for the corresponding column. Note that this is only relevant if different pages in the same column need different default layouts.

For an object page, you can define "defaultLayoutType": "OneColumn". By doing so, in the flexible column layout, this object page moves to the first column. All other object pages that are below the first one in the hierarchy move up accordingly. If they have the same setting, they also move to the first column.

Usually, this setting is made on the main object page. After navigating from the list report, the object page is then displayed in full-screen mode, that is, the main object page replaces the list report. When navigating to an item, the main object remains in the first column and the item is displayed in the second column.

Adapting the Application Header

If required, you can change the application header that is generated from the SAP Fiori launchpad.

List Report, Overview Page, and Analytical List Page

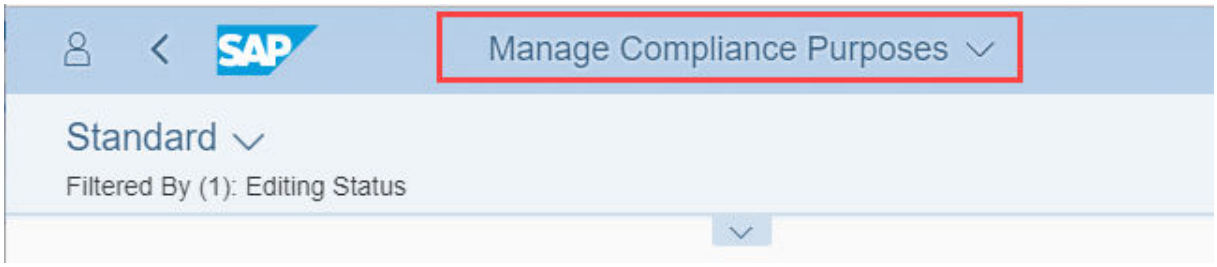


Figure 255: Application Header in List Report

To change the application header, in your project artifacts, change your app's i18n property file under `webapp/i18n/i18n.properties` ->`appTitle`.

The title and description are set in the `sap.app` section in the manifest file, and the image or icon are set in the `sap.ui` section under `icons`. Both the application title and the description should be translatable, that is, the text property name should be specified in the application manifest surrounded by double curly brackets ({{}). In the `icon` property, you can set a URL to an `sap-icon` or a URL to an image.

Sample Code

```
"sap.app": {
        ...
        "title": "{{app_title}}",
        "description": "{{app_description}}",
        ...
},
"sap.ui": {
        "icons": {
            "icon": "sap-icon://dimension",
            "favIcon": "favicon.ico"
        }
}
...
```

Object Page

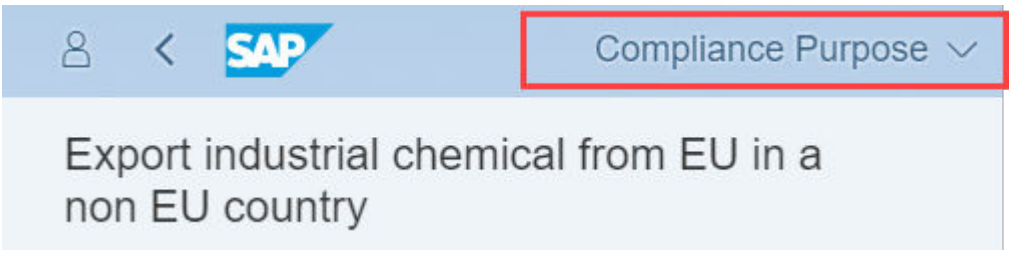


Figure 256: Application Header on Object Page

To change the application header on the object page, change the `@UI.headerInfo.typeName` annotation.

Related Links

- For information about changing the table header in a list report, see Setting the Smart Table Header [page 1737].
- For information about changing object page titles and subtitles, see Adapting the Object Page Title and Subtitle [page 1667].

Managing Variants

Lets you manage variants with different structures in the filter and content areas.

Context

The list report and the analytical list page provide variant management on page level and on control level.

Use page-level variant management to capture filter selection, filter mode, view mode, auto-hide icon state (eye-icon), chart and table configuration (measures and dimensions), sort order, and grouping. The page level variant is enabled by default.

Use control-level variant management to define separate variants for filters, charts, and table sections. Set `smartVariantManagement=false` in the app-descriptor file to use control-level variant management. The filter variant stores filter area selections and the filter mode. When you load the stored variant, both the filter area selection and filter mode appear. It is not possible to:

- Turn off variant management for selected controls (filter, chart, or table area)
- Use control-level variant management and page-level variant management together

> **i** Note
>
> The SAPUI5 standard class `sap.ui.comp.variants.VariantManagement` supports and manages variants.

Related Information

Configuring the Title Area [page 1872]
Creating Key Performance Indicator Tags [page 1873]
Choosing Filter Modes [page 1880]

Responsiveness Options: Example

When using SAP Fiori elements, you can make use of specific responsiveness options. For example, in pages that have toolbars (such as list report tables, object page tables, and smart chart toolbars), the system evaluates the `com.sap.vocabularies.UI.v1.Importance` for each field:

- Fields of high importance are rendered on a mobile phone.
- Fields of high or medium importance are shown on the tablet.
- Fields of high, medium, or low importance are shown on the desktop.

Value Help as a Dropdown List

If your value help contains a fixed number of values, a dropdown list will be rendered.

For more information on how value help annotations are set in CDS, search for *UI Annotations* in the documentation of your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

If the entity set of a value help has a fairly stable number of instances, you can render an input field with a value help and dropdown list box (`sap.m.ComboBox` and in cases of multi selection a `sap.m.MultiComboBox`) using the metadata extension `sap:semantics='fixed-values'` on the entity set level and the `sap:value-list='fixed-values'` on the property level.

In the following example the product category is implemented as a dropdown list box:

Sample Code

$metadata

```
<EntityType Name="SMART_C_ProductType" sap:label="Product" sap:content-
version="1">
    <Key>...</Key>
    ...
    <Property Name="ProductCategory" Type="Edm.String" Nullable="false"
MaxLength="40" sap:label="Category" sap:value-list="fixed-values" />
    ...
    <EntityContainer Name="SMART_PROD_MAN_Entities"
m:IsDefaultEntityContainer="true" sap:supported-formats="atom json xlsx">
    ...
    <EntitySet Name="SEPMRA_I_ProductCategory"
EntityType="SMART_PROD_MAN.SEPMRA_I_ProductCategoryType"
                           sap:creatable="false" sap:updatable="false"
sap:deletable="false" sap:searchable="true" sap:content-version="1"
sap:semantics="fixed-values" />
```

This is the rendering result:

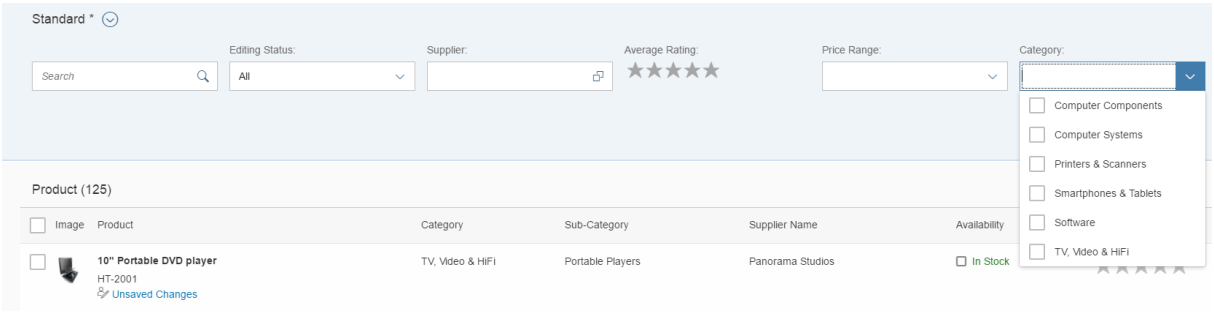


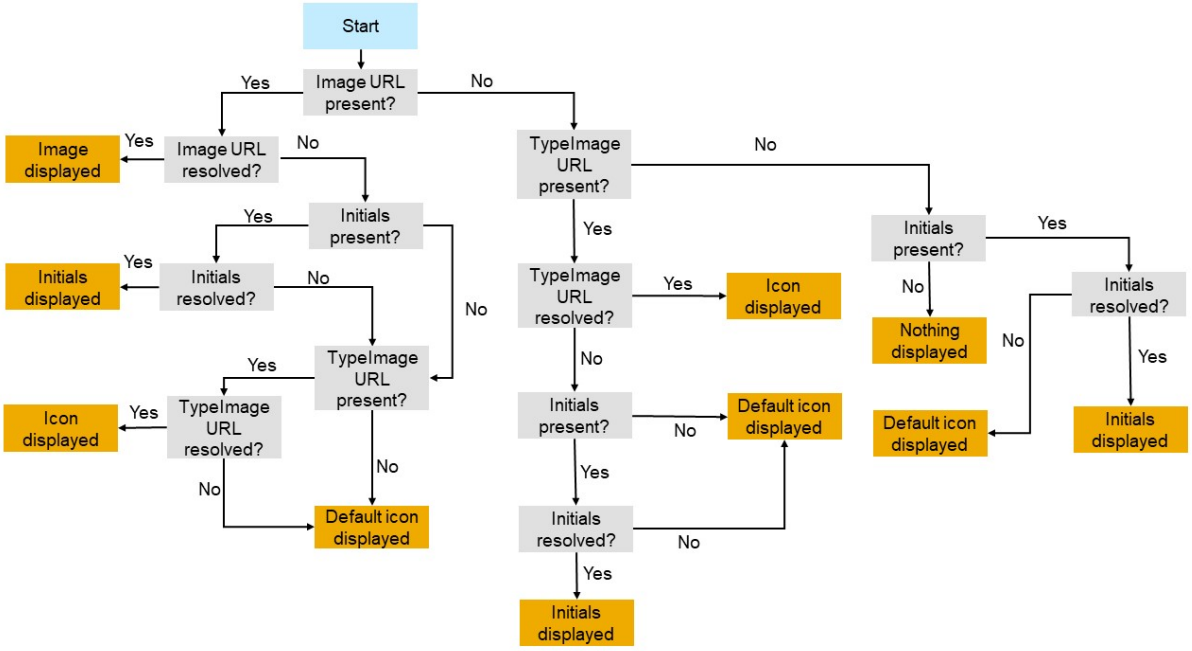
Figure 257: Product Category Values as Dropdown List Box

Using Images, Initials, and Icons

SAP Fiori elements supports the use of images, initials, and icons.

> **i Note**
>
> You can use the SAPUI5 Visual Editor to replace the default icon by any SAP icon.

The system follows this logic for choosing a display option for an object:



> **i Note**
>
> In the object page's header, the header image is an avatar control. By default, the avatar is rendered as a square. If the avatar's source, initials, or icon isn't set or found, a fallback placeholder is displayed. The type of placeholder depends on the shape of the avatar. If the avatar is circular, a person icon is shown. If the avatar is a square, a product icon is shown.

Keyboard Shortcuts

SAP Fiori elements provides keyboard shortcuts for basic operations.

The table shows the available keyboard shortcuts.

Action	Shortcut (Microsoft Windows)	Shortcut (Mac OS)	Prerequisite	Relevant Floorplans	Result
Create	`Ctrl` + `Enter`	`CMD` + `Enter`	The focus should be on the table.	List report Object page	The same as when a user chooses *Create*.
Delete table entry	`Ctrl + D`	`Cmd` + `D`	The focus should be set on the table.	List report Object page	The same as when a user chooses *Delete* in a table **Note** This requires a selection for multiple deletions or a focus position for deleting single items.
Delete page level	`Ctrl` + `Del`	`CMD` + `fn` + `delete` (use ← if there is no delete key	The focus should be set anywhere on the object page.	Object page	The same as when a user chooses *Delete* on the object page
Edit page level	`Ctrl` + `E`	`CMD` + `E`	The focus should be set anywhere on the object page.	Object page	The same as when a user chooses *Edit* on the object page
Save page level	`Ctrl` + `S`	`CMD` + `S`	The page should be in edit mode. The focus should be set anywhere on the object page.	Object page	The same as when a user chooses *Save* on the object page
Share	`Shift` + `Ctrl` + `S`	`Shift` + `CMD` + `S`	The focus should be set anywhere on the list report or object page.	Analytical list page List report Object page	If the focus is on the filter bar, the share-ActionSheet opens near the *Share* button

Action	Shortcut (Microsoft Windows)	Shortcut (Mac OS)	Prerequisite	Relevant Floorplans	Result
Go	`Enter` or `Return`	`Enter` or `Return`	The focus should be set anywhere on the SmartFilterBar.	Analytical list page List report Object page	In the SmartFilterBar, the search is triggered while the focus is within one of the filter bar's input fields.
Go	`Ctrl` + `Enter`	`CMD` + `Enter`	The focus should be on any element in the visual filter panel or on the entire visual filter.	Analytical list page (visual filter)	If the focus is on any element in the visual filter panel or on the entire visual filter, *Go* is triggered.
Select row in analytical or grid tables	`Shift` + `Space`	`Shift` + `Space`	The focus should be on a table cell.	Analytical list page List report Object page	If the focus is on a cell, the entire row is selected
Table settings	`Ctrl` + `,`	`CMD` + `,`	The focus should be set on the table.	Analytical list page List report Object page	The same as when a user chooses the *Table Settings* button.
Export as	`Shift` + `Ctrl` + `E`	`Shift` + `CMD` + `E`	The focus should be set on the table.	Analytical list page List report Object page	The same as when a user chooses *Export to Excel → Export As*.

Initial Expansion Level for Tables in List Reports & Analytical List Pages

You can set the number of expanded levels for tables in List Reports and Analytical List Pages using the `initialExpansionLevel` property of the `PresentationVariant` annotation.

Expected Behavior of Table Types

Table 77: Table Types

Table Type	Expected Behaviour
Analytical Table	The default `initialExpansionLevel` is 0. **i Note**
Responsive Table	Irrespective of the value of the `initialExpansionLevel`, the responsive table expands to one level. The groups are always expanded and you can group using table settings.
Tree Table	In List Reports, the first level is automatically expanded. In Analytical List Pages, the default `initialExpansionLevel` is 0.

The `initialExpansionLevel` should never exceed the number of grouped columns.

Defining Initial Expansion Level in PresentationVariant

For `initialExpansionLevel` to be supported, `PresentationVariant` annotations must exist for the content area in Analytical List Pages/ List Reports and multiple tabs in a List Reports.

i Note

If tabs are not defined for a List Reports, the default `PresentationVariant` (without the qualifier) is considered.

The content area in Analytical List Pages can be associated directly with the `PresentationVariant` in these cases:

- The qualifier is defined in the manifest: There is no matching `SelectionPresentationVariant` with the qualifier but there is a matching `PresentationVariant`.
- The qualifier is not defined in the manifest: There is no default `SelectionPresentationVariant` but there is a default `PresentationVariant`.

The tabs in a List Reports can be associated directly with the `PresentationVariant` when the annotation path of the tab defined in the manifest points to a `PresentationVariant` annotation. In this case, the required `PresentationVariant` has to be defined as shown in the sample code below:

Sample Code

```
<Annotation Term="UI.PresentationVariant" Qualifier="Default">
    <Record>
        <PropertyValue Property="Visualizations">
            <Collection>
                <AnnotationPath>@UI.LineItem#DefaultLineItem</AnnotationPath>
```

```
            </Collection>
        </PropertyValue>
        <PropertyValue Property="GroupBy">
            <Collection>
                <PropertyPath>ProductId</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="InitialExpansionLevel" Int="1"/>
        <PropertyValue Property="SortOrder">
            <Collection>
                <Record>
                    <PropertyValue Property="Property"
PropertyPath="ProductCategory" />
                    <PropertyValue Property="Descending" Bool="false" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Defining the Initial Expansion Level in the SelectionPresentationVariant

If the content area in the Analytical List Pages and tabs in the List Reports are associated with a `SelectionPresentationVariant` that references a `PresentationVariant`, ensure that the `PresentationVariant` is not defined inline but referred to using a path as shown in the sample code below:

Sample Code

```
<Annotation Term="UI.SelectionPresentationVariant" Qualifier="MainContent">
    <Record>
        <PropertyValue Property="ID" String=""/>
        <PropertyValue Property="Text" String=""/>
        <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#Default"/>
        <PropertyValue Property="PresentationVariant"
Path="@UI.PresentationVariant#Default"/>
    </Record>
</Annotation>
```

- Configuring Tables [page 1735]
- Table-Only View [page 1902]

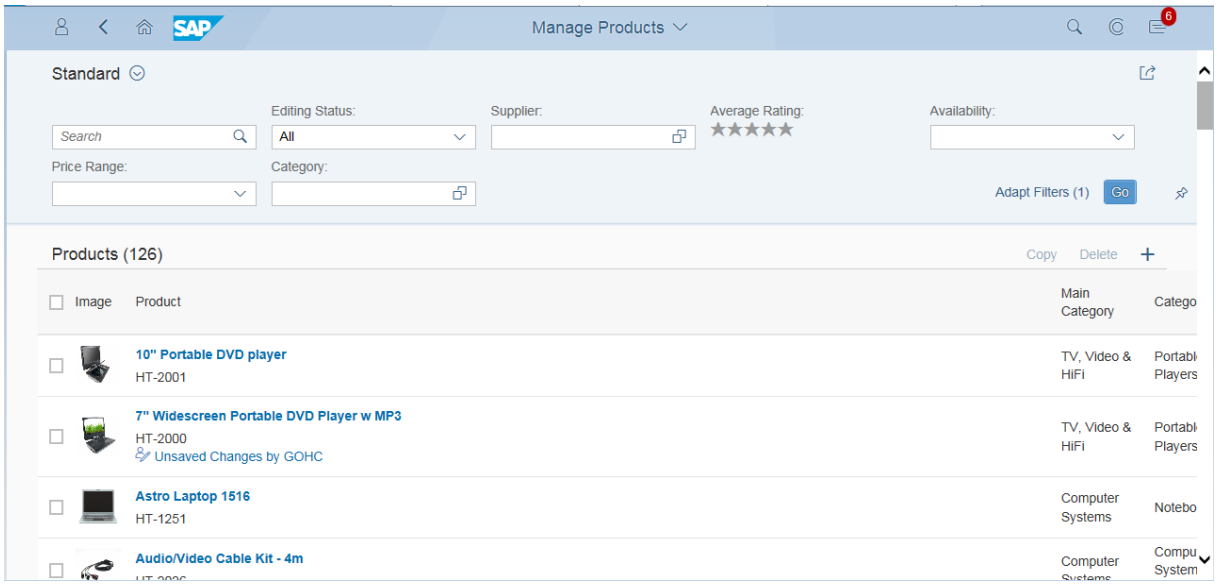
List Report and Object Page

The list report is typically used in conjunction with an object page. This is the main use case when creating SAP Fiori apps. While the list report lets users filter, view, and work with items (objects) organized in list (table) format, object pages let users work with objects, providing functionality to view, edit, and create objects.

The object page can also be used in conjunction with the analytical list page. For more information, see Analytical List Page [page 1868].

List Report Elements

The list report lets the user work with a large list of items. It combines powerful functions for filtering large lists with different ways of displaying the resulting item list.



- Adapting the Application Header [page 1615]
- Enabling Variant Management [page 1637]
- Adapting the Smart Filter Bar [page 1661]
- Actions in the List Report [page 1641]
- Smart Tables [page 1628]

When you launch the application, the system loads the data to the list report by default and the smart filter bar appears collapsed so that more rows can be viewed in table. The `enableAutoBinding` property of the SmartTable has been set to `True` in the UI Adaptation Editor.

> **i Note**
>
> Provide default values for mandatory filter fields to avoid loading of empty tables when you launch the app.

This is the default behavior. In special cases where you don't want the system to load the data when you launch the app, you can disable the settings like this:

- **End Users:**
  End users can disable autoloading by setting `Apply Automatically` to `False` in variant management.
- **Application Developers:**
  Application developers can change the `enableAutobinding` to `False` for smart tables through Visual Editor (UIA).
  For multi-view scenarios, application developers can change the manifest setting `enableAutobinding` to `False` under `quickVariantSelectionX`

Sample Code

```
"pages": {
          "ListReport|<EntitySet> ": {
                  "entitySet": <EntitySet>,
                  "component": {
                  "name": "sap.suite.ui.generic.template.ListReport",
                  "settings": {
                              "quickVariantSelectionX": {
                                      "enableAutoBinding":  false
                               }
                   }
              }
          }
```

Main Elements of the List Report

The list report view includes the following main elements:

- Application header
- Smart filter bar with variant management and a generic *Share* menu that includes the following actions:
  - *Send Email*
  - *Save as Tile*
  - *Share in SAP Jam* (if integration with SAP Jam is configured)
- Smart table
- Footer toolbar that can include optional actions

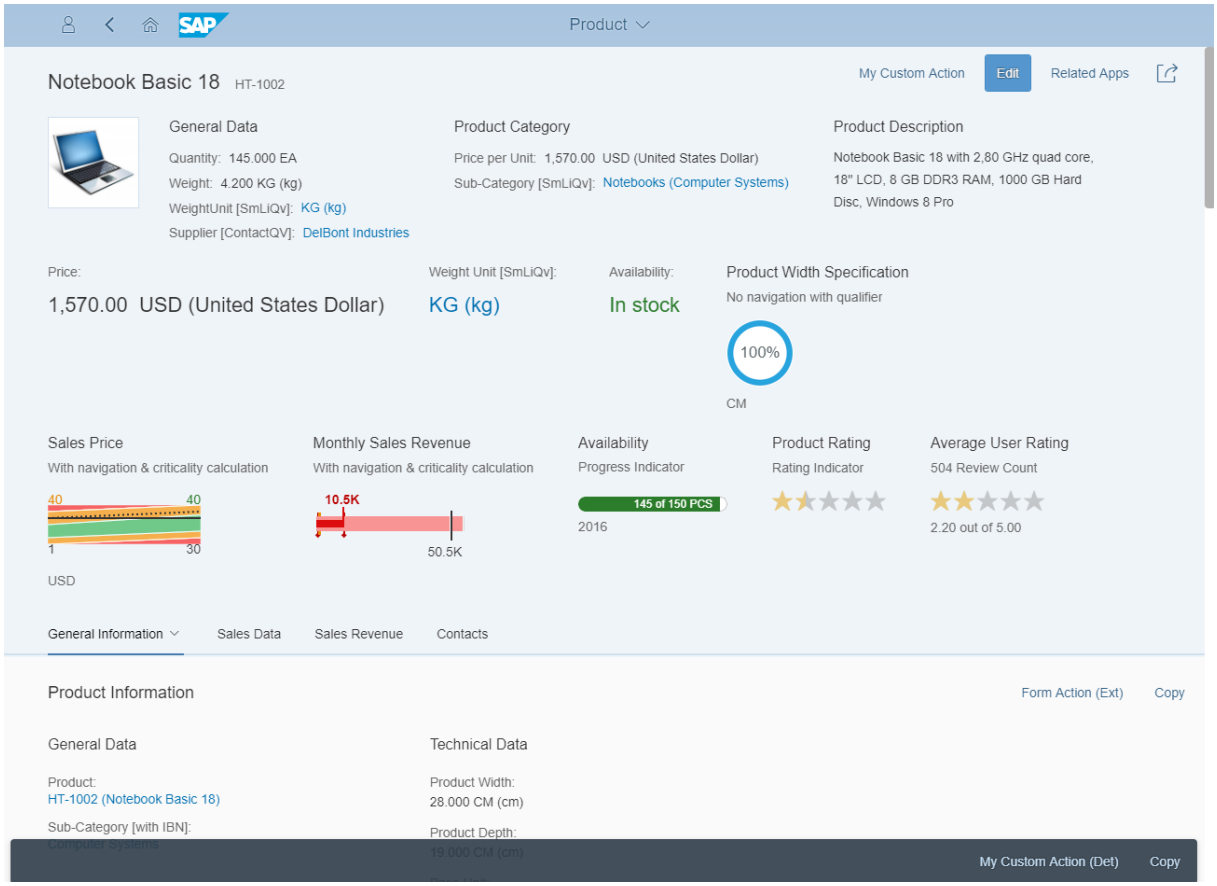
More Information

For more information about the various functions of the list report view, see:

- Social media integration (enabling the *Share in SAP Jam* option in the footer toolbar):
  About SAP Jam Integration
- Configuring List Report Features [page 1637]
- Configuring Further Common Features [page 1778]

Object Page Elements

The object page lets you display, edit, and create objects, as well as save drafts. It is suitable for both simple objects and more complex, multi-faceted objects. The object page view gives you optimal support for multiple devices.



- Enabling the Related Apps Button [page 1697]
- Adding Action Buttons to Forms in Sections [page 1710]
- Header Facets [page 1668]
- Setting up the Object Page Header [page 1665]
- Enabling Actions in Object Page Header [page 1693]
- Setting up the Object Page Header [page 1665]
- Defining and Adapting Sections [page 1698]
- Defining and Adapting Sections [page 1698]
- Object Page Elements [page 1625]

Main Elements of the Object Page

- Page title that is set to the object type, for example, product
- Object header including the following:
  - Title and subtitle
  - Editing status icon (if applicable)
  - Header toolbar, containing generic actions (in *Display* mode)
  - Optional elements, such as:
    - A description
    - An image of the object instance

      > **i Note**
      >
      > If the instance does not provide an image, then the default image of the object type is used.

    - Buttons in the header toolbar for use case-specific actions, for example, *Edit* and *Delete*
    - Header facets to showcase important information relating to the object. Header facets can contain:
      - Label-field pairs, to show, for example, price or availability. We recommend not using more than five label-field pairs.
      - Smart controls, to show, for example, a micro chart detailing sales revenue or a rating indicator to visualize the average of all user-assigned ratings
- Anchor navigation area that lets users navigate to the individual content area sections
- Content area in which data is organized into sections that can contain field groups or a table
- Paginator buttons on the detail page to browse from item to item in the list the user came from

  > **i Note**
  >
  > The paginator buttons are only visible if these conditions are fulfilled:
  > - The user comes from a list to the current page.
  > - This list contains at least two entries.
  > - The user is on a subobject page.

  On the first object page, the paginator buttons are disabled, by default. You can adapt the UI to enable them. For more information, see Adapting the UI: List Report and Object Page [page 1860].
- Footer bar in which actions and the *Show Messages* button are available (if applicable).
  The footer bar of subitem object pages can also include the *Apply* button in create and edit mode. This action concludes the current create or edit activity, saves the draft, and navigates one step up in the object hierarchy. A toast message is displayed when an operation is successful. For more information, see Draft Handling [page 1631].

The object page is made up of the following elements:

More Information

Subject	Link
Controls related to object pages	sap.uxap [page 2481]
Annotations used to set up various elements of object pages	Configuring Object Page Features [page 1664] Configuring Further Common Features [page 1778]

How-To Videos

The following videos provide step-by-step instructions for typical tasks when creating apps with SAP Fiori elements.

YouTube Video	Link to Help
Creating a List Report App with SAP Fiori Elements:	Building an App Using SAP Web IDE [page 1553]
Adding a Field to a List Report Table:	Defining Line Items [page 1762]
Adding a Default Filter to a List Report:	Adapting the Smart Filter Bar [page 1661]
Creating Multiple Views in List Report Tables:	Multiple Views on List Report Tables [page 1645]
Enabling Quick Views for Smart Link Navigation:	Enabling Quick Views for Smart Link Navigation [page 1567]
Adding a Field Group to an Object Page:	Defining and Adapting Sections [page 1698]
Adding a Data Point Header to an Object Page:	Data Points [page 1682]
Creating a Section Extension on an Object Page:	Adding a Section to an Object Page [page 1803]

General Concepts

Smart Tables

A list report contains a smart table control that displays the list report items and uses page mechanisms when loading data.

The smart table in the list report view contains the following:

- Layout management
- Toolbar with actions rendered as text icons, for example, *Personalize* and *Add Item*
- Application-specific actions rendered as text buttons, for example *Copy*, *Approve*, and *Delete*
- Indication of draft status
- Displays items locked by another user

Smart tables can render these table types:

- Responsive tables
  Responsiveness is optimized for mobile use, line items can be viewed with no scrolling or with only vertical scrolling, regardless of the display width.

  > **i Note**
  >
  > On mobile devices, a responsive table is always displayed.

- Grid table
  Desktop-centric table type which allows users to scroll in both directions. This table type can handle a large number of items and columns.
- Analytical table
  Contains data structured in rows and columns. It provides several powerful options for working with data, including advanced grouping and aggregations.
  In contrast to other tables, the analytical data binding used by the analytical table automatically displays an aggregated number in a cell.
- Tree tables
  Users can display hierarchically structured data.

  > **i Note**
  >
  > You can use the tree table only in apps for ready-only scenarios.

The application decides which table representation is most suitable based on the usage. The table toolbar is responsive. For information about available table types, see also Tables: Which One Should I Choose? [page 2286].

Navigation at Row Level

If a page contains a grid, an analytical or tree table, and has an object or subobject page, users can navigate to the object or subobject page at row level. The *Show Detail* button is not displayed in these tables.

Triggering Custom Actions for Items in a Table

By default, the smart table generated by the template is single-selection. Users select an item from the table to trigger a custom action, for example, *Validate*, which then returns the results for the selected item.

Searching for Rows in a Table on an Object Page

In responsive, grid, and analytical tables, if the table is searchable (that is, if an entity set is used for which `sap:searchable` is `true`), a search field is displayed. You can search for particular rows in the table.

Smart Multi-Input Control

Smart multi-input [page 2443] is automatically rendered as a column in responsive and grid tables if a 1:N relationship exists in the association for the given column.

To configure smart multi-input fields on an object page, see Using the Smart MultiInput Control on the Object Page [page 1720].

Vertical Alignment of Responsive Table

You can set the vertical alignment property for a responsive table via manifest property `tableColumnVerticalAlignment` under settings of `sap.ui.generic.app`. You can set the property value as `Top`, `Middle`, or `Bottom`.

Related Information

Configuring Tables [page 1735]
Setting the Table Type [page 1735]
Tables: Which One Should I Choose? [page 2286]
Enabling Multiple Selection in Tables [page 1741]

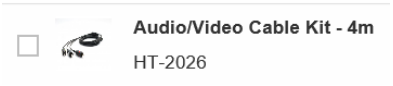
Editing Status

The editing status reflects the state of the object or entry in terms of the processing cycle. For example, it can give the user information about whether the item can be accessed or its level of completion.

Editing Status for Table Items

The list report allows users to view the editing status of the objects displayed. The editing status is calculated from the draft administrative data that is added when using the Business Object Processing Framework (BOPF).

Table 78: Draft Administrative Data: Visualization

Editing Status	Description
	Draft: my own draft
	Active version
	Active version with draft created by another user; no longer locked
	Active version with draft created by another user; locked

Access to Administrative Data

For the statuses *Unsaved Changes* and *Locked*, the name of the user who last changed the object is visible directly in the line item in the list report (see above). Note that if the user's full name is not available in the master data in the back-end system, only the technical user name displays. If the technical name is also not available, the message then indicates that the unsaved changes or the lock on the object belongs to "another user".

As well, for the statuses *Draft*, *Unsaved Changes*, and *Locked*, a link accesses a popover to allow you to view the user who last changed the object and the time of the change. Note that if the user's full name is not available in the master data in the back-end system, only the technical user name displays.

Editing Status Filter

A static filter attribute is available for all applications, which is added to the smart filter bar of the list report for all draft-enabled applications by default. This filter allows users to search for objects or entries in a specific state.

The drop-down values are as follows:

Table 79: Drop-Down Values

Drop-Down Value	Description
All	Shows all results except duplicates. This means that in the case of an edit draft, the drafts shall be shown but not the corresponding active version of the same document.
Own Draft	Shows the drafts that the current user can display or edit.
Locked by Another User	Shows the active versions that are locked by other users. The current user cannot edit these versions.
Unsaved Changes by Another User	Shows the active versions that were edited by other users, but are no longer locked. The current user can edit and overwrite these versions, and the previous draft will be overwritten.
No Changes	Shows active versions that have no corresponding draft.

> **i** Note
>
> The default search filter is *All*, even if the user clears the filter value.

Disabling the Editing Status Filter

If desired, the editing status filter can be disabled after you generate your app. For information, see Disabling the Editing Status Filter [page 1660].

Related Information

Displaying the Editing Status [page 1763]

Draft Handling

A draft is an interim version of a business entity that has not yet been explicitly saved as an active version. Drafts are saved automatically in the background whenever users add or change information within a business entity while it's in edit mode (auto-save). SAP Fiori elements support the creation of apps using draft handling.

Drafts are used as follows:

- To keep unsaved changes when an editing activity is interrupted. This lets users resume editing later.
- To prevent data loss if an app terminates unexpectedly
- As a locking mechanism to prevent multiple users from editing the same object at the same time, and to inform users about unsaved changes by another user.

When a user starts creating a new business entity or edits an existing one, a draft is created in the background. This enables field validation and dynamic field control (showing additional fields based on user interaction) and

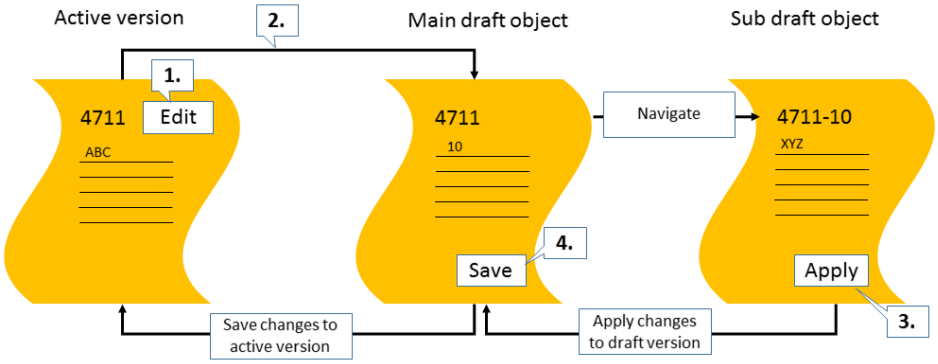
provides default values for fields based on recent data entry. A draft can be validated for consistency and completeness at any time. This returns a list of messages.

While the user is modifying a business entity, an indicator shows when a draft is saved implicitly. The user still needs to choose *Save* to incorporate the changes into an active business document.

Using the Apply Button

The footer bar of a subobject page contains an *Apply* button in create mode and edit mode. When users choose this button they can conclude their current create or edit activity, apply the changes or entries to the draft, and navigate one step up in the object hierarchy.

When a user edits an object, the system behavior of the *Apply* button is as follows:



1. On an object page, the user chooses *Edit*.
2. The system creates a draft version of the object.
3. The user makes changes to the draft version of the object and navigates to a detail page. When *Apply* is chosen, the changes are applied to the draft.
4. When the user chooses *Save*, the changes are saved to the active version of the object.

> **i** Note
>
> If there's an error, for example, network issues, after having chosen *Apply*, a message is displayed. If the user chooses *Cancel*, nothing happens. If the user chooses *Discard Changes*, the system behaves according to the message, for example, the changes are discarded.

Handling Inconsistent Input

Users might enter data that is so inconsistent that the system cannot store it in the draft. For example, characters are entered in a number field, or more characters than the field length allows are entered. If this is the case, the contents of the UI differ from the contents of the draft. Before the draft can be saved by the user, the system displays a message prompting the users to solve these errors. After all errors have been solved, the draft can be saved.

This system behavior is also valid when using the *Apply* button. When choosing *Apply*, the system has to make sure that the contents on the UI and the stored contents of the draft are identical. If the errors described above occur, the system displays the same message prompting the user to solve them.

> **i Note**
>
> This message lists only errors related to technical inconsistencies, not to logical inconsistencies. For example, if a user enters a business partner that does not exist, this error is not displayed. These types of errors are displayed in a state message when saving the object.

Related Links:

- Draft Handling in SAP Fiori Design Guidelines
- Developing New Transactional Apps with Draft Capabilities

Non-Draft Apps

By default, you need to create apps that use draft handling. For general information about draft handling, see https://experience.sap.com/fiori-design-web/draft-handling/ and Draft Handling [page 1631].

You can create non-draft apps, however, you need to consider the specific features and restrictions listed below, as compared to those for draft apps. Create non-draft apps only for simple scenarios without complex flows.

> **i Note**
>
> Example of a complex flow: Creating items and subitems in a single step, before saving.
>
> Example of a simple flow: An app used occasionally to change specific fields.

> **i Note**
>
> Do not combine draft and non-draft entity sets in one app. Exception: A draft-enabled entity set can contain a non-draft child for display purposes only. For example, a sales order might contain a non-draft contact sub-object.

Saving Data

In non-draft scenarios, data is not automatically saved to the back-end system when a user changes data on the UI. Users always have to save the new or changed data when they leave a page, for example, in these cases:

- Creating new subitems
- Editing existing subitems

- Navigating away, for example, by using a chevron in a table
- Navigating back to the list report

The *Save* button is also available on subobject pages. If a user wants to navigate away from the edit screen that contains unsaved data, a data-loss message is displayed.

The following features are also **not** available in non-draft apps:

- Start working on an object, save it as incomplete, and continue later
- Automatic saving and data-loss prevention: Keep working while data is saved automatically and asynchronously
- Navigation within the app to different pages without having to keep saving in between

> **i** Note
>
> On the object page, for fields that contain a combination of a value and text, for example, currency and EUR or text arrangement and unit, changed values are displayed correctly after saving, only if you have defined a dedicated side effect.

Locking

This system doesn't lock objects when data is being edited. Data might be lost if two users work on the same object at the same time. The data of the last user to save is the data in the final version. Consequently, data is not read again from the back-end system when the user starts editing.

Navigation

The following navigation actions or events discard the entered data:

- SAP Fiori actions: For example, Back or Home
  When performing these actions, the SAP Fiori data-loss message is displayed.
- Launchpad signout
  Data is lost, no data-loss message is displayed.
- Browser actions: For example, back, forward, open bookmark, change URL, refresh, and closing the browser.
  When performing these actions, the data is lost, and no data-loss message is displayed.

Extension Points and Secured Execution

Various checks can be executed. For example, check for the needed busy indicator or to see whether the data loss popup is needed. The following table contains the input parameters for the method used in the check, and the corresponding system behavior:

Table 80: Example parameters

Parameter	System Behavior
`busy.set=true`	Triggers a busy indicator when a function is being executeed. Can be set to false in case of immediate completion.
`busy.check=true`	Checks whether the application is currently busy. The function is executed only if it's not busy. Has to be set to false if the function is not triggered by direct user interaction, but as a result of another function that set the application to busy.
`dataloss.popup=true`	Displays a data-loss message before execution of the function if needed (that is, in non-draft cases when the model or registered methods contain pending changes).
`dataloss.navigation=false`	Indicates that execution of the function leads to navigation, that is, leaves the current page, which induces a slightly different text for the data-loss message.

Further Draft Features that Are not Available in Non-Draft Apps

The following draft app features are **not available** in non-draft apps:

- No data loss, connectivity disruption or session time-out
- Device switch: Start on one device and continue on another
- Action and field control adjustments during data entry
- Checks during data entry
- Actions on entered data (without saving or triggering side effects)
- Calculations and defaulting during data entry
- Context-dependent value helps (based on currently entered data)
- Sorting and filtering in editable tables for data entry
- Flexible column layout

Handling Inconsistent Input

Users might enter data that is so inconsistent that the system cannot store it.

For example, you've entered characters in a number field. Or you've entered more characters than the field length allows. The system behaves as described below, depending on whether your app is draft-enabled or not.

- System behavior in draft-enabled apps

In a case like this, the contents of the UI differ from the contents of the draft. Before the user can save the draft, the system displays a message prompting the users to solve these errors. After all errors have been solved, the draft can be saved.
The system also behaves like this when using the *Apply* button. When you choose *Apply*, the system has to make sure that the content on the UI and the stored content of the draft are identical. If the errors described above occur, the system displays the same message prompting the user to solve them.

- System behavior in non-draft apps
  Before users can save the data, the system displays a message prompting the users to solve these errors. After all errors have been solved, the object can be saved.

> **i Note**
>
> This message lists only errors related to technical inconsistencies, not to logical inconsistencies. For example, if a user enters a business partner that does not exist, this error is not displayed. These types of errors are displayed in a state message when you save the object.

Using Analytical Parameters from the Back End

List report and object page support analytical parameters that have been defined in the back end.

They are controlled by the `considerAnalyticalParameters` property, which is defined in the manifest.json of your application. As soon as the `considerAnalyticalParameters` flag is set to `true`, the framework automatically provides filters for the analytical parameter in the list report. These filters automatically become mandatory, and the data on the UI changes according to the value provided in the filter.

> **‹≡› Sample Code**
>
> Manifest.json showing the `considerAnalyticalParameters` property

```
"sap.ui.generic.app": {
            "_version": "1.3.0",
            "settings": {
                  "considerAnalyticalParameters": true,
                  "forceGlobalRefresh": true,
                  "inboundParameters": {
                        "Supplier": {
                              "useForCreate": true
                        }
                  },
                  "objectPageDynamicHeaderTitleWithVM": true
            },
.
.
.
.
```

Configuring List Report Features

You can use annotations to set up various elements in the list report view, such as the table type and smart filter bar.

Enabling Variant Management

Variants let you store settings that users create for the smart filter bar, such as selection fields and layout, and for the smart table, such as sorting and visible columns.

In harmonized variant management for list report, you can save filter setting and values of smart filter bar, and personalisation of smart tables together using the variant for the smart filter bar.

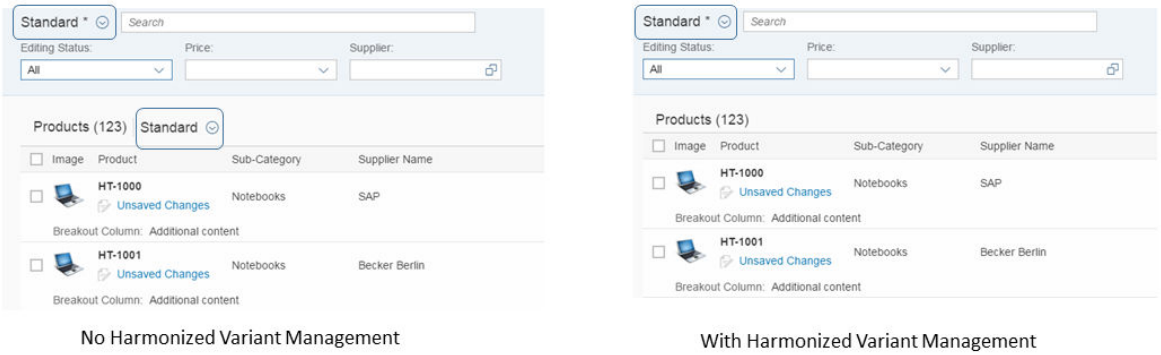


Figure 258: Variant Management

> **i Note**
>
> You can define variants for specific selections of data on the user interface, for example, based on filter settings. In the definition dialog, these variants are called views, however, the feature is called variant management. Therefore, for clarity, we use the term variant management in this section.

Code Sample

In the `manifest.json` file, the `smartVariantManagement` variable is set to `true` by default or `false` based on the option selected in SAP WebIDE Wizard. It can be changed in the manifest as below:

```
"sap.ui.generic.app": {
        "_version":"1.1.0",
        "pages": [
            {
                "entitySet": "XXXXXX_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true,
                    "settings" : {
                        "gridTable" : false,
```

```
                        "multiSelect": false,
                        "smartVariantManagement": true
```

Enabling Table Filters

In harmonized variant management, by default, you cannot set a filter in the table personalization settings. To enable filtering, set the `enableTableFilterInPageVariant` property in the manifest. If `smartVariantManagement` is set to `false`, then table filter is enabled by default.

Sample Code

```
"sap.ui.generic.app": {
        "_version": "1.3.0",
        "settings": {
        "forceGlobalRefresh": true
        },
        "pages": {
            "ListReport|STTA_C_MP_Product": {
                "entitySet": "STTA_C_MP_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true,
                    "settings": {
                        "smartVariantManagement": true,
                        "enableTableFilterInPageVariant": true
                    }
                },
```

Creating a List Report Without Variant Management

To make things easier, you may want to create an app without variant management. In this case, only the title of the app is displayed. You can create a custom title, if required.

Context

Without variant management, and with no custom title added, your app looks as follows:

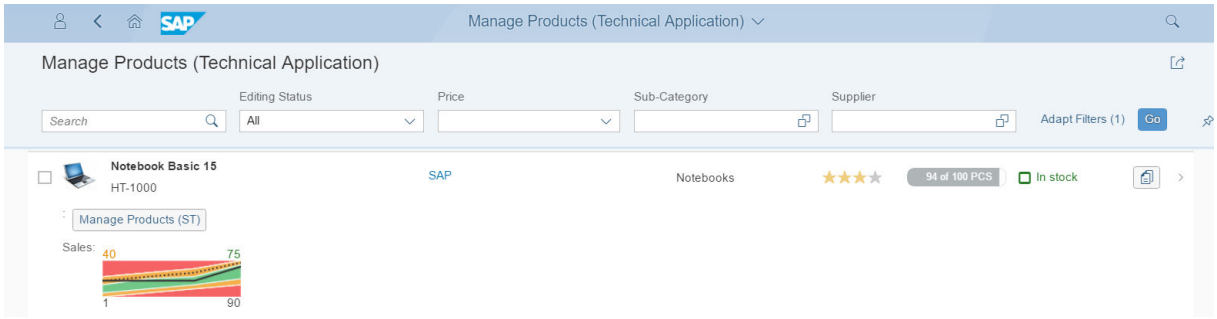
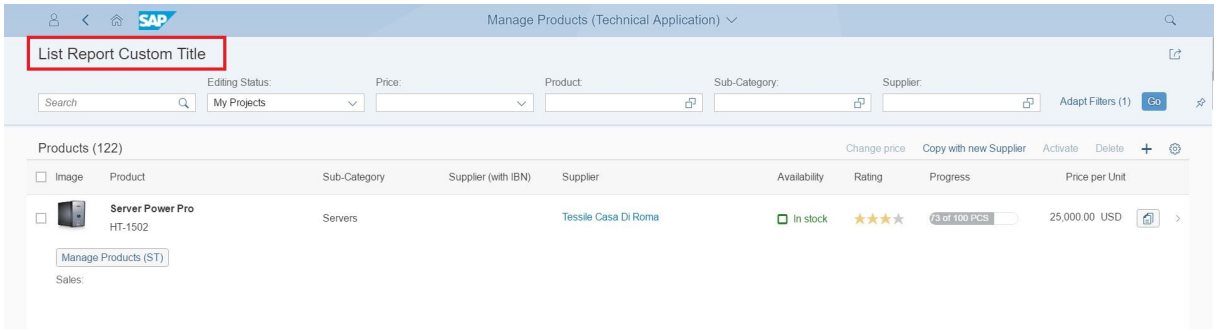


Figure 259: App Without Variant Management

Without variant management and with a custom title added, your app looks like this:

Figure 260: App Without Variant Management and With Custom Title



Procedure

1. Include the `variantManagementHidden` flag in the settings of the list report component in the manifest.json of your app.
   - If you set the flag to `true`, then standard variant management is not available in the app. The app name is displayed instead.
   - If you set the flag to `false` or if the flag is not at all contained in the manifest of the app, the standard variant management is available and can be enabled or disabled. For more information, see Enabling Variant Management [page 1637].

Sample Code

List Report Without Variant Management

```
"sap.ui.generic.app": {
        "_version":"1.1.0",
        "pages": [
            {
                "entitySet": "XXXXXX_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true,
                    "settings" : {
                        "gridTable" : false,
                        "multiSelect": false,
                        "smartVariantManagement": true,
                        "variantManagementHidden": true // Hides Variant
management
                    }
                },
```

2. If you want to use an app-specific title instead of the variant, include the `subTitleIfVariantMgmtHidden` property in the i18n file and enter a text value as shown below:

Sample Code

```
#XTIT,40
subSubTitleIfVariantMgmtHidden = List Report Custom Title
```

3. Add a new property in the manifest.json of the application as shown below.

```
"sap.ui.generic.app": {
        "_version":"1.1.0",
        "pages": [
            {
                "entitySet": "XXXXXX_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true,
                    "settings" : {
                        "gridTable" : false,
                        "multiSelect": false,
                        "smartVariantManagement": true,
                        "variantManagementHidden": true,
                        "subTitleIfVariantMgmtHidden":
"{{subTitleIfVariantMgmtHidden}}" // Adding Custom Title here
                    }
                },
```

Actions in the List Report

The list report supports various types of actions.

Actions in the Table Toolbar

The table toolbar in the list report contains buttons used to trigger actions for the entire list report or for selected items. As shown below, these actions can include generic functions offered by SAP Fiori elements or app-specific actions.

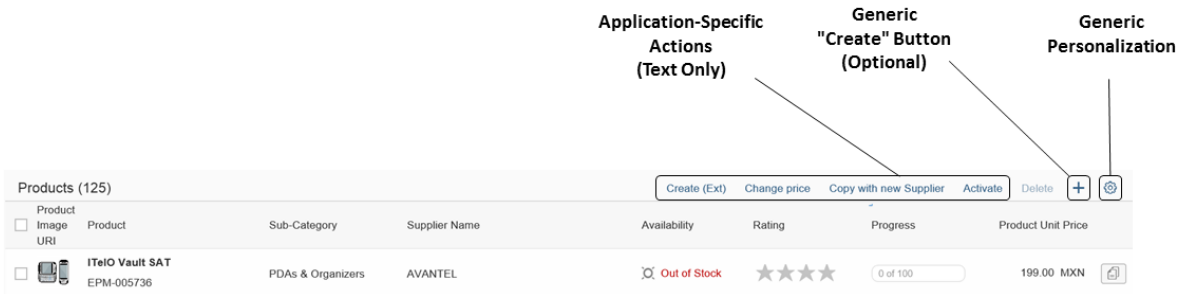


Figure 261: Example: Actions in the Table Toolbar

App-Specific Actions

Depending on your use case, you can define actions that are displayed in the table toolbar for your app, for example, *Copy* and *Approve*. You can define the following types of actions for your app:

- Actions that require user confirmation, for example, for critical actions that have severe consequences. The system opens a dialog in which the user has to confirm the action.
- Actions that require additional user input, for example, an approval comment. The system opens a dialog with one or more entry elements in which the user enters the required data. The system can pre-fill data, if applicable.
- Actions that require none of the above. The system triggers the action.

Generic Actions

You can use the following generic actions in the table toolbar:

- Action that triggers external navigation
- Creation of a new item (+ button) if the entity set can be created
- Deletion of one or more items if the entity set can be deleted

Global Actions and Line Item Actions

For information about setting up global actions that apply to the entire app, see Adding Custom Actions Using Extension Points [page 1831].

For information about defininig line item actions, see Adding Line Item Actions in Tables [page 1743].

More Information

Subject	Link
Actions within the list report	Enabling Actions in the List Report [page 1642]
Actions within a line item in the list report	Adding Line Item Actions in Tables [page 1743]
Determining actions (in the footer of a list report)	Adding Determining Actions [page 1778]
Custom actions using extension points	Adding Custom Actions Using Extension Points [page 1831]

Enabling Actions in the List Report

You can enable certain common actions specific to your use case for the list report. You can display the actions in either the toolbar or within a specific column for line items.

Actions in the Toolbar

You can display actions in the toolbar to allow users to perform an action for one or more lines in the table.

Actions in Toolbar

The table below describes how to set up your annotations so that generic actions provided by SAP Fiori elements and application-specific actions triggering external navigation are rendered in the toolbar:

Action	Setting	Comments
Create (+)	`sap:creatable="true"` for the root entity set	The *Create* feature is enabled by default, as the entity set is already creatable.
Delete	`sap:deletable="true"` for the root entity set	The *Delete* feature is enabled by default, as the entity set is already deletable. Note that if you want to specify conditions for deletion (using the `deletable-path` annotation), you must ensure that the setting `sap:deletable` has not been made.
Action triggering external navigation	Add the following property: `<PropertyValue Property="RequiresContext" Bool="true"/>`	For more information about configuring intent-based navigation, see Configuring External Navigation [page 1563].

Enable or Disable *Delete* Button (Using deletable-path Annotations)

You can enable or disable the *Delete* button in the list report based on conditions specified in the back-end system. For example, you can disable deletion for a sales order that has already been paid. In this case, if a user

selects an item that cannot be deleted, the *Delete* button is disabled. In addition, if the user navigates from this item in the list report to the object page, the *Delete* button is hidden.

In your annotation, set the `deletable-path` to point to a particular property of an object (entity) in the back-end system that is either `true` or `false`. If the value of this property is `true`, the *Delete* button is enabled; if it is `false`, it is disabled. If you want to use the `deletable-path` annotation to specify conditions for deletion, you have to ensure that the setting `sap:deletable` is not present in your annotations.

The code sample below shows you how to set up your annotation to enable or disable the *Delete* button, based on the value of the `Delete_mc` property in the back-end system.

```
<Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_Product">
    <Annotation Term="Org.OData.Capabilities.V1.DeleteRestrictions">
        <Record>
            <PropertyValue Property="Deletable" Path="Delete_mc"/>
        </Record>
    </Annotation>
</Annotations>
```

Application-Specific Actions in Toolbar (Specify Text)

To specify a text for your action, use the `com.sap.vocabularies.UI.v1.DataFieldForAction` property and specify the text to display. The example below shows you how to display an action to create a copy of the list item in the toolbar:

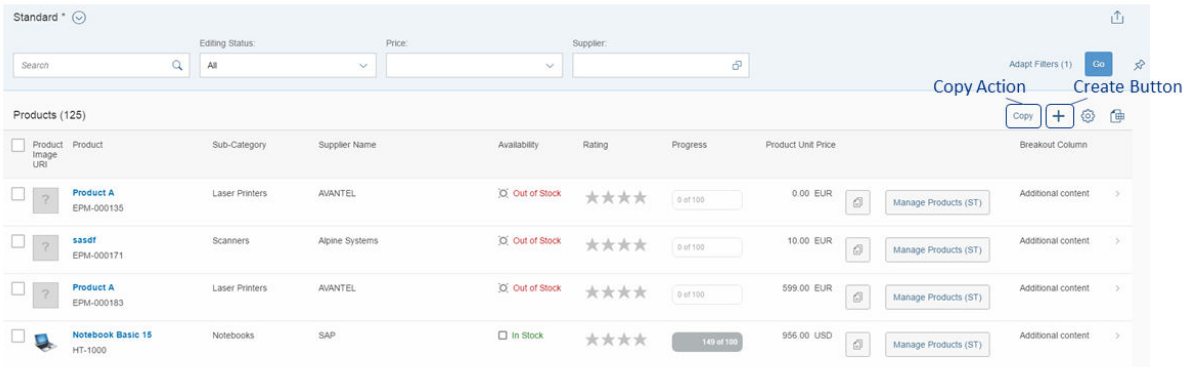


Figure 262: List Report: Annotation DataFieldForAction for Defining Application-Specific Actions

Sample Code

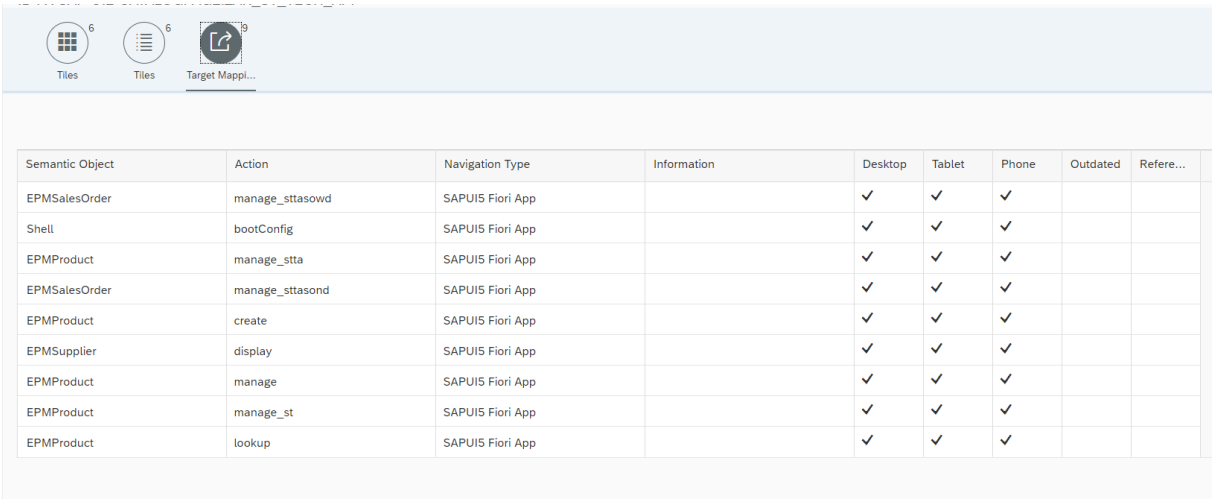
```
<Annotation Term="UI.LineItem">
    <Collection>
        <Record Type="UI.DataFieldForAction">
            <PropertyValue Property="Label" String="Copy"/>
            <PropertyValue Property="Action"
String="SEPMRA_PROD_MAN.SEPMRA_PROD_MAN_Entities/SEPMRA_C_PD_ProductCopy"/>
        </Record>
    <Collection>
        <Record Type="UI.DataField">
        ...
    </Collection>
</Annotation>
```

Enable or Disable Buttons Triggering External Navigation

In a `DataFieldForIntentBasedNavigation`, you can specify `RequiresContext`. Setting it to `True` means that a line needs to be selected for the button to be enabled. Otherwise, it is disabled.

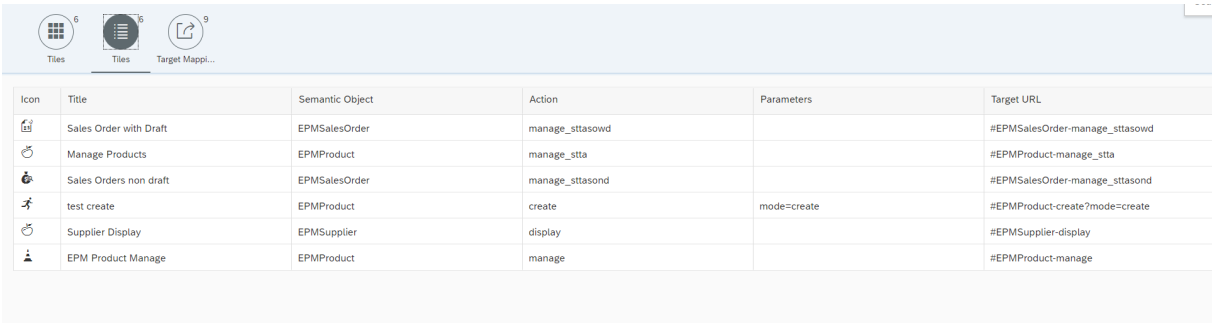
Display or Hide Buttons Triggering External Navigation

You can define that context-independent buttons (`RequiresContext` is set to `False`) triggering external navigation are displayed only if the navigation target is supported on the current device. As a prerequisite, you need to have maintained the navigation target in the SAP Fiori launchpad, as shown in the figures below:



Semantic Object	Action	Navigation Type	Information	Desktop	Tablet	Phone	Outdated	Refere...
EPMSalesOrder	manage_sttasowd	SAPUI5 Fiori App		✓	✓	✓		
Shell	bootConfig	SAPUI5 Fiori App		✓	✓	✓		
EPMProduct	manage_stta	SAPUI5 Fiori App		✓	✓	✓		
EPMSalesOrder	manage_sttasond	SAPUI5 Fiori App		✓	✓	✓		
EPMProduct	create	SAPUI5 Fiori App		✓	✓	✓		
EPMSupplier	display	SAPUI5 Fiori App		✓	✓	✓		
EPMProduct	manage	SAPUI5 Fiori App		✓	✓	✓		
EPMProduct	manage_st	SAPUI5 Fiori App		✓	✓	✓		
EPMProduct	lookup	SAPUI5 Fiori App		✓	✓	✓		

Figure 263: SAP Fiori launchpad: Maintain the supported devices for the combination of semantic object and action.



Icon	Title	Semantic Object	Action	Parameters	Target URL
	Sales Order with Draft	EPMSalesOrder	manage_sttasowd		#EPMSalesOrder-manage_sttasowd
	Manage Products	EPMProduct	manage_stta		#EPMProduct-manage_stta
	Sales Orders non draft	EPMSalesOrder	manage_sttasond		#EPMSalesOrder-manage_sttasond
	test create	EPMProduct	create	mode=create	#EPMProduct-create?mode=create
	Supplier Display	EPMSupplier	display		#EPMSupplier-display
	EPM Product Manage	EPMProduct	manage		#EPMProduct-manage

Figure 264: SAP Fiori launchpad: Maintain the mandatory parameters for semantic object and action

> **i Note**
>
> - As shown above, in the SAP Fiori launchpad, you maintain mandatory parameters for navigation, for example, a sales order ID. If you have specified `RequiresContext: False`, for the combination of semantic object and action, and for this combination you maintain a mandatory parameter in the SAP Fiori launchpad, these settings contradict each other and the button is not displayed.
> - The system checks only those actions that were created using annotations, not extension points.
> - This feature is not relevant for context-dependent buttons.

For information about context-dependent and context-independent actions, see Actions [page 1605].

Actions for Line Items

For information about setting up annotations for actions within a line item of the list report, see Adding Line Item Actions in Tables [page 1743].

More Information

General information on actions: Actions [page 1605]

Multiple Views on List Report Tables

By default, the list report displays only one table. You can define multiple views of a table, and add a chart, if required.

This video shows the step-by-step procedure for defining multiple views:

> **i Note**
>
> You can define variants for specific selections of data on the user interface, for example, based on filter settings. In the definition dialog, these variants are called views, however, the feature is called variant management. Therefore, for clarity, we use the term variant management in this section.

You have the following options:

- **A single table for all views ("single table mode")**: The UI contains a single table instance, one table toolbar, and (if activated) one table variant management. To switch between the views, a segmented button is rendered in the table toolbar. If there are more than three views, a select control is rendered instead of a segmented button.

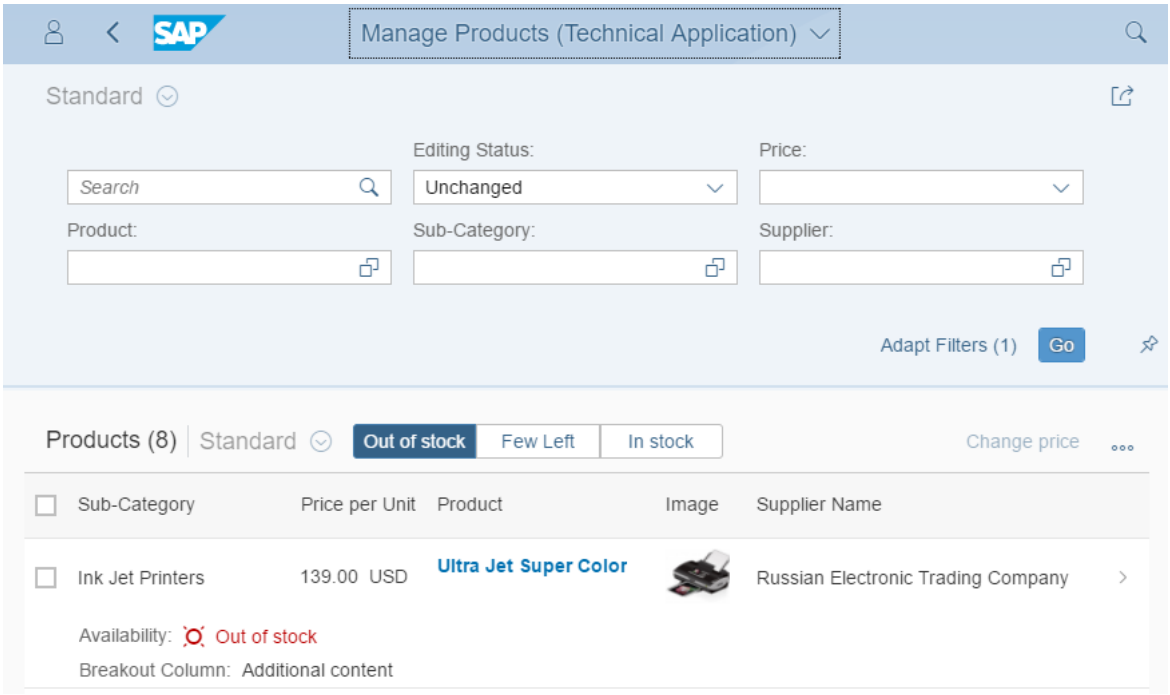


Figure 265: Single Table Mode

- **A separate table for each view ("multiple table mode")**: If there are n views, the UI contains n table instances. This results in n separate table toolbars and n separate table variant managements. An icon tab bar is rendered above the table for switching between the views (table instances). Only the table on the currently selected tab is visible.

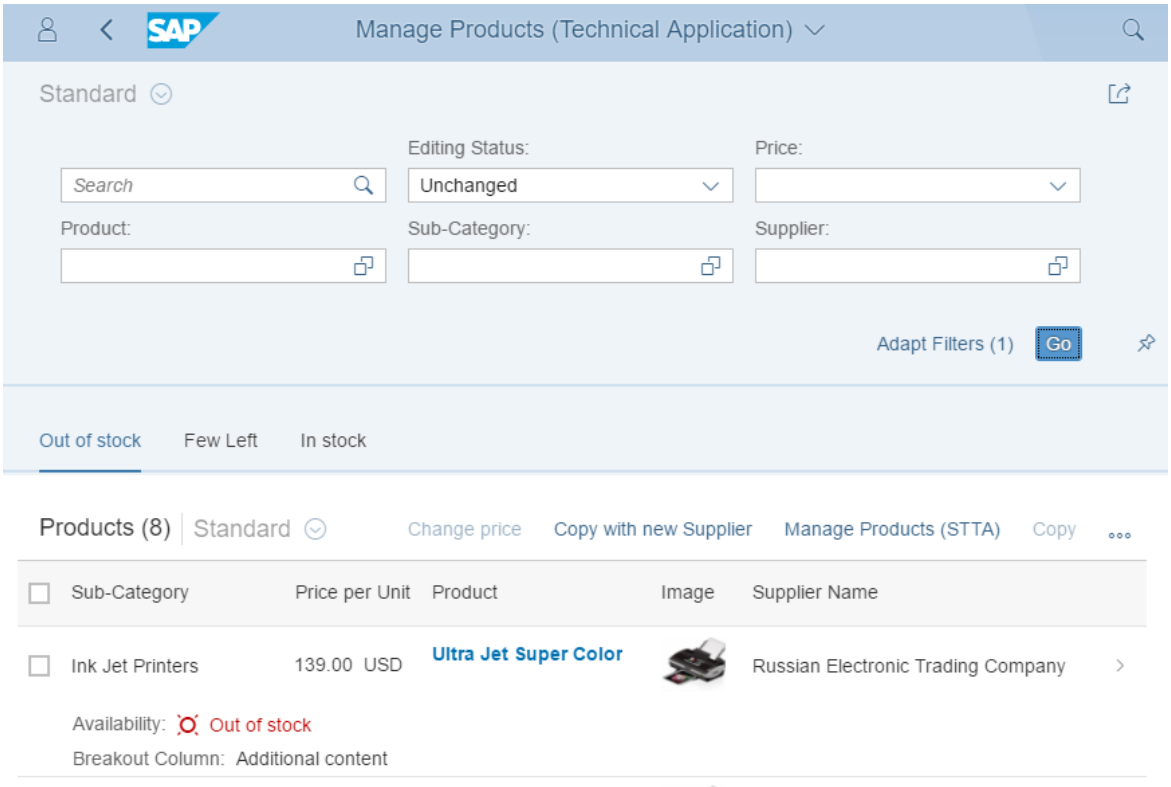


Figure 266: Multiple Table Mode

- In multiple table mode, in addition to tables, you can also display charts on specific tab pages. Tables are displayed by default.

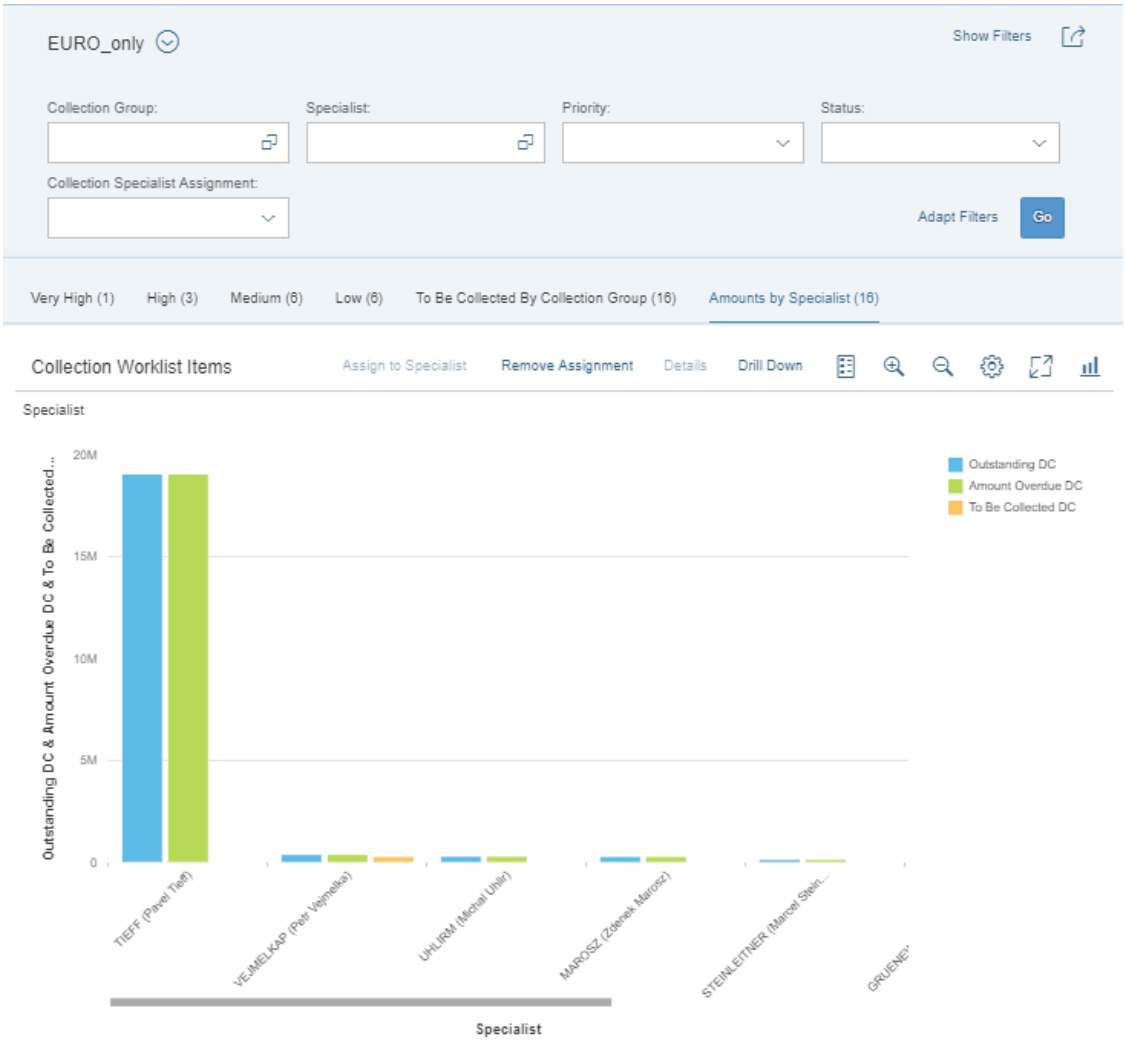


Figure 267: Multiple Table Mode with Charts

- On each tab, you can also display data for different entity sets, for example, a sales order or a supplier. To do so, add the entity set to the corresponding tab in the manifest.

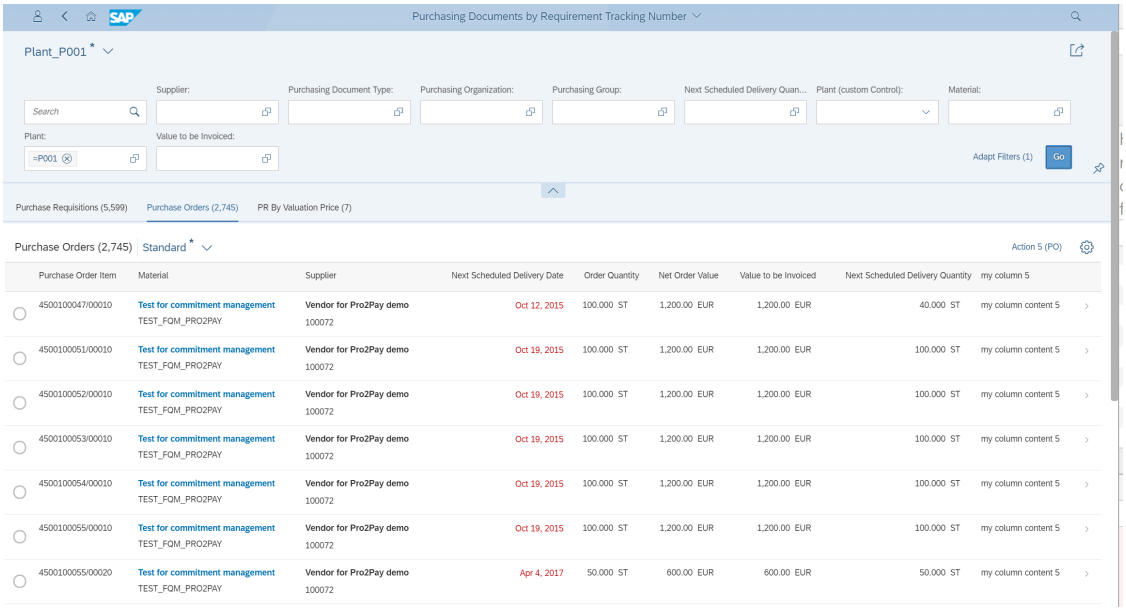


Figure 268: Multiple views on a list report with different entity sets

Which Annotations Should I Use?

- If you only want to describe **which** data should be displayed in a view, you can define a `SelectionVariant` containing filter criteria for the data. See Defining Multiple Views on a List Report Table - Single Table Mode [page 1649].
- If you also want to describe **how** the data should be displayed (for example, different sort orders in a table or a different visualization in a table or chart), you can define a `SelectionPresentationVariant`. Note that you can use this annotation only for multiple table mode and multiple table mode with charts. See Defining Multiple Views on a List Report Table - Multiple Table Mode [page 1651].
- If all you want to do is use a different visualization, you can define a `PresentationVariant`.

> **i Note**
>
> For information about `SelectionVariants`, `PresentationVariants`, and `SelectionPresentationVariants`, see the OData vocabulary at https://wiki.scn.sap.com/wiki/display/EmTech/OData+4.0+Vocabularies+-+SAP+UI.

Related Information

Defining Multiple Views on a List Report Table - Single Table Mode [page 1649]
Defining Multiple Views on a List Report Table - Multiple Table Mode [page 1651]
Defining Multiple Views on a List Report with Different Entity Sets and Table Settings [page 1656]

Defining Multiple Views on a List Report Table - Single Table Mode

You can define multiple views of a table and display them in single table mode. Users can switch between views using a segmented button.

Context

To define multiple views using single table mode, perform the following steps:

Procedure

1. Add `SelectionVariants` to your annotations file.

   ### Sample Code

   `SelectionVariant` that filters for items that cost a certain amount (for example, at least 5,000 euros).

```
<Annotation Term="UI.SelectionVariant" Qualifier="Expensive">
    <Record>
        <PropertyValue Property="Text" String="Expensive">
        </PropertyValue>
        <PropertyValue Property="SelectOptions">
            <Collection>
                <Record Type="UI.SelectOptionType">
                    <PropertyValue Property="PropertyName"
                        PropertyPath="GrossAmount" />
                    <PropertyValue Property="Ranges">
                        <Collection>
                            <Record Type="UI.SelectionRangeType">
                                <PropertyValue Property="Option"

EnumMember="UI.SelectionRangeOptionType/GE" />
                                <PropertyValue Property="Low"
String="5000" />
                            </Record>
                        </Collection>
                    </PropertyValue>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

   ### Note

   For the `SelectionVariant`, the following applies:

   - The `FilterExpression` of the `SelectionVariantType` is not supported.

- The following `SelectionRangeOptionTypes` are supported without any wildcards, for example, *, ?, ...:
  - EQ: Equal to
  - BT: Between
  - LE: Less than or equal to
  - GE: Greater than or equal to
  - NE: Not equal to
  - GT: Greater than
  - LT: Less than

For the `PresentationVariant`, `SortOrders` is supported.

2. Extend the manifest.json to switch on the multiple view feature and link to the variants you have added to your annotations. You do this in the list report settings section under `sap.ui.generic.app`. Use `quickVariantSelection` for single table mode.
   - The `variants` section (lines 11-20) contains a set of entries that point to the variants defined in the annotations.
   - For each entry under `variants` (for example, lines 12-15), define an `annotationPath` (line 14) for a specific variant.
   - Provide a key entry (line 13) that is used for initializing the corresponding `SegmentedButton` item. This is a mandatory entry.

Sample Code

```
1 ...
2 "sap.ui.generic.app": {
3     "pages": [
4        {
5            "entitySet": "C_STTA_SalesOrder_WD_20",
6            "component": {
7                "name": "sap.suite.ui.generic.template.ListReport",
8                "list": true,
9                "settings": {
10                    "quickVariantSelection": {
11                        "variants": {
12                            "0": {
13                                "key": "_tab1",
14                                "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#Expensive"
15                            },
16                            "1": {
17                                "key": "_tab2",
18                                "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionPresentationVariant#Cheap"
19                            }
20                        }
21                    }
22                    ...
```

Related Information

Adding Segmented Buttons to a Table Toolbar [page 1766]

Defining Multiple Views on a List Report Table - Multiple Table Mode

You can define multiple views of a table and display them in multiple table mode. In addition to tables, you can display charts in the views. Users can switch between views using an icon tab bar.

Prerequisites

If you want to use charts in multiple table mode, you need to have defined a `UI.Chart` annotation, including a qualifier, as follows:

```
<Annotation Term="UI.Chart" Qualifier="Chart1">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Revenue by Customer"/>
        <PropertyValue Property="Description" String="Net Revenue by Customer"/>
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Column"/>
        <PropertyValue Property="Dimensions">
            <Collection>
                <PropertyPath>ProductCategory</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Measures">
            <Collection>
                <PropertyPath>NetAmount</PropertyPath>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Context

To define multiple views using multiple table mode, perform the following steps:

Procedure

1. Add `SelectionVariants` or `SelectionPresentationVariants` to your annotations file.

   > **i Note**
   >
   > You can reference different `UI.LineItem` annotations for different tabs. To do so, reference the annotation under `PresentationVariant/Visualizations`. If there is no `PresentationVariant/Visualizations` for a tab, a default `UI.LineItem` (without a qualifier) is taken into account.

   - Multiple table mode with table

Sample Code

`SelectionVariant` that filters for items that cost a certain amount, (for example, at least 5,000 euros).

```
<Annotation Term="UI.SelectionVariant" Qualifier="Expensive">
    <Record>
        <PropertyValue Property="Text" String="Expensive">
        </PropertyValue>
        <PropertyValue Property="SelectOptions">
            <Collection>
                <Record Type="UI.SelectOptionType">
                    <PropertyValue Property="PropertyName"
                        PropertyPath="GrossAmount" />
                    <PropertyValue Property="Ranges">
                        <Collection>
                            <Record Type="UI.SelectionRangeType">
                                <PropertyValue Property="Option"

EnumMember="UI.SelectionRangeOptionType/GE" />
                                <PropertyValue Property="Low"
String="5000" />
                            </Record>
                        </Collection>
                    </PropertyValue>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Sample Code

`SelectionPresentationVariant` containing a `SelectionVariant` and a `PresentationVariant`. The `SelectionVariant` filters for items with a price less than a certain amount (for example, 5,000 euros), the `PresentationVariant` defines the ascending sort order:

```
<Annotation Term="UI.SelectionPresentationVariant" Qualifier="Cheap">
    <Record>
        <PropertyValue Property="Text" String="Cheap">
        </PropertyValue>
        <PropertyValue Property="SelectionVariant">
            <Record>
                <PropertyValue Property="Text" String="Cheap">
                </PropertyValue>
                <PropertyValue Property="SelectOptions">
                    <Collection>
                        <Record Type="UI.SelectOptionType">
                            <PropertyValue Property="PropertyName"
                                PropertyPath="GrossAmount" />
                            <PropertyValue Property="Ranges">
                                <Collection>
                                    <Record
Type="UI.SelectionRangeType">
                                        <PropertyValue Property="Option"

EnumMember="UI.SelectionRangeOptionType/LT" />
                                        <PropertyValue Property="Low"
String="5000" />
                                    </Record>
                                </Collection>
                            </PropertyValue>
                        </Record>
                    </Collection>
```

```
                </PropertyValue>
            </Record>
        </PropertyValue>
        <PropertyValue Property="PresentationVariant">
            <Record>
                <PropertyValue Property="SortOrder">
                    <Collection>
                        <Record>
                            <PropertyValue Property="Property"
                                PropertyPath="GrossAmount" />
                            <PropertyValue Property="Descending"
Bool="false" />
                        </Record>
                    </Collection>
                </PropertyValue>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

- Multiple table mode with chart
  Reference the `UI.Chart` annotation in your `SelectionPresentationVariant` or `PresentationVariant` for your view.

```
<Annotation Term="UI.SelectionPresentationVariant" Qualifier="Chart1">
    <Record>
        <PropertyValue Property="Text" String="Chart1"/>
            <PropertyValue Property="SelectionVariant">
                <Record>
                    ....
                </Record>
            </PropertyValue>
        <PropertyValue Property="PresentationVariant">
            <Record>
                <PropertyValue Property="Visualizations">
                    <Collection>
                        <AnnotationPath>@UI.Chart#Chart1</AnnotationPath>
                    </Collection>
                </PropertyValue>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

i Note

For the `SelectionVariant`, the following applies:

- The `FilterExpression` of the `SelectionVariantType` is not supported.
- The following `SelectionRangeOptionTypes` are supported without any wildcards, for example, *, ?, ...:
  - EQ: Equal to
  - BT: Between
  - LE: Less than or equal to
  - GE: Greater than or equal to
  - NE: Not equal to
  - GT: Greater than
  - LT: Less than

For the `PresentationVariant`, `SortOrders` and visualizations are supported.

2. Extend the manifest.json to switch on the multiple view feature and link to the variants you have added to your annotations. You do this in the list report settings section under `sap.ui.generic.app`. Use `quickVariantSelectionX` for multiple table mode or multiple table mode with charts.
   - The `variants` section (lines 11-20) contains a set of entries that point to the variants defined in the annotations.
   - For each entry under `variants` (for example, lines 12-15), define an `annotationPath` (line 14) of a specific variant.
   - Provide a key entry (line 13) that is used for initializing the corresponding `IconTabBar` item. This entry is mandatory.

Sample Code

```
1 ...
2 "sap.ui.generic.app": {
3     "pages": [
4         {
5             "entitySet": "C_STTA_SalesOrder_WD_20",
6             "component": {
7                 "name": "sap.suite.ui.generic.template.ListReport",
8                 "list": true,
9                 "settings": {
10                    "quickVariantSelectionX": {
11                        "variants": {
12                            "Expensive": {
13                                "key": "Expensive",
14                                "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#Expensive"
15                            },
16                            "Cheap": {
17                                "key": "Cheap",
18                                "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionPresentationVariant#Cheap"
19                            }
20                        }
21                    }
22                    ...
```

Note

- If you want to enable auto-binding, do not use key user adaptation for changing the smart table's `enableAutoBinding` properties. From a performance perspective, this leads to backend requests for each table instance. To achieve the required behavior (that is, only rebind the currently visible table), you can add an entry `enableAutoBinding: true` under `quickVariantSelectionX`. This ensures the behavior on list report page level.
- Stable IDs: As there are separate table instances for each tab, table-specific IDs (such as IDs for tables, toolbar actions, draft indicators in table columns) get a suffix "-<key>", where <key> is the variant key you have specified in the manifest (line 13). This avoids duplicate ID errors and allows you to adapt specific tables via runtime adaptation (for example, hiding a toolbar action for a specific table).

3. If you use charts in multiple table mode, you can implement the following features:
   - Actions in toolbars for charts

For charts in multiple table mode, actions from the annotations (UI.Chart/Actions only) and custom actions that were added using extension points are supported. If custom action buttons are relevant to selection, they are disabled if no chart bar is selected. If not, they are enabled.

```
<Annotation Term="UI.Chart" Qualifier="Chart4">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Column"/>
        <PropertyValue Property="Dimensions">
            <Collection>
                <PropertyPath>CompanyCode</PropertyPath>
                <PropertyPath>Customer</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Measures">
            <Collection>
                <PropertyPath>AmountInTransactionCurrency</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Actions">
            <Collection>
                <Record Type="UI.DataFieldForAction">
                    <PropertyValue Property="Action"
String="ZFAR_CUSTOMER_LINE_ITEMS2_SRV.ZFAR_CUSTOMER_LINE_ITEMS2_SRV_Entitie
s/Create"/>
                    <PropertyValue Property="Label" String="Action 1"/>
                </Record>
                <Record Type="UI.DataFieldForIntentBasedNavigation">
                    <PropertyValue Property="SemanticObject"
String="Customer"/>
                    <PropertyValue Property="Action"
String="postPayment2"/>
                    <PropertyValue Property="Label" String="SO Navigation
(M)"/>
                    <Annotation
Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/Medium"/>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

- Navigation for charts
  To enable navigation for charts, you have to set the property "`showItemNavigationOnChart`" in the manifest to "true" and maintain an internal navigation target in the manifest.

```
"quickVariantSelectionX": {
    "showCounts": true,
    "variants": {
       "0": {
            "key": "0",
            "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#VAR1"
            },
        "1": {
            "key": "1",
            "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionPresentationVariant#VAR4",
            "showItemNavigationOnChart": true
             }
    }
}
```

For information about how to maintain an internal navigation target, see Configuring Internal Navigation [page 1581].

If navigation is enabled, the *Show details* button is displayed in the popup after you select a chart bar and choose the *Details* button.

On each tab, you can also display data for different entity sets with different table types and other settings, for example, a sales order or a supplier. To do so, add the entity set and/or table settings to the corresponding tab in the manifest. For more information, see Defining Multiple Views on a List Report with Different Entity Sets and Table Settings [page 1656].

Defining Multiple Views on a List Report with Different Entity Sets and Table Settings

You can configure your app to display data for different entity sets and table settings, for example, sales orders or suppliers.

Prerequisite: You have completed the procedure Defining Multiple Views on a List Report Table - Multiple Table Mode [page 1651].

To specify table settings on tab pages, you need to add `tableSettings` to the corresponding tab in manifest.json of your application.

ℹ Note

Implement this feature with caution, and, for example, take the following into account:

- While this feature provides a combined view of different objects, it does not replace dedicated applications, each with their specific purpose.
- Use this feature only to search for and work on similar business objects that have a subset of common fields. Do not use it for random business objects. Changing common fields in the smart filter bar always has an effect on the tab that is currently open, as well as on all other tabs. While you can implement any entity set from a technical perspective, you should take the business and usability perspective into account. Moreover, as this feature affects performance, you should also check any changes in performance when adding entity sets. Note that if you don't follow these recommendations, the application will be responsible for usability and performance.
- Do not combine draft and non-draft entity sets in one list report.
- You cannot use flexible column layout in combination with this feature.
- You can specify different table types for each tab, but there should not be a mix of responsive and non-responsive (grid, tree and analytical) tables.
- You can define custom actions using extension points only for main entity sets. Actions defined for other entity sets are not supported.

To include different entity sets and table settings in multiple views, specify an entity set for each tab in the`"quickVariantSelectionX"` section. See lines 10 to 27 in the code snippet below.

Sample Code

```
"sap.ui.generic.app": {
       "pages": [{
           "entitySet": "C_RequirementTrackingPurReq",
           "component": {
```

```
                "name": "sap.suite.ui.generic.template.ListReport",
                "list": true,
                "settings": {
                    "condensedTableLayout": true,
                    "smartVariantManagement": false,
                   "quickVariantSelectionX": {
                       "showCounts": true,
                       "variants": {
                           "1": {
                               "key": "1",
                               "entitySet": "C_RequirementTrackingPurReq",
                               "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#VAR1",
                           "tableSettings": {
                               "type": "GridTable",
                               "multiSelect": false,
                               "selectAll": false,
                               "selectionLimit": 200
                           }
                           },
                           "2": {
                               "key": "2",
                               "entitySet": "C_RequirementTrackingPurOrd",
                               "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#VAR5",
                           "tableSettings": {
                               "type": "GridTable",
                               "multiSelect": false,
                               "selectAll": false,
                               "selectionLimit": 200
                           }
                           },
                           "3": {
                               "key": "3",
                               "entitySet": "C_RequirementTrackingPurReq",
                               "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionPresentationVariant#VAR6",
                               "showItemNavigationOnChart": true,
                           "tableSettings": {
                               "type": "TreeTable",
                               "multiSelect": false,
                               "selectAll": false,
                               "selectionLimit": 200
                           }
                           }
                       }
                   }
               }
           }
       }]    }
```

Under `"sap.ui.generic.app"/"pages"`, specify the leading entity set. This is used for the smart filter bar and for the footer. Each table or chart has its own entitySet which you specify under `"quickVariantSelectionX"/"variants"`. If you do not specifiy an entity set under `"/"variants"`, the leading entity set is used as a default.

System Behavior for Filters

Filter values from the smart filter bar are applied to the currently visible table only if the corresponding properties are available in the entity type of the table or chart. If not, they are ignored.

The same is true for the display of counts on each tab. For example, as you can see below, the *Plant* field is displayed in each entity set. It influences the number of items displayed on each tab:

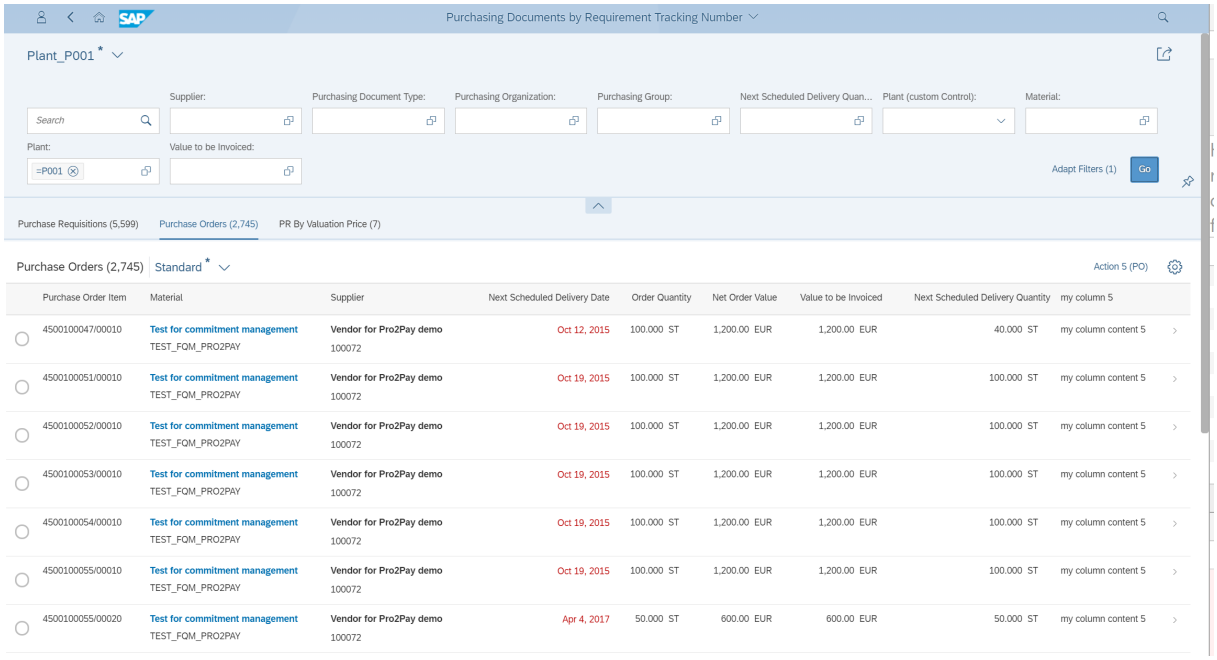


Figure 269: Filtering example

If you add a second filter value, for example, the *Purchasing Group* field, which is found only in the entity type of the second table, only the count of the second tab changes. The counts of the first and third tabs don't change as this field is not relevant for the entity sets. The system displays a message to inform the user about this.

For example, if you add a filter to the *Purchasing Requisition* tab that is not applicable to the *Purchase Orders* tab entity set and switch to the *Purchase Orders*tab, the system displays a message about this. If you close this message and add another filter that is not applicable to the *Purchase Orders* tab entity set, the system displays an updated message saying both filters are not relevant to this entity set.

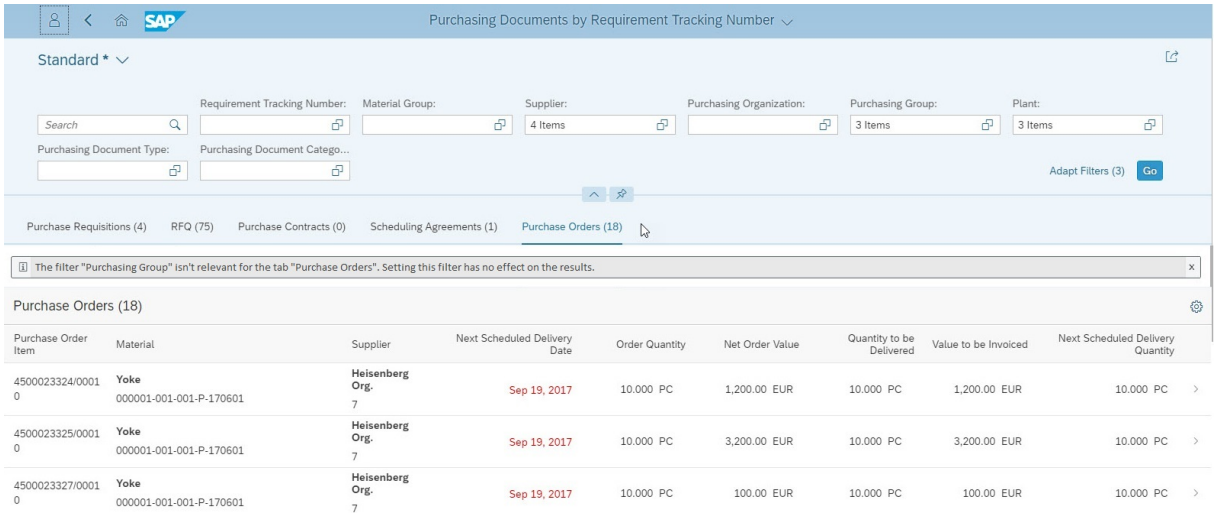


Figure 270: Filtering example

System Behaviour for Different Table Type Settings

Table type settings can be set for each variant under `quickVariantSelectionX` in manifest.json. If table settings are not specified, the system picks the overall table setting and applies them for the variant. It is not possible to have a combination of responsive and non-responsive tables on the same tab. The tables in List Report can either be all responsive or a mix of non-responsive, such as grid, tree or analytical tables. This ensures a consistent scrolling behaviour.

Different tabs on an object page can render different table types. For example, first tab can be a tree table while the second tab can be a grid table.

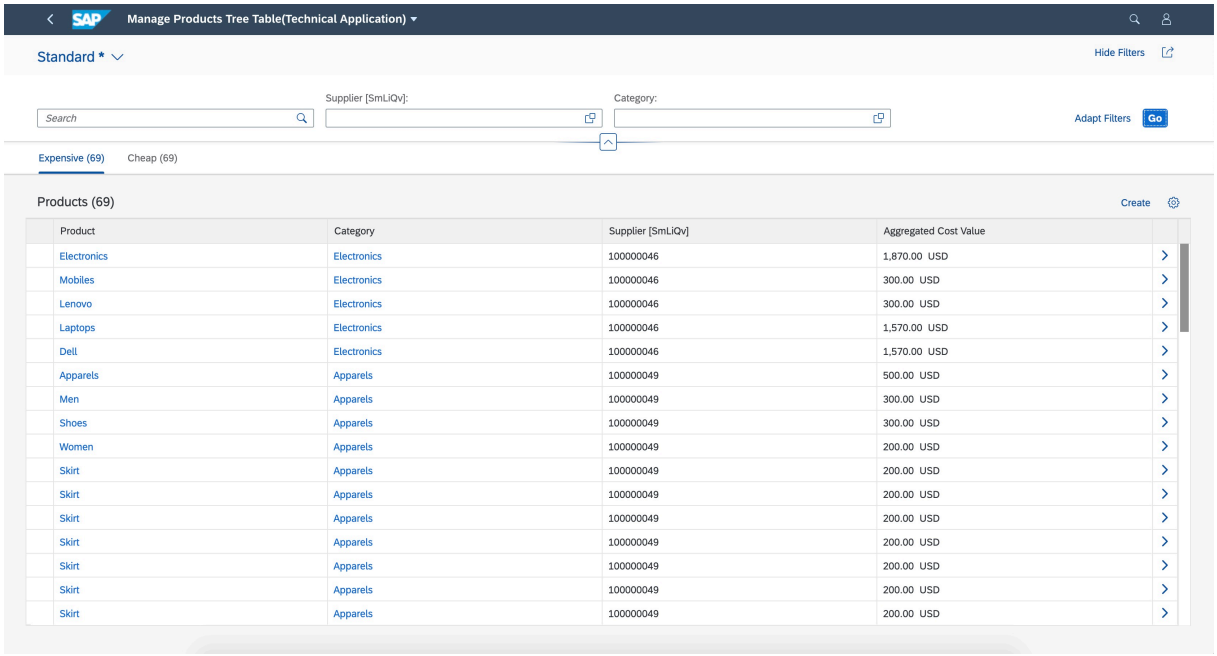
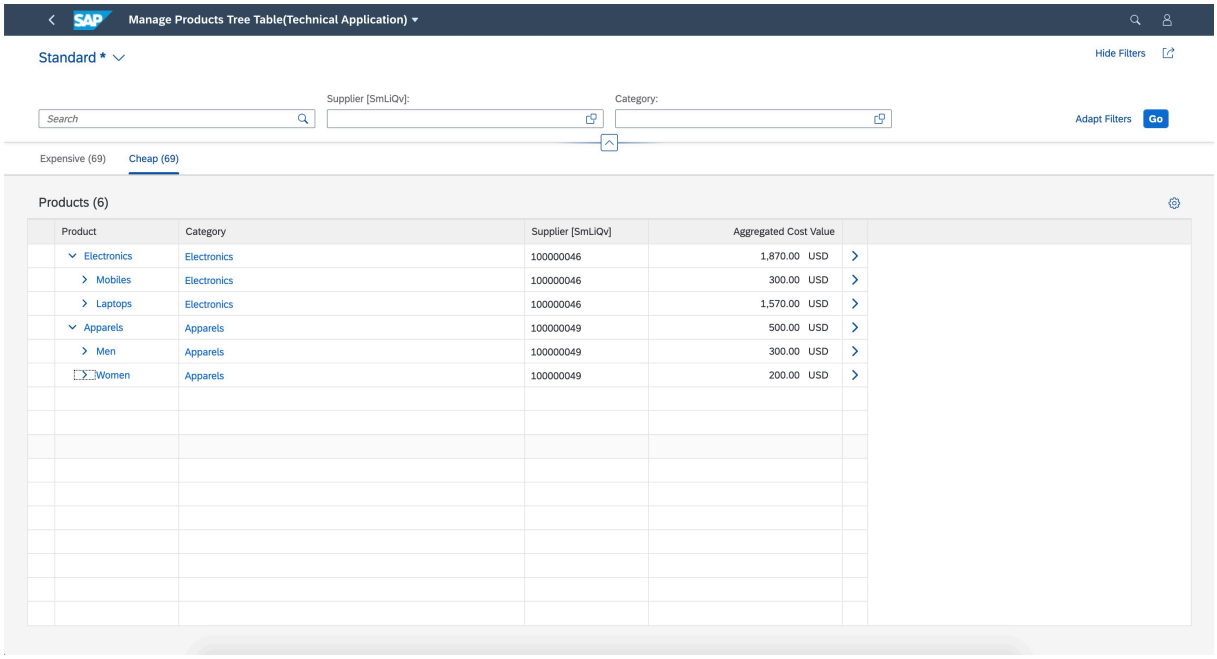


Figure 271: Example of an object page with two tabs of different table types



Related Information

Example: Enable Internal Navigation for a List Report to Object Pages of Different Entity Sets [page 1820]

Disabling the Editing Status Filter

The editing status filter is enabled by default in the list report.

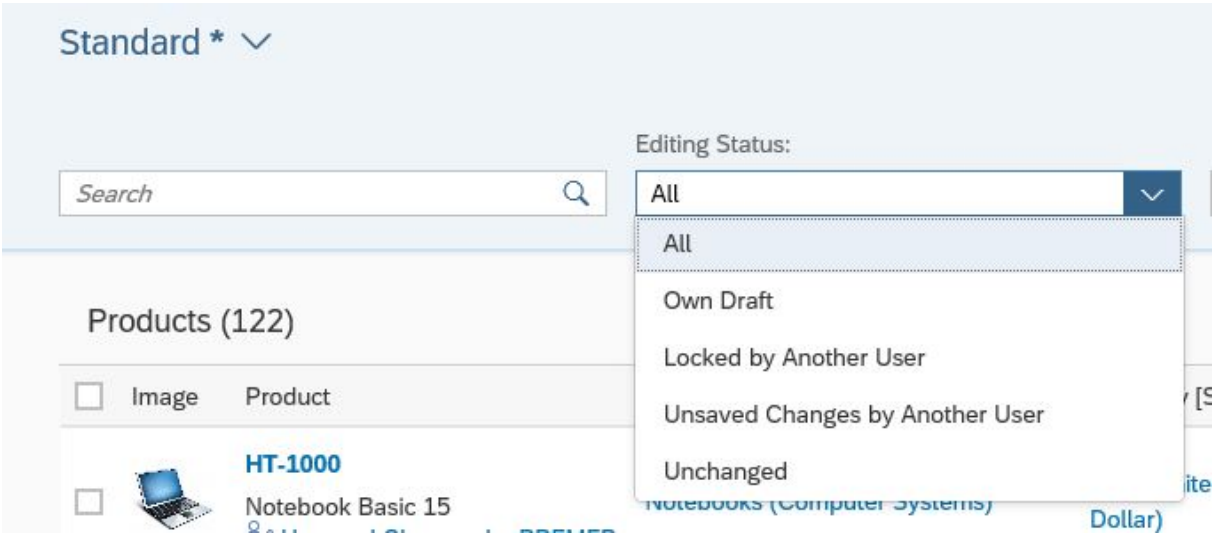


Figure 272: Editing status filter

If required, you can disable this filter for your app. To do so, you need to adapt your OData service in the backend. The resulting metadata.xml file looks as shown below:

```
                    <Annotation Term="Capabilities.NavigationRestrictions">
                          <Record>
                                <PropertyValue
Property="RestrictedProperties">
                                      <Collection>
                                            <Record>
                                                  <PropertyValue
Property="NavigationProperty" NavigationPropertyPath="DraftAdministrativeData" />
                                                  <PropertyValue
Property="FilterRestrictions">
                                                        <Record>

<PropertyValue Property="Filterable" Bool="false" />
                                                        </Record>
                                                  </PropertyValue>
                                            </Record>
                                      </Collection>
                                </PropertyValue>
                          </Record>
                    </Annotation>
```

Adapting the Smart Filter Bar

For the `SmartFilterBar`, you can define application-specific selection fields by using `com.sap.vocabularies.UI.v1.SelectionFields` and field groups for the filter popup.

> **i Note**
>
> Field groups are used in the `SmartFilter` only to group the fields. The grouping can only be seen in the *Adapt Filters* popup. Any label specified in the annotations is used to override the property's default label.
>
> To explicitly define which field groups are to be displayed in the *Adapt Filters* popup, use the `UI.FilterFacets` annotation. If you don't define any field groups using the annotation, the smart filterbar displays all field groups. See also SmartFilterBar.

This video shows the step-by-step procedure for adding a default filter to the smart filter bar:

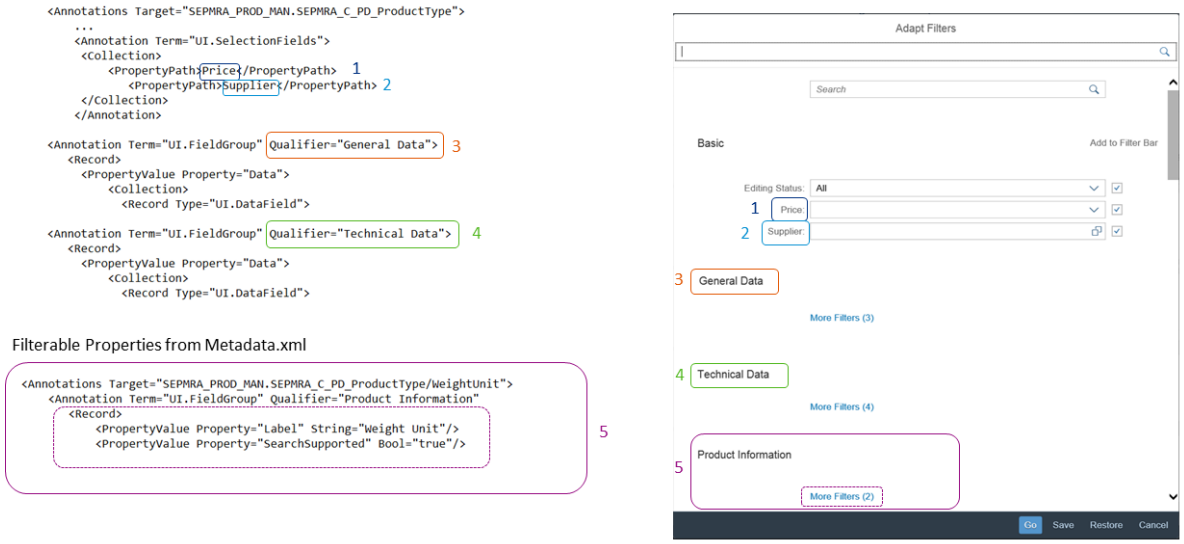


Figure 273: List Report: Adapt Filters Popup

> **i Note**
>
> The *Editing Status* filter is added automatically if you have a draft service.

Adding Filters

Use `com.sap.vocabularies.UI.v1.SelectionFields` to add filter fields to the smart filter bar. Specify additional `PropertyPath` values as shown in the code sample below

Annotation XML

```
<Annotations Target="SEPMRA_PROD_MAN.SEPMRA_C_PD_ProductType">
    ...
    <Annotation Term="UI.SelectionFields">
        <Collection>
```

```
            <PropertyPath>Price</PropertyPath>
            <PropertyPath>Supplier</PropertyPath>
        </Collection>
    </Annotation>
```

Setting Default Filter Value

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType/Supplier">
    <Annotation Term="Common.FilterDefaultValue" String="100000047"/>
</Annotations>
```

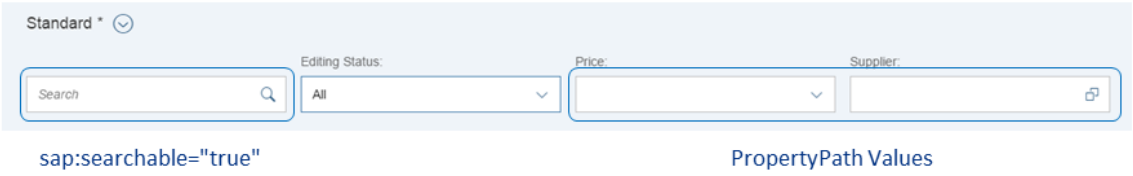
Related Information

Smart Filter Bar [page 2413]

Enabling the Search Function

To enable the search function, `sap:searchable` must be set to `true` for the root entity set.

Figure 274: List Report: Search



Code Sample

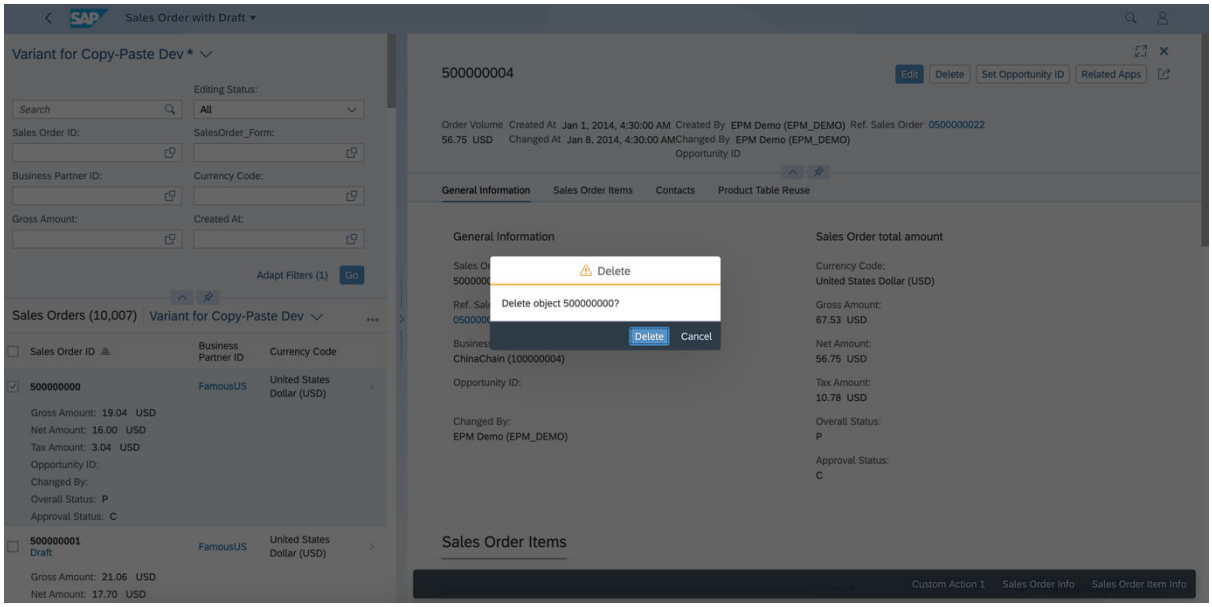
Metadata XML

```
<EntitySet
Name="SEPMRA_C_PD_Product"EntityType="SEPMRA_PROD_MAN.SEPMRA_C_PD_ProductType"
sap:searchable="true" sap:content-version="1"/>
```

Configuring the Delete Dialog Box

You can adapt the text in the Delete dialog box to match your requirements

When a user deletes a record from the list report, the text in the dialog box "Delete Object 500000000?" is displayed in the delete confirmation, informing the user that object 500000000 is being deleted.



The corresponding i18n key for the text used by the SAP Fiori elements framework in the delete confirmation dialog is `ST_GENERIC_DELETE_SELECTED`. You can adapt it by maintaining a text for this key in the app's i18n file.

The context displayed in the Delete dialog box is taken from the `Title` property of the `HeaderInfo` annotation. In the example below, the value mapped to the `"so_id"` property is shown in the dialog text.

Sample Code

```
<Annotation Term="UI.HeaderInfo">
<Record>
  <PropertyValue Property="TypeName" String="Sales Order"/>
     <PropertyValue Property="TypeNamePlural" String="Sales Orders"/>
     <PropertyValue Property="Title">
       <Record Type="UI.DataField">
         <PropertyValue Property="Value" Path="so_id"/>
       </Record>
     </PropertyValue>
   </Record>
 </Annotation>
```

Related Information

Adapting Texts in the Delete Dialog Box (List Report) [page 1843]

Configuring Object Page Features

You can use annotations to set up various elements on the object page, such as the header and sections.

Defining the SmartForm Column Layout

The column layout is used by default in the `SmartForm` on the object page. Do not include any layout or other container controls into the `GroupElement`. Views are also not supported. This could damage the visual layout, keyboard support, and screen-reader support.

You can use the `useColumnLayoutForSmartForm` switch in the manifest, at `sap.ui.generic.app\settings`, to change from the default layout for the `SmartForm` (column layout) to the responsive grid layout. To do so, set the switch to `false`.

Sample Code

```
"sap.ui.generic.app": {
                                "_version": "1.3.0",
                                "settings": {
                                                " useColumnLayoutForSmartForm ":
false
                                }
```

Displaying Actions on the Object Page

Specific rules apply when displaying actions on the object page.

Order of Actions

The order of the application-specific actions follows the order defined in the metadata file. Generic actions are placed after application-specific actions.

Display Based on Mode

The display of actions depends on which mode the user is in:

- In *Display* mode, the relevant actions are displayed in the header toolbar (see Object Page Elements [page 1625]).
- In *Edit* or *Create* mode, the footer bar contains the relevant actions, for example, *Save* and *Cancel* (see figure below).

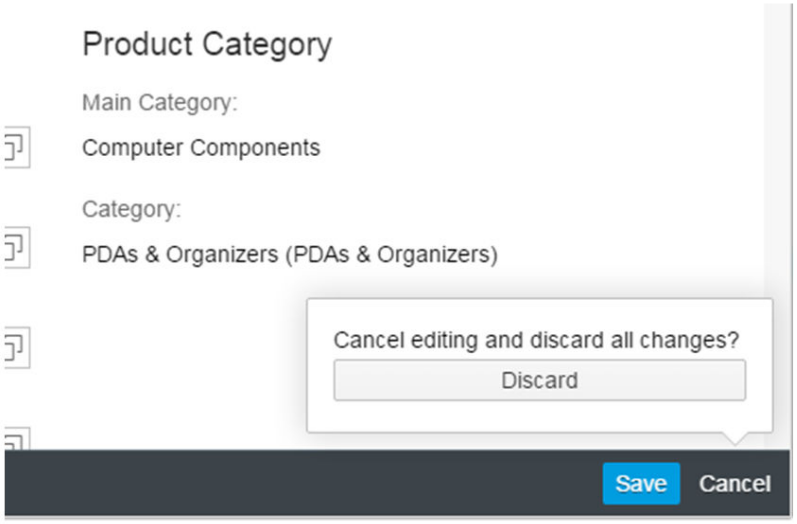


Figure 275: Object Page: Action Triggered from Footer Bar (Edit Mode)

> **i** Note
>
> In non-draft applications, users can choose the *Save and Edit* button to save the current changes. The object page stays in edit mode so that they can continue editing. You can enable this feature via SAPUI5 Visual Editor. For more information, see Adapting the UI: List Report and Object Page [page 1860].
>
> If this feature is enabled, you have to make following changes:
>
> - Change the type of the *Save and Edit* button to `Emphasized` and of the *Save* button to `Transparent`. You can do so by changing the `Type` property of both buttons in SAPUI5 Visual Editor.
> - Place the *Save and Edit* button as the first button in the footer and the *Save* button as the second button, using SAPUI5 Visual Editor.

Setting up the Object Page Header

You have various options for defining the object page header.

The object page supports the standard header and the dynamic header. To ensure consistency across all floorplans and to provide more flexibility, it is mandatory to use the dynamic header. See also Object Page Headers [page 2488].

> Sample Code

```
"sap.ui.generic.app": {
                "_version": "1.3.0",
                "settings": {
                        "forceGlobalRefresh": false,
                        "objectPageHeaderType": "Dynamic",
```

```
                    "showDraftToggle": false
                },
```

The object page header display is determined by the following vocabularies:

- `com.sap.vocabularies.UI.v1.HeaderInfo/Title/Value` determines the object title.
- `com.sap.vocabularies.UI.v1.HeaderInfo/Description/Value` determines the subtitle.
- `com.sap.vocabularies.UI.v1.HeaderInfo/ImageUrl` determines the image.
- `com.sap.vocabularies.UI.v1.HeaderInfo/TypeName` is used as the text for the link that navigates back to the list report.
- `com.sap.vocabularies.UI.v1.HeaderInfo/TypeImageUrl` determines the icon.
- `com.sap.vocabularies.UI.v1.HeaderInfo/Initials` determines the initials

Main Elements

This figure shows how to set up the following basic elements for your object page header in your annotations:

Label in Figure	Element
1	Title (Object Type)
2	Image of the object instance
3	Language-dependent product text in SAP back-end systems
4	Product title in SAP back-end systems

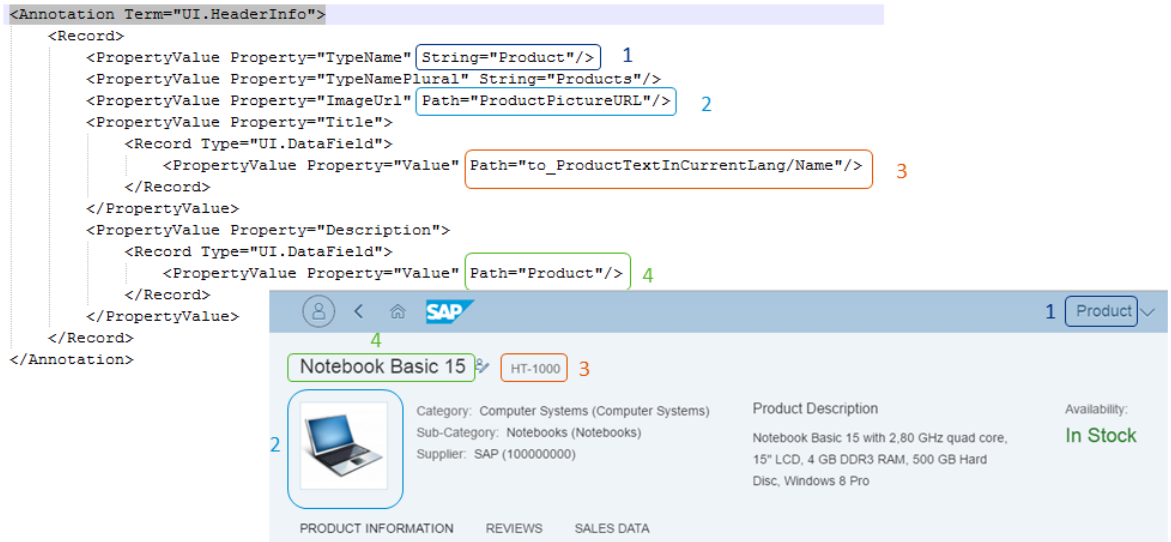


Figure 276: Object Page Header

Sample Code

This sample code is a selectable version of the code shown above for setting up the main elements on the object page header.

```
<Annotation Term="UI.HeaderInfo">
    <Record>
        <PropertyValue Property="TypeName" String="Product"/>
        <PropertyValue Property="TypeNamePlural" String="Products"/>
        <PropertyValue Property="ImageUrl" Path="ProductPictureURL"/>
        <PropertyValue Property="Title"
            <Record Type="UI.DataField">
                <PropertyValue Property="Value"
Path="to_ProductTextInCurrentLang/Name"/>
            </Record>
        </PropertyValue>
        PropertyValue Property="Description">
            <Record Type="UI.DataField">
                <PropertyValue Property="Value" Path="Product"/>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

For information on display options for a object, see Using Images, Initials, and Icons [page 1618].

Adapting the Object Page Title and Subtitle

You can use annotations to adapt the object page title and subtitle.

Figure 277: Object Page Title and Subtitle

To define or change the object page title and subtitle, adapt the OData annotations:

Sample Code

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
            <Annotation Term="UI.HeaderInfo">
                        <Record>
                                <PropertyValue Property="Title">
```

```
                                        <Record Type="UI.DataField">
                                              <PropertyValue
Property="Value" Path="to_ProductTextInOriginalLang/Name"/>
                                        </Record>
                                  </PropertyValue>
                                  <PropertyValue Property="Description">
                                        <Record Type="UI.DataField">
                                              <PropertyValue
Property="Value" Path="ProductForEdit"/>
                                        </Record>
                                  </PropertyValue>
                            </Record>
               </Annotation>
</Annotations>
```

Header Facets

You can include various types of header facets in your object page header, for example, displaying contact data or a rating indicator.

You can use the annotation term `UI.HeaderFacets` to define which information is displayed in the header. For example, you can define content blocks with several fields and contact details. You can also apply content blocks for the shipping address, price, and category.

The last header section is filled with the data points of your metadata document (`com.sap.vocabularies.UI.v1.DataPoint`). The title and value are taken into account.

Available header facets:

- Plain Text Facet [page 1669]
- Contact Facet [page 1671]
- Smart Micro Chart Facet [page 1673]
- Rating Indicator Facet [page 1683]
- Header Field Group [page 1681]
- Data Points [page 1682]
- Form Facet [page 1690]
- Address Facet in the Object Page Header [page 1692]

Enabling Simple Header Facets

If you want to display more content in your header facet, you can enable the simple header facet.

To do so, you must add the `simpleHeaderFacets:true` flag manually under `sap.ui.generic.app` → `pages (Object page)` → `component` → `settings` in the app's manifest.json file.

> **i** Note
>
> The simple header facet does not support the use of complex data points and `DataFieldForAnnotation`, such as ratings, progress indicators, or charts.

The following figure shows a sample simple header facet:

Figure 278: Simple Header Facet

Sample Code

Enabling simple header facet

```
"sap.ui.generic.app": {
        "_version": "1.2.0",

        "pages": [{
                "entitySet": "STTA_C_MP_Product",
                "component": {
                        "name": "sap.suite.ui.generic.template.ListReport",
                        "list": true,
                        "settings": {
                                "gridTable": false,
                                "multiSelect": true,
                                "smartVariantManagement": true
                        }
                },
                "pages": [{
                        "entitySet": "STTA_C_MP_Product",
                        "component": {
                                "name":
"sap.suite.ui.generic.template.ObjectPage",
                                "settings": {
                                        "showRelatedApps": true,
                                        "gridTable": false,
                                        "editableHeaderContent": true,
                                        "simpleHeaderFacets":true, // This
Enables Simple Header Facet on the Object Page
                                }
                        },
                        "pages": [{
                                "navigationProperty": "to_ProductText",
                                "entitySet": "STTA_C_MP_ProductText",
                                "component": {
                                        "name":
"sap.suite.ui.generic.template.ObjectPage"
                                }
                        }]
                }]
        }]
},
```

Plain Text Facet

You can add a plain text facet to the header area. This type of facet is suitable if you wish to add a single field or block of text with a title, such as a description.

A plain text facet shows the label property of the `UI.ReferenceFacet` as the title. It also shows the value property of the `DataField` complex type of the `FieldGroup` annotation as the description.

To add a plain text facet, use the `UI.HeaderFacet` term and include the `UI.ReferenceFacet` complex type, and then reference the `FieldGroup` annotation.

This is displayed as shown below within the object page header:


```
<Record Type="UI.ReferenceFacet">                                    1
    <PropertyValue Property="Label" String="{@i18n>@ProductDescription}"/>
    <PropertyValue Property="Target" AnnotationPath="to_ProductTextInCurrentLang/@UI.FieldGroup#PlainText"/>
</Record> </Collection></Annotation>
```

Figure 279: Plain Text Facet in Object Header

Code Samples

The following code samples show an example of how to create your annotations for a plain text header:

UI.ReferenceFacet

```
<Record Type="UI.ReferenceFacet">
    <PropertyValue Property="Label" String="{@i18n>@ProductDescription}"/>
    <PropertyValue Property="Target" AnnotationPath="to_ProductTextInCurrentLang/
@UI.FieldGroup#PlainText"/>
</Record> </Collection>
```

UI.FieldGroup

```
<Annotation Term="UI.FieldGroup" Qualifier="PlainText">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="Description"/>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

UI.MultilineText

In addition, you must include a property annotation to indicate that this property contains a multiline text, as shown below:

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductTextType/Description">
    <Annotation Term="UI.MultiLineText"/>
</Annotations>
```

CDS Annotations

If desired, you can set this up using a CDS annotation, as shown below:

CDS Annotation Definition

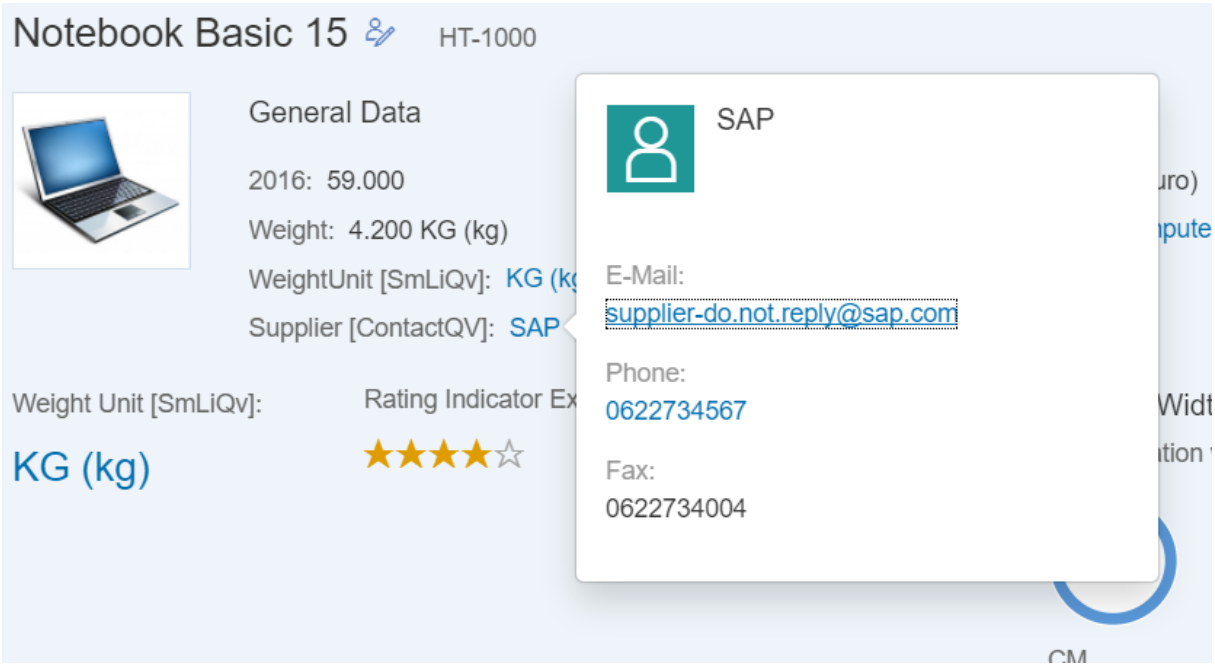
```
//@Scope: #ELEMENT
multiLineText: Boolean default true;
```

CDS Source

```
@UI.multiLineText
Description: String;
```

Contact Facet

Using the `@Communication.Contact` annotation, you can enable a quick view contact within the header form facet, as shown below:



Code Samples

In the example, the `UI.DataFieldForAnnotation` points to a contact annotation on a different entity, which has a 1:1 relation to the root entity, and so one contact is displayed in the header. The label, for example, *Supplier*, is taken from `UI.DataFieldForAnnotation`, and the value, for example *SAP*, is the `fn` property of the contact annotation.

Sample Code

Contact link

```
<Annotation Term="UI.FieldGroup" Qualifier="GeneralInformationForHeader">
                              <Record>
                                    <PropertyValue Property="Data">
                                          <Collection>
                                                <Record Type="UI.DataField">
                                                      <PropertyValue
Property="Value" Path="to_StockAvailability/Quantity"/>
                                                      <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/Low"/>
                                                </Record>
                                                <Record Type="UI.DataField">
                                                      <PropertyValue
Property="Value" Path="Weight"/>
                                                      <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                                                </Record>
                                                <Record Type="UI.DataField">
                                                      <PropertyValue
Property="Label" String="WeightUnit [SmLiQv]"/>
                                                      <PropertyValue
Property="Value" Path="WeightUnit"/>
                                                      <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                                                </Record>
                                                <Record
Type="UI.DataFieldForAnnotation">
                                                      <PropertyValue
Property="Label" String="Supplier [ContactQV]"/>
                                                      <PropertyValue
Property="Target" AnnotationPath="to_Supplier/@Communication.Contact"/>
                                                      <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                                                </Record>
                                          </Collection>
                                    </PropertyValue>
                                    <PropertyValue Property="Label"
String="{@i18n>@GeneralInfoFieldGroupLabel}"/>
                              </Record>
                        </Annotation>
```

Sample Code

Popover

```
<Annotation Term="Communication.Contact">
                           <Record>
                                <PropertyValue Property="fn"
Path="CompanyName"/>
                                <PropertyValue Property="email">
                                     <Collection>
                                          <Record>
                                                <PropertyValue Property="type"
EnumMember="Communication.ContactInformationType/work"/>
                                                <PropertyValue
Property="address" Path="EmailAddress"/>
                                          </Record>
                                     </Collection>
                                </PropertyValue>
                                <PropertyValue Property="tel">
                                     <Collection>
                                          <Record>
```

```
                                                    <PropertyValue Property="type"
EnumMember="Communication.PhoneType/preferred Communication.PhoneType/work"/>
                                                    <PropertyValue Property="uri"
Path="PhoneNumber"/>
                                               </Record>
                                               <Record>
                                                    <PropertyValue Property="type"
EnumMember="Communication.PhoneType/fax"/>
                                                    <PropertyValue Property="uri"
Path="FaxNumber"/>
                                               </Record>
                                          </Collection>
                                     </PropertyValue>
                                </Record>
                           </Annotation>
```

More Information

For more information about the `Communication.Contact` annotation, see Adding a Contact Facet [page 1703].

Smart Micro Chart Facet

You can add a `SmartMicroChart` control to a facet within the header area on the object page.

A smart micro chart facet contains a title, subtitle, `SmartMicroChart` control, and a footer. The `SmartMicroChart` control supports the following micro charts in the object page header:

- Smart area micro chart
- Smart bullet micro chart
- Smart radial micro chart
- Smart column micro chart
- Smart line micro chart
- Smart harvey micro chart
- Smart stacked bar micro chart

To add a smart micro chart facet, in the local annotations file, use a `UI.HeaderFacets` term and the complex type `UI.ReferenceFacet` and reference the `UI.Chart` as shown in the sample code below.

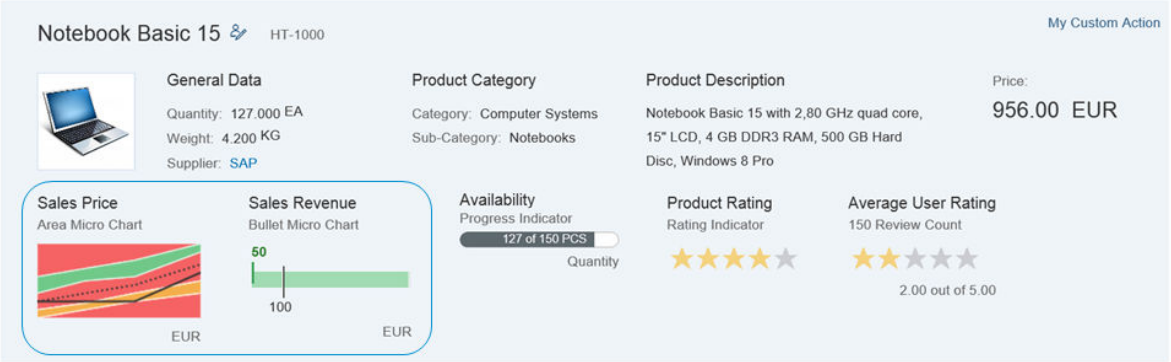


Figure 280: Object Page Header

Code Samples

UI.HeaderFacets and UI.ReferenceFacet

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
    <Annotation Term="UI.HeaderFacets">
        <Collection>
            <Record Type="UI.ReferenceFacet">
                <PropertyValue Property="Target"
AnnotationPath="to_ProductSalesPrice/@UI.Chart"/>
            </Record>
        </Collection>
    </Annotation>
</Annotations>
```

UI.Chart Annotations

The `UI.Chart Title` property is used for the title.

The `UI.Chart Description` property is used for the subtitle.

Smart Area Micro Chart

```
<Annotation Term="UI.Chart">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Sales Price" />
        <PropertyValue Property="Description" String="Area Micro Chart" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Area" />
        <PropertyValue Property="Dimensions">
            <Collection>
                <PropertyPath>PriceDay</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Measures">
            <Collection>
                <PropertyPath>AreaChartPrice</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="AreaChartPrice" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
```

```
                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#AreaChartPrice" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Smart Bullet Micro Chart

```
<Annotation Term="UI.Chart">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Sales Revenue" />
        <PropertyValue Property="Description" String="Bullet Micro Chart" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Bullet" />
        <PropertyValue Property="Measures">
            <Collection>
                <PropertyPath>BulletChartRevenue</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="BulletChartRevenue" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#BulletChartRevenue" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Smart Radial Micro Chart

Sample Code

```
<Annotation Term="UI.Chart" Qualifier="SpecificationWidthRadialChart">
                    <Record Type="UI.ChartDefinitionType">
                        <PropertyValue Property="Title" String="Product Width
Specification"/>
                        <PropertyValue Property="Description" String="No
navigation with qualifier"/>
                        <PropertyValue Property="ChartType"
EnumMember="UI.ChartType/Donut"/>
                        <PropertyValue Property="Measures">
                            <Collection>
                                <PropertyPath>Width</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="MeasureAttributes">
                            <Collection>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="Width"/>
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1"/>
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Width"/>
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
```

Smart Column Micro Chart

Sample Code

```
<Annotation Term="UI.Chart" Qualifier="SpecificationWidthColumnChart">
                        <Record Type="UI.ChartDefinitionType">
                        <PropertyValue Property="Title" String="Product Width
Specification Column Chart"/>
                        <PropertyValue Property="Description"
String="Describe Column Chart"/>
                        <PropertyValue Property="ChartType"
EnumMember="UI.ChartType/Column"/>
                        <PropertyValue Property="Criticality"
Path="criticalityValue"/>
                        <PropertyValue Property="Measures">
                            <Collection>
                                <PropertyPath>Width</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="Dimensions">
                            <Collection>
                                <PropertyPath>Day</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="MeasureAttributes">
                            <Collection>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="Width"/>
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1"/>
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Width"/>
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
```

Smart Line Micro Chart

Sample Code

```
<Annotation Term="UI.Chart" Qualifier="SpecificationWidthtLineChart">
                        <Record Type="UI.ChartDefinitionType">
                        <PropertyValue Property="Title" String="Product Width
Specification Line Chart"/>
                        <PropertyValue Property="Description"
String="Describe Line Chart"/>
                        <PropertyValue Property="ChartType"
EnumMember="UI.ChartType/Line"/>
                        <PropertyValue Property="Measures">
                            <Collection>
                                <PropertyPath>Width</PropertyPath>
                                <PropertyPath>Depth</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="Dimensions">
                            <Collection>
                                <PropertyPath>Day</PropertyPath>
                                <PropertyPath>Day</PropertyPath>
                            </Collection>
                        </PropertyValue>
```

```
                        <PropertyValue Property="MeasureAttributes">
                            <Collection>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="Width"/>
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis2"/>
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Width"/>
                                </Record>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="Depth"/>
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis2"/>
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Depth"/>
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
```

Smart Harvey Micro Chart

Sample Code

```
<Annotation Term="UI.Chart" Qualifier="SpecificationWidthHarveyChart">
                    <Record>
                        <PropertyValue EnumMember="UI.ChartType/Pie"
Property="ChartType" />
                        <PropertyValue Property="Title" String="Sold to
Produced Harvey Chart" />
                        <PropertyValue Property="Description" String="Harvey
Chart" />
                        <PropertyValue Property="Measures">
                            <Collection>
                                <PropertyPath>Sold</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="MeasureAttributes">
                            <Collection>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint" />
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
```

Smart Stacked Bar Micro Chart

Sample Code

```
<Annotation Term="UI.Chart" Qualifier="SalesPriceStackedBarChart">
                    <Record Type="UI.ChartDefinitionType">
                        <PropertyValue Property="Title" String="Sales Price"/>
                        <PropertyValue Property="Description" String="Stacked
BarChart"/>
                        <PropertyValue Property="ChartType"
EnumMember="UI.ChartType/BarStacked"/>
                        <PropertyValue Property="Dimensions">
                            <Collection>
```

```
                                <PropertyPath>Revenue</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="Measures">
                            <Collection>
                                <PropertyPath>Width</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="MeasureAttributes">
                            <Collection>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="Width"/>
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1"/>
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Width"/>
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
```

i Note

The template does not currently support the use of navigation properties in the `UI.Chart` term for the smart micro chart (see example below).

```
<Annotation Term="UI.Chart" Qualifier="ChartQualifier"> Not supported
        <Record Type="UI.ChartDefinitionType">
                <PropertyValue Property="Title" String="Gross Sales Revenue"/>
                <PropertyValue Property="Description" String="With navigation & criticality"/>
                <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Bullet"/>
                <PropertyValue Property="Measures">
                        <Collection>
                                <PropertyPath>to_SalesData/Revenue</PropertyPath>  Not supported
                        </Collection>
                </PropertyValue>
        </Record>
</Annotation>
```

Figure 281: Navigation Property

UI.DataPoint Annotation

The `DataPoint` property of the `MeasureAttributes` of the `Chart` annotation has to point to the `UI.DataPoint` annotation.

The smart micro chart supports both the `Criticality` and `CriticalityCalculation` properties of a `UI.DataPoint`. For an example of how to use the `CriticalityCalculation`, see the smart area micro chart annotation example. For an example of how to use the `Criticality` property, see the smart bullet micro chart annotation example.

i Note

Although the `Title` for the `UI.DataPoint` is mandatory, the smart micro chart doesn't use it.

Smart Area Micro Chart

```
<Annotation Term="UI.DataPoint" Qualifier="AreaChartPrice">
```

```
    <Record>
        <PropertyValue Property="Title" String="Sales Price" />
        <PropertyValue Property="Value" Path="AreaChartPrice" />
        <PropertyValue Property="TargetValue" Path="TargetPrice" />
        <PropertyValue Property="CriticalityCalculation">
            <Record>
                <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Target" />
                <PropertyValue Property="DeviationRangeHighValue"
Path="DeviationUpperBoundPrice" />
                <PropertyValue Property="DeviationRangeLowValue"
Path="DeviationLowerBoundPrice" />
                <PropertyValue Property="ToleranceRangeHighValue"
Path="ToleranceUpperBoundPrice" />
                <PropertyValue Property="ToleranceRangeLowValue"
Path="ToleranceLowerBoundPrice" />
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

Smart Bullet Micro Chart

```
<Annotation Term="UI.DataPoint" Qualifier="BulletChartRevenue">
    <Record>
        <PropertyValue Property="Title" String="Sales Revenue" />
        <PropertyValue Property="Value" Path="BulletChartRevenue" />
        <PropertyValue Property="TargetValue" Path="TargetRevenue" />
        <PropertyValue Property="ForecastValue" Path="ForecastRevenue" />
        <PropertyValue Property="MinimumValue" Decimal="100" />
        <PropertyValue Property="MaximumValue" Decimal="300" />
        <PropertyValue Property="Criticality" Path="Criticality" />
    </Record>
</Annotation>
```

Smart Radial Micro Chart and Smart Column Micro Chart

Sample Code

```
 <Annotation Term="UI.DataPoint" Qualifier="Width">
                     <Record>
                         <PropertyValue Property="Value" Path="Width"/>
                         <PropertyValue Path="Day1" Property="Title" />
                         <PropertyValue Property="Description" String="Bullet
 Micro Chart"/>
                         <PropertyValue Property="TargetValue" Path="Weight"/>
                         <PropertyValue Property="ForecastValue"
 Path="Height"/>
                         <PropertyValue Property="MinimumValue" Decimal="0"/>
                         <PropertyValue Property="MaximumValue" Decimal="100"/>
                         <PropertyValue Property="Criticality"
 Path="criticalityValue"/>
                     </Record>
                 </Annotation>
```

Smart Line Micro Chart

For the width Datapoint annotation, see the UI.DataPoint Annotation for Smart Radial Micro Chart.

Depth Datapoint annotation:

Sample Code

```
 <Annotation Term="UI.DataPoint" Qualifier="Depth">
```

```
                    <Record>
                        <PropertyValue Property="Value" Path="Depth"/>
                        <PropertyValue Path="Day2" Property="Title" />
                        <PropertyValue Property="MinimumValue" Decimal="0"/>
                        <PropertyValue Property="MaximumValue" Decimal="200"/>
                        <PropertyValue Property="Criticality"
Path="criticalityValue"/>

                    </Record>
```

Smart Harvey Micro Chart

Sample Code

```
<Annotation Term="UI.DataPoint">
                    <Record>
                        <PropertyValue Path="Sold" Property="Value" />
                        <PropertyValue Path="Produced"
Property="MaximumValue" />
                        <PropertyValue Path="criticalityValue"
Property="Criticality" />
                    </Record>
                </Annotation>
```

Smart Stacked Bar Micro Chart

For the width Datapoint annotation, see the UI.DataPoint annotation for the Smart Radial Micro Chart.

Note

The following must all point to the same property in the `entityType`:

- `Measures` property of the `Chart` annotation
- `Measure` property of the `MeasureAttributes` property of the `Chart` annotation
- `Value` property of the `DataPoint` annotation

Unit of Measure Annotations

The `Unit of Measure` is used for the footer of the smart micro chart. An annotation for the `Unit of Measure` is included in the example below. The example uses the `Measures.ISOCurrency` term and it is applied to the entity type property that is used as the value property of the `UI.DataPoint`.

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductSalesPriceType/AreaChartPrice">>
    <Annotation Term="Measures.ISOCurrency" Path="Currency"/>
</Annotations>
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductSalesRevenueType/BulletChartRevenue">>
                    <Annotation Term="Measures.ISOCurrency" Path="Currency"/>
                </Annotations>
            <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductSalesRevenueType/
BulletChartMonthRevenue">>
                    <Annotation Term="Measures.ISOCurrency" Path="Currency"/>
                </Annotations>
                <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductSalesRevenueType/BulletChartNetRevenue">
                    <Annotation Term="Measures.ISOCurrency" Path="Currency"/>
                </Annotations>
                <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductType/SpecificationWidthHarveyChart">
```

```
                    <Annotation Term="Measures.ISOCurrency" Path="Currency"/>
                </Annotations>
                <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductType/Sold">
                    <Annotation Term="Measures.ISOCurrency" Path="WeightUnit" />
            </Annotations>
```

Header Field Group

A field group defines the fields displayed within a specific reference facet. For example, the figure below shows the following fields within the *General Information* section (created with a header facet):

- *Quantity*
- *Weight*
- *Supplier*

Figure 282: Object Page: Header Field Group

Code Sample

In the example below, the first field group refers to general information.

```
<Annotation Term="UI.FieldGroup" Qualifier="GeneralInformationForHeader">
        <Record>
        <PropertyValue Property="Data">
        <Collection>
        <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="to_StockAvailability/Quantity"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/Low"/>
        </Record>
        <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="Weight"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
        </Record>
        <Record Type="UI.DataFieldForAnnotation">
        <PropertyValue Property="Label" String="Supplier"/>
        <PropertyValue Property="Target" AnnotationPath="to_Supplier/
@Communication.Contact"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
        </Record>
        </Collection>
        </PropertyValue>
        <PropertyValue Property="Label" String="{@i18n>@GeneralInfoFacetLabel}"/>
        </Record>
        </Annotation>
```

Data Points

A data point represents a single point of data. It is typically a number but can also be textual, for example, a status value.

The image below shows the data points *Product Category* and *Price* in the object page header.

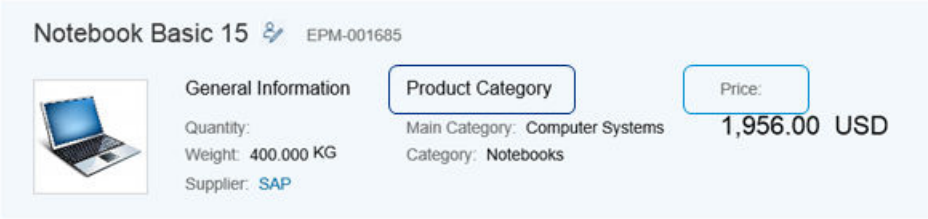


Figure 283: Object Page: DataPoints

This video shows the step-by-step procedure for adding a data point header to an object page:

Code Samples

UI.Reference Facet

If you add a `UI.ReferenceFacet` that points to `UI.DataPoint`, the title and value of the `UI.DataPoint` will be rendered.

```
<Annotation Term="UI.HeaderFacets">
    <Collection>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="{@i18n>@TechnicalData}"/>
            <PropertyValue AnnotationPath="@UI.FieldGroup#TechnicalData"
Property="Target"/>
        </Record>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue AnnotationPath="@UI.DataPoint#Price"
Property="Target"/>
        </Record>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue AnnotationPath="@UI.DataPoint#ProductCategory"
Property="Target"/>
        </Record>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="Employee"/>
            <PropertyValue AnnotationPath="to_Supplier/@Communication.Contact"
Property="Target"/>
        </Record>
</Annotation>
```

UI.DataPoint

Each `UI.DataPoint` annotation term must point to a qualifier, as shown below:

```
<Annotation Term="UI.DataPoint" Qualifier="Price">
    <Record>
```

```
        <PropertyValue Property="Value" Path="Price"/>
        <PropertyValue Property="Title" String="Price"/>
    </Record>
</Annotation>
<Annotation Term="UI.DataPoint" Qualifier="ProductCategory">
    <Record>
        <PropertyValue Property="Value" Path="ProductCategory"/>
        <PropertyValue Property="Title" String="Category"/>
    </Record>
</Annotation>
```

Rating Indicator Facet

You can add a read-only rating indicator to the object page header.

The rating indicator allows you to visually represent the rating types described below.

Aggregated Rating

This rating shows an average of all ratings recorded for the object and is displayed as shown below in the object page header (*Display* mode):

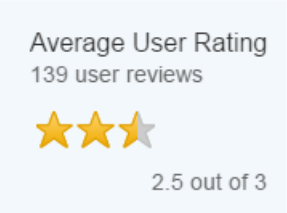


Figure 284: Aggregated Rating Indicator

When the rating indicator shows an aggregated rating, it contains the following elements:

- Title
- Subtitle, displaying the total number of ratings

  > **i Note**
  >
  > You have the option to specify a text in the subtitle to be more descriptive (example: *139 user reviews*, where "139" is the number of reviews received, and "user reviews" is the text added). For information, see the code sample below.

- `Rating` control, displaying the visual representation of the rating (stars)
- Footer text, displaying the calculated average of all ratings

> **i Note**
>
> If you wish to use an aggregated rating, you must use local annotations (as opposed to CDS annotations). For information, see the code sample below.

Non-aggregated Rating

This rating shows a single rating, such the user's own rating for the object. In the examples below, the non-aggregated rating is used to display the user's own rating. This rating type is displayed as shown below:

Figure 285: Non-aggregated Rating Indicator

When the rating indicator shows a non-aggregated rating, it contains the following elements:

- Title
- Subtitle
- `Rating` control, displaying the visual representation of the rating (stars)

> **i Note**
>
> There is no footer for this rating type.
>
> If you wish to use a non-aggregated rating, you can use either local annotations or CDS annotations. See the code samples below.

Rating Indicator in Edit Mode

When the object page is in *Edit* mode, the rating indicator moves into the header facet and appears as shown below with the title only. For an aggregated rating, the number of ratings is shown in parentheses after the stars. Note that the rating indicator is still read-only in this mode.

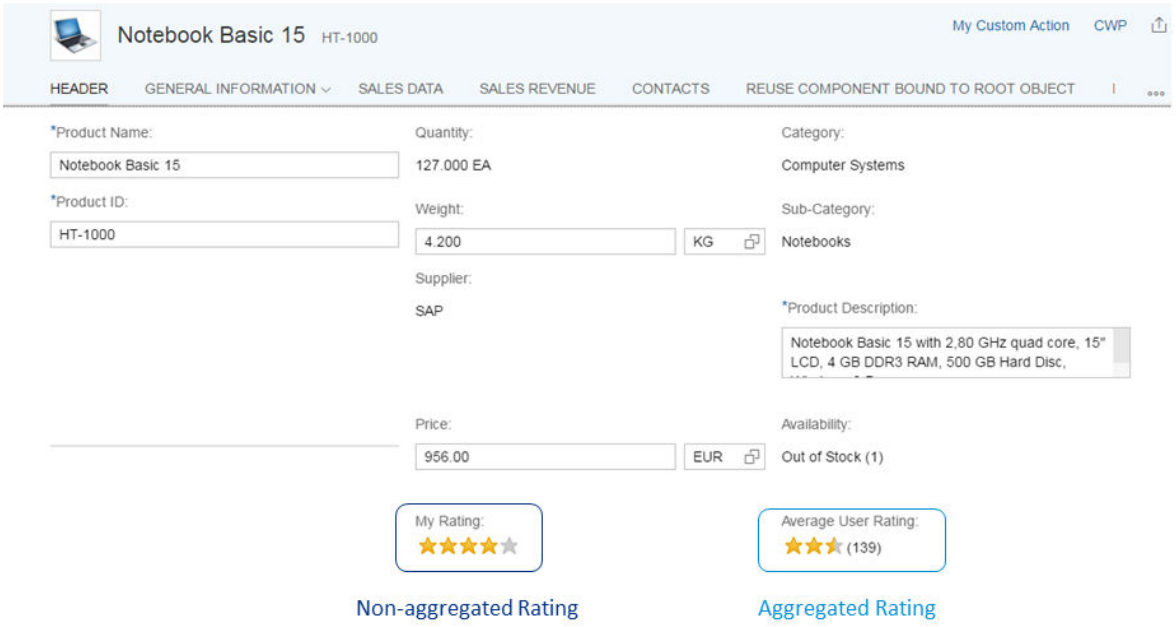


Figure 286: Rating Indicator in Edit Mode

Code Samples

To add a rating indicator facet to the object page header, use a `UI.ReferenceFacet` that points to a `UI.DataPoint` with `Rating` as the `UI.VisualizationType` as shown in the code sample below.

The rating indicator uses the values of the `UI.DataPoint`, which contains the path to the field in the back-end system that provides the rating value.

Set the Maximum Number of Stars

The maximum number of stars (`TargetValue`) can be set in one of the following ways:

- Specified in the annotation, as shown in the sample code for the aggregated rating
- Determined by a path to a specific field in the back-end system, as shown in the sample code for the non-aggregated rating

Change the Subtitle

The subtitle is set differently for the aggregated and non-aggregated ratings.

To render a subtitle for an aggregated rating, set the `SampleSize` property for the term `UI.DataPoint`. The `SampleSize` property value (for example, *139*) is then concatenated with a text (for example, *user reviews*). You can change this text by annotating the `SampleSize` with `Common.Label`, as shown in the sample code for the aggregated rating. Otherwise, the default text (*ratings*) is used.

To render a subtitle for a non-aggregated rating, the `Description` property needs to be set for the term `UI.DataPoint`, as shown in the sample code for the non-aggregated rating.

Aggregated Rating

```
<Record Type="UI.ReferenceFacet">
    <PropertyValue Property="Target" AnnotationPath="to_ProductRating/
@UI.DataPoint#Aggregated"/>
```

```
</Record>
<Annotation Term="UI.DataPoint" Qualifier="Aggregated">
   <!--aggregated rating -->
   <Record>
      <PropertyValue Property="Title" String="{@i18n>@ProductAverageRating}"/>
      <PropertyValue Property="Value" Path="AverageRating"/>
      <PropertyValue Property="TargetValue" Int="3"/>
      <PropertyValue Property="Visualization" EnumMember="UI.VisualizationType/
Rating"/>
      <PropertyValue Property="SampleSize" Path="ReviewCount"/>
   </Record>
</Annotation>

<!--replace default subtitle text with custom text -->

<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductRatingType/ReviewCount">
    <Annotation Term="Common.Label" String="user reviews"/>
</Annotations>
```

Non-aggregated Rating

```
<Record Type="UI.ReferenceFacet">
    <PropertyValue Property="Target" AnnotationPath="to_ProductRating/
@UI.DataPoint#NonAggregated"/>
</Record>
<!-- non aggregated rating -->
<Annotation Term="UI.DataPoint" Qualifier="NonAggregated">
   <Record>
      <PropertyValue Property="Title" String="{@i18n>@ProductUserRating}"/>
      <PropertyValue Property="Description" String="@i18n>@MyRating}"/>
      <PropertyValue Property="Value" Path="Rating"/>
      <PropertyValue Property="TargetValue" Path="MaxRating"/>
      <PropertyValue Property="Visualization" EnumMember="UI.VisualizationType/
Rating"/>
   </Record>
</Annotation>
```

CDS Annotation Source (Non-aggregated Rating Only)

```
@UI.dataPoint: {
    title:'Product Rating',
    description: 'Rating Indicator',
    targetValueElement: 'MaxRating',
    visualization: #RATING
}

@EndUserText.label: 'Rating'
ProductRating.Rating     as Rating
```

Progress Indicator Facet

You can add a progress indicator to a header facet on the object page.

The progress indicator allows you to visually represent the level of completion of a goal or target, such as a project's progress, sales progress for the current year's goal, the development stage of a product, stock availability, and so on. The figure below shows a progress indicator within the object page header.

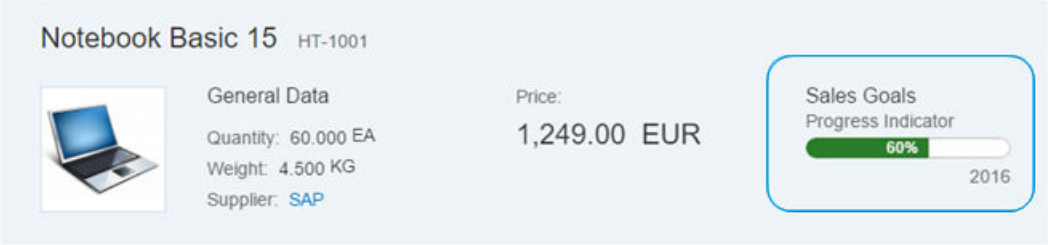


Figure 287: Progress Indicator in Object Page Header

As shown below, progress can be expressed either as a percentage or in absolute numbers (for example, "8 of 10"), and it can include a unit of measure, such as PC, GB, and so on.

Figure 288: Progress Indicator: Percent or Number

The progress indicator in the header facet is made up of sections to include a title, subtitle, and footer:

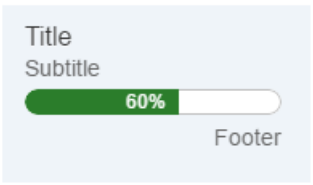


Figure 289: Sections of Progress Indicator

Code Samples

UI.ReferenceFacet

To display the progress indicator in the object page header, add a record to the `UI.HeaderFacets` collection. This record must be of type `UI.ReferenceFacet` and contain an `AnnotationPath` that points to a `UI.DataPoint` with the visualization type `Progress`. The properties for the data point can be included in either the `entityType` being annotated (`Target`) or in another `entityType` different from the `Target`, in which case the `AnnotationPath` contains a navigation path as shown below.

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProjectType">
   <Annotation Term="UI.HeaderFacets">
      <Collection>
```

```
         <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Target"
            AnnotationPath="to_ProgressType/@UI.DataPoint#Progress"/>
         </Record>
      </Collection>
   </Annotation>
</Annotations>
```

> **i Note**
>
> In the example above, "UI" is an alias for the `com.sap.vocabularies.UI.v1` vocabulary.

CDS: UI.DataPoint

In CDS, annotate the `EntityType` containing the properties required for the data point as shown below.

```
@UI.dataPoint: {
    title:'{@i18n>Title}',
    description: {@i18n>SubTitle},
    targetValue: 150,
    criticality: 'Criticality',
    visualization: #PROGRESS
}
ProjectProgress.Progress
```

> **i Note**
>
> - The data point annotation is for a `Property` even if the UI vocabulary specifies an `EntityType` as the `Target`.
> - The property name will be used as the `Qualifier` in the resulting (generated) annotation.

The generated annotation will be similar to the example below:

UI.DataPoint

Annotate the `entityType` containing the properties required for the data point as shown below.

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProgressType">
   <Annotation Term="UI.DataPoint" Qualifier="Progress">
      <Record>
         <PropertyValue Property="Title" String="{@i18n>Title}"/>
         <PropertyValue Property="Description" String="{@i18n>SubTitle}"/>
         <PropertyValue Property="Value" Path="Progress"/>
         <PropertyValue Property="TargetValue" Decimal="150"/>
         <PropertyValue Property="Criticality" Path="Criticality "/>
         <PropertyValue Property="Visualization"
EnumMember="UI.VisualizationType/Progress"/>
      </Record>
   </Annotation>
</Annotations>
```

UoM and Common.Label

Additionally, for the unit of measure (UoM) and the footer, annotate the `entityType`'s property so that it includes the path of the `Value` property for the data point. For example, in the code sample above, the path for the data point `Value` property is `Value`, which is then used to annotate the `entityType`. In the examples below, this is represented by `<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProgressType/Value">`.

Note that the unit of measure can be annotated with `Unit` or `ISOCurrency` as shown below. For the footer, the term `Common.Label` needs to be applied.

Progress Indicator: UoM.Unit

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProgressType/Value">
   <Annotation Term="UoM.Unit" Path="UoM">
   <Annotation Term="Common.Label" String="{@i81n>Footer}">
</Annotations>
```

Progress Indicator: UoM.ISOCurrency

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ ProgressType/Value">
   <Annotation Term="UoM.ISOCurrency" Path="UoM">
   <Annotation Term="Common.Label" Path="Footer”>
</Annotations>
```

> **i** Note
>
> In the examples above, `UoM` is an alias for the `Org.OData.Measures.V1` vocabulary.

Rendering Rules

- The `Value` and `Title` properties are mandatory. Without a value, the progress cannot be calculated. A title should always be provided for an object page header facet, as this is required by the `DataPoint` term in the UI vocabulary.
- The `TargetValue` property is mandatory when using a `UoM` that is not expressed as a percentage (for example, currency, CM, PC, and so on), or if no `UoM` is provided.
- The remaining properties - `Unit of Measure`, `Subtitle`, and `Footer` - are optional.
- If the value of the `Unit of Measure` property is "%", then the `Value` property will be used directly as a percentage.
- If the value of the `Unit of Measure` is not "%" or is not provided, then the progress will be calculated using the `Value` and `TargetValue` properties according to the formula `Progress = Value / TargetValue`.
- Additionally, the following checks will be done:
  - Division by zero will result in progress being zero (since it cannot be calculated).
  - Progress must be a value between 0 and 100.
    - If the progress is less than zero, then no color will appear in the progress bar.
    - If the progress is greater than 100, then the progress bar will be fully colored.
    - In both cases the actual value of the progress will be displayed in the bar as shown below.

Figure 290: Values in Progress Indicator

Key Value Facet

If you add a `UI.ReferenceFacet` that points to `UI.DataPoint`, the title and value of the `UI.DataPoint` are rendered as follows:

Sample Code

```
<Annotation Term="UI.DataPoint" Qualifier="ProductCategory">
      <Record>
            <PropertyValue Property="Value" Path="ProductCategory"/>
                  <PropertyValue Property="Title"
String="{@i18n>@ProductCategory}"/>
            </Record>
</Annotation>
```

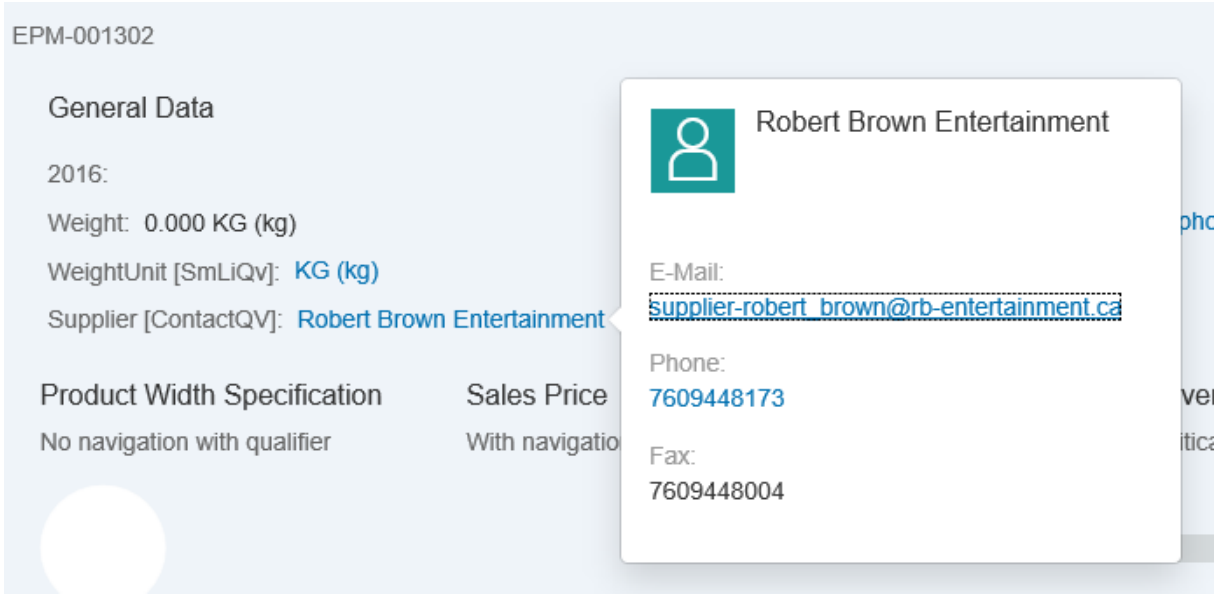
Form Facet

You can add a form facet to the object page header. To do so, add a `UI.ReferenceFacet` that points to `UI.FieldGroup` or `UI.Identification`. If you provide a label in the `UI.ReferenceFacet`, it is used as the form's title.

Sample Code

```
<Record Type="UI.ReferenceFacet">
    <PropertyValue Property="Label" String="{@i18n>@TechnicalData}"/>
    <PropertyValue AnnotationPath="@UI.FieldGroup#TechnicalData"
Property="Target"/>
</Record>
```

The header form facet supports the display of a contact with a quick view, as shown below:



The sample code below shows the `UI.FieldGroup`.

Sample Code

```
<Annotation Term="UI.FieldGroup" Qualifier="Test">
   <Record>
      <PropertyValue Property="Data">
         <Collection>
            <Record Type="UI.DataField">
               <PropertyValue Property="Value" Path="Product" />
               <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/
High" />
            </Record>
```

```
            <Record Type="UI.DataField">
               <PropertyValue Property="Value" Path="ProductCategory" />
               <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/
Medium" />
            </Record>
            <Record Type="UI.DataFieldForAnnotation">
               <PropertyValue Property="Label" String="Supplier" />
               <PropertyValue Property="Target" AnnotationPath="to_Supplier/
@Communication.Contact" />
            </Record>
            <Record Type="UI.DataField">
               <PropertyValue Property="Value" Path="Price" />
               <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/
Medium" />
            </Record>
            <Record Type="UI.DataFieldWithIntentBasedNavigation">
               <PropertyValue Property="Label" String="Weight (with IBN)" />
               <PropertyValue Property="Value" Path="Weight" />
               <PropertyValue Property="SemanticObject" String="EPMProduct" />
               <PropertyValue Property="Action" String="manage_st" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Label" String="Product Information" />
   </Record>
</Annotation>
```

In the sample code above, the third record is a `UI.DataFieldForAnnotation`, which, in this case, points to a contact annotation on a different entity that has a 1:1 relation to the root entity. The label is derived from the label in the `UI.DataFieldForAnnotation` and the value is the fn property of the contact annotation.

The last record is a `UI.DataFieldWithIntentBasedNavigation` to render the property value as a link, allowing for navigation to the semantic object.

> **i** Note
>
> Contacts on 1:n relations are not supported in the header.

Address Facet in the Object Page Header

If you add a `UI.ReferenceFacet` that points to an address annotation, an address facet is displayed in the object page header.

It shows the label of the `UI.ReferenceFacet` and, below, only the label property of the address annotation. This is why the label property needs to contain the whole formatted address, with \n for new lines.

> **i** Note
>
> Other properties of the address annotation are not interpreted and rendered.

Example value for the label property: "Mail Drop: TNE QB\n123 Main Street\nAny Town, CA 91921-1234\nU.S.A.". This is shown as follows:

Sample Code

```
<Record Type="UI.ReferenceFacet">
    <PropertyValue Property="Label" String="Shipping Address"/>
    <PropertyValue AnnotationPath="@Communication.Address" Property="Target"/>
</Record>
```

Enabling Actions in Object Page Header

You can enable generic actions in your object header.

All data fields with `com.sap.vocabularies.UI.v1.DataFieldForAction` are interpreted as actions. The system renders a button within the header displaying the text of the data field label.

Data fields for actions that you annotate with high importance are displayed to the left of the *Edit*/*Delete* buttons, and those without high importance are displayed to the right. In addition to the importance, you need to specify the action for the data field and implement the action handling in the OData service (DPC: `execute_action`), as shown below for the *Copy* button. See the code samples below for information on where to place the data fields for actions.

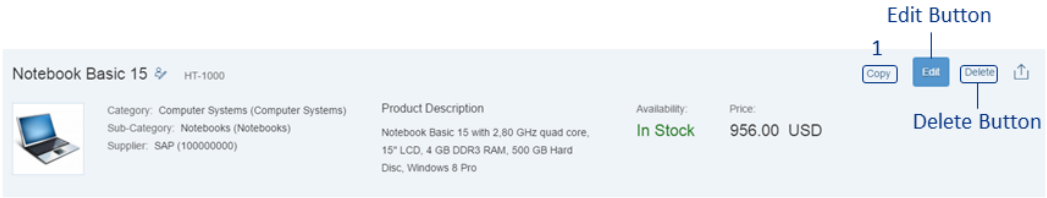


Figure 291: Object Page: Actions Defined for the EntitySet

Annotations for the *Copy* button:

```
<Annotation Term="UI.Identification">
      <Collection>
            <Record Type="UI.DataFieldForAction">
                  <PropertyValue Property="Label" String="Copy" />
                  <PropertyValue Property="Action"
                        String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
STTA_C_MP_ProductCopy" />
```

```
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/
High" />
                </Record>
...
      </Collection>
</Annotation>
```

Edit and Delete Buttons

The *Edit* and *Delete* buttons are displayed as shown above, if the following conditions are met:

- *Edit* button: `sap:updatable` is not set to `false`
- *Delete* button: `sap:deletable` is not set to `false`

> **i Note**
>
> If you want to specify conditions for deletion or updates (using the `deletable-path` or `updatable path` annotation), you need to ensure that you have not made the `sap:deletable` or `sap:updatable` setting in your annotations.

Show or Hide Edit and Delete Buttons (Using updatable-path and deletable-path Annotations)

You can choose to display or hide the *Delete* or *Edit* button on the object page based on certain conditions in your back-end system. For example, you may wish to disable editing or deletion for a sales order that has already been paid.

Within your annotation, you set the `deletable-path` (for the *Delete* button) or `updatable-path` (for the *Edit* button) to point to a particular property of an object (entity) in the back-end system that is either `true` or `false`. If the value of this property is `true`, the *Delete* or *Edit* button is displayed; if it is `false`, it is hidden.

> **i Note**
>
> - This is done in conjunction with the `deletable path` annotation for the list report.
> - If you want to specify conditions for deletion or updates (using the `deletable-path` or `updatable-path` annotation), you need to ensure that you have not made the `sap:deletable` or `sap:updatable` setting in your annotations.

Code Samples

deletable-path

The code sample below shows you how to set up your annotation to display or hide the *Delete* button, based on the value of the `Delete_mc` property in the back-end system.

```
<Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_Product">
    <Annotation Term=" Org.OData.Capabilities.V1.DeleteRestrictions">
        <Record>
            <PropertyValue Property="Deletable" Path="Delete_mc"/>
        </Record>
    </Annotation>
</Annotations>
```

updatable-path

The code sample below shows you how to set up your annotation to display or hide the *Edit* button, based on the value of the `Updatable_mc` property in the back-end system.

```
<Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_Product">
    <Annotation Term="Org.OData.Capabilities.V1.UpdateRestrictions">
        <Record>
            <PropertyValue Property="Updatable" Path="Updatable_mc"/>
        </Record>
    </Annotation>
</Annotations>
```

Smart Controls in Object Page Header

You can use various smart controls within the object page header. You use annotations to set up the smart controls.

Enabling Editable Header Fields

You can enable an option to make fields in the object page header editable in edit mode.

To do this, set the `editableHeaderContent` parameter to `true` in the `manifest.json` file as shown in the example below. This provides a form group for the title and subtitle of the `HeaderInfo` and one form group for each header facet.

Sample Code

```
"sap.ui.generic.app": {
        "pages": [
            {
                "entitySet": "SEPMRA_C_PD_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true
                },
                "pages": [
```

```
                    {
                        "entitySet": "SEPMRA_C_PD_Product",
                        "component": {
                            "name":
"sap.suite.ui.generic.template.ObjectPage",
                            "settings": {
                                "editableHeaderContent": true
                            }
                        },
                        "pages": [
                            {
                                "navigationProperty": "to_ProductText",
                                "entitySet": "SEPMRA_C_PD_ProductText",
                                "component": {
                                    "name":
"sap.suite.ui.generic.template.ObjectPage"
                                }
                            }
                        ]
                    }
                ]
            }
        ]
    },
```

Adding Subpages

If required, you can add additional subpages based on the object page template to your app.

The SAP Web IDE wizard allows you to create one subpage (detail page) for the object page when you create your app. This is expressed in the code as a 1:n association to the root entity and is rendered as a table on the object page.

Removing the Subpage

By default, navigation from the subpage automatically generated by the wizard is available from the object page through the preset definition of a subpage for the corresponding entity set. Simply remove this to delete the subpage and corresponding link from the object page.

Adding Subpages

You can add further subpages manually in the `manifest.json` file post-generation, as shown below:

```
"sap.ui.generic.app": {
    "pages": [
        {
            "entitySet": "SEPMRA_I_ProductWithDraft",
            "component": {
                "name": "sap.suite.ui.generic.template.ListReport",
                "list": true
            },
```

```
            "pages": [
                {
                    "entitySet": "SEPMRA_I_ProductWithDraft",
                    "component": {
                        "name": "sap.suite.ui.generic.template.ObjectPage"
                    },
                    "pages": [
                        {
                            "entitySet": "SEPMRA_I_ProductTextWithDraft",
                            "navigationProperty": "to_ProductText",
                            "component": {
                                "name":
"sap.suite.ui.generic.template.ObjectPage"
                            }
                        }
                    ]
                }
            ]
        }
    ]
},
```

More Information

For information about defining an external navigation target using intent-based navigation, see Changing Navigation to Object Page [page 1583].

Enabling the Related Apps Button

By default, the *Related Apps* button is disabled on object pages created with the object page template. If you want, you can enable this button, which allows you to provide a link to any of the navigation targets of the semantic object.

The *Related Apps* button is displayed on the object page if you set the `showRelatedApps` parameter to `true` in the `manifest.json` as shown below:

```
"sap.ui.generic.app": {
        "pages": [
            {
                "entitySet": "SEPMRA_C_PD_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true
                },
                "pages": [
                    {
                        "entitySet": "SEPMRA_C_PD_Product",
                        "component": {
                            "name": "sap.suite.ui.generic.template.ObjectPage",
                            "settings": {
                                "showRelatedApps": true
                            }
                        },
                        "pages": [
                            {
                                "navigationProperty": "to_ProductText",
```

```
                                "entitySet": "SEPMRA_C_PD_ProductText",
                                "component": {
                                    "name":
"sap.suite.ui.generic.template.ObjectPage"
                                }
                            }
                        ]
                    }
                ]
            }
        ]
    },
```

Defining and Adapting Sections

The object page content is arranged into sections and subsections that you can configure.

You use the `com.sap.vocabularies.UI.v1.Facets` annotations to build sections. Different facets have been defined to display important information in the content area sections.

> **i Note**
>
> All facets are displayed on the same page. The link from a facet leads you to the related section on the same page. The facet annotation label is used twice: Once for the facet in the header area and once for the section's title.

A section can consist of several subsections. If a `UI.CollectionFacet` contains several `UI.CollectionFacets`, each of these is a subsection, as shown in the example below. You can include several content blocks, such as forms or tables in a subsection.

```
<Annotation Term="UI.Facets">
   <Collection>
      <!-- This facet is displayed as before in a section -->
      <Record Type="UI.ReferenceFacet">
         <PropertyValue Property="Label"
String="{@i18n>@GeneralInfoFacetLabel}"/>
         <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#GeneralInformation" />
      </Record>
      <Record Type="UI.CollectionFacet">
         <PropertyValue Property="ID" String="FurtherData"/>
         <PropertyValue Property="Label" String="{@i18n>@FurtherData}"/>
         <PropertyValue Property="Facets">
            <Collection>
               <!-- This CollectionFacet becomes a subsection with a form with
groups -->
               <Record Type="UI.CollectionFacet">
```

This video shows the step-by-step procedure for adding a field group to a section on the object page: .

You can hide and display sections based on properties like this:

> **‹≔› Sample Code**
>
> ```
> <Record Type="UI.ReferenceFacet">
>    <Annotation Term="UI.Hidden" Path="IsActiveEntity"/>
>    <PropertyValue Property="Label" String="{@i18n>@SalesData}" />
> ```

```
    <PropertyValue Property="Target" AnnotationPath="to_ProductSalesData/
@UI.Chart" />
</Record>
```

A facet contains collection facets (`UI.CollectionFacet`) as well as reference facets (`UI.ReferenceFacet`). Collection facets are made up of a list of records, each of which represents a reference facet to a field group, `UI.LineItem`, `UI.Chart`, or another annotation. Reference facets represent a reference, for example, to a `UI.LineItem` (list on the object page), `UI.Chart` (Chart), or `UI.Identification` annotation.

You can define a hierarchy level. Instead of a reference facet, you can add a collection facet that consists of several reference facets. The contents of these reference facets are arranged underneath.

In the figure below, the collection facet for *Product Information* combines three reference facets. Each reference facet refers to a field group or to an identification annotation.

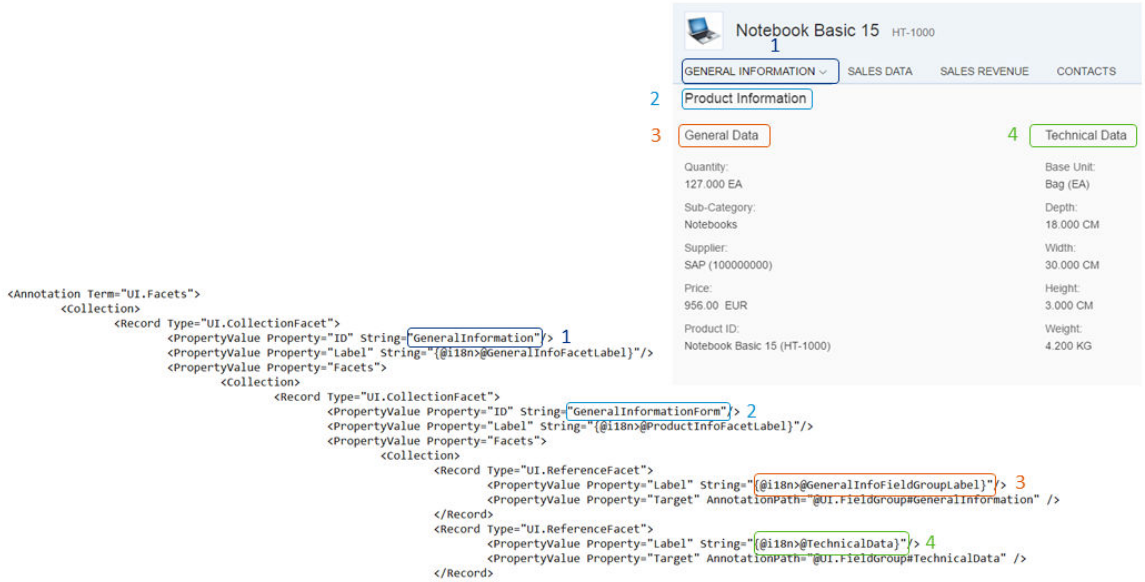


Figure 292: Object Page: CollectionFacet

Further reference facets refer to identification sections, the field group, contact, or line item annotations. For line items, a list is rendered.

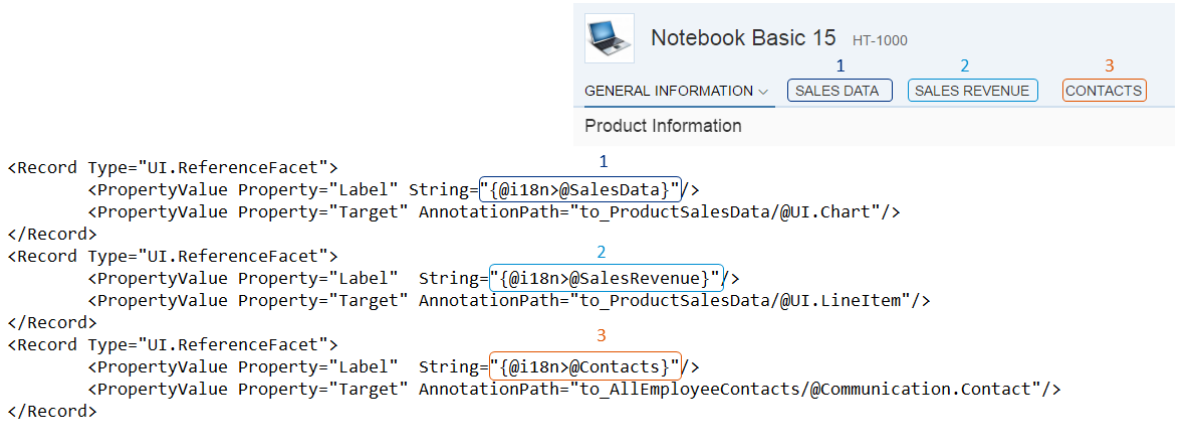


Figure 293: Object Page: ReferenceFacets

Rendering a Smart Table in a Section

To render a smart table in a section, here's what you do:

1. Include a list in the section, indicated by `com.sap.vocabularies.UI.v1.LineItem`.
2. To render an *Add Entry* button, set `Org.OData.Capabilities.V1.InsertRestrictions/Insertable/Bool` to `true` for the entity set.
   To make the related entity set insertable, take into account that:
   - The system gives priority to the `Org.OData.Capabilities.V1.NavigationRestrictions` of the root entity set. Depending on the value of the `Insertable` property of `InsertRestrictions`, the related entity set is made insertable or not insertable.
     If `Insertable` = true, the related entity set is insertable.
     If `Insertable` = false, the related entity set is not insertable.

     ### Sample Code

     ```
     <Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
     STTA_C_MP_Product">
       <Annotation Term="Capabilities.NavigationRestrictions">
         <Record>
             <PropertyValue Property="RestrictedProperties">
               <Collection>
                 <Record>
                    <PropertyValue Property="NavigationProperty"
     NavigationPropertyPath="to_ProductText"/>
                    <PropertyValue Property="InsertRestrictions">
                       <Record>
                          <PropertyValue Property="Insertable" Bool="true"/>
                       </Record>
                    </PropertyValue>
                 </Record>
               </Collection>
               </PropertyValue>
         </Record>
       </Annotation>
     </Annotation>
     ```

> **Note**
>
> `NavigationRestrictions` works only if `InlineCreate` is enabled for the related entity table. The creation from a new object page is not supported using this annotation.

- If `InsertRestrictions` is not defined for the source entity set's `NavigationRestrictions`, the `Insertable` property of the related entity set's `InsertRestrictions` is considered.
  If `Insertable` = true, the related entity set is insertable.
  If `Insertable` = false, the related entity set is not insertable.

  > **Sample Code**
  >
  > ```
  > <Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
  > STTA_C_MP_ProductText">
  >         <Annotation Term="Capabilities.InsertRestrictions">
  >                         <Record>
  >             <PropertyValue Property="Insertable" Bool="true"/>
  >                         </Record>
  >                 </Annotation>
  >                                 </Annotations>
  > ```

- If `InsertRestrictions` is not defined for the related entity set or for the source entity set's `NavigationRestrictions`, the related entity set is insertable.

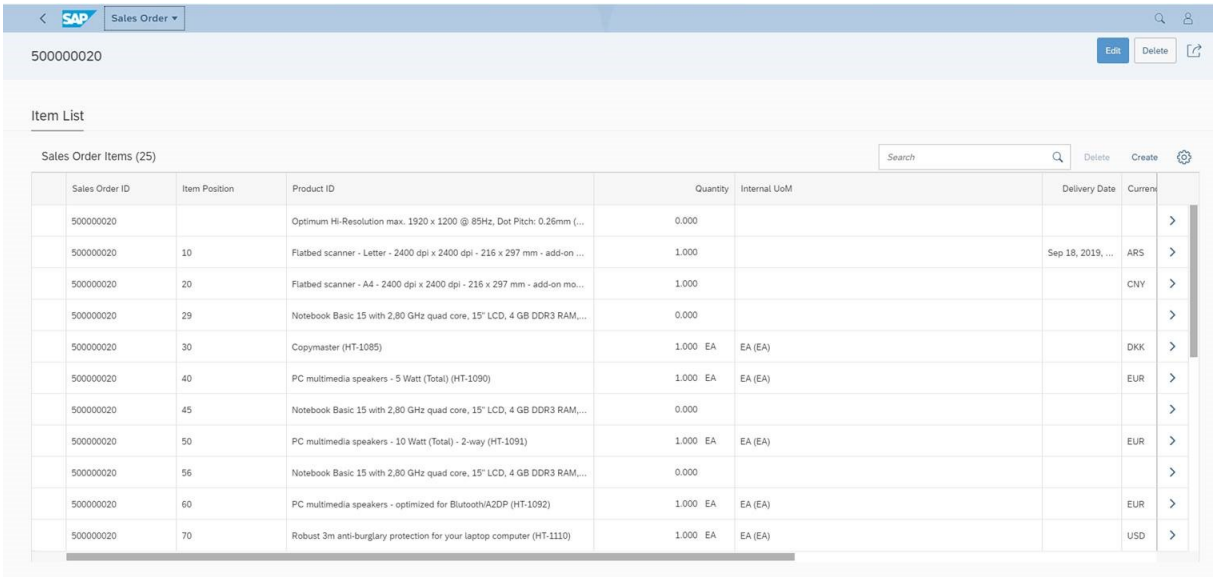
See also Enabling Inline Creation of Table Entries on Object Page [page 1769].

Increased Section and Table Height to use Available Free Space on the Object Page

If the object page contains only one section with just one table or if the object page uses an icon tab bar for sections and any section has only one table, the following system behavior applies:

If the table is a `ui.table`, the section and table expand to use the full page height, showing more rows in the table.

If the table is a `sap.m.table`, the section and table expand to show 25 rows.



Note

For information about the icon tab bar, under Adapting the UI: List Report and Object Page [page 1860], see *Adapting the UI: Object Page* > *Switch to tabs*.

You can also optimize the table visualization by using the condensed mode. For more information, see Using the Condensed Table Layout [page 1773].

If your table has a lot of entries, see the information regarding the `MultiSelectionPlugin` at Enabling Multiple Selection in Tables [page 1741].

IDs for Collection Facets

To enable extensions, personalization, and automated testing, for example, you need to have stable IDs for views and controls. In most cases, they are derived automatically from existing annotations. For collection facets, you can use an annotation to set a stable ID. The ID should be meaningful and must be unique within the entity type. You should use only characters in camel case and without spaces.

If you define your facets in an annotation file in your project, you can add the ID there directly, as in this example:

Sample Code

```
<Annotation Term="UI.Facets">
    <Collection>
        <Record Type="UI.CollectionFacet">
            <PropertyValue Property="ID" String="GeneralInformation"/>
```

Adding a Contact Facet

To render a contacts list and contact facet, you add a `UI.ReferenceFacet` that points to a contact annotation (`type`). It shows the label of the `UI.ReferenceFacet` and, below, the `fn` property of the contact annotation. If you click on the name, a quickview with the contact details is displayed as shown below:

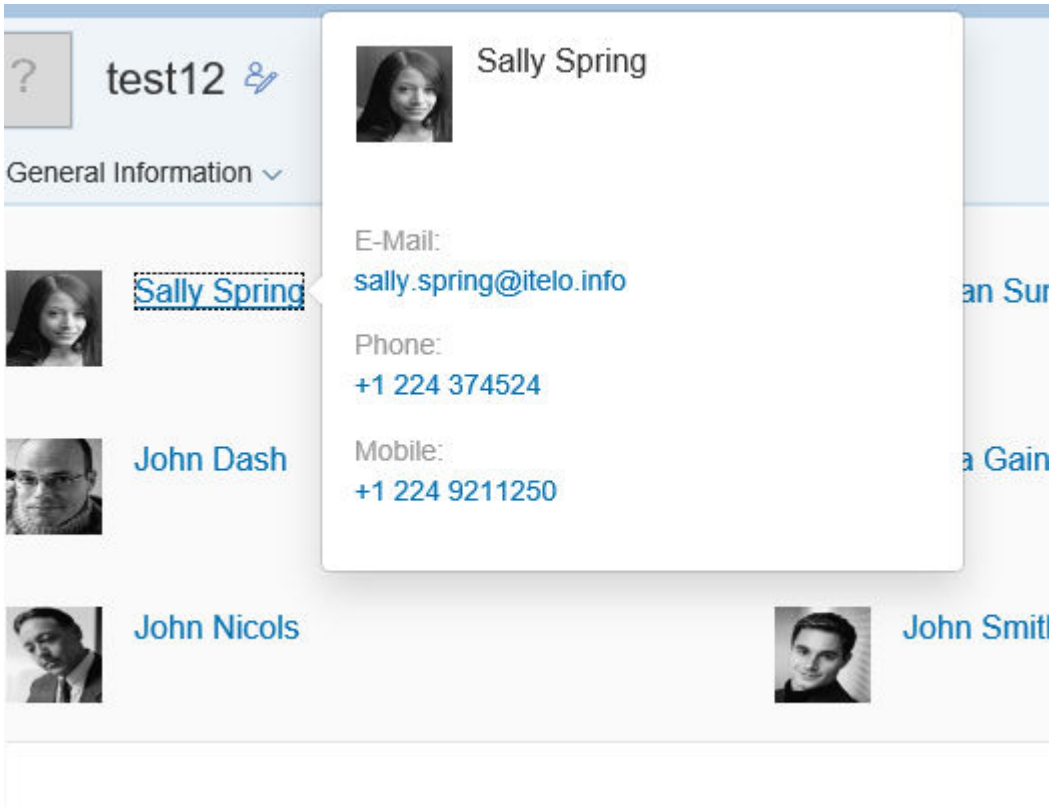


Figure 294: Object Page: Contacts

The facet annotation looks as follows:

Sample Code

```
<Record Type="UI.ReferenceFacet">
   <PropertyValue Property="Label" String="Supplier"/>
   <PropertyValue AnnotationPath="to_Supplier/@Communication.Contact"
Property="Target"/>
</Record>
```

The Contact annotation looks as follows:

Sample Code

```
<Annotations Target="SEPMRA_PROD_MAN.SEPMRA_I_EmployeeType">
    <Annotation Term="Communication.Contact">
        <Record>
            <PropertyValue Property="fn" Path="FormattedName"/>
            <PropertyValue Property="title" Path="JobTitle"/>
            <PropertyValue Property="org" Path="CompanyName"/>
            <PropertyValue Property="role" Path="OrganizationRole"/>
```

```
            <PropertyValue Property="n">
                <Record>
                    <PropertyValue Property="given" Path="FirstName"/>
                    <PropertyValue Property="additional" Path="MiddleName"/>
                    <PropertyValue Property="surname" Path="LastName"/>
                </Record>

            </PropertyValue>
            <PropertyValue Property="photo" Path="EmployeePictureURL"/>
            <PropertyValue Property="tel">
                <Collection>
                    <Record>
                        <PropertyValue Property="type"
EnumMember="Communication.PhoneType/fax"/>
                        <PropertyValue Property="uri" Path="FaxNumber"/>
                    </Record>
                    <Record>
                        <PropertyValue Property="type"
EnumMember="Communication.PhoneType/cell"/>
                        <PropertyValue Property="uri"
Path="MobilePhoneNumber"/>
                    </Record>
                    <Record>
                        <PropertyValue Property="type"
EnumMember="Communication.PhoneType/work"/>
                        <PropertyValue Property="uri" Path="PhoneNumber"/>
                    </Record>
                    <Record>
                        <PropertyValue Property="type" EnumMember=""/>
                        <PropertyValue Property="address" Path="FaxNumber"/>
                    </Record>
                </Collection>
            </PropertyValue>
            <PropertyValue Property="email">
                <Collection>
                    <Record>
                        <PropertyValue Property="type"
EnumMember="Communication.ContactInformationType/work"/>
                        <PropertyValue Property="address"
Path="EmailAddress"/>
                    </Record>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

Address Facet in Sections

If you add a `UI.ReferenceFacet` that points to an address annotation, an address facet is displayed in the object page sections or in a quick view.

It shows the label of the `UI.ReferenceFacet` and, below, only the label property of the address annotation. Therefore, the label property should contain the whole formatted address, with \n for new lines.

> **i** Note
>
> Other properties of the address annotation are not interpreted and rendered.

Example value for the label property: "Mail Drop: TNE QB\n123 Main Street\nAny Town, CA 91921-1234\nU.S.A.". This is shown as follows:

The address facet can be used in various places:

- As a separate section

```
<Record Type="UI.ReferenceFacet">
   <PropertyValue Property="Label" String="Communication Address" />
   <PropertyValue Property="Target" AnnotationPath="to_Address/
@Communication.Address" />
</Record>
```

- As part of a field group within a section and in a quick view with smart link navigation

```
<Annotation Term="UI.FieldGroup" Qualifier="GeneralInformation">
   <Record>
      <PropertyValue Property="Data">
         <Collection>
            <Record Type="UI.DataFieldForAnnotation">
               <PropertyValue Property="Label" String="Communication
Address" />
               <PropertyValue Property="Target" AnnotationPath="to_Address/
@Communication.Address"/>
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Label" String="Product Information"/>
   </Record>
</Annotation>
```

Related Information

Enabling Quick Views for Smart Link Navigation [page 1567]

Smart Chart Facet

You can add a smart chart facet to a content section within the object page.

A smart chart facet contains a `SmartChart` control and is suitable to use if you wish to present data graphically for analysis.

To add a smart chart facet, use the `UI.Facet` term and include the `UI.ReferenceFacet` complex type, and then reference the `UI.Chart` annotation. This is displayed as shown below within a content section of the object page:

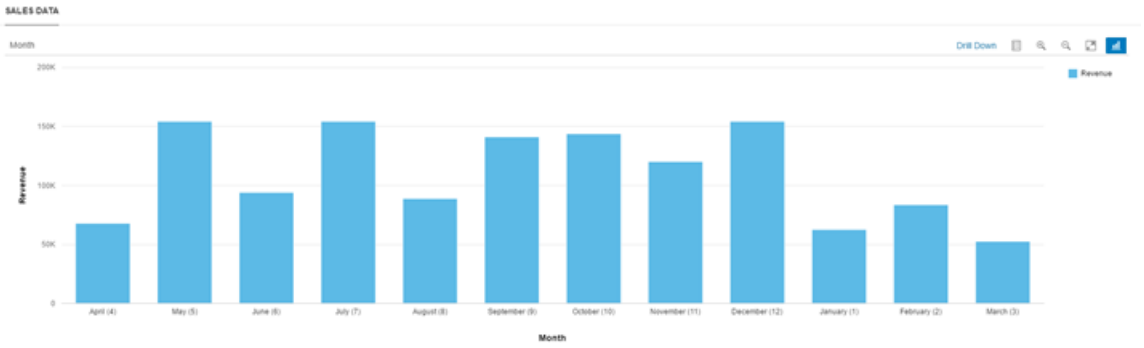


Figure 295: Smart Chart in Object Page

Code Samples

The following code samples show an example of how to create your annotations for the smart chart facet:

UI.ReferenceFacet

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
   <Annotation Term="UI.Facets">
      <Record Type="UI.ReferenceFacet">
         <PropertyValue Property="Label" String="{@i18n>@SalesData}"/>
         <PropertyValue Property="Target" AnnotationPath="to_ProductSalesData/
@UI.Chart"/>
      </Record>
   </Annotation>
</Annotations>
```

UI.Chart

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductSalesDataType">
   <Annotation Term="UI.Chart">
      <Record>
         <PropertyValue Property="Title" String="Test Smart Chart"/>
         <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Column"/>
         <PropertyValue Property="Dimensions">
            <Collection>
               <PropertyPath>DeliveryMonth</PropertyPath>
            </Collection>
         </PropertyValue>
         <PropertyValue Property="Measures">
            <Collection>
               <PropertyPath>Revenue</PropertyPath>
            </Collection>
         </PropertyValue>
      </Record>
   </Annotation>
</Annotations>
```

The chart definition contains measures, on which the aggregations calculations are done, and dimensions, which categorize these measures. In the chart, these are displayed as labels on the x and on the y axis.

> **Note**
>
> The object page does not support `UI.Chart` with qualifier.
>
> The use of navigation properties within the `UI.Chart` term for the smart chart is not supported (see example below).

```
<Annotation Term="UI.Chart" Qualifier="ChartQualifier"> Not supported
        <Record Type="UI.ChartDefinitionType">
                <PropertyValue Property="Title" String="Gross Sales Revenue"/>
                <PropertyValue Property="Description" String="With navigation & criticality"/>
                <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Bullet"/>
                <PropertyValue Property="Measures">
                        <Collection>
                                <PropertyPath>to_SalesData/Revenue</PropertyPath>  Not supported
                        </Collection>
                </PropertyValue>
        </Record>
</Annotation>
```

Figure 296: Navigation Property

Defining Actions in Smart Chart Toolbar

You can add action buttons to the smart chart toolbar by defining the `Actions` property in the annotation term `UI.Chart`. The `Actions` property must be a collection of the types `UI.DataFieldForAction` and `UI.DataFieldForIntentBasedNavigation`.

> **Note**
>
> To make the button visible in the smart chart toolbar, the properties `Determining` and `Inline` for `UI.DataFieldForAction` and `UI.DataFieldForIntentBasedNavigation` must be set to `false` or not defined.

The action buttons are displayed as shown below in the smart chart toolbar:

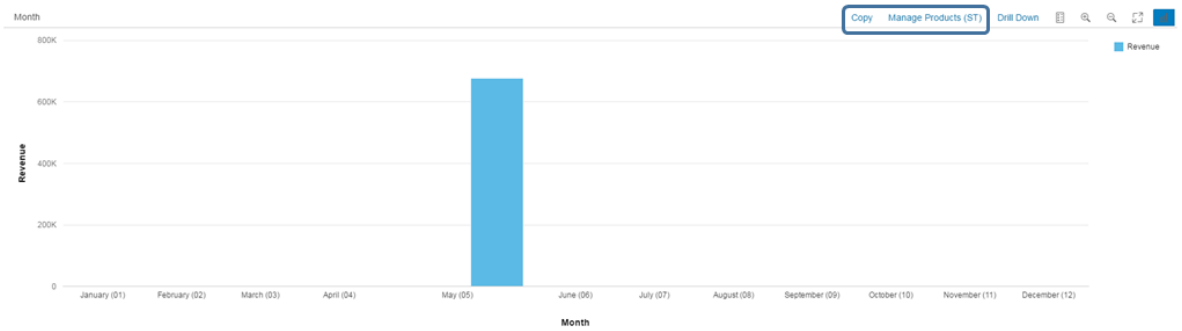


Figure 297: Actions in Smart Chart Toolbar

Enabling and Disabling Actions in Smart Chart Toolbar

To control whether `UI.DataFieldForAction`-based actions are enabled or disabled in the smart chart toolbar, see Using Action Control for Context-Dependent Actions [page 1778].

To control whether `UI.DataFieldForIntentBasedNavigation`-based actions are enabled or disabled in the smart chart toolbar, use the property `RequiresContext`. If this property is set to `true` (default value), the user has to select an item in the smart chart before the action can be performed. If this property is set to `false`, then the button will always be enabled.

Limitation

If you are using the `applicable-path` setting to control the visibility of actions in the smart chart toolbar, the property specified must point to a property within the same entity and cannot be a navigation property (see below).

Code Samples

The code samples below show how to add actions to the smart chart toolbar.

UI.Chart for Smart Chart

```
<Annotations Target="<entity type>">
     <Annotation Term="UI.Chart">
          <Record>
               <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Column"/>
               <PropertyValue Property="Dimensions">
                    <Collection>
                         <PropertyPath><property path></PropertyPath>
                    </Collection>
               </PropertyValue>
               <PropertyValue Property="Measures">
                    <Collection>
                         <PropertyPath><property path></PropertyPath>
                    </Collection>
               </PropertyValue>
               <PropertyValue Property="Actions">
                    <Collection>
                         <Record Type="UI.DataFieldForAction">
                              <PropertyValue Property="Label" String="<label>"/>
                              <PropertyValue Property="Action" String="<entity
type>"/>
                              <PropertyValue Property="InvocationGrouping"
EnumMember="UI.OperationGroupingType/Isolated"/>
                         </Record>
                         <Record Type="UI.DataFieldForIntentBasedNavigation">
                              <PropertyValue Property="Label" String="<label>"/>
                              <PropertyValue Property="SemanticObject"
String="<semantic object>"/>
                              <PropertyValue Property="Action"
String="<action>"/>
                         </Record>
                    </Collection>
```

```
                </PropertyValue>
            </Record>
        </Annotation>
</Annotations>
```

Hiding and Showing Reference Facets Using See More and See Less Links

You can hide or show reference facets on the UI using the *See more* or *See less* links.

The *See more* link appears in the bottom right corner if hidden facets are available.

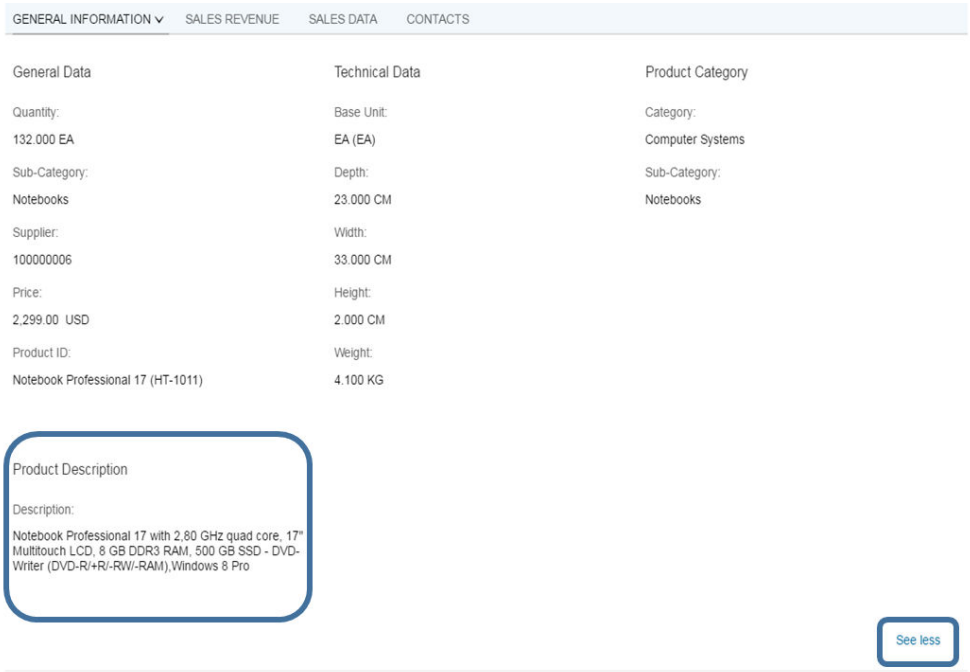


Figure 298: See Less

To enable this feature, set the `PartOfPreview` annotation of the relevant reference facet to `false` as in the following example:

Sample Code

```
<Record Type="UI.ReferenceFacet">
<PropertyValue Property="Label" String="{@i18n>@ProductDescription}"/>
<PropertyValue Property="Target" AnnotationPath="to_ProductTextInCurrentLang/
@UI.FieldGroup#PlainText"/>
        <Annotation Term="UI.PartOfPreview" Bool="false"/>
    </Record>
```

Note

This feature is available for reference facets that are implemented under a collection facet.

Adding Action Buttons to Forms in Sections

You can add action buttons to the forms contained in sections. These forms are indicated by `com.sap.vocabularies.UI.v1.FieldGroup`. A form action button is then displayed in the toolbar of the object page section that contains the form.

You can add the following action buttons:

- Buttons calling an OData function (using `DataFieldForAction`)
- Buttons triggering intent-based navigation (using `DataFieldForIntentBasedNavigation`)

To do so, add a `DataFieldForAction` or `DataFieldForIntentBasedNavigation` to the data of your `FieldGroup` definition. The following example shows the definition of a form with two buttons:

Sample Code

```
<Annotation Term="UI.FieldGroup" Qualifier="TechnicalData">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="Width"/>
                    <Annotation Term="UI.Importance"
EnumMember="UI.ImportanceType/Medium"/>
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="Depth"/>
                    <Annotation Term="UI.Importance"
EnumMember="UI.ImportanceType/Medium"/>
                </Record>
                ...
                <Record Type="UI.DataFieldForAction">
                    <PropertyValue Property="Label" String="Copy"/>
                    <PropertyValue Property="Action"
String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopy"/>
                </Record>
                <Record Type="UI.DataFieldForIntentBasedNavigation">
                    <PropertyValue Property="Label" String="Manage Products
(ST)" />
                    <PropertyValue Property="SemanticObject"
String="EPMProduct" />
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Label" String="Technical Data"/>
    </Record>
</Annotation>
```

For more information, see also Configuring External Navigation [page 1563].

Hiding Features Using the UI.Hidden Annotation

You can use the `UI.Hidden` annotation to hide or display specific features on the object page.

The default value of the `UI.Hidden` annotation is `true`, that is, a feature using the `UI.Hidden` annotation term is not visible on the UI. These are the values you can set:

Annotation	System Behavior
`<Annotation Term="UI.Hidden" Bool="false"/>`	The feature is visible
`<Annotation Term="UI.Hidden" Bool="true"/>`	The feature is not visible
`<Annotation Term="UI.Hidden" Path="Edit_ac"/>`	The feature is visible if the path evaluates to `false` and is not visible if the path evaluates to `true`. **i Note** The path must point to a boolean property. Expression bindings, for instance, using a negation with !, are not supported behind the path. For more information, see Expression Binding [page 845].
`<Annotation Term="UI.Hidden" />`	The default value is `true`

Header Facets on the Object Page

You can hide header facets as shown below:

i Note

You cannot use the `UI.Hidden` annotation to hide the entire `UI.FieldGroup` and `UI.Identification`. If you want to hide these, you have to hide all `DataField` records in them.

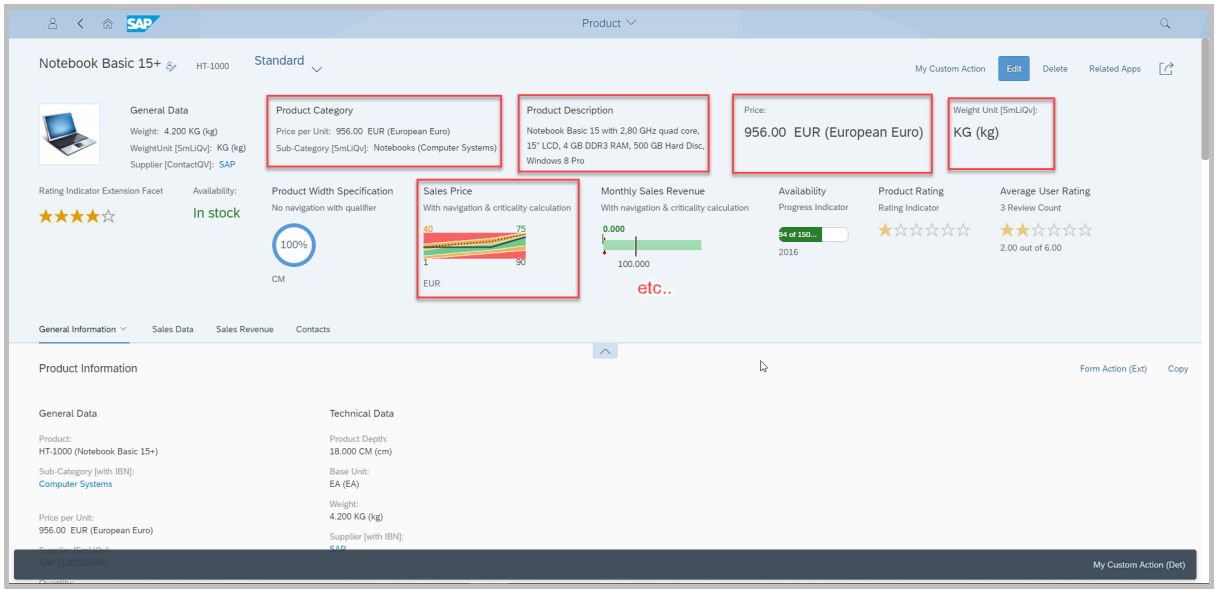


Figure 299: Header Facets on Object Page

```
<Annotation Term="UI.HeaderFacets">
    <Collection>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label"
                String="{@i18n>@GeneralInfoFieldGroupLabel}" />
            <PropertyValue Property="Target"
                AnnotationPath="@UI.FieldGroup#GeneralInformationForHeader" />
                <Annotation Term="UI.Hidden" Bool="false"/>
        </Record>

        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="{@i18n>@ProductCategory}" />
            <PropertyValue Property="Target" AnnotationPath="@UI.Identification" />
            <Annotation Term="UI.Hidden" Path="Edit_ac"/>
        </Record>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label"
                String="{@i18n>@ProductDescription}" />
            <PropertyValue Property="Target" AnnotationPath="to_ProductTextInOriginalLang/@UI.FieldGroup#P
            <Annotation Term="UI.Hidden"/>
        </Record>
```

Sections on Object Page

You can hide an entire section. If you're using collection facets, you can also hide just one subsection. You can hide content within a section. See also DataField Records in Facets [page 1715].

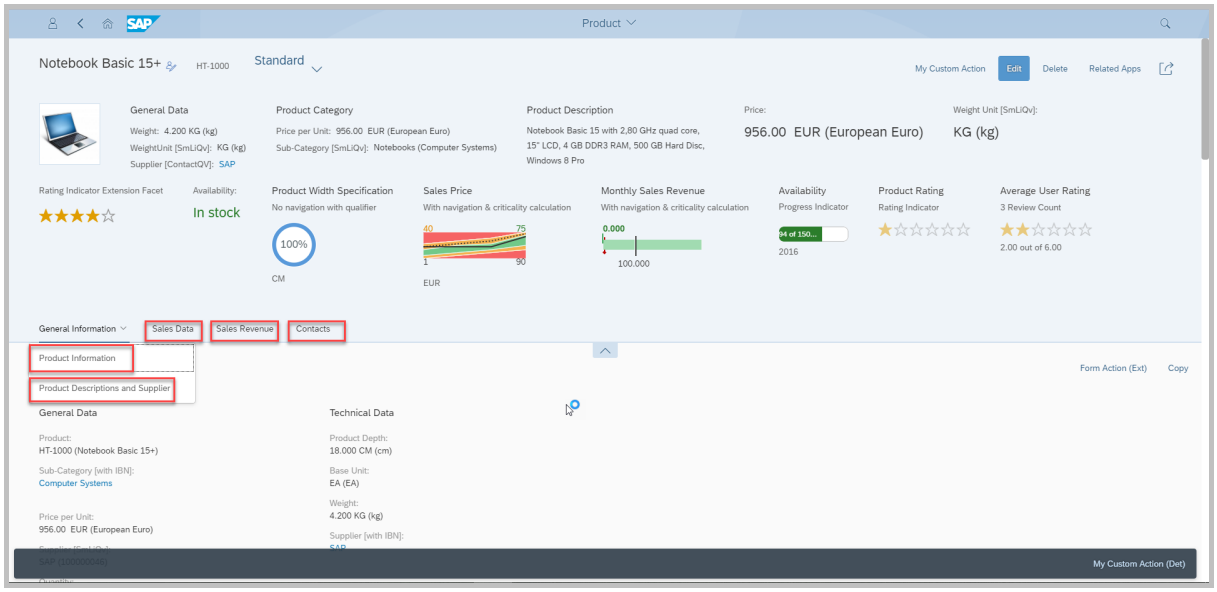


Figure 300: Sections on Object Page

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
    <Annotation Term="UI.Facets">
        <Collection>
            <Record Type="UI.CollectionFacet">
                <PropertyValue Property="ID" String="GeneralInformation" />
                <PropertyValue Property="Label"
                    String="{@i18n>@GeneralInfoFacetLabel}" />
                <PropertyValue Property="Facets">
                    <Collection>
                        <Record Type="UI.CollectionFacet">
                            <PropertyValue Property="ID" String="GeneralInformationForm" />
                            <PropertyValue Property="Label"
                                String="{@i18n>@ProductInfoFacetLabel}" />
                            <PropertyValue Property="Facets">
                                <Collection>
                                    <Record Type="UI.ReferenceFacet">
                                        <PropertyValue Property="Label"
                                            String="{@i18n>@GeneralInfoFieldGroupLabel}" />
                                        <PropertyValue Property="Target"
                                            AnnotationPath="@UI.FieldGroup#GeneralInformation" />
                                    </Record>
                                    <Record Type="UI.ReferenceFacet">
                                        <PropertyValue Property="Label"
                                            String="{@i18n>@TechnicalData}" />
                                        <PropertyValue Property="Target"
                                            AnnotationPath="@UI.FieldGroup#TechnicalData" />
                                    </Record>
                                    <Record Type="UI.ReferenceFacet">
                                        <PropertyValue Property="Label"
                                            String="{@i18n>@ProductCategory}" />
                                        <PropertyValue Property="Target"
                                            AnnotationPath="@UI.Identification" />
                                        <Annotation Term="UI.PartOfPreview" Bool="false" />
                                    </Record>
                                </Collection>
                            </PropertyValue>
                            <Annotation Term="UI.Hidden" Path="Edit_ac"/>
                        </Record>
```

Content in Quick Views

You can hide content in quick views, such as field groups like this:

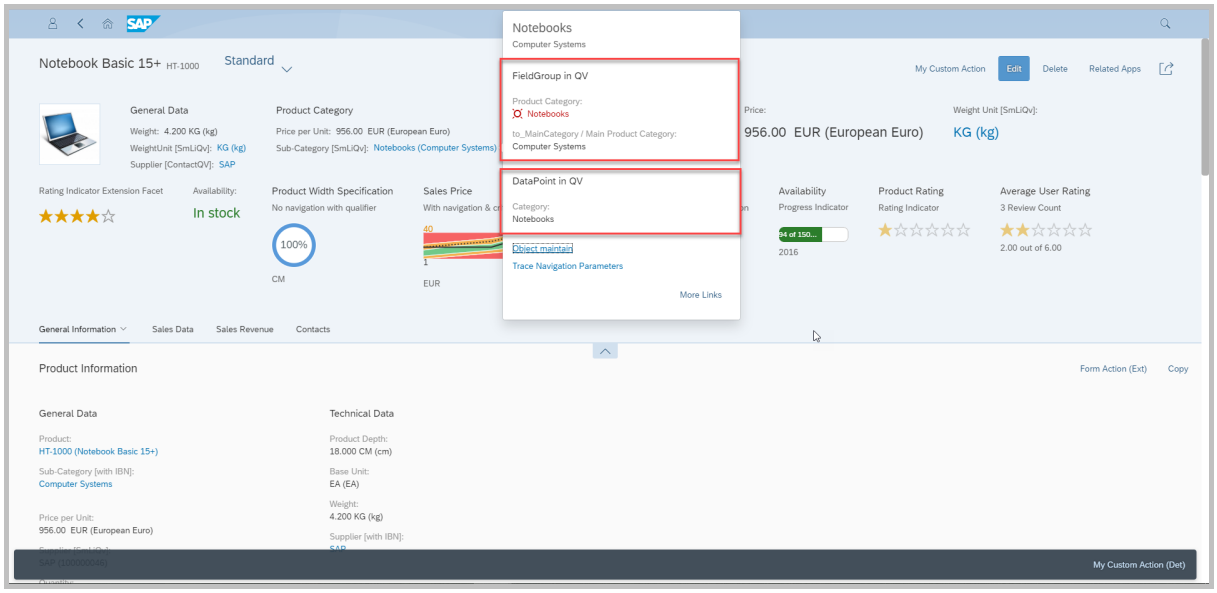


Figure 301: Field Groups in Quick Views on Object Page

```
<Annotation Term="UI.QuickViewFacets">
    <Collection>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="FieldGroup in QV" />
            <PropertyValue Property="Target" AnnotationPath="@UI.FieldGroup#ProductCategoryQuickViewPOC_FieldGroup_1" />
            <Annotation Term="UI.Hidden" Path="Edit_ac" />
        </Record>
        <Record Type="UI.ReferenceFacet">
            <PropertyValue Property="Label" String="DataPoint in QV" />
            <PropertyValue Property="Target" AnnotationPath="@UI.DataPoint#Product" />
            <Annotation Term="UI.Hidden" Path="Activation_ac" />
        </Record>
    </Collection>
</Annotation>
```

You can also use this annotation to hide the content in quick views in the list report.

DataField Records in Header Facets

You can hide DataField records, such as UI.DataField or UI.DataFieldForAnnotation in header facets as shown below:

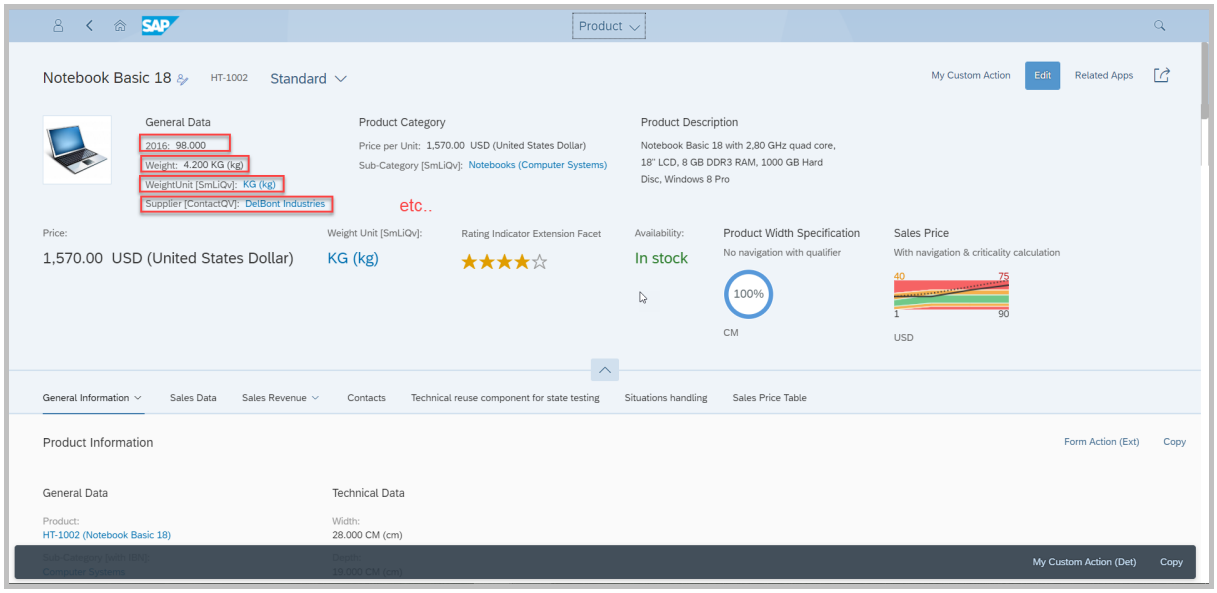


Figure 302: DataField Records in Header Facets

```
<Annotation Term="UI.FieldGroup" Qualifier="GeneralInformationForHeader">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="to_StockAvailability/Quantity"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/Low"/>
                    <Annotation Term="UI.Hidden" Path="Edit_ac" />
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="Weight"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Label" String="WeightUnit [SmLiQv]"/>
                    <PropertyValue Property="Value" Path="WeightUnit"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                </Record>
                <Record Type="UI.DataFieldForAnnotation">
                    <PropertyValue Property="Label" String="Supplier [ContactQV]"/>
                    <PropertyValue Property="Target" AnnotationPath="to_Supplier/@Communication.Contact"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Label" String="{@i18n>@GeneralInfoFieldGroupLabel}"/>
    </Record>
</Annotation>
```

DataField Records in Facets

You can hide DataField records, for example, UI.DataField, UI.DataFieldForAnnotation in facets as shown below:

> **i Note**
>
> You cannot use the `UI.Hidden` annotation to hide the entire `UI.FieldGroup` and `UI.Identification`. If you want to hide these, you have to hide all `DataField` records within them.

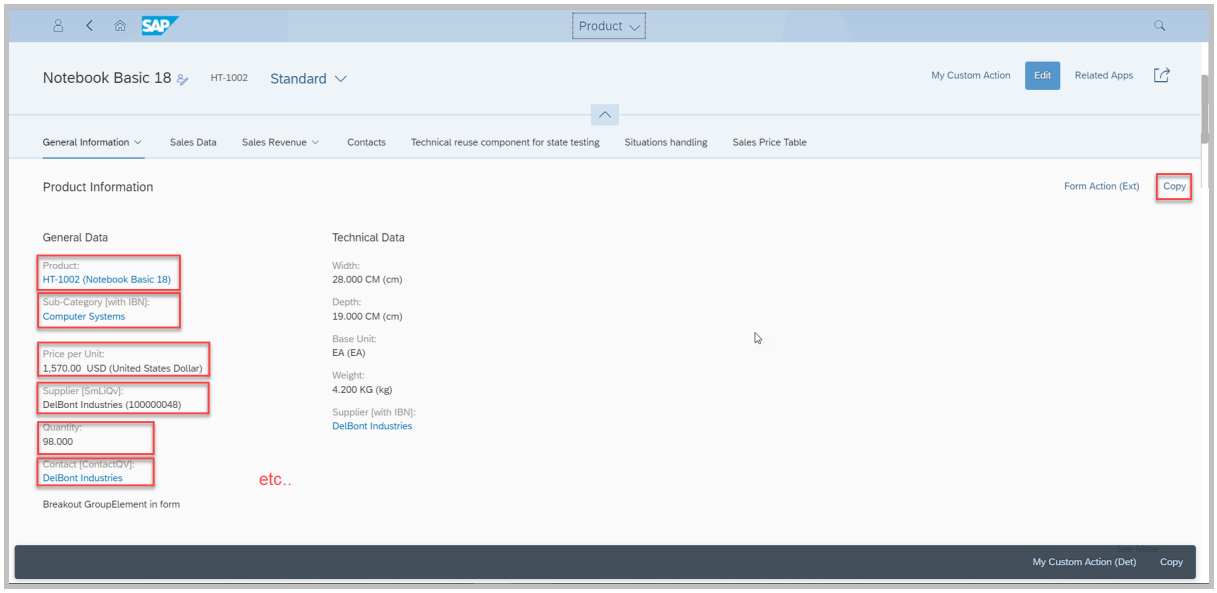


Figure 303: DataField Records in Facets

```
<Annotation Term="UI.FieldGroup" Qualifier="GeneralInformation">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataFieldForAction">
                    <PropertyValue Property="Label" String="Copy"/>
                    <PropertyValue Property="Action" String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopy"/>
                    <PropertyValue Property="InvocationGrouping" EnumMember="UI.OperationGroupingType/Isolated"/>
                    <Annotation Term="UI.Hidden" Path="Edit_ac" />
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Label" String="Product"/>
                    <PropertyValue Property="Value" Path="ProductForEdit"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                    <Annotation Term="UI.Hidden" Path="Edit_ac" />
                </Record>
                <Record Type="UI.DataFieldWithIntentBasedNavigation">
                    <PropertyValue Property="Label" String="Sub-Category [with IBN]"/>
                    <PropertyValue Property="SemanticObject" String="EPMProduct"/>
                    <PropertyValue Property="Action" String="manage_stta"/>
                    <PropertyValue Property="Value" Path="ProductCategory"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/Medium"/>
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="Price"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/Medium"/>
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Label" String="Supplier [SmLiQv]"/>
                    <PropertyValue Property="Value" Path="Supplier"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                </Record>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="to_StockAvailability/Quantity"/>
                    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
                </Record>
```

DataField Records in Smart Tables

To hide table columns, the `UI.Hidden` path needs to refer to the property of the header instance, for example:

```
<Annotation Term="UI.Hidden" Path="to_Product/Edit_ac">
```

In the following example, `STTA_C_MP_Product` is the entity set of the object page header and `STTA_C_MP_ProductText` is the entity set of the smart table on the object page, and `to_Product` is the navigation property from `STTA_C_MP_ProductText` to `STTA_C_MP_Product`.

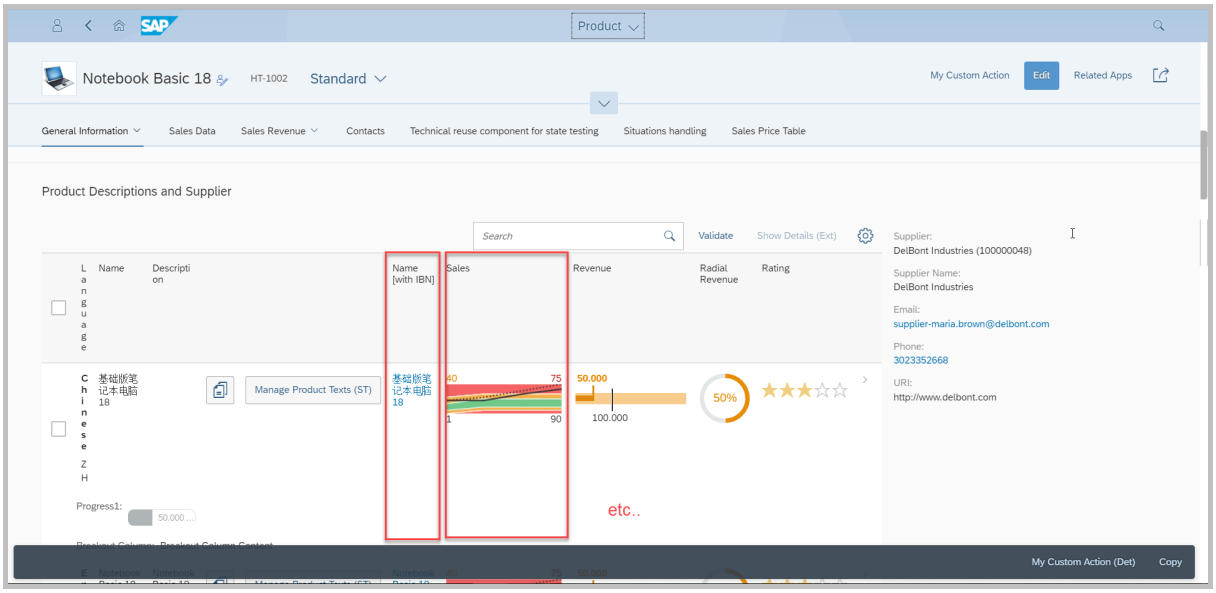


Figure 304: DataField Records in Smart Tables

```
<Annotation Term="UI.LineItem">
    <Collection>
        <Record Type="UI.DataField">
            <PropertyValue Property="Label" String="Language"/>
            <PropertyValue Property="Value" Path="LanguageForEdit"/>
            <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
        </Record>
        <Record Type="UI.DataField">
            <PropertyValue Property="Label" String="Name"/>
            <PropertyValue Property="Value" Path="Name"/>
            <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
        </Record>
        <Record Type="UI.DataField">
            <PropertyValue Property="Label" String="Description"/>
            <PropertyValue Property="Value" Path="Description"/>
            <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/Medium"/>
        </Record>
        <Record Type="UI.DataFieldForAction">
            <PropertyValue Property="Label" String="Copy Text"/>
            <PropertyValue Property="IconUrl" String="sap-icon://copy"/>
            <PropertyValue Property="Action" String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopyText"/>
            <PropertyValue Property="Inline" Bool="true"/>
            <PropertyValue Property="InvocationGrouping" EnumMember="UI.OperationGroupingType/Isolated"/>
        </Record>
        <Record Type="UI.DataFieldForIntentBasedNavigation">
            <PropertyValue Property="Label" String="Manage Product Texts (ST)"/>
            <PropertyValue Property="SemanticObject" String="EPMProduct"/>
            <PropertyValue Property="Action" String="manage_st"/>
            <PropertyValue Property="Inline" Bool="true"/>
        </Record>
        <Record Type="UI.DataFieldWithIntentBasedNavigation">
            <PropertyValue Property="Label" String="Name [with IBN]"/>
            <PropertyValue Property="Value" Path="Name"/>
            <PropertyValue Property="SemanticObject" String="EPMProduct"/>
            <PropertyValue Property="Action" String="manage_stta"/>
            <Annotation Term="UI.Hidden" Path="to_Product/Edit_ac" />
        </Record>
```

Notes on Hiding DataField Records

- Even if you hide all determining actions in the footer, the footer is still displayed on the UI. This also applies to determining actions added via the manifest.

- The *See More* button is visible even if the content within it is hidden.

Configuring a Confirmation Popup for Messages

In draft scenarios, the system displays warning messages during the save process, for example, if fields have not been filled consistently or if entries are missing.

You can enable a confirmation popup that displays all warning and error messages, asking users whether they still want to save the data.

> **i Note**
>
> The *Save* option is availabe only if there are no error messages.

To enable this popup, in the manifest.json under the object page settings, set the `showConfirmationOnDraftActivate` flag to `true`.

```
"pages": {
                        "ObjectPage|STTA_C_MP_Product": {
                                  "entitySet": "STTA_C_MP_Product",
                                  "component": {
                                           "name":
"sap.suite.ui.generic.template.ObjectPage",
                                           "settings": {
                                                      "showRelatedApps": true,
                                                      "tableType":
"ResponsiveTable",
                                                      "editableHeaderContent":
true,
"showConfirmationOnDraftActivate": true,
                                                      "sections": {
"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
"navigationProperty": "to_ProductText",
"entitySet": "STTA_C_MP_ProductText",
"multiSelect": true,
"createMode": "inline",
"tableType": "ResponsiveTable"
                                                                                 }
                                                                }
                                                      }
           },
.
.
.
```

> **i Note**
>
> During the activation of a draft document, if the backend responds with response code 412, the system displays a confirmation popup for the user. This is independent of the `showConfirmationOnDraftActivate` flag.

Save and Navigation Options on the Object Page

When creating, editing, and saving draft or non-draft records, the user stays on the object page by default.

Save and Close Feature - Navigation to List Report

You can add save and close logic to the *Save* button on the object page to enable users to navigate directly back to the list report. If the user clicks the button, the draft record is saved and the user automatically navigates to the list report page.

To enable this feature, set the `navToListOnSave` flag to `true` in the manifest.json.

```
"pages": {
                                    "ObjectPage|STTA_C_MP_Product": {
                                           "entitySet": "STTA_C_MP_Product",
                                           "component": {
                                                  "name":
"sap.suite.ui.generic.template.ObjectPage",
                                                  "settings": {
                                                         "navToListOnSave": true,
                                                         "showRelatedApps": true,
                                                         "tableType":
"ResponsiveTable",
                                                         "editableHeaderContent":
true,
"showConfirmationOnDraftActivate": true,
                                                         "sections": {
"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
"navigationProperty": "to_ProductText",
"entitySet": "STTA_C_MP_ProductText",
"multiSelect": true,
"createMode": "inline",
"tableType": "ResponsiveTable"
                                                                       }
                                                         }
                                                  }
                                           },
```

If the flag is set to `true`, the user automatically navigates to the list report page as soon as the draft record has been saved.

If the flag is set to `false` or if the flag is not set at all, the user stays on object page which is also the default behavior.

Using the Smart MultiInput Control on the Object Page

For fields in which users can enter more than one value, the Smart MultiInput control is rendered on the object page if specific conditions are met.

If the system identifies a 1:N association of a `DataField`, the Smart MultiInput control is activated automatically.

> **i Note**
>
> You must have maintained this 1:N association in the back-end system. Without this 1:N association, a SmartField is rendered.

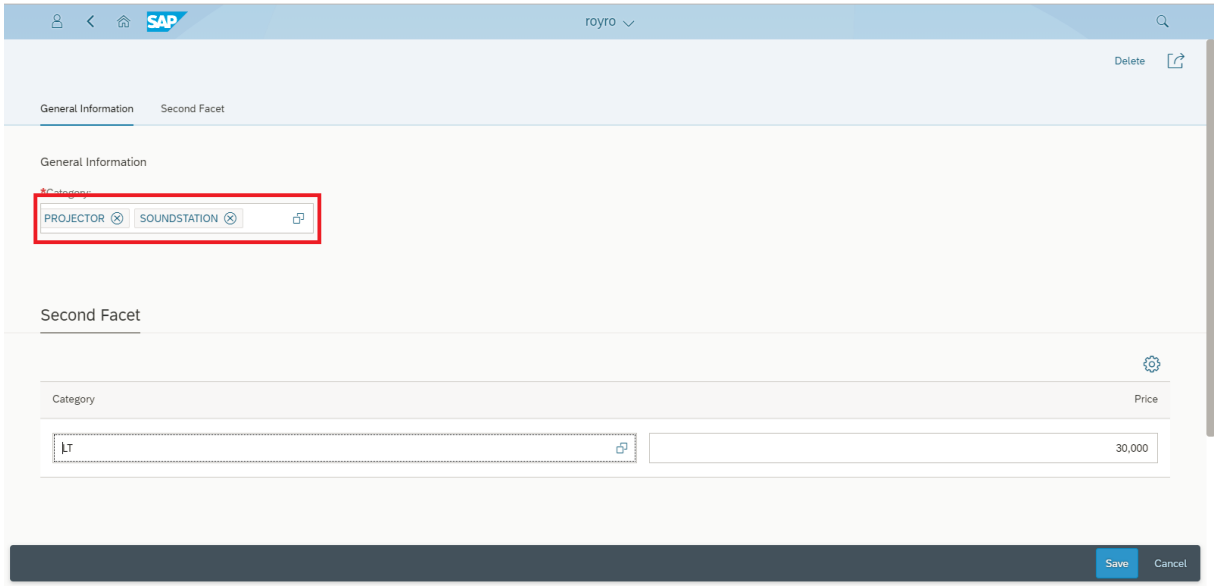


Figure 305: Smart MultiInput Control on the Object Page Form

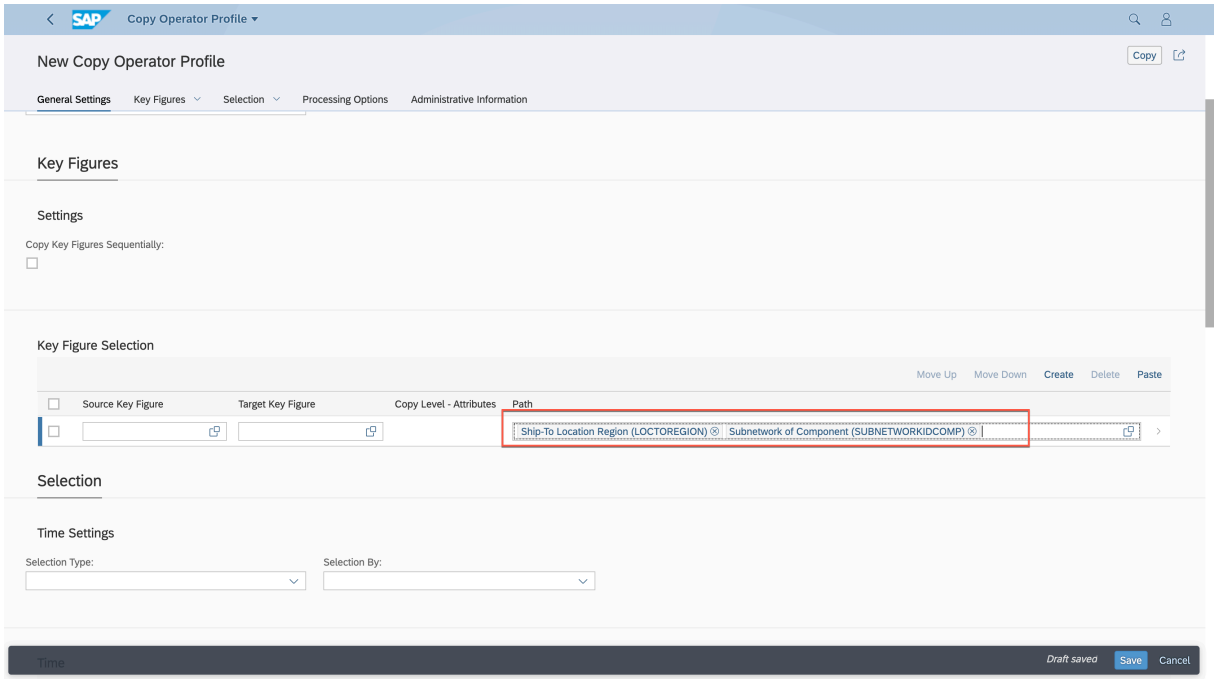


Figure 306: Smart MultiInput Control on the Object Page Table

> **i Note**
>
> You cannot export to excel or copy and paste data from the smart multi-input field.

Related Information

Smart Tables [page 1628]

Including Reuse Components on an Object Page

You can embed a reuse component as a section in the object page of your app.

A reuse component is an encapsulated UI area that can be included (embedded) into an SAP Fiori elements-based application. Reuse components are mainly used for reusable UIs which are used consistently across several applications, for example, output management or attachment services. However, you can also create a one-time reuse component which is used in only one application on one page.

Not every piece of reusable code should be encapsulated into a reuse component. Reuse components are always rectangular UI areas.

> **i Note**
>
> A reuse component that can be used within a SAP Fiori elements-based application cannot be used in a freestyle application. If reuse component owners want to provide the functionality for both types of applications, they need to provide two components, one for each use case.

We recommend factoring out as much coding as possible into artifacts that are shared by both components.

Reuse components that are used on an object page are often used on several pages. If this is not the case, consider whether they can be realized via view or fragment extensions.

To include a reuse component on an object page, adapt the manifest.json of your app, in the pages section of the object page which is to contain the reuse component.

The following example embeds two reuse components:

Sample Code

```
...
 "sap.ui.generic.app": {
    ...
    "pages": {
        "ListReport|myRootEntity": {
            "entitySet": "myRootEntity",
            "component": {
                "name": "sap.suite.ui.generic.template.ListReport",
                "list": true,
                ...
            },
            "pages": {
                "ObjectPage|myRootEntity": {
                    "entitySet": "myRootEntity",
                    "component": {
                        "name": "sap.suite.ui.generic.template.ObjectPage",
                        ...
                    },
                    "embeddedComponents": {
                        "myFirstComponentEmbedding": {
                            "id": "myFirstComponentEmbedding",
                            "componentUsage": "myUsage",
                            "title":
"{{I18N_KEY_FOR_FIRST_REUSE_COMPONENT_TITLE}}",
                            "settings": {
                                "documentNumber": "{documentNumber}"
                            }
                        },
                        "mySecondComponentEmbedding": {
                            "id": "mySecondComponentEmbedding",
                            "componentName":
"theOtherReuseComponentQualifiedName",
                            "title":
"{{I18N_KEY_FOR_SECOND_REUSE_COMPONENT_TITLE}}",
                            "binding": "myNavProperty",
                        }
                    },
                    "pages": {
                        // add list of sub-pages of the root object page here
                    }
                }
            }
        }
    }
```

The logical names of the embedded components (`myFirstComponentEmbedding` and `mySecondComponentEmbedding`) each appear twice within the codeblock. This is because the name of the property (within `embeddedComponents`) which defines the reuse component instance and the value of the property ID in this definition must be identical.

The sample definitions above show two options for addressing a reuse component:

- Faciliating a use of the component that has been defined in the corresponding section of the manifest. For more information, see also https://sapui5.hana.ondemand.com/#/api/sap.ui.core.Component/methods/createComponent
- Directly addressing the component name within the definition

Use the `title` property to provide a title for the section that is to contain the reuse component. To provide a translatable title, this property must point to a key in the i18n file of the app.

If the reuse component exposes properties other than the predefined ones (see Developing Reuse Components [page 1726]), the embedding may contain binding information for these properties. See the property settings under `myFirstComponentEmbedding` above. The corresponding property in this case is `documentNumber`. The value for this property can be any valid binding string, for example, "`{documentNumber}`" can also be replaced by an expression binding like "`{= ${documentNumber} || ${documentId} }`" if relevant for your app.

The reuse component instance cannot only receive information about the object that is on the embedding object page by transferring dedicated properties. As described under Developing Reuse Components [page 1726], the binding context that has been set for the OData model for the embedding page is also valid for the reuse component. However, within the definition of the embedding, this binding context might be modified using the `binding` property. See the second embedding definition above for this. The path that is in the value for this property defines a path relative to the current binding context for the whole object page. Thus, the binding context that is valid for the second reuse component instance is determined by applying this relative path to the binding context that is valid for the whole page.

Defining the Default Visibility

Most reuse components are always visible on the object page they have been placed on. However, you can define that a reuse component is only visible for specific instances of the corresponding entity. In your reuse component implementation, use the `setSectionHidden` method of the extensionAPI to show or hide the reuse component on the object page. For more information, see Developing Reuse Components [page 1726].

Set the default visibility to the value that is correct for most cases. You can use the `hiddenByDefault` property for this purpose, on the same level as ID, title, settings, and so on. If this property is set to `true`, the section is hidden by default when the object page is opened with a new instance. It is only visible when the reuse component unhides itself using the `setSectionHidden` method..

Placing Reuse Component Instances

Each reuse component instance defined for an object page of a SAP Fiori elements-based app is realized by a section that contains a sub-section which in turn contains the content of the reuse component. By default, these sections are added to the end of the object page in the same order the reuse component instances are defined in the manifest. You can move these sections using the SAP Visual Editor. For more options, see Placing Reuse Component Instances on the Object Page [page 1724].

Dependencies

If the reuse component uses its own OData service, make sure you declare it as a dependency both under `"sap.ui5"/"dependencies"/"libs"` and under `"sap.ui5"/"dependencies"/"component"`.

Placing Reuse Component Instances on the Object Page

Each reuse component instance defined for an object page belonging to an SAP Fiori elements-based app is realized by a section that contains a subsection which in turn contains the content of the reuse component.

By default, these sections are added to the end of the object page in the same order as the reuse component instances are defined in the manifest. You can use the SAPUI5 Visual Editor to move these sections.

Adding a Reuse Component as a Sub-Section to an Existing Section

You can add a reuse component to any existing section as a final subsection, using the manifest property `"leadingSectionIdOrPath"` to link it with the desired section. In the annotations, collection facets are used with their IDs, and reference facets are used with their annotation paths. This means you need to provide either the ID or the path of the required facet as a value to this property. During templating, the reuse component is attached as a subsection to the section. See an example manifest below:

Sample Code

```
"embeddedComponents": {
                    "tableTest": {
                        "id":"tableTest",
                        "componentName": "STTA_MP.reuseComponents.tableTest",
                        "title": "{{SalesPriceReuse}}",
                        "leadingSectionIdOrPath": "GeneralInformation",
                        "settings": {
                            "navigationProperty": "to_ProductSalesPrice"
                        }
                    }
                }
```

Grouping Reuse Components into a Single Section

You can group reuse components into a single section. Each reuse component has an ID. You can choose one to be the leading component, and the others can be grouped with it. See the example manifest below.

Use the `"groupTitle"` property for the title of the grouped reuse components. If there is no title, use the existing title as the group (section) title. If the configuration is incorrect, for example, if a reuse component that has already been used, is used in a grouping, the fallback solution is to show it as a separate section.

Sample Code

```
"embeddedComponents": {
                    "stateTest": {
                        "id": "stateTest",
                        "componentUsage": "stateTest",
                        "title": "{{stateReuse}}",
                        "groupTitle":"{{reuseGroupTitle}}"
                    },
                    "situationsTest": {
                        "id": "situationsTest",
                        "componentName":
"STTA_MP.reuseComponents.situationsTest",
                        "title": "{{situationsReuse}}",
                        "settings": {
                            "productKey": "{ProductForEdit}"
                        },
                        "leadingSectionIdOrPath": "stateTest",
                    }
                }
```

Reuse components grouped into a subsection in a existing section can be moved within the section, using the SAP Visual Editor. You can move all reuse components grouped together as a single section within the section.

Hiding Reuse Components via API

You can use the extension API `SetSectionHidden` to hide reuse components. Use this API to hide only reuse components, not to hide other sections.

`sap.suite.ui.generic.template.ObjectPage.extensionAPI.ExtensionAPI.SetSectionHidden` accepts only one boolean argument, either `"true"` or `"false"`.

Call this API in the component.js of the reuseable components.

Sample Code

```
function fnRegisterOnPageDataLoaded(oExtensionAPI){
        oExtensionAPI.attachPageDataLoaded(function(oEvent){
            var oContextData = oEvent.context.getObject();
            (oExtensionAPI.setSectionHidden || jQuery.noop)
(oContextData.ProductCategory !== "Notebooks");
        });
    }
```

Refreshing Reuse Components

Reuse components expose a `stRefresh` function which is called by the SAP Fiori elements framework to refresh the component. You can trigger it using the manifest property `stRefreshTrigger`.

For example, if the reuse component is to react to the value change of more than one property, such as a price change, then annotate a side effect and add the target property`PriceTrigger`, as shown in the sample below.

This can also be achieved via a function import, by annotating a side effect against a function import.

Sample Code

Trigger refresh when changing the value of a property

```
"embeddedComponents": {
          "priceComponentEmbedding": {
             "id": "priceComponentEmbedding",
             "componentUsage": "priceComponentUsage",
             "title": "{{I18N_KEY_FOR_PRICE_COMPONENT_TITLE}}",
             "settings": {
                  "stRefreshTrigger": "{Price}"
             }
       },
```

Sample Code

Trigger refresh when changing the values of one of the properties

```
"embeddedComponents": {
          "priceComponentEmbedding": {
             "id": "priceComponentEmbedding",
             "componentUsage": "priceComponentUsage",
             "title": "{{I18N_KEY_FOR_PRICE_COMPONENT_TITLE}}",
             "settings": {
                  "stRefreshTrigger": "{Price}{Supplier}"
             }
       },
```

Sample Code

Trigger refresh when chaning the values of a combination of properties

```
"embeddedComponents": {
          "priceComponentEmbedding": {
             "id": "priceComponentEmbedding",
             "componentUsage": "priceComponentUsage",
             "title": "{{I18N_KEY_FOR_PRICE_COMPONENT_TITLE}}",
             "settings": {
                  "stRefreshTrigger": "{= ${Price}+${Supplier}}"
             }
       },
```

Developing Reuse Components

Follow these guidelines when developing reuse components that are to be included as sections in object pages.

General Guidelines

As a provider of a reuse component that is used in several apps, create a library which can be included by the apps.

When providing a one-time reuse component provide the reuse component within the same project that implements the application that uses the reuse component.

Technically, a reuse component is a subclass of UIComponent which calls ReuseComponentSupport.mixInto within its `init()` method.

> **i** Note
>
> - The reuse component must not define any method starting with the prefix st or _st. The only exceptions are the methods defined in this documentation. These methods have to be overwritten so you can use the functions provided by SAP Fiori elements.
> - If you want to use the component model functionality providedy by SAP Fiori elements, note that the method name `getComponentModel` is reserved by SAP Fiori elements.
> - The reuse component can define (API) properties on its own. These properties can be used to communicate with the application that embeds the reuse component. However, certain property names are predefined by SAP Fiori elements. These properties have specific semantics and cannot be used to communicate with the embedding application.

Reuse Components and Reuse Component Instances

A **reuse component** is a software artifact that can be used to embed a UI module into an object page . You can embed the same reuse component several times in the same app or even in one object page.

A **reuse component instance** is one occurence of a reuse component within an application.

This means that the implementation of a reuse component must not store any information that should be considered at instance level, in a singleton object.

Handling Properties

A reuse component may expose properties that can be used to communicate with the embedding application. For each of these properties, setter and getter methods are created automatically. The embedding application does not communicate with the reuse component instance by directly calling these methods. It does not even have access to the reuse component instance. For more information, see Including Reuse Components on an Object Page [page 1721] .

The reuse component has two options for consuming the properties:

Option 1: Overriding the Setters

If the reuse component is to react via coding to changes of its properties, override the setter method of the corresponding property within the reuse component.

Option 2: Component Model

When calling the `ReuseComponentSupport.mixInto` method, the reuse component can provide an optional second (string) parameter. If this parameter is not faulty, it is used as the name of the JSON model this is automatically attached to the reuse component. You can retrieve this model from the reuse component by calling the `getComponentModel()` method that is automatically added to the reuse component.

The component model may contain arbitrary properties. However, if any of the reuse component's properties is changed, the corresponding property in the component model is changed accordingly. This can be used in particular for declaratively using the property values within binding definitions. You can also achieve this by creating a corresponding property binding and attach a change handler to this binding.

> **i Note**
>
> If a property in the JSON model is modified by other means, this change is not transferred to the corresponding property of the reuse component. If you want to transfer the change, you can use the `bTwoWaySync` parameter of the `ReuseComponentSupport.mixInto` method. Handle this functionality with care, as there is a risk of accidentally modifying data in the enclosing application.
>
> The reuse component may also use the component model to handle additional properties that aren't exposed as properties of the reuse component. These properties are controlled exclusively by the reuse component. They cannot be used to communicate with the embedding application. However, they can be used for data transfer between declarative and programmatic parts of the implementation of the reuse component.

Predefined Properties

Although some properties are predefined by the SAP Fiori elements framework, the reuse component must declare them in the metadata. The values are then provided by the SAP Fiori elements framework.

Property names with the prefix st are reserved for the SAP Fiori elements framework. These properties are supported:

- `uiMode`
  The possible values for this property are "Create", "Display", and "Edit". Note that this property should not provide information for setting up backend requests (for example, whether the active or the draft version of the object is currently displayed). Use this property only to provide the correct mode for controls within the reuse component (for example, input fields).
- `semanticObject`
  The semantic object displayed on the current object page.
- `stIsAreaVisible`
  This boolean property is set to true if the reuse component is in the visible area of the screen.
  We recommended postponing any performance-critical action if the value of this property is false. In this case, the action should be performed only if the value of this property is true (if it is still necessary). This type of system behavior is called lazy loading.

The following example shows a snippet of a reuse component that uses the predefined properties and an additional property `documentNumber` that is used to pass a key to the reuse component:

> **Sample Code**

```
sap.ui.define([
 "sap/ui/core/UIComponent",
 "sap/suite/ui/generic/template/extensionAPI/ReuseComponentSupport"
], function(UIComponent, ReuseComponentSupport) {
 "use strict";
 /* Definition of the reuse component */
 return UIComponent.extend("MyReuseComponent", {
  metadata: {
   manifest: "json",
   library: "myLibrary",
   properties: {
    /* Standard properties for reuse components */
```

```
    uiMode: {
     type: "string",
     group: "standard"
    },
    semanticObject: {
     type: "string",
     group: "standard"
    },
    stIsAreaVisible: {
     type: "boolean",
     group: "standard"
    },

    /* Component specific properties */

    documentNumber: {
     type: "string",
     group: "specific",
     defaultValue: ""
    }
   }
  },

  // Standard life time event of a component. Used to transform this
component into a reuse component for Fiori Elements
  init: function(){
    //Transform this component into a reuse component for Fiori Elements:
    ReuseComponentSupport.mixInto(this, "myPropertiesModelName");
    // Defensive call of init of the super class:
    (UIComponent.prototype.init || jQuery.noop).apply(this, arguments);
  }
 });
});
```

Now the reuse component can declaratively use all the properties defined above, as shown in the following example:

Sample Code

```
<sfo:SmartForm xmlns:sfo="sap.ui.comp.smartform"
    editable="{= ${myPropertiesModelName>/uiMode} !== 'Display' }"
</sfo:SmartForm>
```

If the reuse component is to adapt to changes of the property programmatically, two alternative techniques can be used. They are shown in the following code snippet:

Sample Code

```
sap.ui.define([
 "sap/ui/core/UIComponent",
 "sap/suite/ui/generic/template/extensionAPI/ReuseComponentSupport"
], function(UIComponent, ReuseComponentSupport) {
 "use strict";
 /* Definition of the reuse component */
 return UIComponent.extend("MyReuseComponent", {
  metadata: {
   manifest: "json",
   library: "myLibrary",
   properties: {
    /* Standard properties for reuse components */
    uiMode: {
     type: "string",
     group: "standard"
```

```
    },
    semanticObject: {
     type: "string",
     group: "standard"
    },
    stIsAreaVisible: {
     type: "boolean",
     group: "standard"
    },

    /* Component specific properties */

    documentNumber: {
     type: "string",
     group: "specific",
     defaultValue: ""
    }
   }
  },

  // Technique 1: Redefine the predefined setter-method
  setStIsAreaVisible: function(bIsAreaVisible){
   if (bIsAreaVisible !== this.getStIsAreaVisible()){
     this.setProperty("stIsAreaVisible", bIsAreaVisible); // ensure that the
property is updated accordingly
     // ... (any code that wants to consume the changed visibility of the
reuse component)
   }
  },

  // Standard life time event of a component. Used to transform this
component into a reuse component for Fiori Elements and do some initialization
  init: function(){
   // Defensive call of init of the super class:
   (UIComponent.prototype.init || jQuery.noop).apply(this, arguments);
    //Transform this component into a reuse component for Fiori Elements:
    ReuseComponentSupport.mixInto(this, "myPropertiesModelName");
    // Technique 2: Attach to changes of the component model
    var oMyPropertiesModel = this.getComponentModel();
    var oPropertyBinding = oMyPropertiesModel.bindProperty("/documentNumber");
    oPropertyBinding.attachChange(function(){
      var sDocumentNumber = oMyPropertiesModel.getProperty("/documentNumber");
      // ... (any code that wants to consume the changed document number)
    });
  }
 });
});
```

Model Propagation

The unnamed model (the OData model) and a JSON model named 'ui' that are defined in the application are propagated to the reuse component.

> **i** Note
>
> Additional JSON models which are used internally may also be propagated to the reuse component. However, the reuse component must not access these models in any way.

The binding context for the unnamed model is already preset. By default, if the reuse component is used in an object page, the binding context of the object page is propagated to the reuse component. The embedding

application may add a relative binding to the definition of the embedding. This modifies the binding context of the reuse component accordingly.

> **i Note**
>
> The propagation of the unnamed model is especially important if the data that is accessed by the reuse component is provided by the same OData service that is used by the embedding application.

Change Events for the Reuse Components

For the implementation of the reuse component to be able to react to a change of the object key attached to the page it is embedded in , ensure that the model propagation has already taken place. The reuse component can then rely on the fact that the bindings of all controls have already been updated accordingly. In this case the reuse component can implement the `stRefresh(oModel, oBindingContext, oExtensionAPI)` method. This method is called if at least one of the following occurs:

- A new binding context is defined for the embedding page
- The binding context of the embedding page is forced to refresh
- The key information for the embedding page changes

Note that method is called only after the model propagation to the reuse component.

The reuse component might also implement the `stStart(oModel, oBindingContext, oExtensionAPI)` method. This method is called at the same time as described for `stRefresh`, but `stStart` is only called once during the lifetime of the app. If both `stStart` and `stRefresh` have been implemented for the first occurence of the specified events, `stStart` is called. For all other occurences, `stRefresh` is called.

Note that the signature of the `stStart` and `stRefresh` methods are identical:

- oModel is the standard OData model of this app
- oBindingContext is the binding context of the current page. Note that this might be faulty, for example, if the reuse component is embedded on an object page that is currently in create mode in a non-draft app.
- oExtensionAPI is the instance of the extension API attached to this reuse component (instance).

If the reuse component is to be used in change scenarios, we recommend implementing the `stRefresh` method. In this case, the framework might call this method to indicate that the reuse component should refresh its data (even if the object instance displayed on the embedding object page did not change).

Lazy Loading and Refresh

The following coding example shows how lazy loading and the refresh function are combined:

> **Sample Code**

```
sap.ui.define([
 "sap/ui/core/UIComponent",
 "sap/suite/ui/generic/template/extensionAPI/ReuseComponentSupport"
], function(UIComponent, ReuseComponentSupport) {
 "use strict";
```

```
 /* Definition of the Reuse Component */
 return UIComponent.extend("MyReuseComponent", {
  metadata: {
   manifest: "json",
   library: "myLibrary",
   properties: {
    stIsAreaVisible: {
     type: "boolean",
     group: "standard"
    },
  },

  setStIsAreaVisible: function(bIsAreaVisible){
   if (bIsAreaVisible !== this.getStIsAreaVisible()){
     this.setProperty("stIsAreaVisible", bIsAreaVisible); // ensure that the
property is updated accordingly
     this.refreshImpl();
   }
  },

  // Standard life time event of a component. Used to transform this
component into a Reuse Component for Fiori Elements and do some initialization
  init: function(){
   // Defensive call of init of the super class:
   (UIComponent.prototype.init || jQuery.noop).apply(this, arguments);
    //Transform this component into a reuse component for Fiori Elements:
    ReuseComponentSupport.mixInto(this);
  },
  stRefresh: function(oModel, oBindingContext, oExtensionAPI) {
    this.oModel = oModel;
    this.oBindingContext = oBindingContext;
    this.oExtensionAPI = oExtensionAPI;
    this.refreshImpl();
  },

  refreshImpl: function(){
     if (this.oBindingContext && this.getStIsAreaVisible()){
       // ... (any code that loads the data for the Reuse Component according
to this.oBindingContext)
       this.oBindingContext = null;
     }
  }
 });
});
```

Extension API

To each reuse component instance, an instance of the extension API is attached. This instance is a variant of the extension API which is provided for the corresponding object page or canvas page. It is passed to the `stStart` and `stRefresh` method as a third parameter.

The `ReuseComponent.mixInto` method which has to be called by every reuse component within its `init()` method returns a Promise that resolves to this instance of the extension API.

Compared to the standard extension API attached to the current page, the extension API instance attached to the reuse component instance has the following modifications:

- The `getNavigationController()` method provides a modified version of the NavigationController which is also able to navigate to the pages defined in the reuse component and shares a common communication object with these pages.

- The `getCommunicationObject()` method, when called with no parameter (or a number not greater 0 as a parameter) behaves as usual. Calling `getCommunicationObject(1)` provides a communication object that can be used for communication with the pages defined in the reuse component. The subpages can retrieve this communication object via calling `getCommunicationObject()` (without parameters).
- If the reuse component is embedded in an object page, the extension API has an additional `setSectionHidden(bHidden)` method that can be used to show or hide the section implemented by the reuse component instance.

Reuse Components with Subpages

You can define additional pages for a reuse component. You can add them to the embedding app and use them to provide navigation within the reuse component.

The embedding application needs to provide the definitions of the possible subpages within its manifest. That is, the provider of the reuse component has to publish a snippet that defines the subpages of the reuse component, and every consumer of the reuse component has to add this snippet to the declaration of his useof the reuse component.

The definition to be added within the definition of the reuse component must have the same structure as the one that is required for the normal way the subpages of an object page are defined within the pages structure of the manifest. See the following example for this:

Sample Code

```
...
"embeddedComponents": {
    "myComponentEmbedding": {
        "id": "myComponentEmbedding",
        "componentName": "theEmbeddedComponent",
        "title": "{{I18N_KEY_FOR_REUSE_COMPONENT_TITLE}}",
        "settings": {
            "documentNumber": "{documentNumber}"
            },
        "pages":{
            // add sub-pages here
        }
    }
}
```

Using Standard Object Pages as Subpages of a Reuse Component

Optimally, use a standard object page as a subpage for a reuse component. The manifest looks like this:

Sample Code

```
...
"embeddedComponents": {
    "myComponentEmbedding": {
        "id": "myComponentEmbedding",
        "componentName": "theEmbeddedComponent",
        "title": "{{I18N_KEY_FOR_REUSE_COMPONENT_TITLE}}",
        "settings": {
            "theNavigationPropertyName":
"myNavigationProperty"         },
        "pages":{
```

```
                "ObjectPage|myNavigationProperty":{
                "navigationProperty": "myNavigationProperty",
                "entitySet": "myDependingEntitySet",
                "component": {
                    "name": "sap.suite.ui.generic.template.ObjectPage"
                }
            }
        }
    }
}
```

There should be a navigation property called `myNavigationProperty` from the entity set of the object page hosting the reuse component, to `myDependingEntitySet`.

You need to explicitly trigger the navigation to the subpage. Use NavigationController.navigateInternal() for this purpose, as shown below:

Sample Code

```
var oBindingContext = oEvent.getSource().getBindingContext();  // or any
other way you have to get a binding context for the target page
var oNavigationController = oExtensionAPI.getNavigationController(); //
oExtensionAPI is the instance of the extension api which has been discussed
above
var sNavigationProperty =
this.getOwnerComponent().getTheNavigationPropertyName();   // retrieve the
name of the navigation property (see comments below)
var oNavigationData = {
    navigationProperty: sNavigationProperty
};
oNavigationController.navigateInternal(oBindingContext, oNavigationData);
```

The name of the navigation property (in this case, `myNavigationProperty`) must be available for the developer of the reuse component in order to trigger the navigation. However, this name is part of the OData service of the embedding application. To achieve a loose coupling, the developer of the reuse component does not need to hard-code this name. The reuse component should have a `theNavigationPropertyName` property which is used to transfer the name of the navigation property to the reuse component.

General Structure of a Reuse Component Project

A reuse component that can be used within a SAP Fiori elements-based application cannot be used in a freestyle application. Provide two separate components that refer to a common implementation.

Stable IDs

For some purposes it is helpful if the IDs of the controls used within a reuse component instance are stable. As a prerequisite, the ID of the view hosting the content of the reuse component is stable. You should define this view declaratively. Specify the view within the manifest.json of the reuse component.

Configuring Tables

You can use the annotations and entries in the manifest.json to control various aspects of tables.

For information on smart tables, see also: https://sapui5.hana.ondemand.com/sdk/#/api/sap.ui.comp.smarttable.SmartTable and Tables: Which One Should I Choose? [page 2286]

Setting the Table Type

In the `manifest.json` file, and through annotations, you can control which table type is rendered in the list report and on the object page.

These are the `type` properties available within `tableSettings` :

- `ResponsiveTable`
- `GridTable`
- `AnalyticalTable`
- `TreeTable`

> **i Note**
>
> - Grid tables, analytical tables, and tree tables cannot be rendered on smartphones. On smartphones, always responsive tables are shown.
> - List report only: If the `type` property within `tableSettings` is `AnalyticalTable`, set the annotation `sap:semantics` to `aggregate` for the specified entity type. Note that `sap:semantics` is a back-end entity type definition and cannot be changed in the SAP Web IDE.
> - If you do not maintain the `type` property within `tableSettings` , and if `sap:semantics`to `aggregate` has been set in the back-end, an analytical table is rendered.

Examples

Set the `type` property within `tableSettings` to the required value in the `sap.ui.generic.app` section of the manifest.json:

Example for the list report:

```
"sap.ui.generic.app": {
            "pages": [{
                "entitySet": "Zfarvd_Bs_Hd_Bo",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true,
                    "settings": {
                        "tableSettings": {
                            "type": "GridTable"
                            }
                    }
                },
```

Examples for the object page:

```
"sap.ui.generic.app": {
        "pages": [
            {
                "entitySet": "STTA_C_MP_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true
                },
                "pages": [
                    {
                        "entitySet": "STTA_C_MP_Product",
                        "component": {
                            "name": "sap.suite.ui.generic.template.ObjectPage"
                            "settings": {
                                "sections": {

"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
                                        "navigationProperty": "to_ProductText",
                                        "entitySet": "STTA_C_MP_ProductText",
                                        "tableSettings": {
                                    "type": "TreeTable"
                                 }
                                    }
                                }
                            }
                        },
```

Defining `tableTypes` under the settings is supported for backward compatibility. However, using `tableSetings` is recommended for defining table types.

> **i Note**
>
> If you maintain the `type` property within `tableSettings` in `sections`, it has a higher priority than the `type` property of `tableSettings` in Object Page .

```
"pages": [{
                "entitySet": "STTA_C_MP_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ObjectPage",
                    "settings": {
                        "showRelatedApps": true,
                        "tableSettings": {
                            "type":"ResponsiveTable"
                        },
                        "sections": {

"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
                                "navigationProperty": "to_ProductText",
                                "entitySet": "STTA_C_MP_ProductText",
                                "tableSettings": {
                                    "type":"GridTable"
                                }
                            }
                        }
                    }
```

More Information

For a description of the available table types, see Smart Tables [page 1628].

For information about setting up tables in the list report through annotations, see Settings for List Report Tables [page 1761].

For information about setting up tables in the object page, see Settings for Object Page Tables [page 1765].

Setting the Smart Table Header

The header of the smart table is set using `com.sap.vocabularies.UI.v1.HeaderInfo TypeNamePlural`.

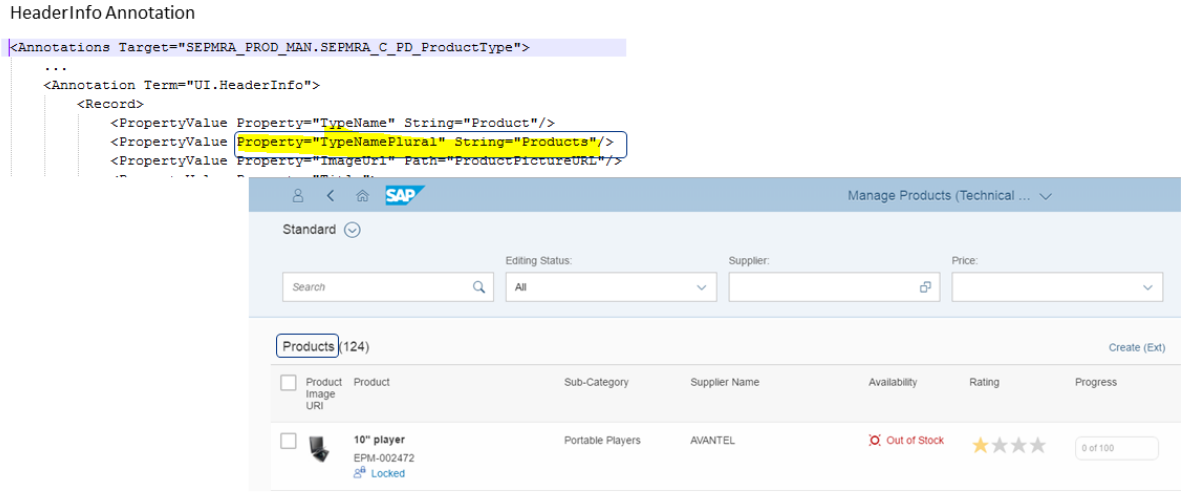


Figure 307: List Report: Page Header

Defining the Default Sort Order

You can define whether the default sort order for tables is ascending or descending by using the `com.sap.vocabularies.UI.v1.PresentationVariant` annotation term and the `SortOrder` property.

Sample Code

Presentation Variant

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="ZFAR_CUSTOMER_LINE_ITEMS2_SRV.Item">
    <Annotation Term="com.sap.vocabularies.UI.v1.PresentationVariant">
        <Record>
            <PropertyValue Property="Visualizations">
                <Collection>
                    <AnnotationPath>@UI.LineItem</AnnotationPath>
                </Collection>
            </PropertyValue>
            <PropertyValue Property="RequestAtLeast">
                <Collection>
```

```
                    <PropertyPath>Customer</PropertyPath>
                    <PropertyPath>CompanyCode</PropertyPath>
                </Collection>
            </PropertyValue>
            <PropertyValue Property="SortOrder">
                <Collection>
                    <Record>
                        <PropertyValue Property="Property"
PropertyPath="CompanyCode"/>
                        <PropertyValue Property="Descending" Bool="true"/>
                    </Record>
                    <Record>
                        <PropertyValue Property="Property"
PropertyPath="Customer"/>
                    </Record>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

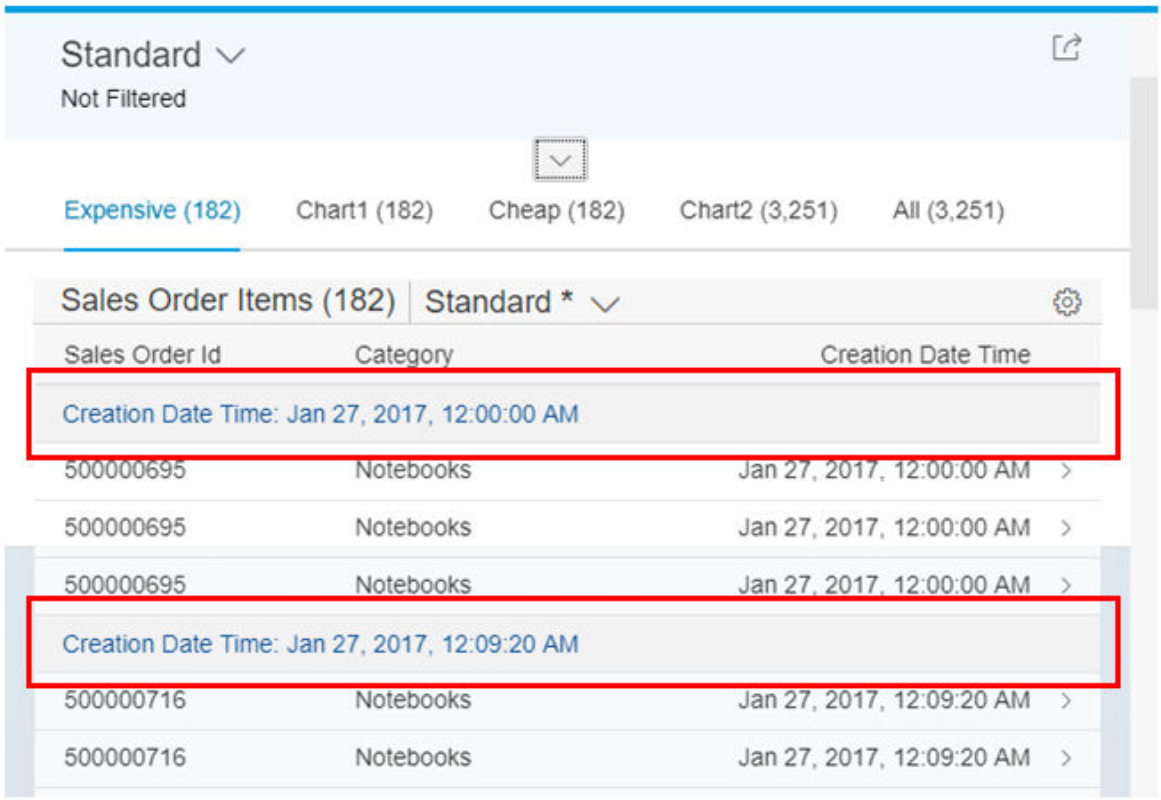
If no sort order is defined, the property value is `Ascending`.

Table Groupings

SAP Fiori elements handle table groupings automatically.

The grouping of rows is supported in responsive tables (`sap.m.Table`) and analytical tables (`sap.ui.Table.AnalyticalTable`). Grid tables and tree tables are not supported.

Table grouping of rows looks like this:



Grouping Header Text

These group header formatters are automatically set by SAP Fiori elements:

- Responsive table: Set a grouping function on the sorter: `fnGroup of sap.ui.model.Sorter`.
- Analytical table: Set property `groupHeaderFormatter` of `sap.ui.table.AnalyticalColumn`.

Grouping Header Types

SAP Fiori elements supports these group header formatters:

OData Type	SAP Display Format	OData Example	Unformatted	Formatted	Comments
Edm.DateTimeOffsetDate	Date	/Date(1485471600000 +0000)/	Fri Jan 27 2017 00:00:00 GMT +0100 (Central European Standard Time)	Jan 26, 2017	UTC

OData Type	SAP Display Format	OData Example	Unformatted	Formatted	Comments
Edm.DateTimeOffset		/Date(1485471600000 +0000)/	Fri Jan 27 2017 00:00:00 GMT +0100 (Central European Standard Time)	Jan 27, 2017, 12:00:00 AM	
Edm.DateTime	Date	/Date(1485471600000 +0000)/	Fri Jan 27 2017 00:00:00 GMT +0100 (Central European Standard Time)	Jan 26, 2017	UTC
Edm.DateTime		/Date(1485471600000 +0000)/	Fri Jan 27 2017 00:00:00 GMT +0100 (Central European Standard Time)	Jan 27, 2017, 12:00:00 AM	
Edm.Time	Time	PT11H13M01S	[object Object] is: {ms: 43980000, __edmType: "Edm.Time"}	11:13:01 AM	
Edm.String	Date	20180313	20180313	Mar 13, 2018	
Edm.Boolean		true	true	Yes	
Edm.Decimal		10	10	10.000 m*	Unit of measure
Edm.Decimal		2498.00	2498.00	2,498.00 EUR*	Currency
Edm.String		m	m	Meter (m)*	Unit of measure and TextArrangement
Edm.String		EUR	EUR	Euro (EUR)*	Currency and TextArrangement

* Units of measure or currencies in the grouping header are supported only in responsive tables.

Limitations

For analytical tables, take the following limations into account:

- Grouping can only be activated for dimension columns.
- Grouping takes place on the server. The header formatter has no effect on the grouping itself. This means that groups stay separate even if they have the same header after formatting.
- The `TextArrangement` annotation is not supported. A property with a text property is displayed as follows: “m – Meter”. For more information, see TextArrangement.
- Displaying values with units of measure is not supported.

Enabling Multiple Selection in Tables

Single selection in tables is enabled by default. If you want, you can enable multi-select..

This allows users to select multiple items from list report or object page tables, and then choose a custom action button. After triggering the action, the user stays on the same page (no navigation). Note that the custom action button cannot have action parameters and that the results have to be displayed on the page on which the action was triggered.

List Report Settings

When `multiSelect` is set to `true` in the `manifest.json` file of a list report table, the table switches from single-select to multi-select, as shown in the this sample code:

```
"sap.ui.generic.app": {
  "_version": "1.3.0",
  "pages": {
  "ListReport|STTA_C_MP_Product": {
    "entitySet": "STTA_C_MP_Product",
    "component": {
    "name": "sap.suite.ui.generic.template.ListReport",
    "list": true,
    "settings" : {
      "tableSettings": {
        "multiSelect": true
      }
    }
```

Object Page Settings

You have two options:

- You can enable multiple selection at object page level for all tables on the object page.
  To do so, set `multiSelect` to `true` in the `manifest.json` file of your object page.

  ```
  "pages": {
    "ObjectPage|STTA_C_MP_Product": {
    "entitySet": "STTA_C_MP_Product",
    "name": "sap.suite.ui.generic.template.ObjectPage",
      "settings": {
        "showRelatedApps": true,
        "editableHeaderContent": true,
        "tableSettings": {
          "multiSelect": true
        },
        "sections": {
          "to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
            "navigationProperty": "to_ProductText",
            "entitySet": "STTA_C_MP_ProductText",
            "createMode": "inline"
          }
  ```

- You can enable multiple selection at table level, that is, individually for each table.

To do so, set `multiSelect` to `true` for a specific table in the `manifest.json` file of your object page:

```
"pages": {
  "ObjectPage|STTA_C_MP_Product": {
  "entitySet": "STTA_C_MP_Product",
  "name": "sap.suite.ui.generic.template.ObjectPage",
    "settings": {
      "showRelatedApps": true,
      "editableHeaderContent": true,
      "sections": {
        "to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
          "navigationProperty": "to_ProductText",
          "entitySet": "STTA_C_MP_ProductText",
          "createMode": "inline",
          "tableSettings": {
            "multiSelect": true
          }
        }
```

MultiSelectionPlugin

Use the `MultiSelectionPlugin`for grid tables, analytical tables, and tree tables. The *Select All* button is disabled by default. If you want to enable it, set `selectAll : true`. If `selectAll : false`, the following applies:

- The user can still select a range. If new data needs to be loaded from the back-end system for this, the number of lines to be loaded is restricted to the specified limit. The default value for this limit is 200. If `selectAll : true`, the selection limit is not evaluated.
- The *Undo Selection* button is displayed instead of the *Select All* button. The user can choose this button to reset all selections.

> **i Note**
>
> If the user chooses *Select All*, the system loads all data from the back-end system, possibly in multiple sequential requests. For performance reasons, set `selectAll : true` only if the expected amount of data is not too high.

For more information, see MultiSelectionPlugin and the sample for the MultiSelectionPlugin.

The following sample code shows example table settings for a `GridTable` with `MultiSelectionPlugin`. You can place these `"tableSettings"` underneath `"settings"` at list report level as well as at object page level.

> **Sample Code**

```
    "tableSettings": {
          "type": "GridTable",
          "multiSelect": true,
          "selectAll": false,
          "selectionLimit": 150
    }
```

Adding Line Item Actions in Tables

You can add a line item action button as a column within the `SmartTable` control in both the list report and the object page.

Line item actions are used to trigger actions directly for a single table row. The following types of line item actions are supported:

- Actions that trigger a back-end call through the OData service, for example, *Approve* or *Unblock*, represented by the complex type `DataFieldForAction`
- Actions that trigger navigation, for example, to a different app, represented by the complex type `DataFieldForIntentBasedNavigation`

To add a line item action to a smart table, in the annotation term `UI.LineItem`, set the `Inline` property to `true` for the complex types listed above. The line item actions are then displayed as shown below:

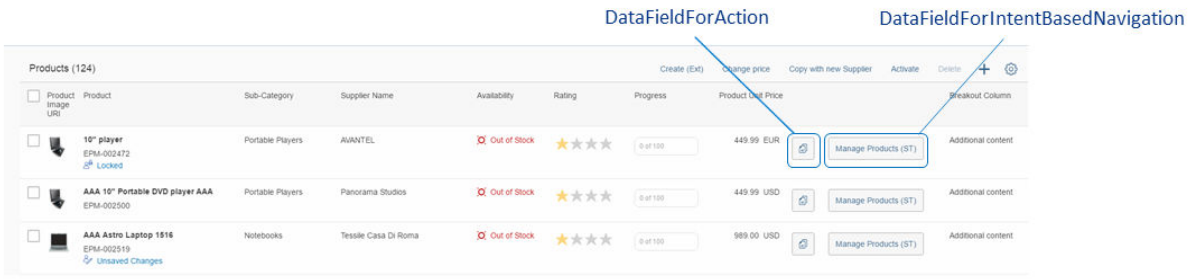


Figure 308: Line Item Actions (List Report Shown)

Code Sample

The following code sample shows how to create your annotations for line item actions. Note that the `UI.LineItem` vocabulary term is used to define the columns for the smart table.

UI.LineItem

```
<Annotation Term="UI.LineItem">
   <Collection>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy with new Supplier"/>
         <PropertyValue Property="Action"
            String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
STTA_C_MP_ProductCopywithparams"/>
         <Annotation Term="UI. InvocationGrouping"
            EnumMember="UI.OperationGroupingType/Isolated"/>
      </Record>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Activate"/>
         <PropertyValue Property="Action"
            String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
STTA_C_MP_ProductActivation"/>
         <Annotation Term="UI. InvocationGrouping"
            EnumMember="UI.OperationGroupingType/ChangeSet"/>
      </Record>
      <Record Type="UI.DataField">
         <PropertyValue Property="Value" Path="Product"/>
         <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
         <PropertyValue Property="Value" Path="ProductCategory"/>
```

```
         <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
         <PropertyValue Property="Value" Path="to_Supplier/CompanyName"/>
         <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
         <PropertyValue Property="Criticality" Path="to_StockAvailability/
StockAvailability"/>
         <PropertyValue Property="Value" Path="to_StockAvailability/
StockAvailability"/>
         <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
         <PropertyValue Property="Value" Path="Price"/>
         <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy"/>
         <PropertyValue Property="Action"
            String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopy"/>
         <PropertyValue Property="Inline" Bool="true"/>
         <Annotation Term="UI. InvocationGrouping"
            EnumMember="UI.OperationGroupingType/Isolated"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
         <PropertyValue Property="Label" String="Manage Products (ST)"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct"/>
         <PropertyValue Property="Action" String="manage_st"/>
         <PropertyValue Property="Inline" Bool="true"/>
      </Record>
      <Record Type="UI.DataFieldWithIntentBasedNavigation">
         <PropertyValue Property="Label" String="Weight (with IBN)" />
         <PropertyValue Property="Value" Path="Weight"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct" />
         <PropertyValue Property="Action" String="manage_st"/>
      </Record>
   </Collection>
</Annotation>
```

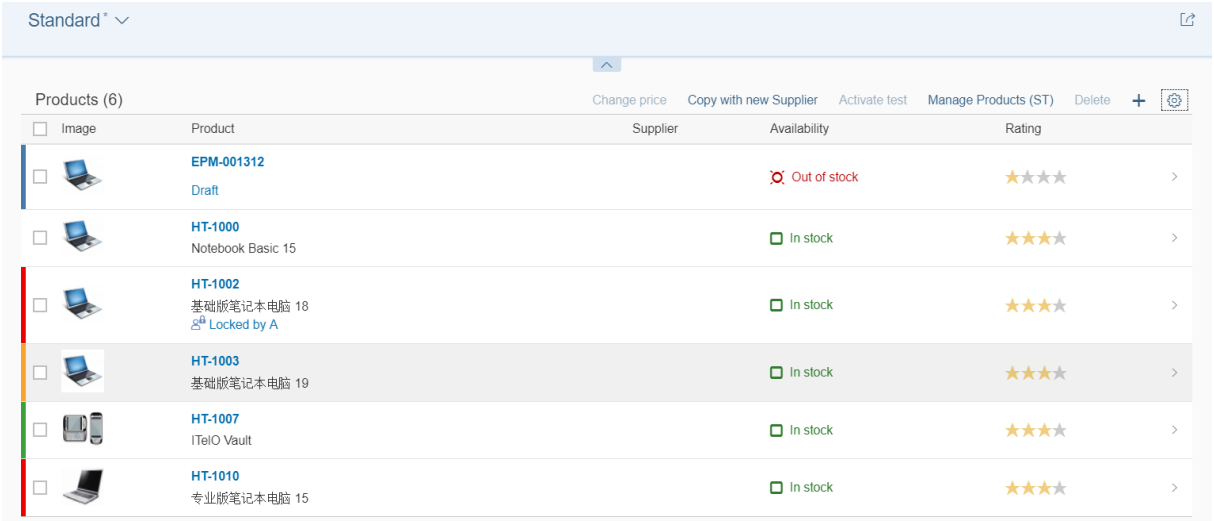
In the example above, the order in which the record types are presented in the annotation determines the order in which they appear in the table columns:

- For the first two record types, the `DataFieldForAction` complex type does not contain the `Inline` property, which means that the action button will appear in the smart table toolbar. If the `Inline` property is there and set to `false`, the action button is also displayed in the smart table toolbar.
- With the next five record types, the `DataField` complex type is used to define the data for a column within the smart table.
- With the last two record types, the `DataFieldForAction` and `DataFieldForIntentBasedNavigation` complex types are used and contain the `Inline` property which is set to `true`. This means the action buttons will appear in every row in the appropriate column within the smart table.
- With the last record type, the `DataFieldWithIntentBasedNavigation` complex type is used to render the property value as a link allowing for navigation to the semantic object.

Highlighting Line Items Based on Criticality

You can add semantic highlights to line items in tables, based on their criticality.

The figure below shows an example of this:



To do so, add a `LineItem` criticality annotation for the line items of the entity type that is used by a table, as follows:

Sample Code

```
<Annotation Term="UI.LineItem">
  <Annotation Term="UI.Criticality"
Path="Element_transporting_criticality_of_complete_LineItem" />          //
LineItem Criticality annotation
  <Collection>
    <Record Type="UI.DataField">
      ...
    </Record>
  </Collection>
</Annotation>
```

If this annotation has been defined based on the criticality value received for the corresponding line item, the following highlights are displayed:

- 0 - None (no color)
- 1 - Error (red)
- 2 - Warning (yellow)
- 3 - Success (green)

Note

Newly created line items in draft status are always highlighted in blue. After saving the line item, blue is replaced by the color for the criticality of the line item.

Adding a Rating Indicator to a Table

You can add a read-only rating indicator to a table.

The rating indicator allows you to visually represent the value of a field from the back-end system with the corresponding number of stars (configurable). This field can indicate a rating or classification for a specific object or item.

Decimal values are rounded up or down accordingly. If a value falls between x.25 and x.74, a half star is displayed.

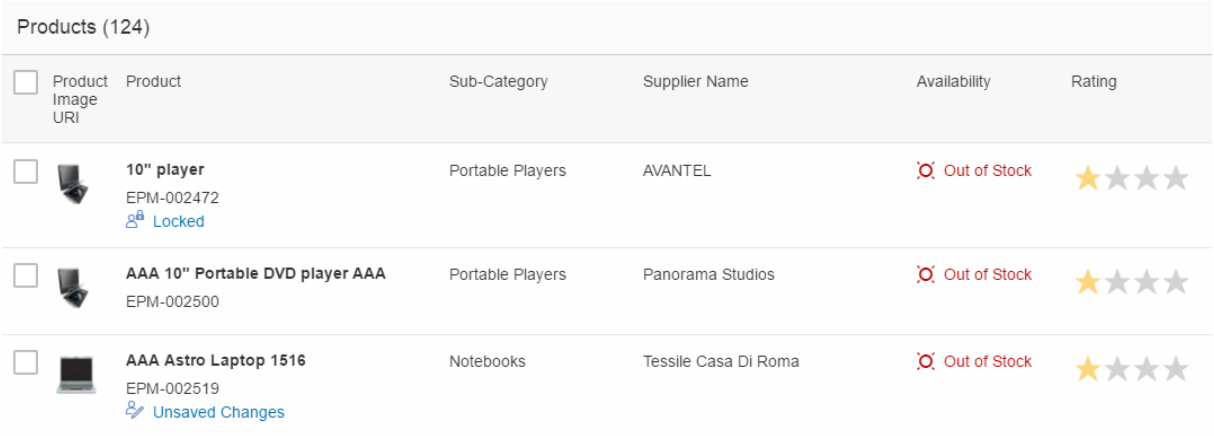


Figure 309: Rating Indicator in Table

Code Samples (OData Annotations)

Use the following annotations to enable the rating indicator and define the maximum number of stars:

Sample Code

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
              <Annotation Term="UI.DataPoint" Qualifier="Rating">
                            <Record>
                                          <PropertyValue Property="Value"
Path="to_StockAvailability/StockAvailability" />
                                          <PropertyValue
Property="TargetValue" Decimal="4" />
                                          <PropertyValue
Property="Visualization" EnumMember="UI.VisualizationType/Rating" />
                            </Record>
              </Annotation>
              <Annotation Term="UI.LineItem">
                            <Collection>
                                          <Record
Type="UI.DataFieldForAnnotation">
                                                        <PropertyValue
Property="Label" String="Rating" />
                                                        <PropertyValue
Property="Target" AnnotationPath="@UI.DataPoint#Rating" />
                                          </Record>
                                          …
                            </Collection>
                            …
              </Annotation>
              …
</Annotations>
```

> **Note**
>
> The `TargetValue` property defines the maximum number of stars. In this case, it is 4.

Adding a Progress Indicator to a Table

You can add a progress indicator to a table.

The progress indicator allows you to visually represent the level of completion of a project or a goal, for example.

It can be used to express completion values either as a percentage or as absolute numbers (for example, 8 of 10).

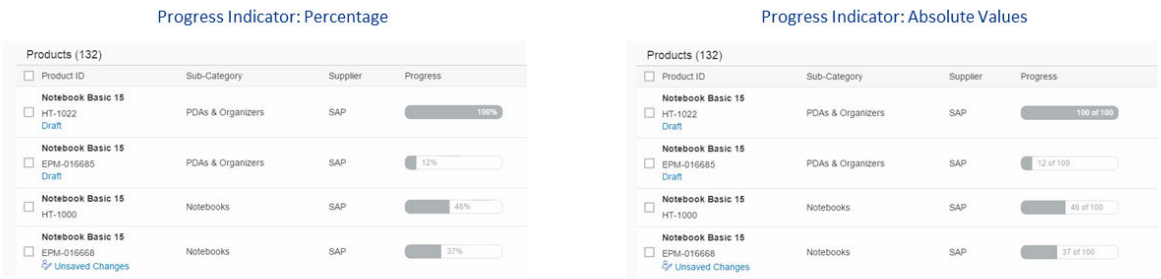


Figure 310: Progress Indicator

Code Samples

The following code sample shows how to implement a progress indicator using annotations. `AnnotationPath="@UI.DataPoint#Progress` references the `Qualifier="Progress"`. `EnumMember="UI.VisualizationType/Progress"` defines the actual visualization as a progress indicator.

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
                <Annotation Term="UI.DataPoint" Qualifier="Progress">
                                <Record>
                                                <PropertyValue Property="Value"
Path="to_StockAvailability/Quantity" />
                                                <PropertyValue
Property="TargetValue" Decimal="100" />
                                                <PropertyValue
Property="Visualization" EnumMember="UI.VisualizationType/Progress" />
                                </Record>
                </Annotation>
                <Annotation Term="UI.LineItem">
                                <Collection>
                                                <Record
Type="UI.DataFieldForAnnotation">
                                                                <PropertyValue
Property="Label" String="Progress" />
                                                                <PropertyValue
Property="Criticality" Path="to_StockAvailability/Quantity" />
```

```
                                                                    <PropertyValue
Property="Target" AnnotationPath="@UI.DataPoint#Progress" />
                                                                </Record>
                                    <Collection>
                            </Annotation>
</Annotations>
```

Change Color of Progress Bar

If required, you can set up the progress bar so that it changes color to reflect the state of the progress depending on the `criticality` value as shown in the figure below.

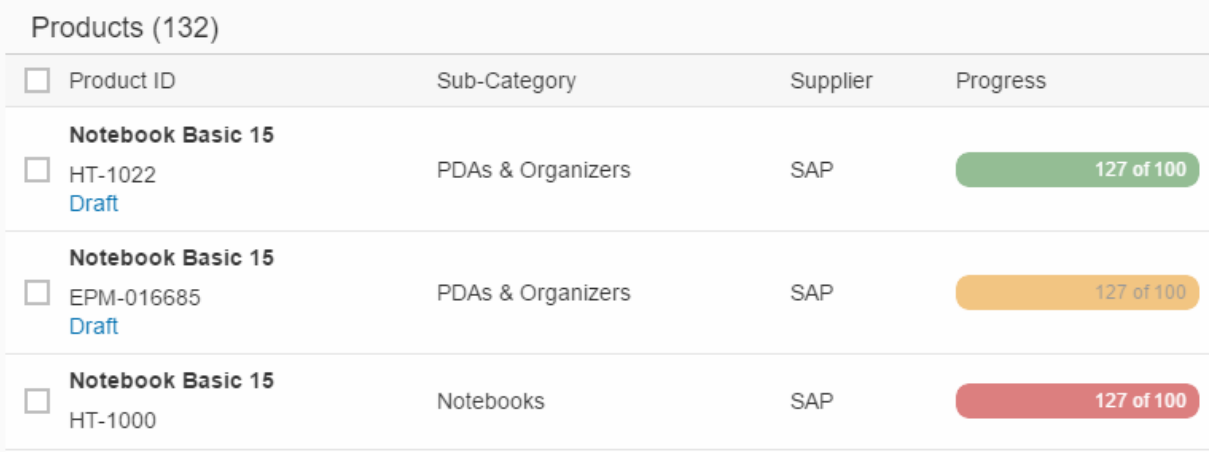


Figure 311: Progress Indicator: Colors Reflect State of Progress

To do so, assign a value to the `criticality` property.

The path references the property (in this case, `Path="StockAvailability"`) that defines the color.

```
<Annotation Term="UI.DataPoint" Qualifier="Quantity">
                <Record>
                        <PropertyValue Property="Value" Path="Quantity"/>
                        <PropertyValue Property="Title"
String="{@i18n&gt;@Availability}"/>
                        <PropertyValue Property="Description" String="Progress
Indicator"/>
                        <PropertyValue Property="TargetValue" Decimal="150"/>
                        <PropertyValue Property="Visualization"
EnumMember="UI.VisualizationType/Progress"/>
                        <PropertyValue Property="Criticality"
Path="StockAvailability"/>
                </Record>
        </Annotation>
```

Adding a Smart Micro Chart to a Table

You can add a smart micro chart to a column within a `SmartTable` control in both the list report and the object page.

To add a smart micro chart to a smart table, use the annotation term `UI.LineItem` and the complex type `DataFieldForAnnotation`. The smart micro charts are then displayed within the table column as shown below:

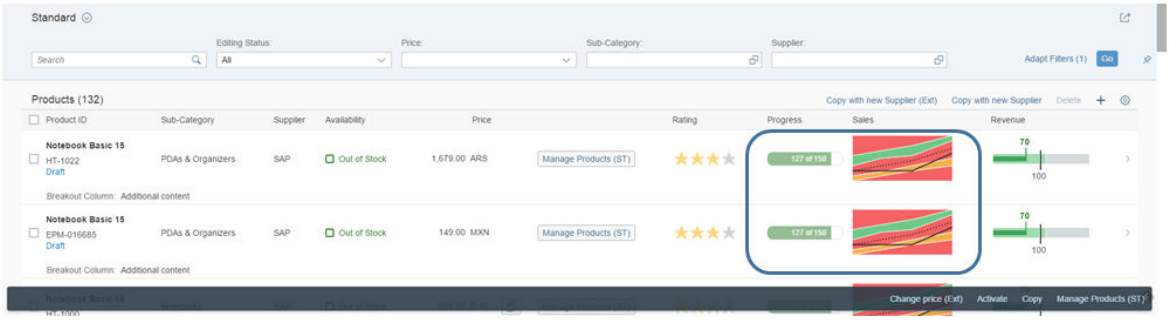


Figure 312: Smart Micro Chart in List Report

Code Samples

UI.LineItem and UI.DataFieldForAnnotation

The `Label` property of the `UI.DataFieldForAnnotation` is used for the text of the table column header.

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
  <Annotation Term="UI.LineItem">
    <Collection>
      <Record Type="UI.DataFieldForAction">
        <PropertyValue Property="Label" String="Copy with new Supplier"/>
        <PropertyValue Property="Action"
String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopywithparams"/>
        <Annotation Term="UI.OperationGrouping"
EnumMember="UI.OperationGroupingType/Isolated"/>
      </Record>
      <Record Type="UI.DataFieldForAction">
        <PropertyValue Property="Label" String="Activate"/>
        <PropertyValue Property="Action"
String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductActivation"/>
        <PropertyValue Property="Determining" Bool="true"/>
        <Annotation Term="UI.OperationGrouping"
EnumMember="UI.OperationGroupingType/ChangeSet"/>
      </Record>
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="Product"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="ProductCategory"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="to_Supplier/CompanyName"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
```

```
        <PropertyValue Property="Criticality" Path="to_StockAvailability/
StockAvailability"/>
        <PropertyValue Property="Value" Path="to_StockAvailability/
StockAvailability"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="Price"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
      </Record>
      <Record Type="UI.DataFieldForAction">
        <PropertyValue Property="Label" String="Copy"/>
        <PropertyValue Property="IconUrl" String="sap-icon://copy"/>
        <PropertyValue Property="Action"
String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopy"/>
        <PropertyValue Property="Inline" Bool="true"/>
        <PropertyValue Property="Determining" Bool="true"/>
        <Annotation Term="UI.OperationGrouping"
EnumMember="UI.OperationGroupingType/Isolated"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
        <PropertyValue Property="Label" String="Manage Products (ST)"/>
        <PropertyValue Property="SemanticObject" String="EPMProduct"/>
        <PropertyValue Property="Action" String="manage_st"/>
        <PropertyValue Property="Inline" Bool="true"/>
        <PropertyValue Property="Determining" Bool="true"/>
      </Record>
      <Record Type="UI.DataFieldForAnnotation">
        <PropertyValue Property="Label" String="Rating"/>
        <PropertyValue Property="Target" AnnotationPath="@UI.DataPoint#Rating"/>
      </Record>
      <Record Type="UI.DataFieldForAnnotation">
        <PropertyValue Property="Label" String="Progress"/>
        <PropertyValue Property="Criticality" Path="to_StockAvailability/
Quantity"/>
        <PropertyValue Property="Target"
AnnotationPath="@UI.DataPoint#Progress"/>
      </Record>
      <Record Type="UI.DataFieldForAnnotation">
        <PropertyValue Property="Label" String="Sales"/>
        <PropertyValue Property="Target" AnnotationPath="to_ProductSalesPrice/
@UI.Chart#AreaChartQualifier"/>
      </Record>
      <Record Type="UI.DataFieldForAnnotation">
        <PropertyValue Property="Label" String="Revenue"/>
        <PropertyValue Property="Target" AnnotationPath="to_ProductSalesRevenue/
@UI.Chart#BulletChartQualifier"/>
      </Record>
    </Collection>
  </Annotation>
</Annotations>
```

UI.Chart Annotations

Smart Area Micro Chart

```
<Annotation Term="UI.Chart" Qualifier="AreaChartQualifier">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Sales Price" />
        <PropertyValue Property="Description" String="Area Micro Chart" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Area" />
        <PropertyValue Property="Dimensions">
            <Collection>
                <PropertyPath>PriceDay</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Measures">
```

```
            <Collection>
                <PropertyPath>AreaChartPrice</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="AreaChartPrice" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#AreaChartPrice" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Smart Bullet Micro Chart

```
<Annotation Term="UI.Chart" Qualifier="BulletChartQualifier">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Sales Revenue" />
        <PropertyValue Property="Description" String="Bullet Micro Chart" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Bullet" />
        <PropertyValue Property="Measures">
            <Collection>
                <PropertyPath>BulletChartRevenue</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="BulletChartRevenue" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#BulletChartRevenue" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

→ Recommendation

Refer the documentation for Smart Micro Chart Facet [page 1673] to see code samples for these micro charts:

- Smart radial micro chart
- Smart column micro chart
- Smart line micro chart
- Smart harvey micro chart
- Smart stacked bar micro chart

Limitation

Currently, the template doesn't currently support the use of navigation properties within the `UI.Chart` term for the smart micro chart (see example below).

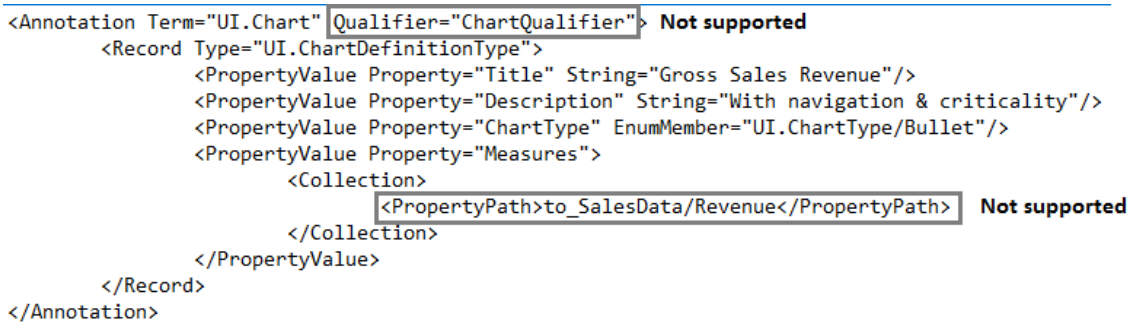


Figure 313: Navigation Property

UI.DataPoint Annotation

The `DataPoint` property of the `MeasureAttributes` of the `UI.Chart` annotation should point to the `UI.DataPoint` annotation.

The `SmartMicroChart` control supports the `Criticality` and `CriticalityCalculation` properties of a `UI.DataPoint`. For an example of how to use the `CriticalityCalculation`, see the smart area micro chart annotation example. For an example of how to use the `Criticality` property, see the smart bullet micro chart annotation example.

> **i Note**
>
> Although the `Title` for the `UI.DataPoint` is mandatory, it is not used by the smart micro chart.

Smart Area Micro Chart

```
<Annotation Term="UI.DataPoint" Qualifier="AreaChartPrice">
    <Record>
        <PropertyValue Property="Title" String="Sales Price" />
        <PropertyValue Property="Value" Path="AreaChartPrice" />
        <PropertyValue Property="TargetValue" Path="TargetPrice" />
        <PropertyValue Property="CriticalityCalculation">
            <Record>
                <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Target" />
                <PropertyValue Property="DeviationRangeHighValue"
Path="DeviationUpperBoundPrice" />
                <PropertyValue Property="DeviationRangeLowValue"
Path="DeviationLowerBoundPrice" />
                <PropertyValue Property="ToleranceRangeHighValue"
Path="ToleranceUpperBoundPrice" />
                <PropertyValue Property="ToleranceRangeLowValue"
Path="ToleranceLowerBoundPrice" />
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

Smart Bullet Micro Chart

```
<Annotation Term="UI.DataPoint" Qualifier="BulletChartRevenue">
    <Record>
        <PropertyValue Property="Title" String="Sales Revenue" />
        <PropertyValue Property="Value" Path="BulletChartRevenue" />
        <PropertyValue Property="TargetValue" Path="TargetRevenue" />
        <PropertyValue Property="ForecastValue" Path="ForecastRevenue" />
        <PropertyValue Property="MinimumValue" Decimal="100" />
        <PropertyValue Property="MaximumValue" Decimal="300" />
```

```
        <PropertyValue Property="Criticality" Path="Criticality" />
    </Record>
</Annotation>
```

> **Note**
>
> The following must all point to the same property in the `entityType`:
>
> - `Measure` property of the `Chart` annotation
> - `Measure` property of the `MeasureAttributes` property of the `Chart` annotation
> - Value property of the `DataPoint` annotation

CDS Annotations

CDS Annotation Definition for UI.Chart

```
chart : array of
{
    qualifier : String(120);
    @LanguageDependency.maxLength : 40
    title : String(60);
    @LanguageDependency.maxLength : 80
    description : String(120);
    chartType : String enum
    {
       COLUMN;
       COLUMN_STACKED;
       COLUMN_STACKED_100;
       COLUMN_DUAL;
       COLUMN_STACKED_DUAL;
       COLUMN_STACKED_DUAL_100;
       BAR;
       BAR_STACKED;
       BAR_STACKED_100;
       BAR_DUAL;
       BAR_STACKED_DUAL;
       BAR_STACKED_DUAL_100;
       AREA;
       AREA_STACKED;
       AREA_STACKED_100;
       HORIZONTAL_AREA;
       HORIZONTAL_AREA_STACKED;
       HORIZONTAL_AREA_STACKED_100;
       LINE;
       LINE_DUAL;
       COMBINATION;
       COMBINATION_STACKED;
       COMBINATION_STACKED_DUAL;
       HORIZONTAL_COMBINATION_STACKED;
       HORIZONTAL_COMBINATION_STACKED_DUAL;
       PIE;
       DONUT;
       SCATTER;
       BUBBLE;
       RADAR;
       HEAT_MAP;
       TREE_MAP;
       WATERFALL;
       BULLET;
       VERTICAL_BULLET;
    };
    dimensions : array of elementRef;
    measures : array of elementRef;
    dimensionAttributes : array of
    {
```

```
        dimension : elementRef;
        role : String(10) enum
        {
            CATEGORY;
            SERIES;
        };
    };
    measureAttributes : array of
    {
        measure : elementRef;
        role : String(10) enum
        {
            AXIS_1;
            AXIS_2;
            AXIS_3;
        };
        asDataPoint : Boolean default true;
    };
}
```

CDS Annotation for UI.Chart: Smart Area Micro Chart

```
@UI.chart:[{
  title: 'Sales Price',
  description: 'Area Micro Chart',
  chartType: #AREA,
  dimensions:['PriceDay'],
  measures:['AreaChartPrice'],
  measureAttributes: [
    { measure: 'AreaChartPrice', role: #AXIS_1, asDataPoint: true }
  ],
  qualifier: 'AreaChartQualifier'
}]
```

CDS Annotation for UI.Chart: Smart Bullet Micro Chart

```
@UI.chart:[{
  title:'Sales Revenue',
  description: 'Bullet Micro Chart',
  chartType: #BULLET,
  measures:['BulletChartRevenue'],
  measureAttributes: [
    { measure: 'BulletChartRevenue', role: #AXIS_1, asDataPoint: true }
  ],
  qualifier: 'BulletChartQualifier'
}]
```

CDS Annotation Definition for UI.DataPoint

```
dataPoint
{
    @LanguageDependency.maxLength : 40
    title : String(60);
    @LanguageDependency.maxLength : 80
    description : String(120);
    @LanguageDependency.maxLength : 190
    longDescription : String(250);
    targetValue : DecimalFloat;
    targetValueElement : elementRef;
    forecastValue : elementRef;
    minimumValue : DecimalFloat;
    maximumValue : DecimalFloat;
    visualization : String enum
    {
        NUMBER;
```

```
        BULLET_CHART;
        DONUT;
        PROGRESS;
        RATING;
    };
    valueFormat
    {
        scaleFactor : DecimalFloat;
        numberOfFractionalDigits : Integer;
    };
    referencePeriod
    {
        @LanguageDependency.maxLength : 80
        description : String(120);
        start : elementRef;
        end : elementRef;
    };
    criticality : elementRef;
    criticalityCalculation
    {
        improvementDirection : String enum { MINIMIZE; TARGET; MAXIMIZE; };
        toleranceRangeLowValue : DecimalFloat;
        toleranceRangeLowValueElement : elementRef;
        toleranceRangeHighValue : DecimalFloat;
        toleranceRangeHighValueElement : elementRef;
        deviationRangeLowValue : DecimalFloat;
        deviationRangeLowValueElement : elementRef;
        deviationRangeHighValue : DecimalFloat;
        deviationRangeHighValueElement : elementRef;
    };
    trend : elementRef;
    trendCalculation
    {
        referenceValue : elementRef;
        isRelativeDifference : Boolean default false;
        upDifference : DecimalFloat;
        upDifferenceElement : elementRef;
        strongUpDifference : DecimalFloat;
        strongUpDifferenceElement : elementRef;
        downDifference : DecimalFloat;
        downDifferenceElement : elementRef;
        strongDownDifference : DecimalFloat;
        strongDownDifferenceElement : elementRef;
    };
    responsible : elementRef;
    responsibleName : String(120);
};
```

CDS Annotation for UI.DataPoint: Smart Area Micro Chart

```
@UI.dataPoint: {
   title: 'Sales Price',
   targetValueElement: 'TargetPrice',
   criticalityCalculation: {
      improvementDirection: #TARGET,
      toleranceRangeLowValueElement: 'ToleranceLowerBoundPrice',
      toleranceRangeHighValueElement: 'ToleranceUpperBoundPrice',
     deviationRangeLowValueElement: 'DeviationLowerBoundPrice',
     deviationRangeHighValueElement: 'DeviationUpperBoundPrice'
   }
}
ProductSalesPrice.Price as AreaChartPrice
```

CDS Annotation for UI.DataPoint: Smart Bullet Micro Chart

```
@UI.dataPoint: {
```

```
    title:'Sales Revenue',
    targetValueElement: 'TargetRevenue',
    forecastValue: 'ForecastRevenue',
    minimumValue: 100,
    maximumValue: 300,
    criticality: 'Criticality'
}
ProductSalesRevenue.Revenue as BulletChartRevenue
```

> **Note**
>
> Currently, the `UI.DataPoint` in the CDS views does not support `elementRef` for `minimumValue` and `maximumValue`. The values for these properties are hard-coded as in the example above.

Adding a Contact Quick View to a Table

Using the `@Communication.Contact` annotation, you can enable a contact quick view in a table.

The quick view can be displayed from the list report and the object page, as shown below.

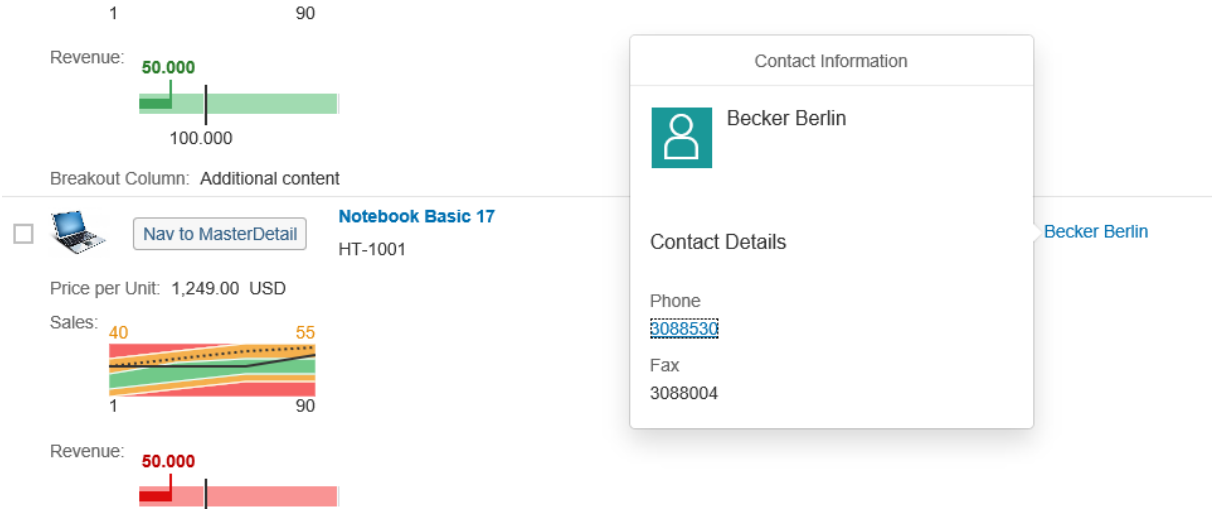


Figure 314: Displaying a Contact Quick View in a Table

Code Sample

In the example, the `UI.LineItem` annotation needs to be added under the `ProductType` entity, as follows:

> **Sample Code**

```
 <Annotation Term="UI.LineItem">
                                    <Collection>

                                            <Record
Type="UI.DataFieldForAction">
                                                   <PropertyValue
Property="Label" String="Activate" />
                                                   <PropertyValue
Property="Action"
```

```
String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductActivation"/>
                                                        <PropertyValue
Property="InvocationGrouping"

EnumMember="UI.OperationGroupingType/ChangeSet" />
                                                    </Record>

                                                     <Record Type="UI.DataField">
                                                        <PropertyValue
Property="Value" Path="ProductPictureURL" />
                                                        <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High" />
                                                    </Record>
                                                    <Record Type="UI.DataField">
                                                        <PropertyValue
Property="Value" Path="ProductForEdit" />
                                                        <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High" />
                                                    </Record>
                                                    <Record Type="UI.DataField">
                                                        <PropertyValue
Property="Value" Path="ProductCategory" />
                                                        <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High" />
                                                    </Record>
                                                    <Record
Type="UI.DataFieldWithIntentBasedNavigation">
                                                        <PropertyValue
Property="Label" String="Supplier (with IBN)" />
                                                        <PropertyValue
Property="Value" Path="Supplier" />
                                                        <PropertyValue
Property="SemanticObject" String="EPMProduct" />
                                                        <PropertyValue
Property="Action" String="manage" />
                                                    </Record>

                                                     <Record
Type="UI.DataFieldForAnnotation">
                                                        <PropertyValue
Property="Label" String="{@i18n&gt;@Supplier}"/>
                                                        <PropertyValue
Property="Target"

AnnotationPath="to_Supplier/@Communication.Contact"/>
                                                        <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High" />
                                                    </Record>
                                                    <Record Type="UI.DataField">
                                                        <PropertyValue
Property="Criticality"

Path="to_StockAvailability/StockAvailability" />
                                                        <PropertyValue
Property="Value"

Path="to_StockAvailability/StockAvailability" />
                                                        <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High" />
                                                    </Record>
                                                    <Record
Type="UI.DataFieldForAnnotation">
                                                        <PropertyValue
Property="Label" String="Rating" />
                                                        <PropertyValue
Property="Target" AnnotationPath="@UI.DataPoint#Rating" />
                                                    </Record>
```

```
                                                <Record
Type="UI.DataFieldForAnnotation">
                                                        <PropertyValue
Property="Label" String="Progress" />
                                                        <PropertyValue
Property="Criticality"

Path="to_StockAvailability/Quantity" />
                                                        <PropertyValue
Property="Target" AnnotationPath="@UI.DataPoint#Progress" />
                                                </Record>
                                                <Record Type="UI.DataField">
                                                        <PropertyValue
Property="Value" Path="Price" />
                                                        <Annotation
Term="UI.Importance" EnumMember="UI.ImportanceType/High" />
                                                </Record>
                                                <Record
Type="UI.DataFieldForAction">
                                                        <PropertyValue
Property="Label" String="Copy" />
                                                        <PropertyValue
Property="IconUrl" String="sap-icon://copy" />
                                                        <PropertyValue
Property="Action"

String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopy" />
                                                        <PropertyValue
Property="Inline" Bool="true" />
                                                        <PropertyValue
Property="Determining" Bool="true" />
                                                        <PropertyValue
Property="InvocationGrouping"

EnumMember="UI.OperationGroupingType/Isolated" />
                                                </Record>
                                                <Record
Type="UI.DataFieldForIntentBasedNavigation">
                                                        <PropertyValue
Property="Label" String="{@i18n&gt;@MANAGE_PRODUCTS_(ST)_2}" />
                                                        <PropertyValue
Property="SemanticObject" String="EPMProduct" />
                                                        <PropertyValue
Property="Action" String="manage_st" />
                                                        <PropertyValue
Property="Inline" Bool="false" />
                                                        <PropertyValue
Property="Determining" Bool="false" />
                                                        <PropertyValue
Property="RequiresContext" Bool="true" />
                                                </Record>

                                                <Record
Type="UI.DataFieldForIntentBasedNavigation">
                                                        <PropertyValue
Property="Label" String="{@i18n&gt;@Inline_Nav_MasterDetail}" />
                                                        <PropertyValue
Property="SemanticObject" String="EPMProduct" />
                                                        <PropertyValue
Property="Action" String="manage" />
                                                        <PropertyValue
Property="Inline" Bool="true" />
                                                        <PropertyValue
Property="Determining" Bool="false" />
                                                        <PropertyValue
Property="RequiresContext" Bool="false" />
                                                </Record>
```

```
                                                <Record
Type="UI.DataFieldForAnnotation">
                                                        <PropertyValue
Property="Label" String="Sales"/>
                                                        <PropertyValue
Property="Target" AnnotationPath="to_ProductSalesPrice/
@UI.Chart#SalesPriceAreaChart"/>
                                                </Record>
                                                <Record
Type="UI.DataFieldForAnnotation">
                                                        <PropertyValue
Property="Label" String="Revenue"/>
                                                        <PropertyValue
Property="Target" AnnotationPath="to_ProductSalesRevenue/
@UI.Chart#GrossSalesRevenueBulletChart"/>
                                                </Record>
                                        </Collection>
                                </Annotation>
```

The `Communication.Contact` annotation term is defined under the `SupplierType` entity as follows:

Sample Code

```
 <Annotation Term="Communication.Contact">
                                        <Record>
                                                <PropertyValue Property="fn"
Path="CompanyName" />
                                                <PropertyValue Property="tel">
                                                        <Collection>
                                                                <Record>

<PropertyValue Property="type"

EnumMember="Communication.PhoneType/fax" />

<PropertyValue Property="uri" Path="FaxNumber" />
                                                                </Record>
                                                                <Record>

<PropertyValue Property="type"

EnumMember="Communication.PhoneType/preferred Communication.PhoneType/work" />

<PropertyValue Property="uri" Path="PhoneNumber" />
                                                                </Record>
                                                        </Collection>
                                                </PropertyValue>
                                                <PropertyValue Property="email">
                                                        <Collection>
                                                                <Record>

<PropertyValue Property="type"

EnumMember="Communication.ContactInformationType/work
Communication.ContactInformationType/pref" />

<PropertyValue Property="address" Path="EmailAddress" />
                                                                </Record>
                                                        </Collection>
                                                </PropertyValue>
                                        </Record>
                                </Annotation>
```

Adding Multiple Fields to One Column in Responsive Tables

You can add multiple IDs, descriptions, and action buttons to one column in a responsive table.

To include these items, the `UI.FieldGroup` needs to be referred to in the `UI.LineItem` annotation. The `UI.FieldGroup` then contains a collection of annotations that can be grouped together semantically.

The `FieldGroup` below contains the fields that are displayed in the same column.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.FieldGroup" Qualifier="TechData">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="com.sap.vocabularies.UI.v1.DataField">
                    <PropertyValue Property="Value" Path="Supplier" />
                </Record>
                <Record Type="com.sap.vocabularies.UI.v1.DataField">
                    <PropertyValue Property="Value" Path="Width" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

The `UI.LineItem` that includes the `UI.FieldGroup` shown above looks like this (second `DataField`):

Sample Code

```
<Annotation Term="UI.LineItem">
    <Collection>
        <Record Type="UI.DataField">
           <PropertyValue Property="Label" String="Project"/>
            <PropertyValue Property="Value" Path="Project"/>
            <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/
High"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation">
          <PropertyValue Property="Label" String="Technical Data" />
<!-- This Label becomes the column header -->
          <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#TechData"/> <!--  This FieldGroup should
always be a part of same entityType as the UI.LineItem (to_NavigationProperty/
@UI.FieldGroup are currently not supported) -->
        </Record>
     </Collection>
</Annotation>
```

Note

For the fields contained in the `FieldGroup` shown above: If a property and its corresponding `sap:text` property are shown in the same column, the `TextArrangement` annotation is not applicable for this field. In this case, the default is `idOnly`. In all other cases, the `textArrangement` annotation can be defined to show the ID and the description in the table column.

You can use the following annotations in `UI.FieldGroup`:

- `UI.DataField`
- `UI.DataFieldForAction`
- `UI.DataFieldForIntentBasedNavigation`
- `UI.DataFieldWithNavigationPath`
- `UI.DataFieldForAnnotation`
  - `Communication.Contact`
  - `UI.Chart`
  - `UI.Visualization/Rating`
  - `UI.Visualization/Progress`

The result looks like this:

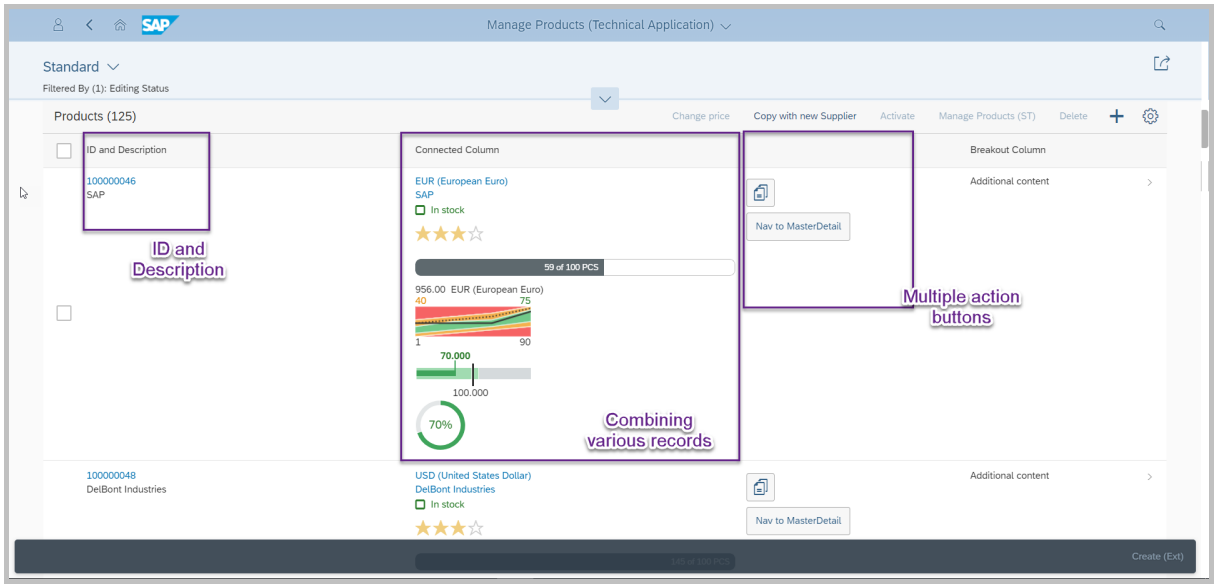


Figure 315: Multiple IDs, Descriptions, and Action Buttons in a Table Column

Limitations

Take the following limitations into account when implementing this feature:

- When using the export to Microsoft Excel feature, only the first field of the semantically connected column is exported, that is, the first visible field in the table column.
- When using table personalization, users can only filter, sort, and group the semantically connected column based on the individual properties (of the same entity type as the `UI.LineItem`) mentioned in the `UI.FieldGroup` collection.
  Example: Height and weight are two properties which are semantically connected. The name of the column header is *Combined*. In this case, in the personalization, you cannot filter, sort, and group on *Combined*. You can only filter, sort, and group on individual properties like height and weight.

Settings for List Report Tables

You can set up various aspects of the list report table through annotations and in the manifest.json file, as described in the sections that follow.

Defining Line Items

To define the line items of a table, use `com.sap.vocabularies.UI.v1.LineItem` as shown in the code samples below. The rendering result is as follows:

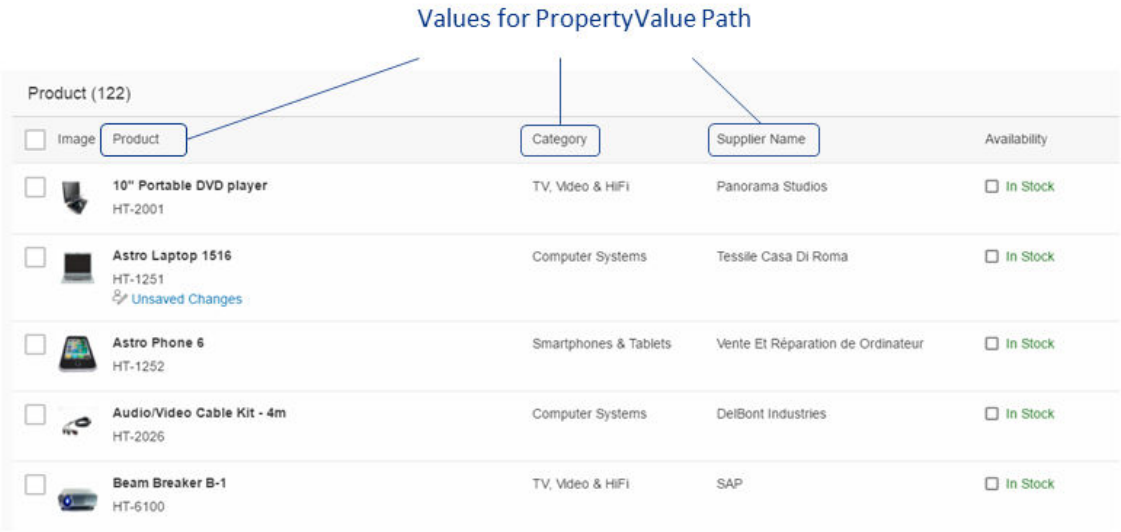


Figure 316: List Report: LineItem of Root EntitySet

This video shows the step-by-step procedure for adding line items to a list report table:

Code Samples

Root entitySet in manifest.json

```
"pages: [
    {
    "entitySet": "SEPMRA_C_PD_Product",
    "component": {
         "name": "sap.suite.ui.generic.template.ListReport",
         "list": true
    },
```

Annotation XML: Determining Column Names

```
...
<Annotation Term="UI.LineItem">
    <Collection>
    <RecordType="UI.DataField">
        <PropertyValue Property="Value" Path="Product"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <RecordType="UI.DataField">
        <PropertyValue Property="Value" Path="ProductCategory"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <RecordType="UI.DataField">
        <PropertyValue Property="Value" Path="Supplier"/>
```

```
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    </Collection>
</Annotation>
```

More Information

For information about adding actions for line items in the list report view, see Enabling Actions in the List Report [page 1642].

For information about responsiveness options in tables, see Responsiveness Options: Example [page 1617].

Displaying Images in Tables

To display images in tables, you must first add a data field with a value that relates to an image URL

This is shown in the following example:

Sample Code

```
<Record Type="UI.DataField">
    <PropertyValue Property="Value" Path="ProductPictureURL"/>
    <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
</Record>
```

You also need to add a property annotation to the local annotation file that specifies that this property contains an image URL, as in the following example:

Sample Code

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductType/ProductPictureURL">
    <Annotation Term="UI.IsImageUrl" />
    <Annotation Term="Common.Text" String="{Product}" />
</Annotations>
```

The `Common.Text` annotation is optional. You can use it to provide textual information for the image, for example, for accessibility purposes. This text is not visible on the UI, but can be read by screen readers. From a technical perspective, the provided string (in this example, the product) is assigned to the `alt` property of the `sap.m.Image` instance.

Displaying the Editing Status

In draft-enabled applications, the edit, locked, unsaved data, or draft status is displayed in the first column to which you have added the semantic key using the semantic key annotation.

For more information, see Editing Status [page 1630].

> **Note**
>
> This is only available for the default `DataField`.

> **Sample Code**

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="STTA_PROD_MAN.STTA_C_MP_ProductType">
    <Annotation Term="Common.SemanticKey">
        <Collection>
            <PropertyPath>ProductForEdit</PropertyPath>
        </Collection>
    </Annotation>
</Annotations>
```

If you have not added a semantic key to the line items, the editing status is displayed in the *Title* or *Description* column, depending on the available information:

- If a semantic key annotation is available for the `entityType`, the value of the first semantic key is displayed.
- If no semantic key annotation is available, the system checks whether the `headerInfo/Title/Value/Path` is available under the `LineItem` annotation.

If applicable, this column is displayed in bold font and the editing status is added, if available.

For more information, see also SemanticKey.

Adapting the Table Content to the Space in the List Report

By default, the analytical, grid, and tree tables in list reports use the entire space of the table container on the page.

For tables in the list report that have their own scrollbar, the`fitContent` property is set to `true` by default. This applies if the `tableType` is `sap.ui.table.Table`, `sap.ui.table.AnalyticalTable`, or `sap.ui.table.TreeTable`.

If you don't want the table to occupy the entire space, set the the property to `false`.

Alternatively, you can use the SAPUI5 Visual Editor to change this setting.

> **Note**
>
> In the default settings for responsive tables the `fitContent` property is set to `false`.

Related Information

https://sapui5.hana.ondemand.com/#/api/sap.f.DynamicPage/methods/getFitContent
Adapting the UI: List Report and Object Page [page 1860]

Settings for Object Page Tables

You can set up various aspects of the object page tables through the manifest.json file, as described in the sections that follow.

Adding Titles to Object Page Tables

You can add a title to an object page table.

To add a title to an object page table, provide a string value to the `TypeNamePlural` property of the `UI.HeaderInfo` annotation for the entity type with which the table is associated.

Sample Code

```
<Annotations Target="STTA_PROD_MAN.STTA_C_MP_ProductTextType">
      <Annotation Term="UI.HeaderInfo">
          <Record>
              <PropertyValue Property="TypeName" String="Product Text" />
              <PropertyValue Property="TypeNamePlural"
String="{@i18n>@TableTitle}" />
              <PropertyValue Property="Title">
                    <Record Type="UI.DataField">
                          <PropertyValue Property="Value" Path="Name" />
                    </Record>
              </PropertyValue>
             <PropertyValue Property="Description">
                    <Record Type="UI.DataField">
                        <PropertyValue Property="Value" Path="Language" />
                    </Record>
             </PropertyValue>
       </Record>
 </Annotation>
```

Take the following into account:

- As `TypeNamePlural` is a mandatory parameter, if the section title and the table title are identical, the table title is not displayed.
- If the `UI.HeaderInfo` annotation hasn't been entered, the table title is also not displayed.
- Make sure you provide the appropriate section titles if the same string is maintained under `TypeNamePlural` in the `UI.HeaderInfo` annotation.

The results look like this:

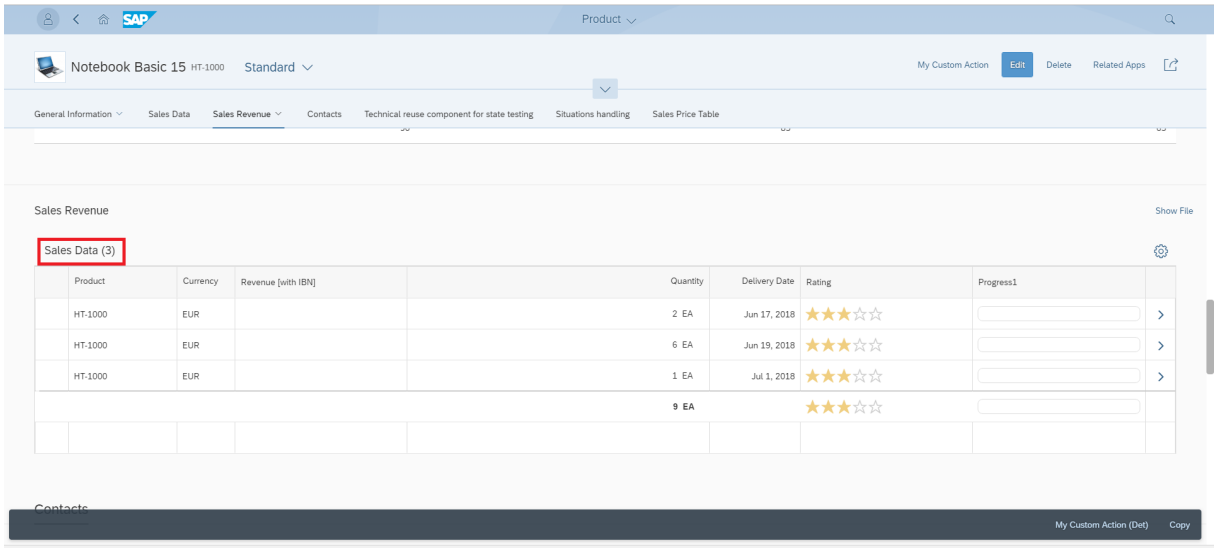


Figure 317: Title for single table in object page section

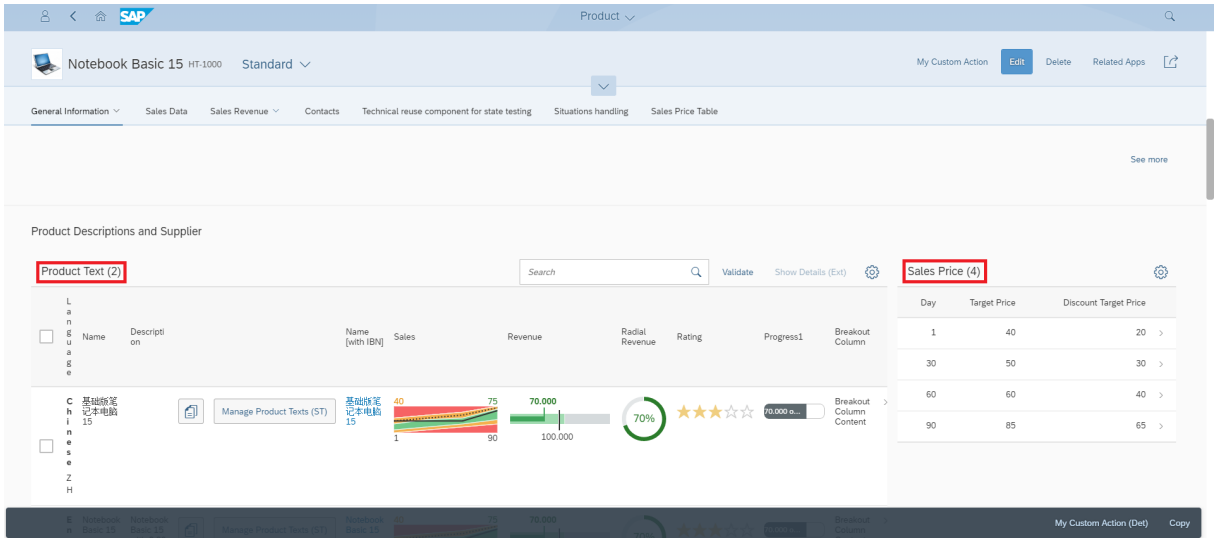


Figure 318: Title for multiple tables in object page section

Adding Segmented Buttons to a Table Toolbar

You can add segmented buttons to the toolbar, to enable switching between the table content using a selection variant annotation.

You can associate every button of the segmented buttons (or every list item in the select box) with a selection variant that filters the table according to the selection variant filters once a user has clicked it. This means that the user has multiple views in a single table of the object page. You can enable this feature for any table on the object page.

To implement this feature, make a `"quickVariantSelection"` entry in the manifest. Every variant corresponds to its filter on the UI. A segmented button is rendered when the number of variants defined is less than or equal to 3. Defining 4 or more variants in the manifest renders a selection box.

The following manifest settings are required for adding segmented buttons:

Sample Code

```
"component": {
  "name": "sap.suite.ui.generic.template.ObjectPage",
  "settings": {
    "showConfirmationOnDraftActivate": false,
    "sections": {
      "SalesOrderItemsID": {
        "navigationProperty": "to_Item",
        "entitySet": "C_STTA_SalesOrderItem_WD_20",
        "createMode": "inline",
        "quickVariantSelection": {
          "showCounts": true,
          "variants": {
            "0": {
              "key": "_tab2",
              "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#SimpleFilter"
            },
            "1": {
              "key": "_tab3",
              "annotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#ComplexFilter"
            }
          }
        }
      },
    "showRelatedApps": true
  }
}
```

To show the number of records available next to the title of the segmented button, set `showCounts` to `true`.

You can define a simple or a complex filter condition inside the `SelectionVariant`. While the simple condition has only one property in the `SelectionVariant`, the complex filter condition can have more than one property to be filtered.

The annotations defined in the variants are:

Sample Code

```
<Annotation Term="UI.SelectionVariant" Qualifier="SimpleFilter">
      <Record>
         <PropertyValue Property="Text" String="Tax amount less than 10
USD" />
         <PropertyValue Property="SelectOptions">
            <Collection>
               <Record Type="UI.SelectOptionType">
                  <PropertyValue Property="PropertyName"
PropertyPath="tax_amount" />
                  <PropertyValue Property="Ranges">
                     <Collection>
                        <Record Type="UI.SelectionRangeType">
                           <PropertyValue Property="Sign"
EnumMember="UI.SelectionRangeSignType/I" />
                           <PropertyValue Property="Option"
EnumMember="UI.SelectionRangeOptionType/LT" />
                           <PropertyValue Property="Low" String="10" />
                        </Record>
                     </Collection>
```

```
                  </PropertyValue>
               </Record>
            </Collection>
         </PropertyValue>
      </Record>
   </Annotation>
   <Annotation Term="UI.SelectionVariant" Qualifier="ComplexFilter">
      <Record>
         <PropertyValue Property="Text" String="Net Amount between 10 and 40
And Gross Amount Less than 100 USD" />
         <PropertyValue Property="SelectOptions">
            <Collection>
               <Record Type="UI.SelectOptionType">
                  <PropertyValue Property="PropertyName"
PropertyPath="net_amount" />
                  <PropertyValue Property="Ranges">
                     <Collection>
                        <Record>
                           <PropertyValue Property="Sign"
EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeSignType/I" />
                           <PropertyValue Property="Option"
EnumMember="UI.SelectionRangeOptionType/BT" />
                           <PropertyValue Property="Low" String="10" />
                           <PropertyValue Property="High" String="40" />
                        </Record>
                     </Collection>
                  </PropertyValue>
               </Record>
               <Record Type="UI.SelectOptionType">
                  <PropertyValue Property="PropertyName"
PropertyPath="gross_amount" />
                  <PropertyValue Property="Ranges">
                     <Collection>
                        <Record Type="UI.SelectionRangeType">
                           <PropertyValue Property="Option"
EnumMember="UI.SelectionRangeOptionType/LT" />
                           <PropertyValue Property="Sign"
EnumMember="UI.SelectionRangeSignType/I" />
                           <PropertyValue Property="Low" String="100" />
                        </Record>
                     </Collection>
                  </PropertyValue>
               </Record>
            </Collection>
         </PropertyValue>
      </Record>
   </Annotation>
```

i Note

If the `SelectionVariant` has multiple `<SelectionOptionType>`, filters that have the same target property are combined with an OR condition. Filters that have different target properties are combined with an AND condition.

The table containing the segmented buttons looks like this:

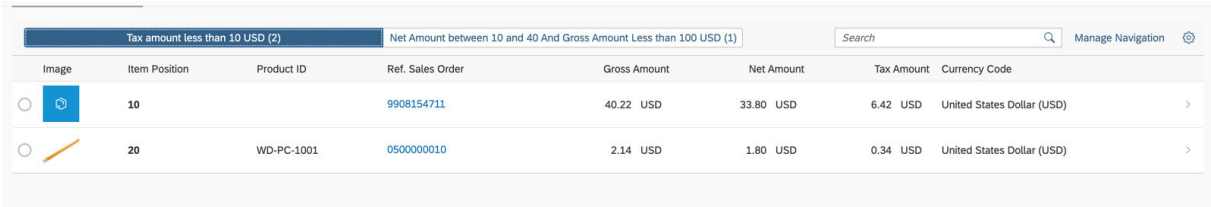


Figure 319: Segmented button 1 selected

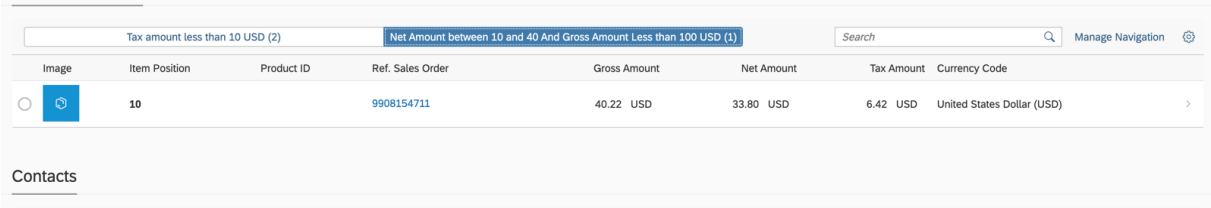


Figure 320: Segmented button 2 selected

> ### i Note
>
> When the table is initially loaded, two calls are made to determine the number of the records in the table. These are shown in the button text.
>
> Setting `showCounts` to `false` will not show the number in the button text. This means that in this case no count calls are made.

Related Information

Defining Multiple Views on a List Report Table - Single Table Mode [page 1649]

Enabling Inline Creation of Table Entries on Object Page

You can enable the inline creation of table entries for apps that use draft handling.

In edit mode, the user can add new entries to a table in a section by choosing *Add Entry*. By default, a new entry is created and the system automatically navigates to the item's object page. You can enable inline creation of entries, that is, a new line is created but automatic navigation isn't triggered. When a new entry is created, the line is highlighted in blue. This highlight disappears once the data is saved.

To enable inline creation, in the `pages` section in the manifest.json of your app, set `createMode` to `inline` like this:

> ### i Note
>
> If you have defined an ID for the reference facet of your table, use this ID instead of the generated one, for example, `to_ProductText::com.sap.vocabularies.UI.v1.LineItem`.

```
"sap.ui.generic.app": {
```

```
        "pages": [
            {
                "entitySet": "SEPMRA_C_PD_Product",
                "component": {
                    "name": "sap.suite.ui.generic.template.ListReport",
                    "list": true
                },
                "pages": [
                    {
                        "entitySet": "SEPMRA_C_PD_Product",
                        "component": {
                            "name": "sap.suite.ui.generic.template.ObjectPage",
                            "settings": {
                                "sections": {

"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {

"navigationProperty": "to_ProductText",
                                                                "entitySet":
"SEPMRA_C_PD_ProductText",
                                                                "createMode":
"inline"
                                                        }
                                                    }
                                                }
                        },
                        "pages": [
                            {
                                "navigationProperty": "to_ProductText",
                                "entitySet": "SEPMRA_C_PD_ProductText",
                                "component": {
                                    "name":
"sap.suite.ui.generic.template.ObjectPage"
                                }
                            }
                        ]
                    }
                ]
            }
        ]
    },
```

> **Note**
>
> For apps based on releases below SAP NetWeaver 7.51 SP01, the following restriction applies: If a user sets a filter in a table that is enabled for inline creation, the filter conditions might not be evaluated correctly. This can result in data being displayed incorrectly and not according to the filter criteria that has been entered. This is relevant only for apps that use draft handling.

Changing the Default Sort Order

Based on the the default sort order, each newly created row is placed at the top of the table. You can disable this default sorting by using the `disableDefaultInlineCreateSort` flag as shown below in the manifest.json. You can then enter your own sorting logic.

This flag is evaluated only if the `"createMode":"inline"` flag is available in the manifest.json.

> **Sample Code**

```
"ObjectPage|STTA_C_MP_Product": {
                        "entitySet": "STTA_C_MP_Product",
                        "component": {
                            "name":
"sap.suite.ui.generic.template.ObjectPage",
                            "settings": {
                                "showRelatedApps": true,
                                "tableType": "ResponsiveTable",
                                "editableHeaderContent": true,
                                "showConfirmationOnDraftActivate": true,
                                "sections": {

"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
                                        "navigationProperty":
"to_ProductText",
                                        "entitySet": "STTA_C_MP_ProductText",
                                        "multiSelect": true,
                                        "createMode": "inline",
                                        "disableDefaultInlineCreateSort":true,
                                        "tableType": "ResponsiveTable"
                                    }
                                }
                            }
                        },
```

> **i** Note
>
> If you disable the default sort order and do not enter a custom sort order, the newly created row is displayed last. Due to the `growingThreshold` setting of a maximum of 10 rows, the row might not be visible if 10 rows are already displayed. To see this row, you then need to scroll down to the last position in the table.

Enabling Action Buttons in Tables on the Object Page

You can use annotations to enable generic actions in tables on the object page.

Display or Hide the + (Create) Button

You can display or hide the + (*Create*) button for entities related to the selected object (entity) based on certain conditions set up in your back-end system. For example, you can prevent users from adding a product description after the product has been archived.

When set up to be visible, the + button is displayed in the table toolbar when the object page is in *Edit* mode, as shown below.

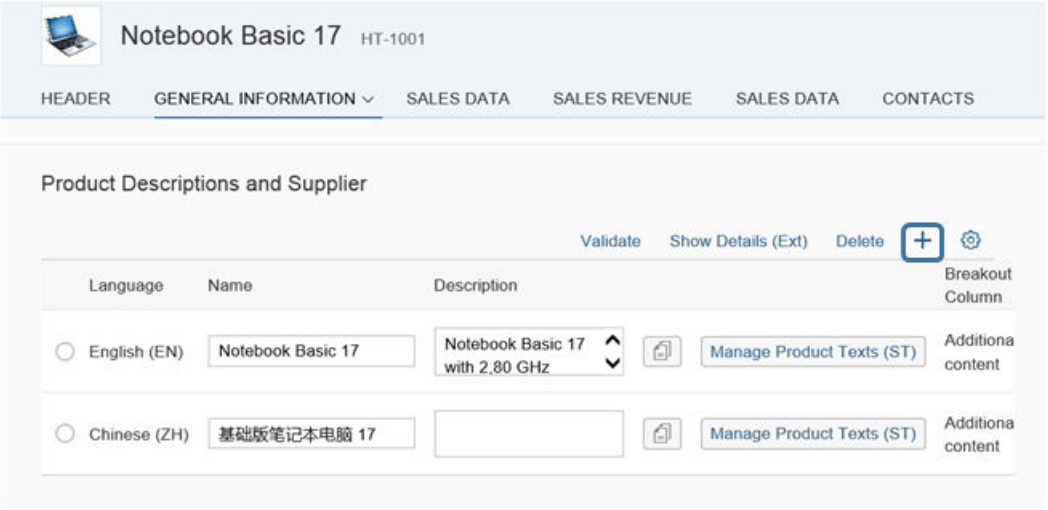


Figure 321: + Button in Table Toolbar

Within your annotations, you set the `creatable-path` to a particular property of the root object (entity) in the back-end system that is either `true` or `false`. If the value of this property is `true`, the + button is displayed; if it is `false`, it is hidden. Note that the `creatable-path` must point to a property of the root entity.

Code Samples

creatable-path: v2 Annotation

```
<NavigationProperty Name="to_ProductText" sap:creatable-
path="CanCreateProductText".../>
```

creatable-path: v4 Annotation

```
<Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_Product">
    <Annotation Term="Capabilities.InsertRestrictions">
        <Record>
            <PropertyValue Property="NonInsertableNavigationProperties">
                <Collection>
                    <If>
                        <Not>
                            <Path>CanCreateProductText</Path>
                        </Not>
                        <NavigationPropertyPath>to_ProductText</
NavigationPropertyPath>
                    </If>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
</Annotations>
```

Enable or Disable Delete Button

You can enable or disable the *Delete* button in the toolbar of tables on the object page based on certain conditions in the back-end system. For example, you can disable the *Delete* button for a product's text if the text is in English.

Within your annotation, you set the `deletable-path` for an entity set to point to a particular Boolean property of the entity that has a value of either `true` or `false`. The *Delete* button is enabled if the selected item's property is `true`. If multiple selection is enabled for the table, the button is enabled if at least one selected item is deletable.

Code Samples

v2 Annotation for Metadata

```
<EntitySet Name="STTA_C_MP_ProductText"
EntityType="STTA_PROD_MAN.STTA_C_MP_ProductTextType" sap:deletable-
path="Delete_mc" sap:content-version="1"/>
```

v4 Annotation

```
<Annotations Target="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductText">
   <Annotation Term="Org.OData.Capabilities.V1.DeleteRestrictions">
        <Record>
            <PropertyValue Property="Deletable" Path="Delete_mc"/>
        </Record>
    </Annotation>
</Annotations>
```

Using the Condensed Table Layout

You can set the content density to `condensed` for `ui.table` on the object page.

To do so, in the manifest.json under the relevant section, set `"condensedTableLayout" : true`. Make this setting for the section in which you want the UI table to adapt the condensed style.

Sample Code

```
"ObjectPage|STTA_C_MP_Product": {
                                        "entitySet": "STTA_C_MP_Product",
                                        "component": {
                                              "name":
"sap.suite.ui.generic.template.ObjectPage",
                                              "settings": {
                                                    "sections": {

"to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {

"condensedTableLayout": true,
                                                                }
                                                    }
                                              }
                                        }
```

> **i Note**
>
> - The `condensedTableLayout` class can be set only if there is just one section that contains only UI.table or if the icon tab bar is used for navigation between sections, and any section has only one table in it. Otherwise, the class is not applied for `ui.table` even if the manifest key is set to true.
> - The app needs to run in compact mode. If it runs in condensed mode, the class is not set, even if the manifest key is set to true.
> - Condensed mode can only be applied to `ui.table`.

Related Information

Content Densities [page 1142]

Copying and Pasting from Microsoft Excel to Editable Tables

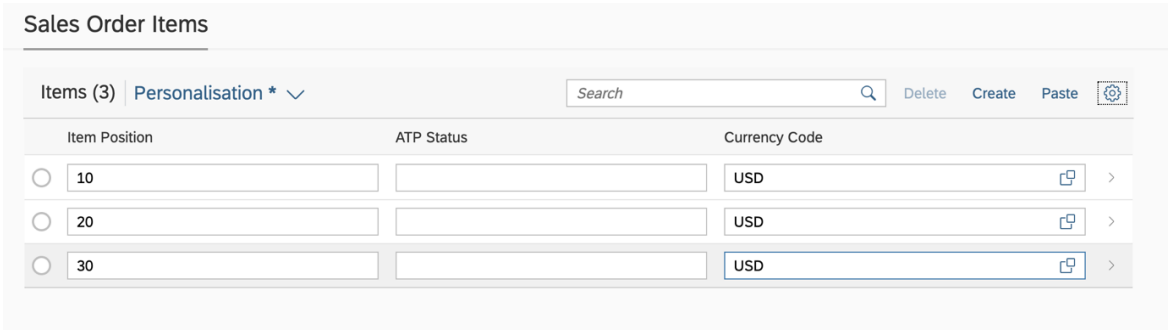
Users can copy and paste data from Microsoft Excel to responsive tables and grid tables.

You need to fulfill these prerequisites to use this feature:

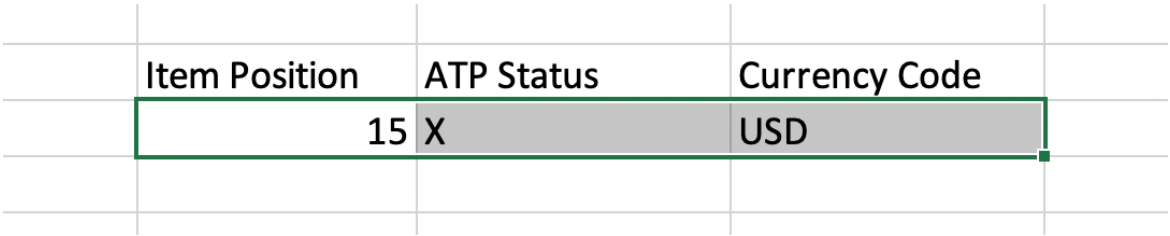
- The app is draft-enabled.
- Inline creation is enabled for the object page table. See also Enabling Inline Creation of Table Entries on Object Page [page 1769].

If these prerequisites are fulfilled, a *Paste* button is shown in the table toolbar. Once the user has copied data from Microsoft Excel, the focus can be anywhere in the table except for the cells. The user needs to trigger the browser paste (`CTRL` + `V` for Microsoft Windows, `CMD` + `V` for MacOS). The paste function in a smart table is available for these scenarios:

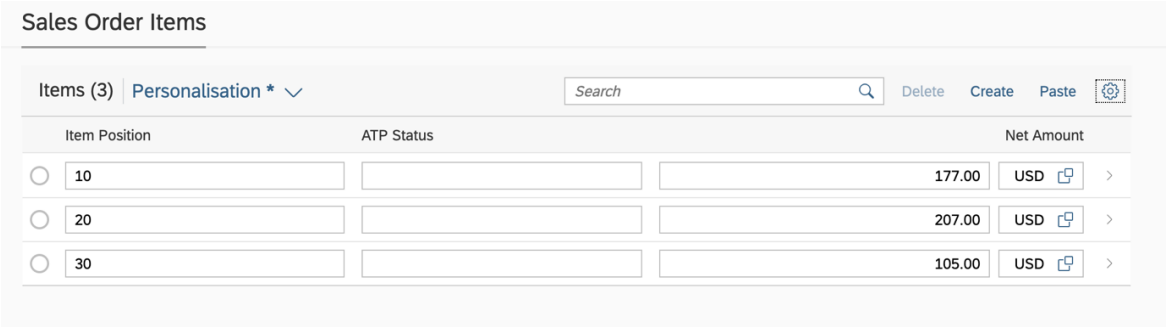
- In edit mode, the smart table only has editable fields.



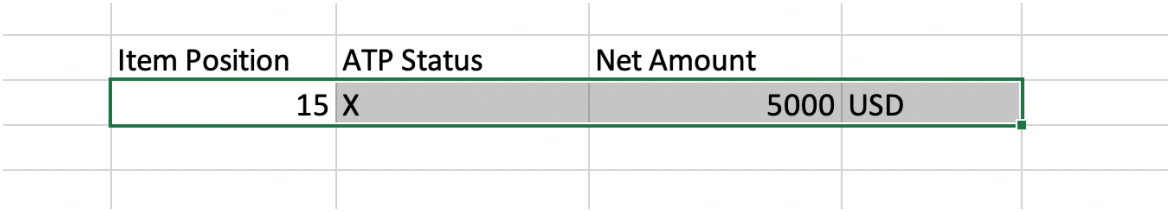
Sample data:



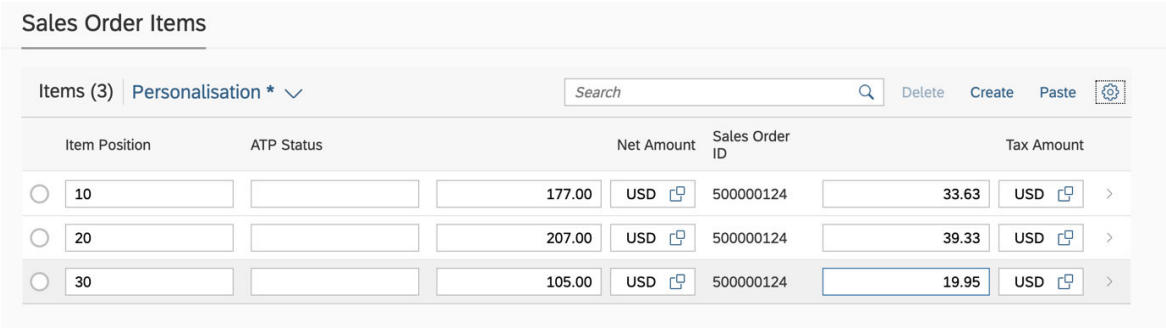
- In edit mode, the smart table only has editable fields. It also contains multiple editable fields in a single column, for example, value and unit of measure.



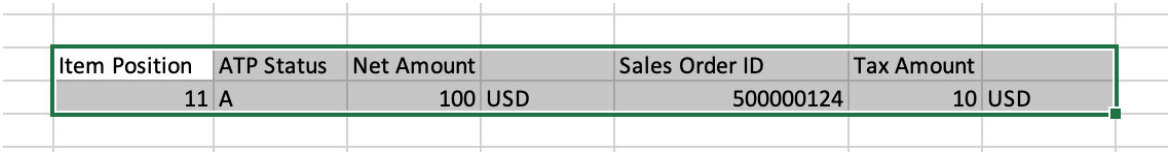
Sample data:



- The smart table has both editable and non-editable fields.

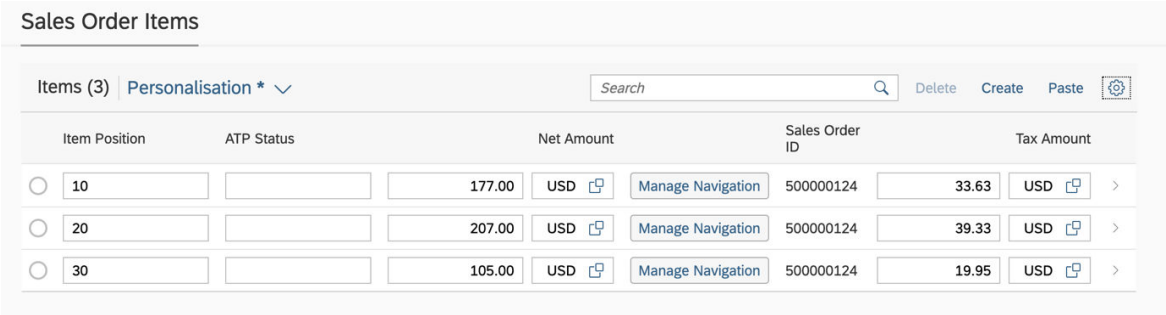


Sample data:

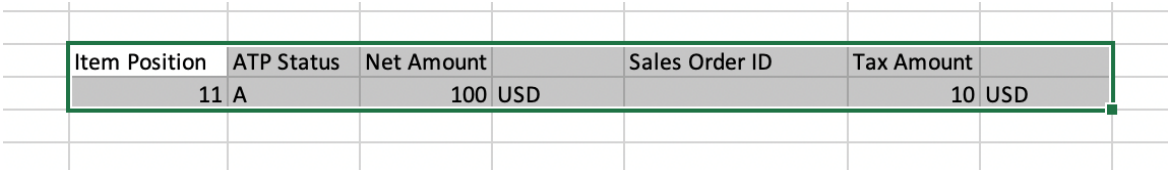


> **i Note**
>
> The user has to maintain the placeholder for non-editable fields in Microsoft Excel. The fields can also be empty.

- The smart table contains an inline action.



Sample data:



Item Position	ATP Status	Net Amount		Sales Order ID	Tax Amount	
11	A	100	USD		10	USD

> **i Note**
>
> Do not make entries for inline actions since the actions are not actual columns in the smart table.

> **i Note**
>
> Only the pasting of simple data fields is supported. Complex fields, such as smart links and images, are not supported.
>
> If there are validation errors, an error message is shown in a dialog box so that the user can take action.
>
> All records that a user pastes are part of one POST batch call. The duration of the POST call increases with the number of records pasted.
>
> The order of the of the data copied from Microsoft Excel might differ from the order in the table after the user has pasted it. SAP Fiori elements cannot control this.
>
> Users cannot paste data into custom columns of tables.
>
> This feature is not supported for tables with custom columns and custom tables.

Adapting Text for Confirmation Dialog Box When Deleting Lines in a Table

When a user deletes a line in a table on the object page, a confirmation dialog box is displayed. You can adapt the displayed default texts for every table.

To do so, you have to provide these custom texts in the application's i18n file, under the respective entitySet. This is the key for those texts:

```
<i18n_Key>|<EntitySet>|<navigationProperty>|com.sap.vocabularies.UI.v1.LineItem
```

If the annotation has a qualifier for the line item annotation, then the i18n key should be:

```
<i18n_Key>|<EntitySet>|<navigationProperty>|com.sap.vocabularies.UI.v1.LineItem|
<qualifier>
```

Example

```
DELETE_SELECTED_ITEM|STTA_C_MP_Product|to_ProductText|
com.sap.vocabularies.UI.v1.LineItem=Delete this row?
```

You can use these i18n keys:

```
#YMSG, 100: Delete selected item text. "item" to be redefined.

DELETE_SELECTED_ITEM=Delete the selected item?

#YMSG, 100: Delete selected items text. "items" to be redefined.

DELETE_SELECTED_ITEMS=Delete the selected items?

#YMSG, 30: Delete success message. Parameter: {0}= deleted items count. "items" to be redefined.

DELETE_SUCCESS_PLURAL_WITH_COUNT={0} items have been deleted.

#YMSG, 30: Delete success message. Parameter: {0}= deleted item as count (1). "item" to be redefined.

DELETE_SUCCESS_WITH_COUNT={0} item has been deleted.

#YMSG, 30: Delete error message. Parameter: {0}= non-deleted items as count."items" to be redefined.

DELETE_ERROR_PLURAL_WITH_COUNT={0} items cannot be deleted.

#YMSG, 30: Delete error message. Parameter: {0}= non-deleted item as count (1)."item" to be redefined.

DELETE_ERROR_WITH_COUNT={0} item cannot be deleted.

#YMSG, 30: Delete error message. "items" to be redefined.

DELETE_ERROR_PLURAL=The selected items cannot be deleted.

#YMSG, 30: Delete error message. "item" to be redefined.

DELETE_ERROR=The selected item cannot be deleted.

#YMSG, 30: Delete success message. "items" to be redefined.

DELETE_SUCCESS_PLURAL=The selected items have been deleted.

#XMSG: Message box text after successfully deleting an object or sub-item. "item" to be redefined.

ITEM_DELETED=Item deleted

#YMSG, 100: Delete undeletable items text: {0}=digit, {1}=digit

DELETE_UNDELETABLE_ITEMS={0} of {1} items cannot be deleted.
```

If you don't provide any custom texts, the system uses the default texts listed above.

Configuring Further Common Features

You can use annotations to set up various elements that are common to the list report and the object page, such as status colors and navigation.

Using Action Control for Context-Dependent Actions

You can use action control to display actions by adding the `sap:action-for` and `sap:applicable-path` terms to your action or function import.

Actions you've defined using these annotations are context-dependent. This means that users have to select an item or a line in a list. Only then are the actions enabled.

In the applicable path, you specify a Boolean property for the entity type that controls whether the function import can be invoked. SAP Fiori elements evaluates the applicable path and sets the visibility of the corresponding action based on the boolean property value.

If the condition as defined by the applicable-path variable is not fulfilled, the action is hidden or disabled as follows:

- hidden if on page header level or for line item buttons in a table
- disabled for header buttons in a table

Sample Code

```
<FunctionImport Name="SEPMRA_C_PD_ProductCopy"
                ReturnType="SEPMRA_PROD_MAN.SEPMRA_C_PD_ProductType"
                EntitySet="SEPMRA_C_PD_Product" m:HttpMethod="POST"
                sap:action-for="SEPMRA_PROD_MAN.SEPMRA_C_PD_ProductType"
                sap:applicable-path="IsActiveEntity">
    <Parameter Name="ProductDraftUUID" Type="Edm.Guid" Mode="In"/>
    <Parameter Name="ActiveProduct" Type="Edm.String" Mode="In"
MaxLength="10"/>
</FunctionImport>
```

Note

If multi-select is available in a table, the action is allowed when at least one of the marked entries fulfils the condition.

Adding Determining Actions

You can add a determining action button to the footer of the list report view or to the footer of the object page.

Determining actions are used to trigger actions directly using the context of the table in the list report, or the context of the page in the object page.

Two types of determining actions are supported:

- Actions that trigger a back-end call through the OData service (`Function`, `FunctionImport`, `Action`, or `ActionImport`), represented by the complex type `DataFieldForAction`
- Actions that trigger intent-based navigation, represented by the complex type `DataFieldForIntentBasedNavigation`

To add a determining action to the footer of the list report or object page, use the annotation term `UI.LineItem` and set the `Determining` property to `true` for the complex type. This is displayed as shown below:

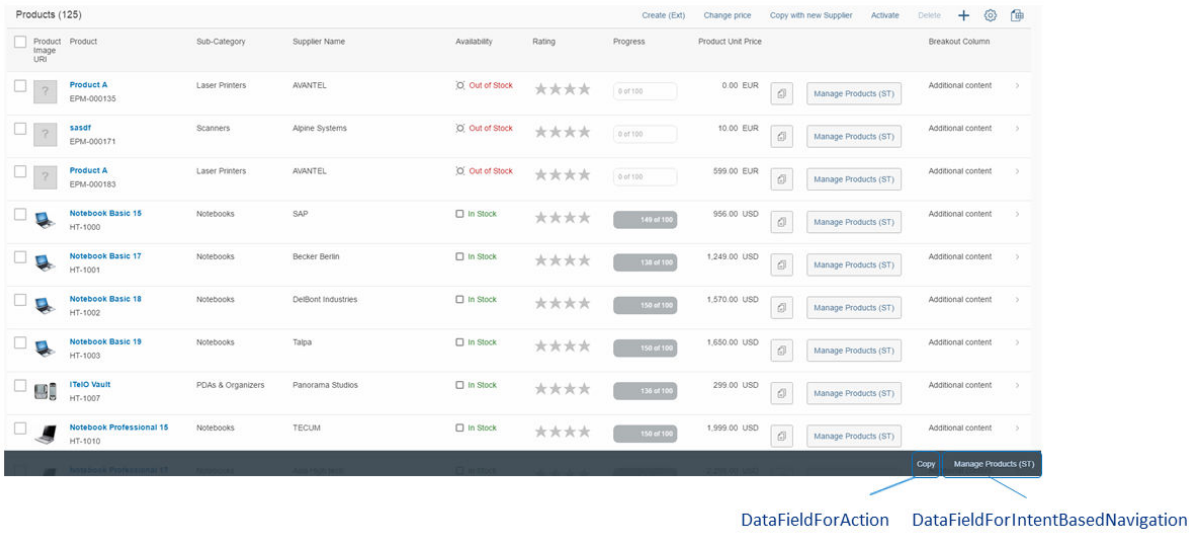


Figure 322: List Report: Determining Action in Footer

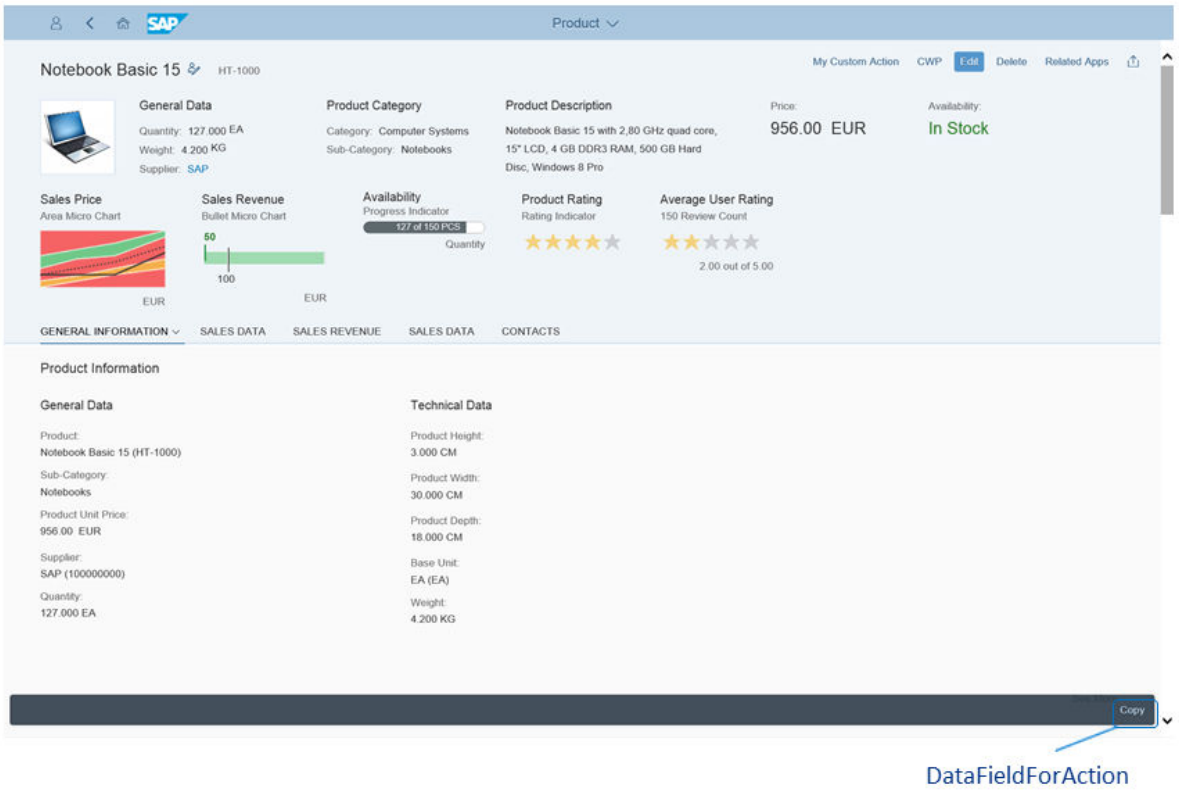


Figure 323: Object Page: Determining Action in Footer

Code Samples

List Report

The following code sample shows an example of how to create your annotations for the determining actions in the list report:

```
<Annotation Term="UI.LineItem">
   <Collection>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy with new Supplier"/>
         <PropertyValue Property="Action"
             String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
            STTA_C_MP_ProductCopywithparams"/>
         <Annotation Term="UI. InvocationGrouping"
            EnumMember="UI.OperationGroupingType/Isolated"/>
      </Record>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy"/>
         <PropertyValue Property="Action"
            String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/STTA_C_MP_ProductCopy"/>
         <PropertyValue Property="Determining" Bool="true"/>
         <Annotation Term="UI. InvocationGrouping"
            EnumMember="UI.OperationGroupingType/Isolated"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
         <PropertyValue Property="Label" String="Manage Products (ST)"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct"/>
         <PropertyValue Property="Action" String="manage_st"/>
         <PropertyValue Property="Determining" Bool="true"/>
      </Record>
```

```
   </Collection>
</Annotation>
```

> **Note**
>
> The `UI.LineItem` vocabulary term is used to define the columns for the smart table.

In the example above for the first record type, the `DataFieldForAction` complex type does not contain the `Determining` property. Therefore, the action button will appear in the smart table toolbar.

With the last two record types, the `DataFieldForAction` and `DataFieldForIntentBasedNavigation`, complex types are used and contain the `Determining` property, which is set to `true`. This means the action buttons will appear in the footer.

Object Page

The following code sample shows how to create your annotations for the determining actions on the object page:

```
<Annotation Term="UI.Identification">
    <Collection>
       <Record Type="UI.DataFieldForAction">
          <PropertyValue Property="Label" String="CWP"/>
          <PropertyValue Property="Action"
             String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
             STTA_C_MP_ProductCopywithparams"/>
          <Annotation Term="UI.OperationGrouping"
             EnumMember="UI.OperationGroupingType/Isolated"/>
          <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
       </Record>
       <Record Type="UI.DataFieldForAction">
          <PropertyValue Property="Label" String="Copy"/>
          <PropertyValue Property="Action"
             String="STTA_PROD_MAN.STTA_PROD_MAN_Entities/
STTA_C_MP_ProductCopy"/>
          <PropertyValue Property="Determining" Bool="true"/>
          <Annotation Term="UI.OperationGrouping"
             EnumMember="UI.OperationGroupingType/Isolated"/>
          <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
       </Record>
    </Collection>
 </Annotation>
```

> **Note**
>
> The `UI.Identification` vocabulary term is used to define the actions on the object page.

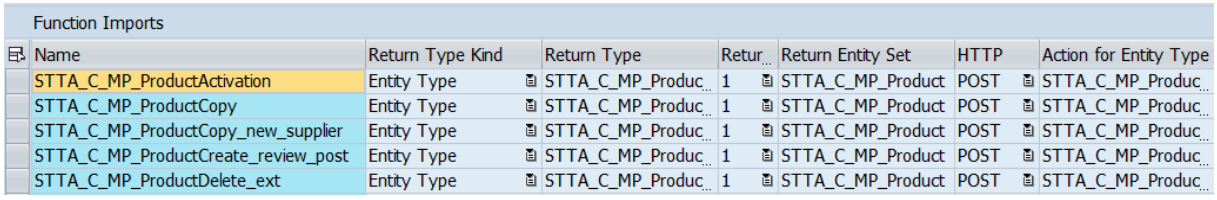
In the example above for the first record type, the `DataFieldForAction` complex type does not contain the `Determining` property. That's why the action button will appear in the object page header.

With the last record type, the `DataFieldForAction` complex type is used and contains the `Determining` property, which is set to `true`. This means the action button will appear in the footer. Note that if the `Determining` property is either not present or is set to `false`, the action will not appear in the footer.

Adding Action-Specific Messages to Confirmation Dialog Boxes

You can create message texts for specific critical actions. They are displayed in the confirmation dialog box for the action.

You can add action-specific messages to the i18n.properties file of your application. The key to be added needs to have the following format: `ACTION_CONFIRM|<FunctionImportName>`, where `FunctionImportName` is specific to a particular action. The figure below shows some examples:



Function Imports

Name	Return Type Kind	Return Type	Retur...	Return Entity Set	HTTP	Action for Entity Type
STTA_C_MP_ProductActivation	Entity Type	STTA_C_MP_Produc...	1	STTA_C_MP_Product	POST	STTA_C_MP_Produc...
STTA_C_MP_ProductCopy	Entity Type	STTA_C_MP_Produc...	1	STTA_C_MP_Product	POST	STTA_C_MP_Produc...
STTA_C_MP_ProductCopy_new_supplier	Entity Type	STTA_C_MP_Produc...	1	STTA_C_MP_Product	POST	STTA_C_MP_Produc...
STTA_C_MP_ProductCreate_review_post	Entity Type	STTA_C_MP_Produc...	1	STTA_C_MP_Product	POST	STTA_C_MP_Produc...
STTA_C_MP_ProductDelete_ext	Entity Type	STTA_C_MP_Produc...	1	STTA_C_MP_Product	POST	STTA_C_MP_Produc...

Figure 324: Function imports

The key can also be interpreted as `ACTION_CONFIRM|<EntitySetName><ActionName>`.

Add the key value pair and the message text to the i18n.properties file of your application, as follows:

Sample Code

```
#XMSG: Messagebox text for confirming an action question
ACTION_CONFIRM|STTA_C_MP_ProductActivation = Are you sure you really want to
activate this product?
```

Note

If you have not specified a custom action for an action, the generic message, "Do you really want to execute the action <Action Label>?" is displayed.

Adding Confirmation Popovers for Actions

You can display a popover when a user triggers an action.

To do so, add the `IsActionCritical` annotation as shown below:

```
<Annotations Target="GWSAMPLE_BASIC.GWSAMPLE_BASIC_Entities/RegenerateAllData">
        <Annotation Term="com.sap.vocabularies.Common.v1.IsActionCritical"
Bool="true"/>
</Annotations>
```

Prefilling Fields When Creating a New Entity

When a user creates a new entity, it is possible to prefill fields with specific values.

If the entity is draft-enabled, the relevant service implementation can be used to prefill the fields.

In some cases, the values result from a user interaction that took place before the creation of the entity was triggered. Then, the relevant information must be transfered from the front end to the back end.

> **i Note**
>
> Use this feature carefully as the user might not expect to come upon prefilled fields.
>
> Do not use this feature if the fields are not (or might not be) visible to the user.

There are two options for supporting the prefilling of fields on the UI. You can use them only for the main object.

- Creation via cross-app navigation
- Prefilling Fields When Creating a New Entity Using an Extension Point [page 1850]

Creation via Cross-App Navigation

When navigating to an app that is based on SAP Fiori elements, set `parameterMode` or `preferredMode` to `create` to indicate that a new instance is to be created (and the user starts with the *Create* screen of this instance).

The source app might add values for properties to be prefilled in the created instance by adding the corresponding name/value-pair as a startup-parameter for the target app.

> **i Note**
>
> The startup-parameters must contain a single value for the property.
>
> You have configured the property to be used in the create case, for the target app.

The configuration in the target app needs to be done in the manifest.json of the target app. The following code sample shows how to ensure that a value for `Supplier` can be passed to the app's create process:

> **Sample Code**

```
...
"sap.ui.generic.app": {
    "_version": "???",
    "settings": {
        "inboundParameters": {
            "Supplier": {
                "useForCreate": true
            }
        },
    }
...
```

Status Colors and Icons

You can define status colors and icons.

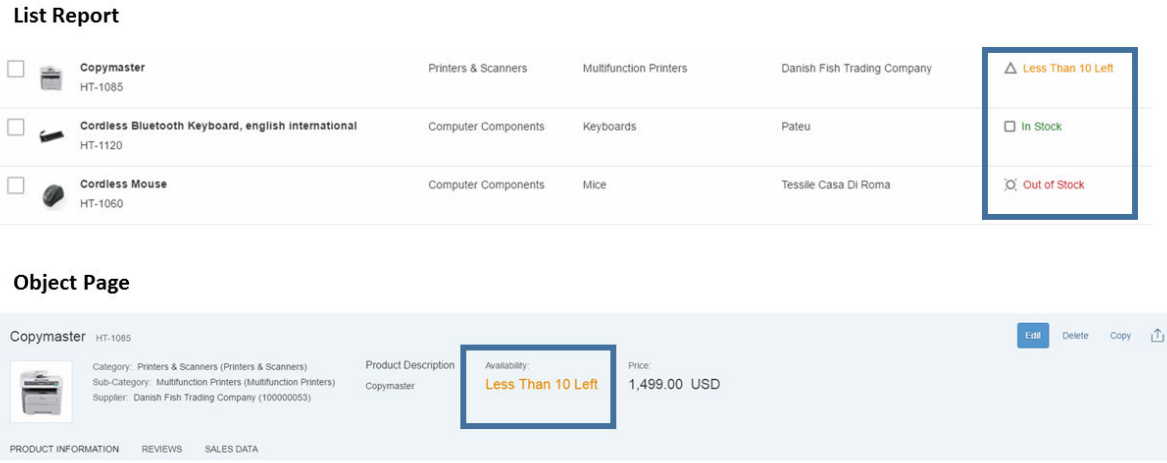
The UI annotations in CDS for `LineItem`, `Identification` and `FieldGroup` have an attribute in the corresponding fields that indicates how critical the field is using colors and icons. This attribute has to refer to another property, which contains the value of the criticality.

```
<Annotation Term="UI.LineItem">
    <Collection>
        ...
        <Record Type="UI.DataField">
            <PropertyValue Property="Criticality" Path="to_ProductStock/
StockCriticality"/>
            <PropertyValue Property="Value" Path="to_ProductStock/
StockEmergencyLevel"/>
            <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/
High"/>
        </Record>
        ...
```

The property containing the criticality can have the following values (derived from the complex type `CriticalityType` of the vocabulary `com.sap.vocabularies.UI.v1`):

- 0 - Neutral
- 1 - Negative
- 2 - Critical
- 3 - Positive

The texts are not static, you can change them to suit your purposes by defining them in the criticality path. The following shows examples of using these values in stock availability of products both in the list report and the object page:



Side Effects

If a user changes the content of a field or performs another activity, this change can potentially influence other fields on the UI. This system behavior is called a side effect.

Side effects are performed in the back end. However, you need to annotate the side effects implemented in the back end using side effect annotations to "inform" the frontend which fields on the UI might be influenced by a change, in order to request new data for these fields. Otherwise, the UI might still display outdated data.

> **i** Note
>
> The so-called default side effects are relevant for the majority of apps. Therefore, you do not need to annotate these side effects.They are available per default. You cannot switch them off.

Default Side Effects

The following side effects are available in SAP Fiori elements by default.

User Action	Side Effect
Creating a new entity / draft version, either in the list report or on the object page	List binding of the parent page is refreshed to show the newly created entity.
Deleting an entity, either in the list report or on the object page	List binding of the parent page is refreshed to remove the deleted entity.
Creating a draft for an active object	List binding of the list report page is refreshed to show the new draft.
Discarding a draft version	List binding of the list report page is refreshed to remove the draft and show the active version.
Activating a draft version	List binding of the list report page is refreshed to remove the draft and show the active version.
Triggering an action	Collection for which the action is annotated is refreshed if the following conditions apply: • The action is a bound action. • The returned instance does not correspond to the bound instance. Example: Copy action.

Side Effects that can be Annotated

A side effect annotation needs to contain the following elements:

- Side effect trigger: A property change, an action, or a structural change (creating or deleting a subitem)
- Data the side effect influences: A property or structural information (to be checked)
- Side effect type: Value change, field control change, or validation
- Qualifier which is sent to the back end to indicate the reason for triggering the request (to be checked)

You can define side effects either in `MPC_EXT` or in local annotation files.

Supported Side Effect Annotation Properties

The following side effect annotations are supported:

- **Source properties**
  You can define a property or a list of properties that create a virtual field group. Once the user leaves this property or field group, the side effect is triggered. .

  > **i Note**
  >
  > You cannot use navigation properties as source properties.

- **Source entities**
  You can specify a 1:n navigation property. The side effect is triggered when structural changes are made (adding or deleting an item). The side effect is not triggered if a property of any entity is changed. This needs to be done in the entity type of the associated entity.

  > **i Note**
  >
  > You cannot specify a 1:1 association or an empty target to ensure that the whole entity is considered as the source.

- **Target properties**
  You can define a property or a list of properties to be refreshed. You can also use 1:1 navigation properties. If this is the case, a request with an expand to this navigation property is triggered.
- **Target entities**
  You can specify 1:1 and 1:n navigation properties.
  In case of 1:n, the request is not sent via the list binding. This means that no paging is considered and new and deleted entries are not updated in the list. Therefore, 1:n should be used carefully. If you specify an empty target, the whole entity is updated.
- **No source properties and no source entities**
  You can define the behavior of the global side effect (user chooses `Enter` without focusing on a field group). You can use this side effect to control whether a prepare or validate, or only a refresh is sent. You can also restrict the updated fields via target properties and target entities.
  See also Using the Global Side Effect [page 1792].

> **i Note**
>
> - Side effects for non-draft apps are supported. The side effects are triggered once the user saves the entity and the save action is successful.
> - You can annotate side effects of actions. If no side effect is annotated, the system does not refresh anymore automatically.
> - If a text arragement annotation is used, especially in combination with a value list annotation, you also need to provide a side effect annotation to indicate that the text must be updated when the user sets a different key.

Side Effect Types

The following side effect types are available:

- ValueChange
  If a prepare action is annotated, a prepare is sent. Note that the complete UI is blocked in this case.
- ValidationMessage
  If a validation is annotated, a validation is sent by the system. This happens asynchronously, and the UI is not blocked.
- FieldControlChange
  Only GETs are sent by the system. The complete UI is blocked.

> **i** Note
>
> Do not misuse the side effect types. If your service has a prepare action but you do not want to trigger the complete prepare in the backend system, you should use the side effect qualifier that is sent to the backend system to decide whether the prepare (and which one) is to be triggered.

Scenario where Local Side Effects cannot be Triggered

As a general rule, no data is sent to the backend until a UI validation error is solved. Hence, side effect is not triggered when there is a validation error related to the source field(s).

For example, if a data field referencing to property `ProductCategory` has field value entered with greater than `MaxLength="40"`, no side effect shall be triggered.

> Sample Code

```
<EntityType Name="SEPMRA_I_ProductCategoryType" sap:label="Category"
sap:content-version="1">
            <Key>
            <PropertyRef Name="ProductCategory"/>
            </Key>
            <Property Name="ProductCategory" Type="Edm.String"
Nullable="false" MaxLength="40" sap:label="Category"/>
            <Property Name="MainProductCategory" Type="Edm.String"
MaxLength="40" sap:label="Main Category" sap:value-list="fixed-values"/>
            <NavigationProperty Name="to_MainCategory"
Relationship="STTA_PROD_MAN.assoc_BBDC3EA034F824A7382F8EEF561C1160"
FromRole="FromRol
       e_assoc_BBDC3EA034F824A7382F8EEF561C1160"
ToRole="ToRole_assoc_BBDC3EA034F824A7382F8EEF561C1160"/>
       </EntityType>
```

Related Information

Side Effect Annotations: Examples [page 1788]
Using the Global Side Effect [page 1792]

Side Effect Annotations: Examples

You define side effects either in the *`MPC_EXT` class or in the local annotation file.

Example: Annotating side effects in the method `DEFINE` of the class

`CL_MM_PUR_PO_AI_MAINT_MPC_EXT`

Sample Code

```
* define Side Effects for Purchase Order:
    DATA lo_ann_target TYPE REF TO /iwbep/if_mgw_vocan_ann_target.   "
Vocabulary Annotation Target                      "#EC NEEDED
    DATA lo_annotation TYPE REF TO /iwbep/if_mgw_vocan_annotation.   "
Vocabulary Annotation                             "#EC NEEDED
    DATA lo_collection TYPE REF TO /iwbep/if_mgw_vocan_collection.   "
Vocabulary Annotation Collection                  "#EC NEEDED
    DATA lo_function   TYPE REF TO /iwbep/if_mgw_vocan_function.     "
Vocabulary Annotation Function                    "#EC NEEDED
    DATA lo_fun_param  TYPE REF TO /iwbep/if_mgw_vocan_fun_param.    "
Vocabulary Annotation Function Parameter          "#EC NEEDED
    DATA lo_property   TYPE REF TO /iwbep/if_mgw_vocan_property.     "
Vocabulary Annotation Property                    "#EC NEEDED
    DATA lo_record     TYPE REF TO /iwbep/if_mgw_vocan_record.       "
Vocabulary Annotation Record                      "#EC NEEDED
    DATA lo_reference  TYPE REF TO /iwbep/if_mgw_vocan_reference.    "
Vocabulary Annotation Reference


    lo_reference = vocab_anno_model->create_vocabulary_reference( iv_vocab_id
= '/IWBEP/VOC_COMMON'

iv_vocab_version = '0001').
    lo_reference->create_include( iv_namespace =
'com.sap.vocabularies.Common.v1' ).
    lo_reference = vocab_anno_model->create_vocabulary_reference( iv_vocab_id
= '/IWBEP/VOC_CORE'

iv_vocab_version = '0001').
    lo_reference->create_include( iv_namespace = 'Org.OData.Core.V1' ).

    lo_ann_target = vocab_anno_model-
>create_annotations_target( 'MM_PUR_PO_AI_MAINTAIN.C_PurchaseOrderEnhWDType' )
 ##NO_TEXT . "Add annotation term for VIPs introduced

    lo_annotation = lo_ann_target->create_annotation( iv_term =
'com.sap.vocabularies.Common.v1.SideEffects' ) ##NO_TEXT .

    lo_record     = lo_annotation->create_record( )  ##NO_TEXT.
    lo_property   = lo_record->create_property( 'SourceProperties' )
##NO_TEXT.
    lo_collection = lo_property->create_collection( ).

    lo_collection->create_simple_value( )->set_property_path( 'Supplier' )
##NO_TEXT .
    lo_collection->create_simple_value( )-
>set_property_path( 'CompanyCode' )  ##NO_TEXT .
    lo_collection->create_simple_value( )-
>set_property_path( 'DocumentCurrency' )  ##NO_TEXT .
```

```
    lo_collection->create_simple_value( )-
>set_property_path( 'PurchasingGroup' )  ##NO_TEXT.
    lo_collection->create_simple_value( )-
>set_property_path( 'PurchasingOrganization' )  ##NO_TEXT.
```

Example: User changes a source property and the system refreshes the price

Sample Code

```
<Annotations Target="NAMESPACE.ENTITYTYPE">
<Annotation Term="com.sap.vocabularies.Common.v1.SideEffects"
Qualifier="PriceChanged">
    <Record>
        <PropertyValue Property="SourceProperties">
            <Collection>
                <PropertyPath>Amount</PropertyPath>
                <PropertyPath>Discount</PropertyPath>
                <PropertyPath>Product</PropertyPath>
            </Collection>
        </PropertyValue>
      <PropertyValue Property="TargetProperties">
         <Collection>
            <PropertyPath>Price</PropertyPath>
         </Collection>
      </PropertyValue>
        <PropertyValue Property="EffectTypes" EnumMember="ValueChange" />
    </Record>
</Annotation>
</Annotations>
```

Example: User changes the supplier and the system refreshes the 1:1 navigation toSupplier

Sample Code

```
<Annotations Target="NAMESPACE.ENTITYTYPE">
<Annotation Term="com.sap.vocabularies.Common.v1.SideEffects"
Qualifier="SupplierChanged">
    <Record>
        <PropertyValue Property="SourceProperties">
            <Collection>
                <PropertyPath>Supplier</PropertyPath>
            </Collection>
        </PropertyValue>
      <PropertyValue Property="TargetEntities">
         <Collection>
            <NavigationPropertyPath>toSupplier</NavigationPropertyPath>
         </Collection>
      </PropertyValue>
        <PropertyValue Property="EffectTypes" EnumMember="ValueChange" />
    </Record>
</Annotation>
</Annotations>
```

Example: User changes a single property, and the system reads the whole entity due to field control

Sample Code

```
<Annotations Target="NAMESPACE.ENTITYTYPE">
<Annotation Term="com.sap.vocabularies.Common.v1.SideEffects"
Qualifier="PriceChanged">
    <Record>
        <PropertyValue Property="SourceProperties">
            <Collection>
                <PropertyPath>Status</PropertyPath>
            </Collection>
        </PropertyValue>
       <PropertyValue Property="TargetEntities">
         <Collection>
            <NavigationPropertyPath></NavigationPropertyPath>
         </Collection>
      </PropertyValue>
        <PropertyValue Property="EffectTypes"
EnumMember="FieldControlChange" />
    </Record>
</Annotation>
</Annotations>
```

Example: Side effect on structural changes of a 1:n association

If any header information or other associated entity needs to be refreshed once a subitem has been created or deleted, you should add side effect annotations as shown in the example below:

Sample Code

```
<Annotations Target="NAMESPACE.ENTITYTYPE">
<Annotation Term="com.sap.vocabularies.Common.v1.SideEffects"
Qualifier="ReactOnItemCreationOrDeletion">
   <Record>
      <PropertyValue Property="SourceEntities">
         <Collection>
            <NavigationPropertyPath>toSalesOrderItems</NavigationPropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="EffectTypes" EnumMember="ValueChange"/>
      <PropertyValue Property="TargetProperties">
         <Collection>
            <PropertyPath>OverallAmount</PropertyPath>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
</Annotations>
```

Example: Side effect after executing an action

After executing an action, but only if the returned entity is different from the entity for which the action was called, the related list binding is refreshed. Therefore, you need to define a side effect annotation for those cases in which any other entity or an association might be changed due to an action call. The target definition's property path that may cover both properties and entities has to express a binding parameter name referring to the entity to which the action is bound.

Sample Code

```
<Annotations
Target="CA_OC_MANAGE_OR_ITEMS_SRV.CA_OC_MANAGE_OR_ITEMS_SRV_Entities/
IssueOutput">
   <Annotation Term="com.sap.vocabularies.Common.v1.SideEffects">
      <Record>
         <PropertyValue Property="EffectTypes" EnumMember="ValueChange"/>
         <PropertyValue Property="TargetProperties">
            <Collection>
                <PropertyPath>_it/to_OutputRequestItemStatus/
OutputRequestItemStatus_Text</PropertyPath>
            </Collection>
         </PropertyValue>
      </Record>
   </Annotation>
</Annotations>
```

Example: Refresh the navigation target

In this case, when the item tax amount is changed, the navigation property leading to the root (`to_SalesOrder`) is updated.

Sample Code

```
<Annotations
Target="STTA_SALES_ORDER_WD_20_SRV.C_STTA_SalesOrderItem_WD_20Type">
              <Annotation Term="com.sap.vocabularies.Common.v1.SideEffects"
Qualifier="TaxAmountChanged">
                     <Record>
                            <PropertyValue Property="SourceProperties">
                                   <Collection>
                                          <PropertyPath>TaxAmount</
PropertyPath>
                                   </Collection>
                            </PropertyValue>
                            <PropertyValue Property="TargetEntities">
                                   <Collection>

<NavigationPropertyPath>to_SalesOrder</NavigationPropertyPath>
                                   </Collection>
                            </PropertyValue>
                            <PropertyValue Property="EffectTypes"
EnumMember="ValueChange"/>
                     </Record>
              </Annotation>
```

Using the Global Side Effect

To optimize performance, make the annotations for the desired side effects as specific as possible.

You can annotate a global side effect (side effect that is triggered when the user chooses `Enter`) for each service and specify the targets (entities or properties) that might be affected by changes to any properties. These targets will be requested whenever the user chooses `Enter`. Only if you annotate a side effect, a page will be refreshed automatically when a change is made.

The `forceGlobalRefresh` parameter in the manifest.json defines the global refresh behavior:

- If the parameter is set to `true`, a global refresh is triggered when the user chooses `Enter`, even if no global side effect has been annotated.
- If the parameter is set to `false`, the system reacts according to what has been annotated.
- If the parameter is not set, for compatibility reasons, the (runtime) default is `true`.

For newly generated apps, the (design time) default is `false`, that is, the SAP WebIDE creation wizard generates `"forceGlobalRefresh": false`.

Sample Code

```
<Annotations Target="NAMESPACE.ENTITYTYPE">
<Annotation Term="com.sap.vocabularies.Common.v1.SideEffects"
Qualifier="GlobalSideEffect">
   <Record>
      <PropertyValue Property="EffectTypes" EnumMember="ValueChange"/>
      <PropertyValue Property="TargetProperties">
         <Collection>
            <PropertyPath>TaxAmount</PropertyPath>
            <PropertyPath>NetAmount</PropertyPath>
            <PropertyPath>GrossAmount</PropertyPath>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
</Annotations>
```

Changing Default Titles for Unnamed Objects

When adding new items to a list report or object page table, the default title *<Unnamed Object>* is displayed for unnamed entities. You can add more specific default titles in your app's i18n file.

In the i18n files of the list report and the object page, change the value of the NEW_OBJECT property to the title you want to be displayed for new entities for the list report or the object page like this:

List report:

```
#XTIT, Default title for unnamed objects

NEW_OBJECT=LR Unnamed Entity
```

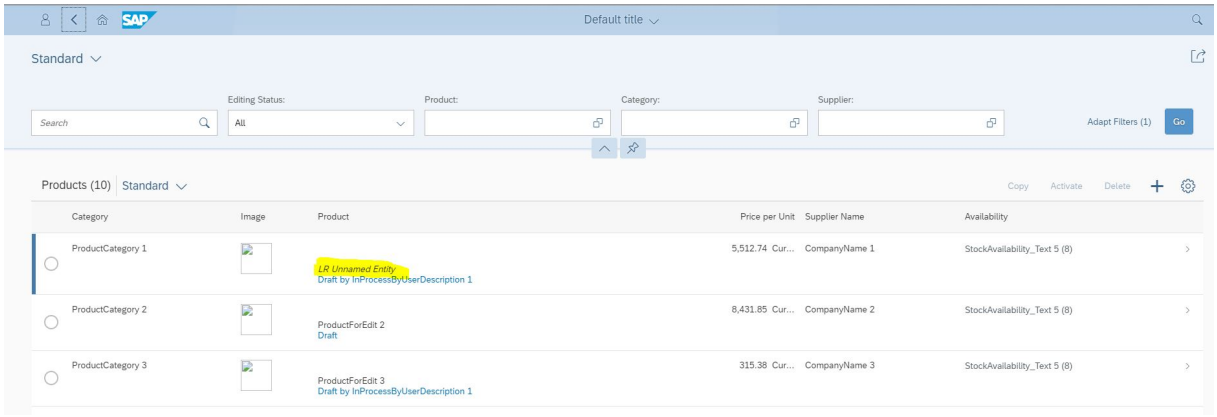


Figure 325: Default title in list report

Object page:

```
#XTIT, Default title for unnamed objects

NEW_OBJECT=OP Unnamed Entity
```

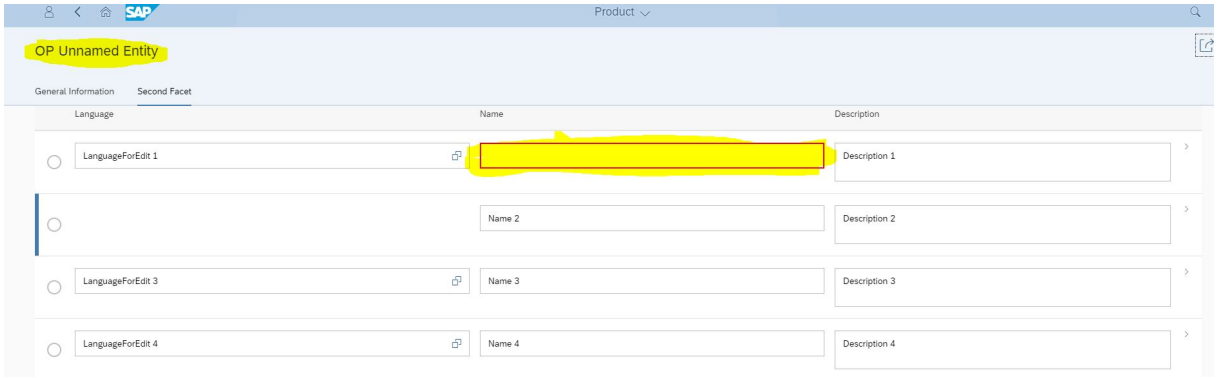


Figure 326: Default title on object page

Enabling Buttons to Display Draft / Saved Values

In the list report and on the object page, you can enable buttons to display and hide draft values.

Sample Code

```
"sap.ui.generic.app": {
            "_version": "1.3.0",
            "settings": {
                "showDraftToggle": true,
            }
},
```

By default, these buttons are hidden. To enable them, set the `showDraftToggle` flag to `True` in the manifest.json.

In the list report table toolbar, this enables the *Hide Draft Values* / *Show Draft Values* button. In the object page header, this enables the *Display Saved Version* / *Return to Draft* button for your draft-enabled applications.

In the list report, the *Hide Draft Values* / *Show Draft Values* button is enabled only if the *Editing Status* in the list report filterbar is set to *All*. Using this button, users can view the saved / active versions of entities if there are any. They can switch back to displaying the modified draft version using the *Show Draft Values* button

On the object page, users see the *Display Saved Version* / *Return to Draft* button when someone lands on their draft.

The following tables show the system behavior for list report and object page:

Mode switch in list report	In list report	When navigating to object page	Navigation back to list report
Hide Draft Values	Only saved objects are displayed in the list report table	Saved object is displayed with *Return to Draft* button if a draft exists for this object.	Only saved objects are displayed in the list report table
Show Draft Values	Saved and draft objects are displayed in the list report table	Object page opens in edit mode for drafts with *Display Saved Version* button	Saved and draft objects are displayed in the list report table

Mode switch on object page	On object page	Navigation back to list report
Display Saved Version	Saved version of record with *Return to Draft* button	State of *Hide / Show Draft* button preserved from when the user last navigated from list report
Return to Draft	Object page opens in edit mode with *Display Saved Version* button	State preserved of *Hide / Show Draft* button from when the user last navigated from list report

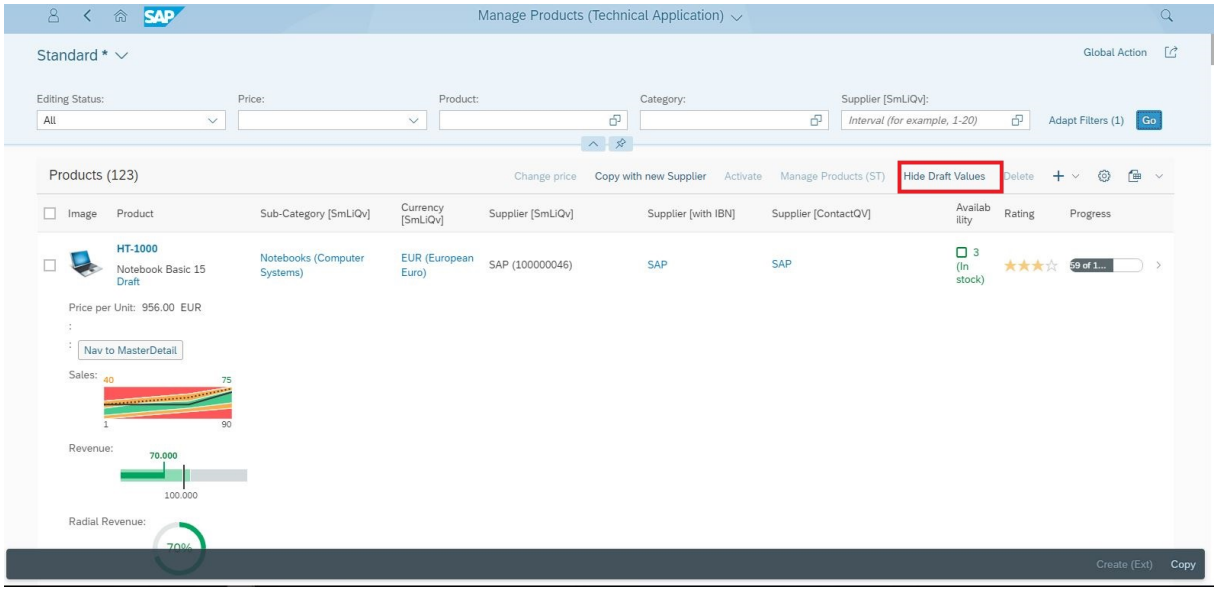


Figure 327: List report - all items are shown

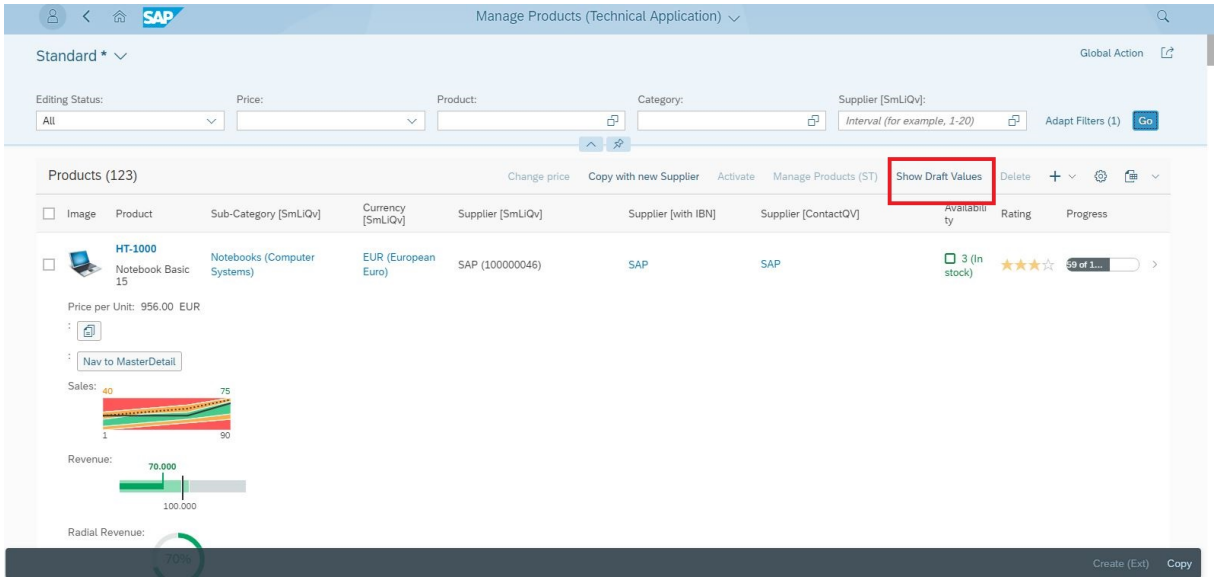


Figure 328: List report - only saved versions of the items are shown

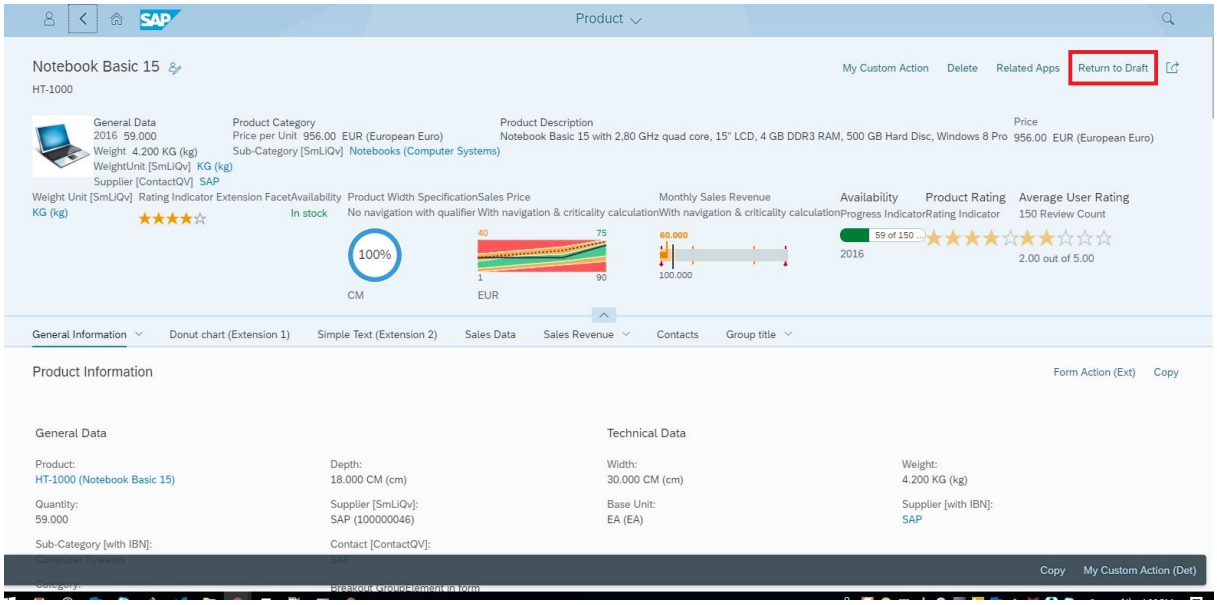


Figure 329: Object page - saved version of the object shown in read mode

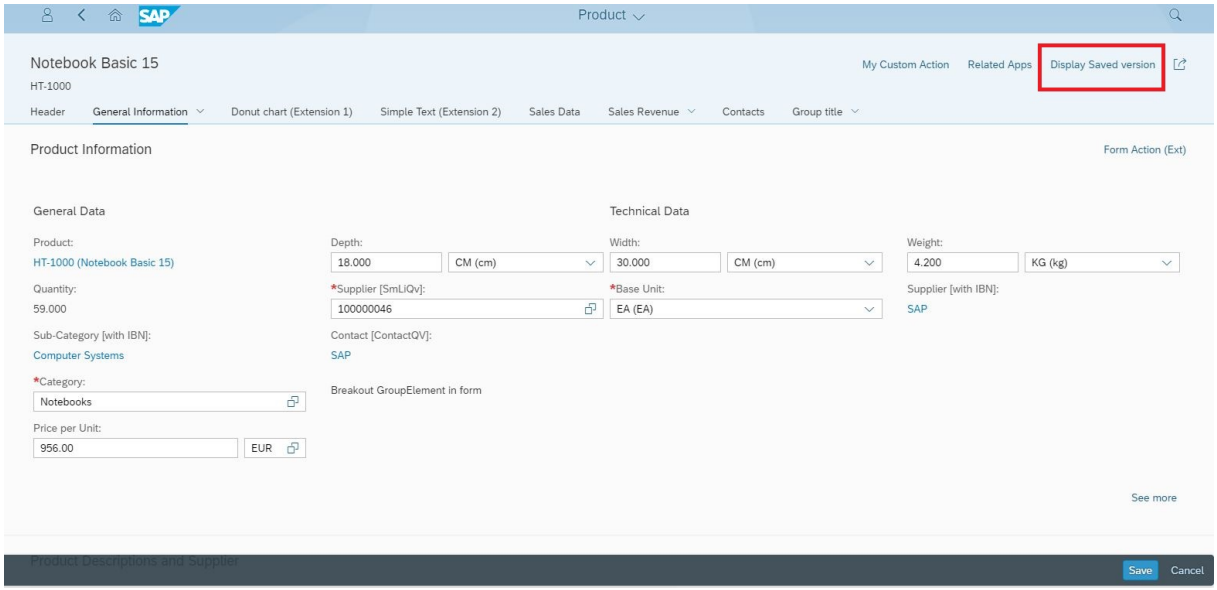


Figure 330: Object page - draft version of the object is shown

Providing Editable Key Fields

Users cannot edit key fields in draft applications. You can introduce an additional key field that is editable.

> **i Note**
>
> This topic is relevant only for applications using existing databases. If you build on a new data model, you can always use an artificial key (typically, a GUID) and model the fields with a semantical meaning only as non-key fields.
>
> However, for existing databases that already use semantically important fields in the key and you need to be able to change them (maybe only provide them by the user upon creation), changing the model in such a way would lead to a data conversion with potentially high impact for customers with huge amounts of data.

> **i Note**
>
> This topic is relevant only for applications that use draft handling. For non-draft applications, as all changes are transported to the backend only in one shot when the object is saved, from a UI point of view, a change in a key property would just mean, that the object returned from the save operation is a different instance than before.

In draft applications, users cannot edit key fields because changes are merged directly into the draft document, and since the key field is part of the draft document's identifier, users would need to navigate to a different document with each merge. From the user experience point of view, this is not a good solution.

As a workaround, you can introduce an additional field that is editable. In an active document, this field's value is always identical to the key field's value. While users work on the draft, only the additional field is changed. Only when the draft is activated, is the additional field's value transferred to the key field. When activated, the system automatically navigates from the draft to the active instance.

On the UI, only the additional field is displayed. However, you need to use the key field for the navigation parameters. It should be consistent with the semantic object attribute.

If a user navigates to a list report or object page application, and if the navigation parameters are not selective enough to directly navigate to the object, you need to set the needed navigation parameters in the filters. As the additional field is used here, SAP Fiori elements need to map the value of the navigation parameter to the additional field. To do this, use the `com.sap.vocabularies.Common.v1.EditableFieldFor` annotation.

Sample Code

```
<Annotations
Target="STTA_SALES_ORDER_WD_20_SRV.C_STTA_SalesOrderItem_WD_20Type/
SalesOrderForEdit">
                        <Annotation
Term="com.sap.vocabularies.Common.v1..EditableFieldFor" String="SalesOrder"/>
                  </Annotations>
                  <Annotations
Target="STTA_SALES_ORDER_WD_20_SRV.C_STTA_SalesOrderItem_WD_20Type/
SalesOrderItemForEdit">
                        <Annotation
Term="com.sap.vocabularies.Common.v1.EditableFieldFor"
String="SalesOrderItem"/>
                  </Annotations>
```

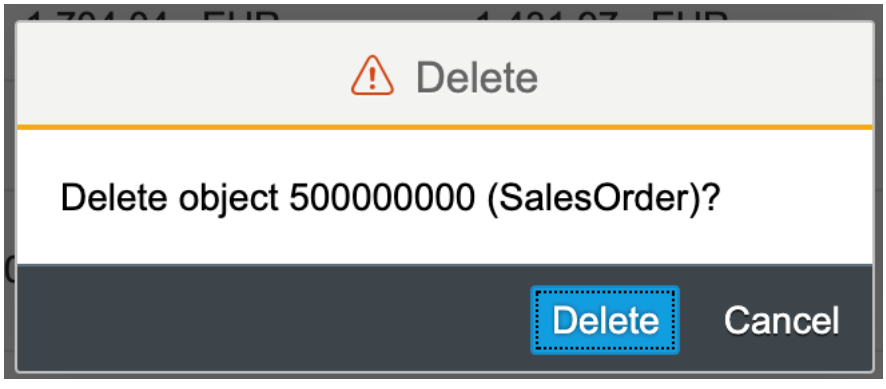
Configuring the Delete Confirmation Dialog Box

You can adapt the text in the Delete dialog box to match your requirements while deleting an object or an item from the list report and object page tables.

The context displayed in the dialog box is taken from the `Title` and `Description` properties of the `UI.HeaderInfo` annotation (defined in the `entitySet` bound to the table).

Depending on the `UI.HeaderInfo` annotation, there can be three different type of text that can appear in the dialog box:

- **When the `UI.HeaderInfo` annotation has both `Title` and `Description` properties defined**
  In this scenario, when you delete a single item from the table, the dialog confirmation would show the message like the one displayed below.

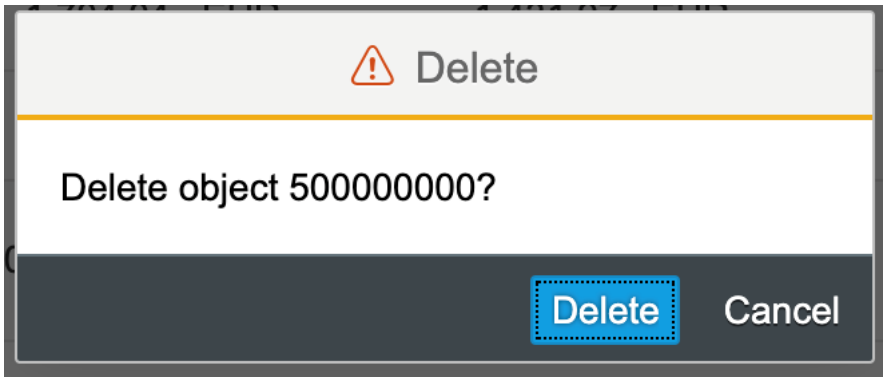


Sample Code

```
<Annotation Term="UI.HeaderInfo">
  <Record>
    <PropertyValue Property="TypeName" String="Sales Order" />
    <PropertyValue Property="TypeNamePlural" String="Sales Orders" />
```

```
    <PropertyValue Property="Title">
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="so_id" />
       </Record>
    </PropertyValue>
    <PropertyValue Property="Description">
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" String="Sales Order" />
      </Record>
    </PropertyValue>
  </Record>
</Annotation>
```

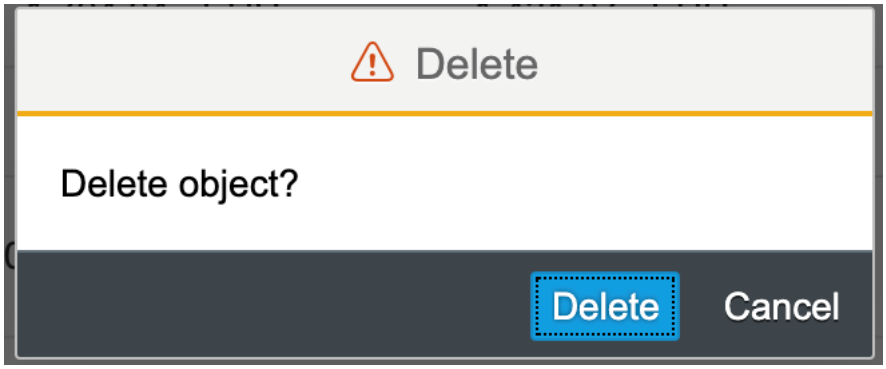
- **When the `UI.HeaderInfo` annotation has only the `Title` property defined**
  In this scenario, when you delete a single item from the table, the dialog confirmation would show the message such as the one displayed below.



Sample Code

```
<Annotation Term="UI.HeaderInfo">
  <Record>
    <PropertyValue Property="TypeName" String="Sales Order" />
    <PropertyValue Property="TypeNamePlural" String="Sales Orders" />
    <PropertyValue Property="Title">
      <Record Type="UI.DataField">
        <PropertyValue Property="Value" Path="so_id" />
       </Record>
    </PropertyValue>
  </Record>
</Annotation>
```

- **When the `UI.HeaderInfo` annotation has neither the `Title` nor the `Description` property defined.**
  In this scenario, when you delete a single item from the table, the dialog confirmation would show the message like the one displayed below.



The applications can override the default text by using the i18 keys mentioned below:

- Main Object: These are applicable when delete is triggered from list report and the main object page.
  - `UI.HeaderInfo` annotation has both `Title` and `Description` defined
    i18n key to be used is `DELETE_WITH_OBJECTINFO`
  - `UI.HeaderInfo` annotation has only `Title` defined
    i18n key to be used is `DELETE_WITH_OBJECTTITLE`
  - `UI.HeaderInfo` annotation has neither `Title` nor `Description` defined
    i18n key to be used is `ST_GENERIC_DELETE_SELECTED`
- Sub Entity: These are appicable when delete is triggered from object page tables or the sub object.
  - `UI.HeaderInfo` annotation has both `Title` and `Description` defined
    i18n key to be used is `DELETE_SELECTED_ITEM_WITH_OBJECTINFO`
  - `UI.HeaderInfo` annotation has only `Title` defined
    i18n key to be used is `DELETE_SELECTED_ITEM_WITH_OBJECTTITLE`
  - `UI.HeaderInfo` annotation has neither `Title` nor `Description` defined
    i18n key to be used is `DELETE_SELECTED_ITEM`

> **i Note**
>
> Applications that are overriding delete message when single entry is being deleted from the table, should revisit the `UI.HeaderInfo` annotation's configuration and adapt the texts accordingly.

Extending List Reports and Object Pages Using App Extensions

Various framework extension points are available for list reports and object pages.

Make sure you have read the following information: Read Before Extending a Generated App [page 1586].

> **⚠ Caution**
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

You can either use the extension wizard in the SAP Web IDE to create extensions or you can create them manually.

In list reports and object pages, you can use SAP Web IDE to add the following extensions:

- List report
  - Filter
  - Action

- Column
- Object page
  - Action
  - Facet
  - Column
  - Header
  - Form

While this documentation describes how to manually define app extensions, the following how-to video shows you how to create a section extension on the object page using the extension wizard: .

Extension Points for Object Page Header Facets

You define application-specific header facets using annotations, but in some cases it might be necessary to integrate components in the front end that are not available with annotations.

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

On the object page, you can use extension points to add additional header facets in the following places:

- Before header facet: The extension is inserted before a given facet.
- Replace header facet: The extension is rendered instead of an existing facet.
- After header facet: The extension is inserted after a given facet.

You must use a view inside the extension to create a header facet extension. Enter the extension information in the following format in the manifest.json of your application:

`<Different_Scenario>|<EntitySet_Name>|headerEditable::<Annotation_Information>`

Specify the extension facet in the form of its annotation path. In addition, you must specify the entity set name, as the same annotation may exist for various entity sets. You also need to define the className, viewName, type and the optional parameter `bVisibleOnEdit`.

> ℹ Note
>
> You only set this optional parameter to "false" if the entire header extension is to be hidden when the object page is edited.

> Sample Code

```
"extends": {
```

```
   "extensions": {
         "sap.ui.viewExtensions": {
                 "sap.suite.ui.generic.template.ObjectPage.view.Details": {
                          "BeforeHeaderFacet|STTA_C_MP_Product|
headerEditable::com.sap.vocabularies.UI.v1.Chart::SpecificationWidthBulletChar
t": {
                                        "className": "sap.ui.core.mvc.View",
                                        "viewName":
"STTA_MP.ext.fragments.HeaderExtensionFacet",
                                        "type": "XML",
                                        "bVisibleOnEdit": true
   }
                          }
            }
      }
}
```

The following extension options are available:

- Standard object header facet: Before, replace, and after scenario:
  - `"BeforeHeaderFacet|<EntitySet Name>|headerEditable::<Annotation information "`
  - `"ReplaceHeaderFacet|<EntitySet Name>|headerEditable::<Annotation information>"`
  - `"AfterHeaderFacet|<EntitySet Name>|headerEditable::<Annotation information>"`
- Simple object header facet: Before, replace, and after scenario:
  `"BeforeSimpleHeaderFacet|<EntitySet Name>|headerEditable::<Annotation information>"`
  `"ReplaceSimpleHeaderFacet|<EntitySet Name>|headerEditable::<Annotation information>"`
  `"AfterSimpleHeaderFacet|<EntitySet Name>|headerEditable::<Annotation information>"`
- Replace the complete object page header with an extension. This means that the `UI.HeaderFacet` annotation is not there. If it is there, remove it from the annotations. The manifest entry should look like this:
  `"ReplaceHeaderExtensionFacet|<EntitySet Name>"`
- Standard object header: If there is no image in the object page header, you can include an extension instead of an image. The manifest entry should look like this:
  `"NoImageExtensionFacet|<EntitySet Name>"`

  > **i Note**
  >
  > You can only use this scenario if there is no value for the `ImageUrl` or `TypeImageUrl` property of the `UI.HeaderInfo` annotation.

  If there is an image in the object page header, you can enter an extension after the image. The manifest entry should look like this:
  `" AfterImageExtensionFacet|<EntitySet Name>"`
- Object page header containing only a `DataPoint` annotation: Before, replace, and after scenario:
  - `"BeforeHeaderDataPoint|<EntitySet Name> | <Annotation Information>"`
  - `"ReplaceHeaderDataPoint|<EntitySet Name> | <Annotation Information>"`
  - `"AfterHeaderDataPoint|<EntitySet Name> | <Annotation Information>"`

Extension Points for Sections on the Object Page

You define application-specific sections in the form of annotations, however, in some cases you might need to integrate components into the front end, for example, charts or attachments.

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

On the object page, you can use extension points to add additional sections in these places:

- `BeforeFacet`: The extension is inserted before a given section.
- `ReplaceFacet`: The extension is rendered instead of an existing section.
- `AfterFacet`: The extension is inserted after a given section.

> ℹ Note
>
> In the manifest.json, you use the term `facet` to add a section to the object page.

You need to specify the section in the form of its annotation path. In addition, you must specify the `entitySet` name, as the same annotation path may exist for various entity sets.

You add this information to the `manifest.json` file, as in these examples:

```
"extends": {
    "extensions": {
        "sap.ui.viewExtensions": {
       "sap.suite.ui.generic.template.ObjectPage.view.Details":
      "BeforeFacet|SEPMRA_C_PD_Product|
to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.BeforeFacetTest",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout before Product Text LineItem"
      }
     },
     "BeforeFacet|SEPMRA_C_PD_Product|
to_ProductText::com.sap.vocabularies.UI.v1.LineItem|1": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.BeforeFacetTestNew",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout before Product Text LineItem",
          "key" : "1"
      }
     }
     "AfterFacet|SEPMRA_C_PD_Product|
to_Supplier::com.sap.vocabularies.UI.v1.Identification": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.AfterFacetTest",
```

```
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout after Supplier Identification"
      }
     },
     "AfterFacet|SEPMRA_C_PD_Product|
to_Supplier::com.sap.vocabularies.UI.v1.Identification|1": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.AfterFacetTest",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout after Supplier Identification",
         "key": 1
      }
}
}
```

> **i Note**
>
> You can specify either a view or a fragment contained in the additional section. Either way, you do not need to use the object page (uxap) tags `ObjectPageSection`, `subSections`, or `ObjectPageSubSection`. These definitions are already part of the template for the object page view. Additional sections are rendered if an extension exists.

For an example with step-by-step instructions, see Adding a Section to an Object Page [page 1803].

For more information, see View Extension [page 2149].

Adding a Section to an Object Page

You can add an additional section to your object page, as described below.

> **⚠ Caution**
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

For this example, you want to add a section called *Product Description* to the object page of the *Manage Products* app.

> **i Note**
>
> This documentation describes how to manually define extensions. The following video provides an example for how to create a section extension on the object page using the extension wizard:

Step 1: Create fragment for the new facet

In the SAP Web IDE, open the folder structure of the Manage Products project and proceed as follows:

1. In the `webapp` folder, create a new subfolder called `ext`.
2. In the folder `ext`, create a new subfolder called `view`.
3. In the `view` folder, create file `DescriptionBreakout.view.xml`.
4. Define the view with its elements, here a `TextArea` that consumes the section title for the product description text in the original language.

Sample Code

```
<core:View xmlns:core="sap.ui.core" xmlns="sap.m">
    <VBox>
        <TextArea id="DescriptionTextArea"
                value="{to_ProductTextInOriginalLang/Description}"
                width="30%"
                editable="false"/>
    </VBox>
</core:View>
```

Step 2: Add section title to the i18n file

To make the section title translatable, add the text to the `i18n` file as follows:

Sample Code

```
#This is the resource bundle for Manage Products

# XTIT: Title of a facet within an object page if not needed in local/
annotations.xml
ProductDescription=Product Description
```

Step 3: Add extension definition to the manifest.json file

To add the extension definition to the `manifest.json` file, use a `viewExtension`.

The extension appears within the `ObjectPage.view` using the `AfterFacet` option.

Sample Code

```
manifest.json
    "extends": {
            "extensions": {
                "sap.ui.viewExtensions": {
                    "sap.suite.ui.generic.template.ObjectPage.view.Details": {
                        "AfterFacet|SEPMRA_C_PD_Product|GeneralInformation": {
                            "className": "sap.ui.core.mvc.View",
                            "viewName":
"ManageProducts.ext.view.DescriptionBreakout",
```

```
                            "type": "XML",
                            "sap.ui.generic.app": {
                                "title": "{{ProductDescription}}"
                            }
                        }
                    }
                }
            }
        },
```

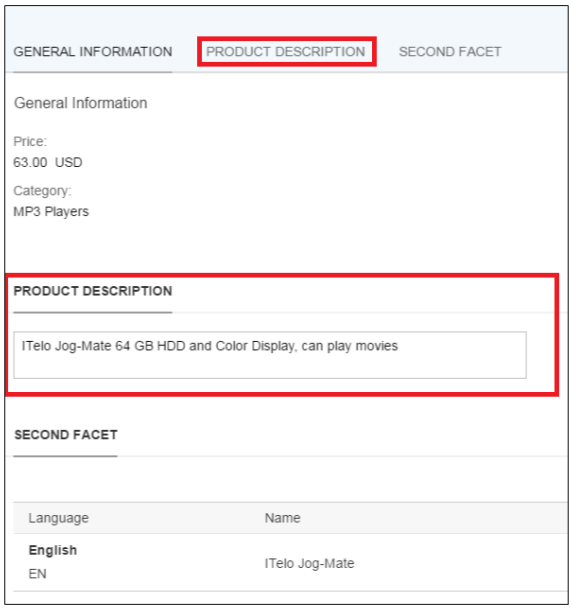
To add multiple sections, the extension name needs to contain a key after the annotation name in the extension entry, for example, `"BeforeFacet|SEPMRA_C_PD_Product|to_ProductText::com.sap.vocabularies.UI.v1.LineItem|1"`, as well as a `key` object in `sap.ui.generic.app`.

Sample Code

```
"extends": {
    "extensions": {
     "BeforeFacet|SEPMRA_C_PD_Product|
to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.BeforeFacetTest",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout before Product Text LineItem"
      }
     },
     "BeforeFacet|SEPMRA_C_PD_Product|
to_ProductText::com.sap.vocabularies.UI.v1.LineItem|1": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.BeforeFacetTestNew",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout before Product Text LineItem",
          "key" : "1"
      }
     }
     "AfterFacet|SEPMRA_C_PD_Product|
to_Supplier::com.sap.vocabularies.UI.v1.Identification": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.AfterFacetTest",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout after Supplier Identification"
      }
     },
     "AfterFacet|SEPMRA_C_PD_Product|
to_Supplier::com.sap.vocabularies.UI.v1.Identification|1": {
      "className": "sap.ui.core.Fragment",
      "fragmentName": "nw.epm.refapps.st.prod.manage.ext.AfterFacetTest",
      "type": "XML",
      "sap.ui.generic.app": {
         "title": "Facet Breakout after Supplier Identification",
         "key": 1
      }
```

Results

The object page of the *Manage Products* app shows the new section *Product Description*:



Adding Dynamic Side Content to Object Page Sections

Sometimes it might be necessary to add additional information that is not available with annotations to object page sections or subsections.

You can use extension points to add additional content to sections in the following places:

- `BeforeMainContent`: The extension is added before a sections' main content.
- `AfterMainContent`: The extension is added after the section's main content.

Main content refers to the information that comes from the annotations.

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Enter the extension information in the manifest.json of your application in the following format:
`<Different_Scenario>|<EntitySet_Name>|<Annotation_Information_of_Subsection>`

Specify the extension facet in the form of the annotation information of the subsection where side content needs to be placed. In addition, specify the entity set name, as the same annotation may exist for various entity sets. You also need to define the `className`, `viewName`, `type`, and the optional parameter `equalSplit`.

To enable the equal split mode, (50:50 percent for main content vs. side content), add the `"equalSplit": true` setting to the manifest.

If `"equalSplit"` is set to `false` or is not defined, the percentage of main content and side content depends on the device on which the app is running.

Sample Code

```
"extends": {
   "extensions": {
         "sap.ui.viewExtensions": {
                 "sap.suite.ui.generic.template.ObjectPage.view.Details": {
                          "AfterMainContent|STTA_C_MP_Product|
GeneralInformationForm":{
                "className": "sap.ui.core.Fragment",
                "fragmentName": "STTA_MP.ext.fragments.SideContentExtension",
                "type": "XML",
        "equalSplit": true
                },
"BeforeMainContent|STTA_C_MP_Product|
to_ProductSalesData::com.sap.vocabularies.UI.v1.Chart":{
               "className": "sap.ui.core.Fragment",
               "fragmentName": "STTA_MP.ext.fragments.SideContentExtension",
               "type": "XML"
                }
                          }
            }
     }
}
```

You can specify either a view or a fragment contained in the section. You do not need to use the object page (uxap) tags, `ObjectPageSection`, `subSections`, or `ObjectPageSubSection`. These definitions are already part of the template for the object page view.

After you have added side content, the system displays a button in the subsection toolbar to show or hide the side content. The default texts for this button are *Show Details* or *Hide Details*. If you want to provide a custom text, specify it by adding the key value pair of the custom label to the i18n.properties file of the specific entity set of object page. The key uniquely defines the subsection for whose side content button you provide the custom text. The structure of the key is as follows:

Note

In the annotation information of the subsection, replace all separators (-- , :: etc) with a | (vertical bar) while forming the key.

- Show the side content button
  `ShowSideContent|<EntitySet>|<Annotation Info of the Subsection>`
  Example: `ShowSideContent|STTA_C_MP_Product|to_ProductSalesData|com.sap.vocabularies.UI.v1.Chart`
  `ShowSideContent|STTA_C_MP_Product|GeneralInformationForm`
- Hide the side content button
  `HideSideContent|<EntitySet>|<Annotation Info of the Subsection>`

Example: `HideSideContent|STTA_C_MP_Product|to_ProductSalesData|com.sap.vocabularies.UI.v1.Chart`
`HideSideContent|STTA_C_MP_Product|GeneralInformationForm`

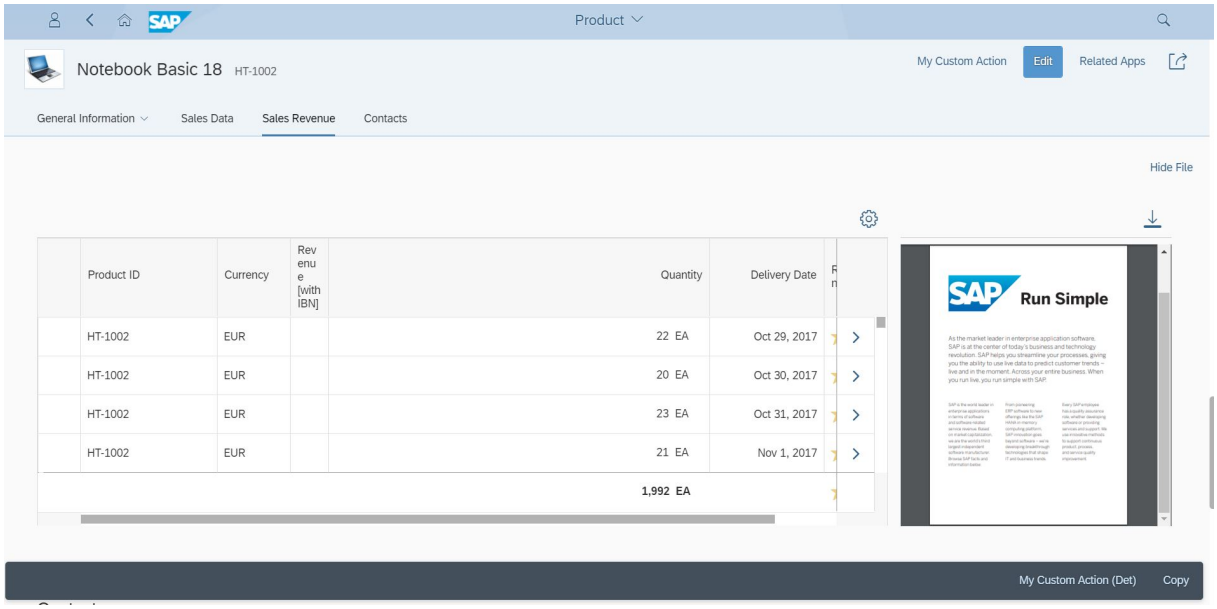


Figure 331: Side Content Added to the Object Page

> **i Note**
>
> - Use dynamic side content for small previews. Do not consider it as an extension of the main panel's content.
> - Do not use tables in the side content panel.
> - Avoid any content that may introduce a horizontal scroll bar.
> - For better content visualization of the dynamic side content, use the 50% screen display of the dynamic side content.

Extension Points for Subsections on the Object Page

On the object page, you can use extension points to add additional subsections.

> **⚠ Caution**
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

You can add additional subsections in existing facets:

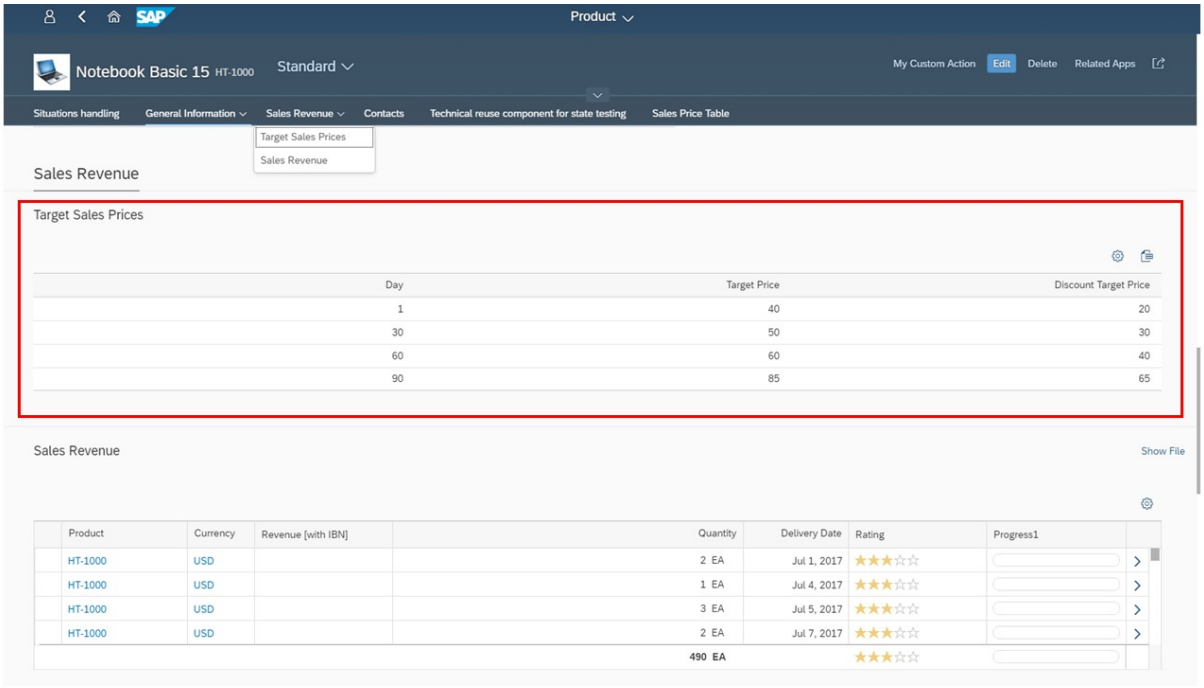
- `BeforeSubSection`: The extension is inserted before a given subsection in a facet
- `AfterSubSection`: The extension is inserted after a given subsection in a facet
- `ReplaceSubSection`: The extension replaces an existing subsection in a facet.

You must specify the subsection in the form of its annotation path. You also have to specify the entitySet name, as the same annotation path may exist for various entity sets. You add this information to the manifest.json file, as in the example. For more information, see Extension Points for Sections on the Object Page [page 1802].

Sample Code

```
"sap.ui.viewExtensions": {
   "sap.suite.ui.generic.template.ObjectPage.view.Details": {
      "BeforeSubSection|STTA_C_MP_Product|
to_ProductSalesData::com.sap.vocabularies.UI.v1.Chart":{
         "className": "sap.ui.core.mvc.View",
         "viewName": "STTA_MP.ext.view.ProductSalesPrice",
         "type": "XML",
         "sap.ui.generic.app": {
            "title": "Target Sales Prices",
            "enableLazyLoading": true
         }
      },
       "AfterSubSection|STTA_C_MP_Product|
to_ProductSalesData::com.sap.vocabularies.UI.v1.LineItem":{
         "className": "sap.ui.core.mvc.View",
         "viewName": "STTA_MP.ext.view.ProductSalesPrice",
         "type": "XML",
         "sap.ui.generic.app": {
            "title": "Target Sales Prices",
            "enableLazyLoading": true
         }
     "ReplaceSubSection|STTA_C_MP_Product|
to_ProductTextType::com.sap.vocabularies.UI.v1.LineItem":{
         "className": "sap.ui.core.mvc.View",
         "viewName": "STTA_MP.ext.view.ProductSalesPrice",
         "type": "XML",
         "sap.ui.generic.app": {
            "title": "Target Sales Prices",
            "enableLazyLoading": true
         }
      },
    .....
```

The result looks as shown below. The highlighted subsection has been added using the extension point.



i Note

You can specify either a view or a fragment contained in the additional subsection. Either way, you do not need to use the object page (uxap) tags `ObjectPageSection`, `subSections`, or `ObjectPageSubSection`. These definitions are already part of the template for the object page view. Additional sections are rendered if an extension exists.

Extension Points for Forms on the Object Page

On the object page, you can use extension points to extend forms in sections.

⚠ Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Use the "`SmartFormExtension|<entity name>|<fieldgroup annotation>`" key in the manifest entry to add new fields to an existing field group. In the example below, an extension is added to the *General Information* field group.

Sample Code

```
"sap.suite.ui.generic.template.ObjectPage.view.Details": {
    "SmartFormExtension|STTA_C_MP_Product|
com.sap.vocabularies.UI.v1.FieldGroup::GeneralInformation": {
        "className": "sap.ui.core.Fragment",
        "fragmentName": "STTA_MP.ext.fragments.SmartFormGroupElement",
        "type": "XML"
    }
}
```

Note

`SmartForm Extension` supports only "sap.ui.core.Fragment" for the "className".

See also: Defining the SmartForm Column Layout [page 1664]

Extension Points for Tables

You can use extension points to enhance tables in SAP Fiori elements apps.

Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

You use the following extension points to add additional columns to tables:

Table Type	SAP Fiori Element	Extension Point	Example
All	Object page	`onBeforeRebindTableExtension`	Example: Applying Custom Logic When a Table is Loaded or Refreshed [page 1813]
Responsive table	List report	`ResponsiveTableColumnsExtension|<Name of the EntitySet>` `ResponsiveTableCellsExtension|<Name of the EntitySet>`	Example: Adding Columns to a Responsive Table in the List Report [page 1814]

Table Type	SAP Fiori Element	Extension Point	Example
	Object page	`ResponsiveTableColumnsExtension|<Name of the table EntitySet>|<Facet ID/Annotation Path>` `ResponsiveTableCellsExtension|<Name of the table EntitySet>|<Facet ID/Annotation Path>`	Example: Adding Columns to a Responsive Table on the Object Page [page 1822]
Grid table	List report	`GridTableColumnsExtension|<Name of the EntitySet>`	Example: Adding Columns to a Grid Table in the List Report [page 1824]
	Object page	`GridTableColumnsExtension|<Name of the table EntitySet>|<Facet ID/Annotation Path>`	Example: Adding Columns to a Grid Table in the Object Page [page 1826]
Analytical table	List report	`AnalyticalTableColumnsExtension|<Name of the EntitySet>`	
	Object page	`AnalyticalTableColumnsExtension|<Name of the table EntitySet>|<Facet ID/Annotation Path>`	Example: Adding Columns to an Analytical Table on the Object Page [page 1827]
Tree table	List report	`TreeTableColumnsExtension|<Name of the EntitySet>`	Example: Adding Columns to a Tree Table in the List Report [page 1829]

> **i Note**
>
> `<Name of the EntitySet>` is the EntitySet of the current page. `<Name of the table EntitySet>` is the EntitySet of the table the extension is meant for. Use the `<name of the table EntitySet>` for all table column extensions on the object page, as opposed to all other view extensions on the object page.

You use extension point `ListReportExtension` to replace default navigation within a responsive table in a list report. For more information, see the following example: Example: Replacing Standard Navigation in a Responsive Table in the List Report [page 1817].

You use extension point `DetailsExtension` to replace default navigation within a responsive table on an object page. For more information, see the following example: Example: Replacing Standard Navigation in a Responsive Table on the Object Page [page 1819].

Example: Applying Custom Logic When a Table is Loaded or Refreshed

This example explains how to use an extension point if you want to apply custom logic when your table is being loaded or refreshed.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Procedure

1. Register your extension point in the manifest.json.

   > Sample Code

   ```
   "extends": {
      "extensions": {
         ...
         "sap.ui.controllerExtensions": {
            ...
            "sap.suite.ui.generic.template.ListReport.view.Details": {
               ...
               "controllerName": "STTA_MP.ext.controller.DetailsExtension",
               ...
            }
         }
         ...
   ```

2. Implement your controller extension.

   You have to implement a `onBeforeRebindTableExtension` function within the object page controller extension. Here, it is `DetailsExtension.controller.js`. In this example, three tables are used on the object page.To identify the tables, you should use the table ID.

   > Sample Code

   ```
   sap.ui.controller("STTA_MP.ext.controller.ListReportExtension", {
   ```

```
    onBeforeRebindTableExtension: function(oEvent) {
      var oID = oEvent.getSource().getId();
      var tableId =
"STTA_MP::sap.suite.ui.generic.template.ObjectPage.view.Details::STTA_C_MP_
Product--to_ProductText::com.sap.vocabularies.UI.v1.LineItem::Table";
      // to select only one specific table
      switch (oID) {
        case tableId:
          // implement your logic for table 1 here
          ...
          break;
        default :
          // implement your default logic for all others here
          ...
          return;
       }
    }
    ...
}
```

Example: Adding Columns to a Responsive Table in the List Report

For responsive tables, you have to implement two extension points to add a custom column.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

For information about the configuration options and examples for defining custom columns in smart tables, see Smart Table [page 2444].

The table containing additional custom columns can look like this:

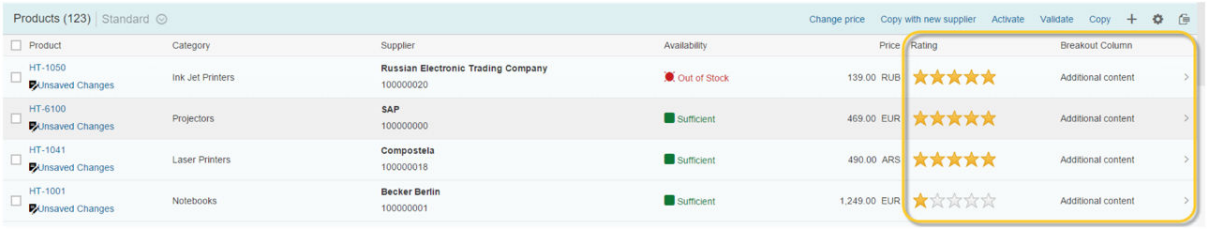


Figure 332: Custom columns in responsive table

Procedure

1. Define a fragment for the view extension.

   For a custom column in a responsive table, you have to implement two extensions. First, implement the definition of the custom columns, then, implement the content of the custom columns.

   You can change the sequence of the columns via the `customData` property `columnIndex`, as shown in the sample code below.

   > **i Note**
   >
   > If the content of your custom column refers to a property (such as `{Price}`), you need to include a corresponding `"leadingProperty"` entry in the `CustomData` of the column definition.

   Sample Code

```
<core:FragmentDefinition xmlns:core="sap.ui.core"
       xmlns="sap.m">
       <Column>
              <Text text="{i18n|sap.suite.ui.generic.template.ListReport|
STTA_C_MP_Product>xfld.Rating}" />
              <customData>
                     <core:CustomData key="p13nData"
                            value='\{"columnKey": "Rating",
"leadingProperty":"Price", "columnIndex" : "100"}' />
              </customData>
       </Column>
       <Column>
              <Text text="{i18n|sap.suite.ui.generic.template.ListReport|
STTA_C_MP_Product>xfld.BreakoutColumn}" />
              <customData>
                     <core:CustomData key="p13nData"
                            value='\{"columnKey": "Test", "columnIndex" :
"101"}' />
              </customData>
       </Column>
</core:FragmentDefinition>
```

   In the example project webapp/ext/fragments/ListReportResponsiveTableColumns.fragment.xml, enter the following:

   Sample Code

```
<core:FragmentDefinition xmlns:core="sap.ui.core" xmlns="sap.m">
       <RatingIndicator value="{= ${Price} > 500 ? 1:5}"></RatingIndicator>
```

```
        <Text text="{i18n|sap.suite.ui.generic.template.ListReport|
STTA_C_MP_Product>xfld.BreakoutColumnContent}"></Text>
</core:FragmentDefinition>
```

2. Register your view extensions in the manifest.json file of your application, as follows:

Sample Code

```
...
 "extends": {
      "extensions": {
        "sap.ui.viewExtensions": {
          "sap.suite.ui.generic.template.ListReport.view.ListReport": {
            "ResponsiveTableColumnsExtension|STTA_C_MP_Product": {
                  "className": "sap.ui.core.Fragment",
                  "fragmentName":
"STTA_MP.ext.fragments.ListReportResponsiveTableColumns",
                  "type": "XML"
            },
            "ResponsiveTableCellsExtension|STTA_C_MP_Product": {
                  "className": "sap.ui.core.Fragment",
                  "fragmentName":
"STTA_MP.ext.fragments.ListReportResponsiveTableCells",
                  "type": "XML"
            },...
```

If you use `QuickVariantSelectionX`, you need to define the extensions per tab. In this case, the names of the extension points are`ResponsiveTableColumnExtension|<EntitySet>|<tabKey>` and `ResponsiveTableCellsExtensions|<EntitySet>|<tabKey>`, respectively. `<tabKey>` is the key provided when defining the `QuickVariantSelectionX`. See also Defining Multiple Views on a List Report Table - Multiple Table Mode [page 1651].

Note

If you do not use |<tab key> as part of the extension point name, for compatibility reasons, the feature will also work. However, you cannot provide stable IDs for the columns and cells.

Sample Code

```
...
"extends": {
  "extensions": {
    "sap.ui.viewExtensions": {
     "sap.suite.ui.generic.template.ListReport.view.ListReport": {
        "ResponsiveTableColumnsExtension|STTA_C_MP_Product|Expensive": {
          "className": "sap.ui.core.Fragment",
          "fragmentName":
"STTA_MP.ext.fragments.ListReportResponsiveTableColumnsExpensive",
          "type": "XML"
        },
        "ResponsiveTableCellsExtension|STTA_C_MP_Product|Expensive": {
          "className": "sap.ui.core.Fragment",
          "fragmentName":
"STTA_MP.ext.fragments.ListReportResponsiveTableCellsExpensive",
          "type": "XML"
        },
        "ResponsiveTableColumnsExtension|STTA_C_MP_Product|Cheap": {
          "className": "sap.ui.core.Fragment",
          "fragmentName":
"STTA_MP.ext.fragments.ListReportResponsiveTableColumnsCheap",
          "type": "XML"
```

```
        },
        "ResponsiveTableCellsExtension|STTA_C_MP_Product|Cheap": {
          "className": "sap.ui.core.Fragment",
          "fragmentName":
"STTA_MP.ext.fragments.ListReportResponsiveTableCellsCheap",
          "type": "XML"
        },...
```

Example: Replacing Standard Navigation in a Responsive Table in the List Report

You can replace the standard navigation from the list report to the object page with your own navigation to an external or internal target.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Procedure

1. Add a navigation target to the manifest.js file.

   In the example below, external navigation (outbound) via the SAP Fiori Launchpad has been added.

   > Sample Code

```
sap.app": {
...
  "crossNavigation": {
     "inbounds": {},
     "outbounds": {
       "EPMProductManageSt": {
         "semanticObject": "EPMProduct",
         "action": "manage",
         "parameters": {
            "preferredMode": {
               "value": {
```

```
                    "value": "display"
                }
            }
        }
    }
  }
....
```

2. Register your extension in the manifest.js file.

Sample Code

```
"extends": {
   "extensions": {
      ...
      "sap.ui.controllerExtensions": {
         ...
         "sap.suite.ui.generic.template.ListReport.view.ListReport": {
            ...
            "controllerName": "STTA_MP.ext.controller.ListReportExtension",
            ...
         }
      }
      ...
```

3. Implement your controller extension.

   You have to implement the `onListNavigationExtension` function within the list report controller extension.

Sample Code

```
onListNavigationExtension: function(oEvent) {
 var oNavigationController = this.extensionAPI.getNavigationController();
 var oBindingContext = oEvent.getSource().getBindingContext();
 var oObject = oBindingContext.getObject();
 // for notebooks we trigger external navigation for all others we use
internal navigation
 if (oObject.ProductCategory == "Notebooks") {
    oNavigationController.navigateExternal("EPMProductManageSt");
 } else {
    // return false to trigger the default internal navigation
    return false;
 }
 // return true is necessary to prevent further default navigation
 return true;
 },
```

4. To ensure that this navigation can be triggered by the user, you must have defined an object page. If you do not want the user to be able to navigate to this object page, follow the procedure described under Changing Navigation to Object Page [page 1583], *Enable External Navigation*.

Example: Replacing Standard Navigation in a Responsive Table on the Object Page

You can replace the standard navigation from the object page with your own navigation to an external or internal target.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Procedure

1. Add a navigation target to the manifest.js file.

   In the example below, external navigation (outbound) via the SAP Fiori Launchpad has been added.

   > Sample Code

   ```
   sap.app": {
   ...
     "crossNavigation": {
        "inbounds": {},
        "outbounds": {
          "EPMProductManageSt": {
            "semanticObject": "EPMProduct",
            "action": "manage",
            "parameters": {
               "preferredMode": {
                  "value": {
                     "value": "display"
                  }
               }
            }
          }
      }
    }
   ....
   ```

2. Register your extension in the manifest.js file.

Sample Code

```
"extends": {
   "extensions": {
      ...
      "sap.ui.controllerExtensions": {
         ...
         "sap.suite.ui.generic.template.ObjectPage.view.Details": {
            ...
            "controllerName": "STTA_MP.ext.controller.DetailsExtension",
            ...
         }
      }
      ...
```

3. Implement your controller extension.

   You have to implement the `onListNavigationExtension` function within the object page controller extension.

Sample Code

```
onListNavigationExtension: function(oEvent) {
 var oNavigationController = this.extensionAPI.getNavigationController();
 var oBindingContext = oEvent.getSource().getBindingContext();
 var oObject = oBindingContext.getObject();
 // for notebooks we trigger external navigation for all others we use
internal navigation
 if (oObject.ProductCategory == "Notebooks") {
    oNavigationController.navigateExternal("EPMProductManageSt");
 } else {
    // return false to trigger the default internal navigation
    return false;
 }
 // return true is necessary to prevent further default navigation
 return true;
 },
```

Example: Enable Internal Navigation for a List Report to Object Pages of Different Entity Sets

You can enable internal navigation to an object page for a list report with different entity sets by using the `onListNavigationExtension` function.

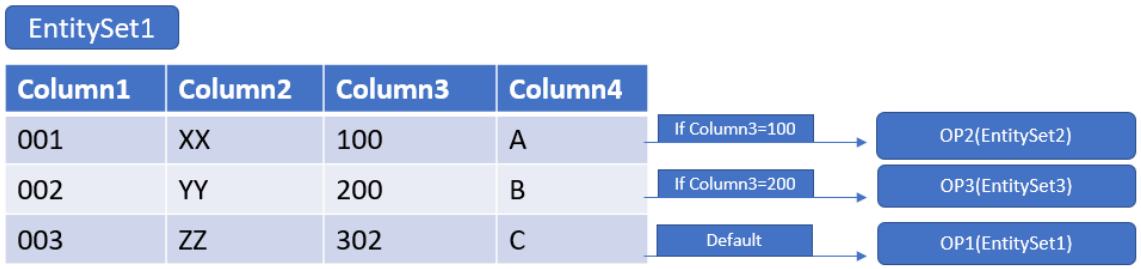
Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

In the extension function, you can define the logic or condition that triggers the navigation to the object page. If none of the conditions mentioned in the extension are met, navigation to the default object page is triggered.

The figure below shows a sample navigation scenario:



The code snippet below shows a sample implementation of the `onListNavigationExtension`.

Sample Code

```
onListNavigationExtension: function(oEvent) {
            var oBindingContext = oEvent.getSource().getBindingContext();
            var oObject = oBindingContext.getObject();
            var sNavigationProperty;
            switch (oObject.Column3){
                case "100":
                    sNavigationProperty = "NavigationProperty1";
                    break;
                case "200":
                    sNavigationProperty = "NavigationProperty2";
                    break;
            }
            if (sNavigationProperty){
                var oExtensionAPI = this.extensionAPI;
                var fnNavigate = function(){
                    return new Promise(function(fnResolve, fnReject){
                        var oModel = oBindingContext.getModel();
                        var oTarget;
                        oModel.createBindingContext(sNavigationProperty,
oBindingContext, {}, function(oTarget){
                            var oNavigationController =
oExtensionAPI.getNavigationController();
                            oNavigationController.navigateInternal(oTarget);
                            fnResolve();
                        });
                    });
                };
            oExtensionAPI.securedExecution(fnNavigate, {
                busy: {
                            check: false
                },
                dataloss: {
                            popup: false
                }
});
                return true;
            }
            return false;
        }
Sample Implementation of Manifest changes:
        "pages": {
                    "ObjectPage|EntitySet1 ": {
                        "entitySet": " EntitySet1",
```

```
                "component": {
                    "name": "sap.suite.ui.generic.template.ObjectPage"
                }
            },
            "ObjectPage| EntitySet2 ": {
                "entitySet": " EntitySet2",
                "component": {
                    "name": "sap.suite.ui.generic.template.ObjectPage"
                }
            },
            "ObjectPage| EntitySet3": {
                "entitySet": " EntitySet3",
                "component": {
                    "name": "sap.suite.ui.generic.template.ObjectPage"
                }
            }
```

Example: Adding Columns to a Responsive Table on the Object Page

For responsive tables, you have to implement two extension points to add a custom column.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

The table containing additional columns can look like this:

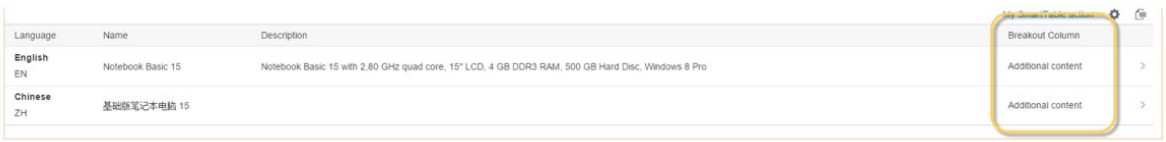


Figure 333: Custom columns in a responsive table on the object page

Procedure

1. Define a fragment for the view extension.

For a custom column in a responsive table, you have to implement two extensions. First, implement the definition of the custom columns and then implement the content of the custom columns.

In the example project: webapp/ext/fragments/ProductTextResponsiveTableColumns.fragment.xml:

Sample Code

```
<core:FragmentDefinition xmlns:core="sap.ui.core"
       xmlns="sap.m">
       <Column>
              <Text text="{i18n|sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumn}" />
              <customData>
                     <core:CustomData key="p13nData"
                            value='\{"columnKey": "Test", "columnIndex" :
"101"}' />
              </customData>
       </Column>
</core:FragmentDefinition>
```

In the example project: ProductTextResponsiveTableCells.fragment.xml:

Sample Code

```
<core:FragmentDefinition xmlns:core="sap.ui.core" xmlns="sap.m">
       <Text text="{i18n|sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumnContent}"></Text>
</core:FragmentDefinition>
```

2. Register your view extensions in the manifest.json file of your application as follows:

   For information on naming, see Extension Points for Tables [page 1811].

Sample Code

```
...
 "extends": {
      "component": "sap.suite.ui.generic.template.ListReport",
      "minVersion": "1.1.0",
      "extensions": {
        "sap.ui.viewExtensions": {
"sap.suite.ui.generic.template.ObjectPage.view.Details": {
            "ResponsiveTableColumnsExtension|STTA_C_MP_ProductText|
to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
                  "className": "sap.ui.core.Fragment",
                  "fragmentName":
"STTA_MP.ext.fragments.ProductTextResponsiveTableColumns",
                  "type": "XML"
            },
            "ResponsiveTableCellsExtension|STTA_C_MP_ProductText|
to_ProductText::com.sap.vocabularies.UI.v1.LineItem": {
                  "className": "sap.ui.core.Fragment",
                  "fragmentName":
"STTA_MP.ext.fragments.ProductTextResponsiveTableCells",
                  "type": "XML"
            }
      },
```

Example: Adding Columns to a Grid Table in the List Report

To add custom columns to a grid table in the list report, follow the steps described below.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

The table containing additional columns can look like this:

Figure 334: Custom columns in a grid table in the list report

Procedure

1. Define a fragment for the view extension

   In the sample project, this is webapp/ext/fragments/ListReportGridTableColumns.fragment.xml.

   You add the extension column to the first position of the grid table. You can change this sequence via the `customData` property `columnIndex`, as shown in the sample code below.

   > ℹ Note
   >
   > If the content of your custom column refers to a property (such as `{Price}`), you need to include a corresponding `"leadingProperty"` entry in the `CustomData` of the column definition.

   > ⁖ Sample Code

   ```
   <core:FragmentDefinition xmlns:core="sap.ui.core"
         xmlns:table="sap.ui.table" xmlns="sap.m">
         <table:Column>
   ```

```
                <Label text="Rating" />
                <table:customData>
                       <core:CustomData key="p13nData"
                              value='\{"columnKey": "Rating",
"leadingProperty":"Price", "columnIndex" : "100"}' />
                </table:customData>
                <table:template>
                       <RatingIndicator value="{= ${Price} > 500 ? 1:5}"></
RatingIndicator>
                </table:template>
         </table:Column>
         <table:Column>
                <Label text="Test" />
                <table:customData>
                       <core:CustomData key="p13nData" value='\{"columnKey":
"Test", "columnIndex" : "101"}' />
                </table:customData>
                <table:template>
                       <Text text="{i18n|
sap.suite.ui.generic.template.ListReport|
STTA_C_MP_Product>xfld.BreakoutColumnContent}"></Text>
                </table:template>
         </table:Column>
</core:FragmentDefinition>
```

2. Register your view extensions in the manifest.json file of your application as follows:

Sample Code

```
...
 "extends": {
      "component": "sap.suite.ui.generic.template.ListReport",
      "minVersion": "1.1.0",
      "extensions": {
        "sap.ui.viewExtensions": {
          "sap.suite.ui.generic.template.ListReport.view.ListReport": {
            "GridTableColumnsExtension|STTA_C_MP_Product": {
                  "className": "sap.ui.core.Fragment",
                  "fragmentName":
"STTA_MP.ext.fragments.ListReportGridTableColumns",
                  "type": "XML"
            },…
```

If you use `QuickVariantSelectionX`, you need to define the extensions per tab. In this case, the name of the extension point is `GridTableColumnExtension|<EntitySet>|<tabKey>`. `<tabKey>` is the key provided when defining the `QuickVariantSelectionX`. See also Defining Multiple Views on a List Report Table - Multiple Table Mode [page 1651].

Note

If you do not use |<tab key> as part of the extension point name, for compatibility reasons, the feature will also work. However, you cannot provide stable IDs for the columns and cells.

Sample Code

```
...
"extends": {
  "extensions": {
    "sap.ui.viewExtensions": {
      "sap.suite.ui.generic.template.ListReport.view.ListReport": {
        "GridTableColumnsExtension|STTA_C_MP_Product|Expensive": {
```

```
            "className": "sap.ui.core.Fragment",
            "fragmentName":
"STTA_MP.ext.fragments.ListReportGridTableColumnsExpensive",
            "type": "XML"
          },
          "GridTableColumnsExtension|STTA_C_MP_Product|Cheap": {
            "className": "sap.ui.core.Fragment",
            "fragmentName":
"STTA_MP.ext.fragments.ListReportGridTableColumnsCheap",
            "type": "XML"
          },...
```

Example: Adding Columns to a Grid Table in the Object Page

To add custom columns to a grid table in the list report, follow the steps described below.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

Procedure

1. Define a fragment for the view extension.

   For a custom column in a responsive table, you have to implement two extensions. First, implement the definition of the custom columns and then implement the content of the custom columns.

   In the example project webapp/ext/fragments/ProductTextGridTableColumns.fragment.xml, two columns are added, one containing text, the other containing an action button:

   > ‹≡› Sample Code

   ```
   <core:FragmentDefinition xmlns:core="sap.ui.core"
   xmlns:table="sap.ui.table" xmlns="sap.m"
   xmlns:sfi="sap.ui.comp.smartfield">
      <table:Column width="150px" >
         <Label text="{i18n|sap.suite.ui.generic.template.ObjectPage|
   STTA_C_MP_Product>xfld.BreakoutColumn}" />
   ```

```
      <table:template>
         <Text text="{i18n|sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumnContent}"></Text>
      </table:template>
   </table:Column>
   <table:Column width="160px">
      <Label text="{i18n|sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumn}" />
      <table:template>
         <Button text="GridTab.Button"
                 press = "onGridTableBreakoutButtonPress"></Button>
      </table:template>
   </table:Column>
</core:FragmentDefinition>
```

2. Register your view extensions in your application's manifest.json file as follows:

Sample Code

```
"extends": {
     "extensions": {
         "sap.ui.controllerExtensions": {
             ....
         }
        "sap.ui.viewExtensions": {
              ....
              "sap.suite.ui.generic.template.ObjectPage.view.Details": {
                     ....
                     "GridTableColumnsExtension|STTA_C_MP_ProductText|
to_ProductTextNavigation::com.sap.vocabularies.UI.v1.LineItem": {
                                "className": "sap.ui.core.Fragment",
                                "fragmentName":
"STTA_MP.ext.fragments.ProductTextGridTableColumns",
                                "type": "XML"
                      },
            ...
```

Example: Adding Columns to an Analytical Table on the Object Page

To add custom columns to an analytical table on the object page, follow the steps described below.

Context

Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's

responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

The table containing additional columns can look like this:

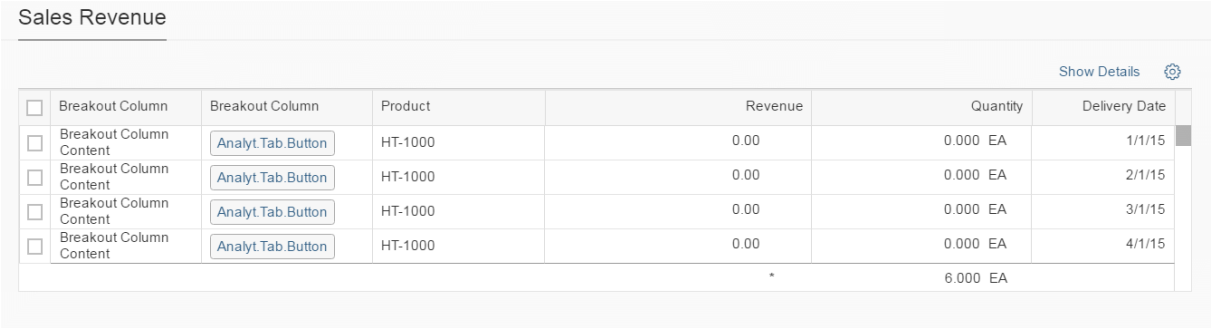


Figure 335: Custom columns in an analytical table on the object page

Procedure

1. Define a fragement for the view extension

   In the example project webapp/ext/fragments/ProductSalesDataAnalyticalTableColumns.fragment.xml, two columns are added: One containing text, the other one containing an action button.

   ### Sample Code

```
<core:FragmentDefinition
                xmlns:core="sap.ui.core"
                xmlns:table="sap.ui.table"
                xmlns="sap.m"
                xmlns:sfi="sap.ui.comp.smartfield">
                <table:AnalyticalColumn width="150px">
      <Label text="{i18n|sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumn}" />
      <table:template>
         <Text text="{i18n|sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumnContent}"></Text>
      </table:template>
      <table:customData>
                              <core:CustomData key="p13nData" value='\
{"columnKey": "some unique key" \}' />
                   </table:customData>
                </table:AnalyticalColumn>

                <table:AnalyticalColumn width="170px">
                              <Label text="{i18n|
sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.BreakoutColumn}" />
                              <table:template>
                                          <Button text="{i18n|
sap.suite.ui.generic.template.ObjectPage|
STTA_C_MP_Product>xfld.AnalyticalTableButton}"

                              press =
"onAnalyticalTableBreakoutButtonPress">
                                          </Button>
                             </table:template>

                              <table:customData>
                                          <core:CustomData
key="p13nData" value='\{"columnKey": some unique key"  \}' />
```

```
                                </table:customData>

                </table:AnalyticalColumn>

</core:FragmentDefinition>
```

2. Register your view extensions in the manifest.json file of your application, as follows:

Sample Code

```
"extends": {
     "extensions": {
         "sap.ui.controllerExtensions": {
             ....
         }
         "sap.ui.viewExtensions": {
              ....
              "sap.suite.ui.generic.template.ObjectPage.view.Details": {
                     ....
                     "AnalyticalTableColumnsExtension|
STTA_C_MP_ProductSalesData|
to_ProductSalesData::com.sap.vocabularies.UI.v1.LineItem": {
                        "className": "sap.ui.core.Fragment",
                        "fragmentName":
"STTA_MP.ext.fragments.ProductSalesDataAnalyticalTableColumns",
                        "type": "XML"
                     },
                     ....
```

Example: Adding Columns to a Tree Table in the List Report

To add custom columns to a tree table in the list report, follow the steps described below.

Context

⚠ Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

The table containing additional columns can look like this:

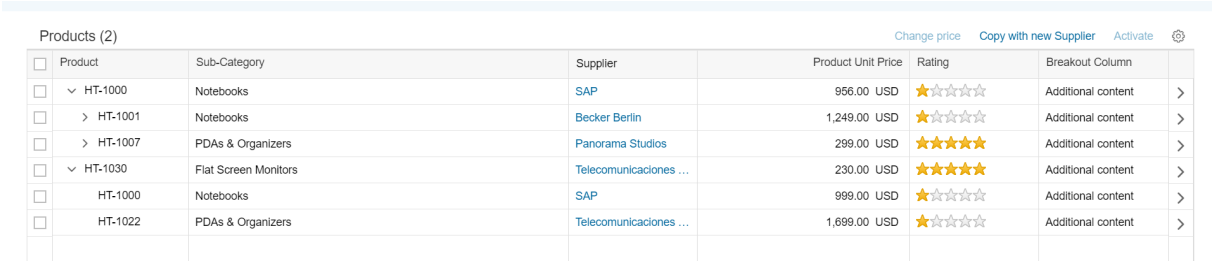


Figure 336: Custom columns in tree table in the list report

Procedure

1. Define a fragment for the view extension

   In the example project webapp/ext/fragments/ListReportTreeTableColumns.fragment.xml, the custom column is added to the first position of the tree table. You can change the sequence via the `customData` property `columnIndex` as shown below.

   > **i Note**
   >
   > If the content of your custom column refers to a property (such as `{Price}`), you need to include a corresponding `"leadingProperty"` entry in the `CustomData` of the column definition.

   Sample Code

   ```
   <core:FragmentDefinition xmlns:core="sap.ui.core"
          xmlns:table="sap.ui.table" xmlns="sap.m">
          <table:Column>
                 <Label text="Rating" />
                 <table:customData>
                        <core:CustomData key="p13nData"
                                    value='\{"columnKey": "Rating",
   "leadingProperty":"Price", "columnIndex" : "100"}' />
                 </table:customData
                 <table:template>
                        <RatingIndicator value="{= ${Price} > 500 ? 1:5}"></
   RatingIndicator>
                 </table:template>
          </table:Column>
          <table:Column>
                 <Label text="Test" />
                 <table:customData>
                        <core:CustomData key="p13nData" value='\{"columnKey":
   "Test", "columnIndex" : "101"}' />
                 </table:customData>
                 <table:template>
                        <Text text="{i18n|
   sap.suite.ui.generic.template.ListReport|
   STTA_C_MP_Product>xfld.BreakoutColumnContent}"></Text>
                 </table:template>
          </table:Column>
   </core:FragmentDefinition>
   ```

2. Register your view extension in the manifest.json file of your application.

   Sample Code

   ```
   ...
   ```

```
 "extends": {
      "component": "sap.suite.ui.generic.template.ListReport",
      "minVersion": "1.1.0",
      "extensions": {
        "sap.ui.viewExtensions": {
          "sap.suite.ui.generic.template.ListReport.view.ListReport": {
            "TreeTableColumnsExtension|STTA_C_MP_Product": {
                  "className": "sap.ui.core.Fragment",
                  "fragmentName":
"STTA_MP.ext.fragments.ListReportTreeTableColumns",
                  "type": "XML"
            },…
```

If you use `QuickVariantSelectionX`, you need to define the extensions per tab. In this case, the name of the extension point is `TreeTableColumnExtension|<EntitySet>|<tabKey>`. `<tabKey>` is the key provided when defining the `QuickVariantSelectionX`. See also Defining Multiple Views on a List Report Table - Multiple Table Mode [page 1651].

> **i Note**
>
> If you do not use |<tab key> as part of the extension point name, for compatibility reasons, the feature will also work. However, you cannot provide stable IDs for the columns and cells.

Sample Code

```
...
"extends": {
  "extensions": {
    "sap.ui.viewExtensions": {
      "sap.suite.ui.generic.template.ListReport.view.ListReport": {
        "TreeTableColumnsExtension|STTA_C_MP_Product|Expensive": {
          "className": "sap.ui.core.Fragment",
          "fragmentName":
"STTA_MP.ext.fragments.ListReportTreeTableColumnsExpensive",
          "type": "XML"
        },
        "TreeTableColumnsExtension|STTA_C_MP_Product|Cheap": {
          "className": "sap.ui.core.Fragment",
          "fragmentName":
"STTA_MP.ext.fragments.ListReportTreeTableColumnsCheap",
          "type": "XML"
        },...
```

Adding Custom Actions Using Extension Points

You can use extension points to add custom actions to the list report and the object page.

Context

> **⚠ Caution**
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori

elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

You can define custom actions for:

- List reports (global action)
  For global actions, you do not have to select a line in the list report table. This type of action refers to the whole list report, for example, *Display Log*. Global actions are placed in the list report filter bar next to the *Share* button.
- Table toolbar of the list report
- Header of the object page
- Table toolbar for a specific table on the object page
- Form in a section on the object page
- Footer bar

These custom actions are displayed as buttons on the UI. When the user selects the action, the system calls a handler function that can be implemented within a controller extension.

Procedure

1. Implement controller extension
   a. In your app, create a `.controller.js` file for your extension.

      In the code sample below, we assume the following:
      - App name: `my_app`
      - File names: `MyListReportExt.controller.js` (extending the `ListReport` controller), `MyObjectPageExt.controller.js` (extending the `ObjectPage` controller)
      - Location of controller files: `my_app/webapp/ext/controller`

   b. In your controller extension, implement the event handler functions to be executed when the user selects the action. For example, if you want to extend the `ListReport` controller, your controller extension should look like this:

```
sap.ui.controller("my_app.ext.controller.MyListReportExt", {
  onCustomAction1 : function(oEvent) { … },
  onCustomAction2 : function(oEvent) { … },
  …
}
```

   When implementing the handler functions for your custom actions, you must use the Using the ExtensionAPI [page 1588].
2. Extend the manifest.json file

   In your app's `manifest.json` file, under `sap.ui5 → extends → extensions`, you can specify extensions for the `ListReport` and the `ObjectPage` controllers.

Specify the following information and extend the manifest files as described below:

`<entity set>`	Entity set that is displayed on the list report or on the object page (for example, `SMART_C_Product`) **i Note** If you use multiple views with different `entity sets` on the list report page, `Actions` need to be defined only for main entity set. It is not possible to execute `Actions` defined for other `entity sets`.
`<Action 1>, <Action 2>, ...`	Action names
`<id>`	ID to be used for the action button **i Note** The values of the action name and the ID should be identical.
`<button text>`	nullText to be displayed on the button (typically a binding to an i18n entry, for example, null<button text>null-null`{i18n>MY_BUTTON_TEXT}`)
`<handler function>`	Handler function that is called when the user selects the action button
`<global>` (required)	Indicates whether this is a global action. The default value is `false`. **i Note** If a determining property is set along with the global property, the action is rendered as a global action since this takes precedence.
Relevant only for table toolbar actions in the list report and object page: `<requiresSelection>` (optional)	Property that indicates whether the action requires a selection of items. The default value is `true`.
Relevant only for list report actions and object page header actions: `<determining>` (optional)	Property that indicates whether the action should be displayed in the footer of the page. The default value is `false`.
Relevant only for object page actions: `<SmartTable Facet ID>`	ID that either comes from the annotation in which you have provided an ID for the facet or that's made up of the annotation term plus the navigation property. For example: <entity type association>::com.sap.vocabularies.UI.v1.LineItem

- Table toolbar action for the list report

```
...
"extends": {
  "extensions": {
    "sap.ui.controllerExtensions": {
      "sap.suite.ui.generic.template.ListReport.view.ListReport": {
        "controllerName": "my_app.ext.controller.MyController",
          "sap.ui.generic.app": {
            "<entity set>": {
              "EntitySet": "<entity set>",
              "Actions": {
                "<Action 1>": {
                  "id" : "<id>",
                  "text" : "<button text>",
                  "press" : "<handler function>",
                  "requiresSelection": <true|false>
                },
                "<Action 2>": {
                  ...
                },
                ...
              }
              ...
```

- Action for the object page header

```
...
"extends": {
  "extensions": {
    "sap.ui.controllerExtensions": {
      ...
      "sap.suite.ui.generic.template.ObjectPage.view.Details": {
        "controllerName": "my_app.ext.controller.DetailsExtension",
          "sap.ui.generic.app": {
            "<entity set>": {
              "EntitySet": "<entity set>",
              "Header" : {
                "Actions": {
                  "<Action 1>": {
                    "id" : "<id>",
                    "text" : "<button text>",
                    "press" : "<handler function>"
                  },
                  "<Action 2>": { ... }
                }
              },
...
```

- Table toolbar action for the object page

```
...
"extends": {
  "extensions": {
    "sap.ui.controllerExtensions": {
      ...
      "sap.suite.ui.generic.template.ObjectPage.view.Details": {
        "controllerName": "my_app.ext.controller.DetailsExtension",
          "sap.ui.generic.app": {
            "<entity set>": {
              "EntitySet": "<entity set>",
              "Sections": {
                "<SmartTable Facet ID>": {
                  "id" : "<SmartTable Facet ID>",
                  "Actions": {
```

```
                    "<SmartTable Action 1>": {
                      "id" : "<id>",
                      "text" : "<button text>",
                      "press" : "<handler function>",
                      "requiresSelection": <true|false>
                    },
                    "<SmartTable Action 2>": { ... }
                  }
...
```

- Form action for the object page

```
...
"extends": {
  "extensions": {
    "sap.ui.controllerExtensions": {
      ...
      "sap.suite.ui.generic.template.ObjectPage.view.Details": {
        "controllerName": "my_app.ext.controller.DetailsExtension",
          "sap.ui.generic.app": {
            "<entity set>": {
              "EntitySet": "<entity set>",
              "Sections": {
                "<Form Facet ID>": {
                  "id": "<Form Facet ID>",
                  "Actions": {
                    "<Action 1>": {
                      "id" : "<id>",
                      "text" : "<button text>",
                      "press" : "<handler function>"
                    },
                    "<Action 2>": { ... }
                  }
...
```

- Footer bar action in the list report:

```
"sap.ui5": {
   "extends": {
      "extensions": {
         "sap.ui.controllerExtensions": {
            "sap.suite.ui.generic.template.ListReport.view.ListReport": {
               "controllerName": "my_app.ext.controller.MyController",
               "sap.ui.generic.app": {
                  "<entity set>": {
                     "EntitySet": "<entity set>",
                     "Actions": {
                        "<Action 1>": {
                           "id": "<id>",
                           "text": "<button text>",
                           "press": "<handler function>",
                           "determining": true
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
```

- Footer bar action in the object page:

```
"sap.ui5": {
```

```
   "extends": {
      "extensions": {
         "sap.ui.controllerExtensions": {
            "sap.suite.ui.generic.template.ObjectPage.view.Detail": {
               "controllerName": "my_app.ext.controller.MyController",
               "sap.ui.generic.app": {
                  "<entity set>": {
                     "EntitySet": "<entity set>",
                     "Header": {
                        "Actions": {
                           "<Action 1>": {
                              "id": "<id>",
                              "text": "<button text>",
                              "press": "<handler function>",
                              "determining": true
                           }
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
```

- List report (global action)

```
...
"extends": {
  "extensions": {
    "sap.ui.controllerExtensions": {
      "sap.suite.ui.generic.template.ListReport.view.ListReport": {
        "controllerName": "my_app.ext.controller.MyController",
          "sap.ui.generic.app": {
            "<entity set>": {
              "EntitySet": "<entity set>",
              "Actions": {
                "<Action 1>": {
                  "id" : "<id>",
                  "text" : "<button text>",
                  "press" : "<handler function>",
                  "global": <true|false>
                },
                "<Action 2>": {
                  ...
                },
                ...
              }
              ...
```

Display of Actions Added Using Extension Points

You can control the display of actions added using extension points in the list report and object page through certain settings in the `manifest.json` file.

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori

elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

In the list report and in tables on the object page, you can specify that a user must make a selection before an action button is enabled. In addition, you can use the `applicablePath` setting for the action, which will then use data from the back-end system to determine whether the action is valid for the selection, and thus whether the button should be enabled or disabled.

If more than one row is selected in the list report or table, or if one `applicablePath` setting from a selection evaluates to `true`, the action button will be enabled. The back-end system must then return the appropriate message for cases in which the action cannot be performed.

When an action is placed in the header or footer of the object page, it is enabled by default. In this case, the actions consider only the `applicablePath` setting for the product represented by the object page, as in this instance the object page itself is considered the selection. Therefore, when the `applicablePath` for the product evaluates to `false`, the action button is hidden (not visible) on the object page.

i Note

Using the `applicablePath` setting is optional. If you do not use it, the action is always enabled.

Code Samples

To set up and control the display of your actions, in the `manifest.json` file, use the properties `requiresSelection` and `applicablePath`.

List Report (Action in Table Header)

The following code sample shows an example of how to set up your `manifest.json` file to determine whether to enable or disable an action in the list report:

```
"sap.ui5": {
   "extends": {
      "extensions": {
         "sap.ui.controllerExtensions": {
            "sap.suite.ui.generic.template.ListReport.view.ListReport": {
               "controllerName": "my_app.ext.controller.ListReportExtension",
               "sap.ui.generic.app": {
                  "<entity set>": {
                     "EntitySet": "<entity set>",
                     "Actions": {
                        "<Action 1>": {
                           "id": "<id>",
                           "text": "<button text>",
                           "press": "<handler function>",
                           "requiresSelection": true,
                           "applicablePath": "<entity type property>"
                        },
                        "<Action 2>": {
```

```
                           "id": "<id>",
                           "text": "<button text>",
                           "press": "<handler function>",
                           "requiresSelection": true
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
```

Object Page (Action in Header or Footer)

The following code sample shows an example of how to set up your `manifest.json` file to determine whether an action is visible in the header or footer of the object page:

```
"sap.ui5": {
   "extends": {
      "extensions": {
         "sap.ui.controllerExtensions": {
            "sap.suite.ui.generic.template.ObjectPage.view.Details": {
               "controllerName": "my_app.ext.controller.DetailsExtension",
               "sap.ui.generic.app": {
                  "<entity set>": {
                     "EntitySet": "<entity set>",
                     "Header": {
                        "Actions": {
                           "<Action 1>": {
                              "id": "<id>",
                              "text": "<button text>",
                              "press": "<handler function>",
                              "applicablePath": "<entity type property>"
                           }
                        }
                     },
                     "Sections": {
                        "<entity type
association>::com.sap.vocabularies.UI.v1.LineItem": {
                        "id": "<SmartTable Facet ID>",
                           "Actions": {
                              "<Action 2>": {
                                 "id": "<id>",
                                 "text": "<button text>",
                                 "press": "<handler function>",
                                 "requiresSelection": true,
                                 "applicablePath": "<entity type property>"
                              }
                           }
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
```

Displaying Custom Action Buttons Depending on the Mode

In case of draft-enabled applications, if the draft information of the object page needs to be found out in the controller / view extension (if the object page is an own draft version or an active version with an existing own draft), you can use the `DraftAdministrativeData` navigation property. For example, you might want to show a custom action button on the object page only in edit mode.

Sample Code

```
onAfterRendering: function(oEvent) {
    var oButton =
sap.ui.getCore().byId("STTA_MP::sap.suite.ui.generic.template.ObjectPage.view.
Details::STTA_C_MP_Product--action::ObjectPageCustomAction");
    oButton.bindElement("DraftAdministrativeData");
    oButton.bindProperty("visible", {
        path: "DraftIsCreatedByMe"
    });
},
```

Adding Custom Fields to the Smart Filter Bar

You can extend the filter bar by using a custom filter field.

Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

To enable this, you need to add a view extension and a corresponding controller extension, as in the following example:

Sample Code

```
"extends": {
            "extensions": {
                "sap.ui.controllerExtensions": {
                  "sap.suite.ui.generic.template.ListReport.view.ListReport":
{
                    "controllerName":
"<myNamespace>.ext.controller.CustomFilter"
                  }
                },
                "sap.ui.viewExtensions": {

"sap.suite.ui.generic.template.ListReport.view.ListReport": {
                        "SmartFilterBarControlConfigurationExtension|
<myEntityset>": {
```

```
                                "className": "sap.ui.core.Fragment",
                                "fragmentName":
"<myNamespace>.ext.fragment.CustomFilter",
                                "type": "XML"
                            },
```

You can add additional controls to the smart filter bar. The following methods are mandatory:

- Using `onBeforeRebindTable`, you evaluate the settings in the custom fields and add the corresponding filters to the `bindingParameters` of the table.
- Using `getCustomAppStateData`, you read the state of all custom fields and store that state in the object provided to enable the templates to use it for navigation.
- Using `restoreCustomAppStateData`, you get the custom app state object you provided in `getCustomAppStateData` and set the corresponding values to your custom controls. For example, you call this method after returning from a navigation.

The `onInitSmartFilterBar` method is optional. You use it if you need to bind a custom control to its own model or if you want value changes to trigger an action.

The enhanced controller methods each call a corresponding extension method:

- `onBeforeRebindTableExtension`
- `getCustomAppStateDataExtension`
- `restoreCustomAppStateDataExtension`
- `onInitSmartFilterBarExtension`

> **i** Note
>
> The filterable fields are usually defined by metadata annotations. You only have to use the extension option if the filter attribute can only be calculated by the client.

For an example with step-by-step instructions, see Adding Filterable Field to the Smart Filter Bar [page 1840].

Adding Filterable Field to the Smart Filter Bar

The following example shows the development steps for adding an additional filter to the smart filter bar of the list report.

> **⚠** Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

In this example, we assume that you want to add a *Price* field with two filter options to the smart filter bar of the Manage Products app's list report. To do so, you have to complete the following steps:

1. Create a controller for a new facet on the list report
2. Add field name and filter option texts to the `i18n` file
3. Define a view and a controller extension in the `manifest.json` file

> **Note**
>
> You can only add new fields to the smart filter bar, not change existing ones. If you want to change existing fields, you must remove them first completely from the annotations, then add them again as new fields.

Step 1: Create a controller for a new facet in the list report

In the SAP Web IDE, open the folder structure of the Manage Products project and then proceed as follows:

1. In the `webapp` folder, create a new subfolder called `ext`.
2. In the folder `ext`, create a new subfolder called `fragment`.
3. In the `fragment` folder, create file `Custom.Filter.fragment.xml`.
4. In the `controller` folder, create file `Custom.Filter.controller.js`.
5. Define the fragment by adding `ControlConfiguration` to the smart filter bar. You can see the options for the `ComboBox` in the following example:

> **Sample Code**

```
<core:FragmentDefinition
        xmlns="sap.m"
        xmlns:smartfilterbar="sap.ui.comp.smartfilterbar"
        xmlns:core="sap.ui.core">
    <!-- Price Filter-->
    <smartfilterbar:ControlConfiguration key="CustomPriceFilter" index="3"
    label="{i18n|sap.suite.ui.generic.template.ListReport|
SEPMRA_C_PD_Product>xfld.Price}"
                            visibleInAdvancedArea="true" groupId="_BASIC">
        <smartfilterbar:customControl>
            <ComboBox id="CustomPriceFilter-combobox">
                <core:Item id="CustomPriceFilterItem0" key="0"
                        text="{i18n|
sap.suite.ui.generic.template.ListReport|
SEPMRA_C_PD_Product>xtit.Price_0-100}"/>
                <core:Item id="CustomPriceFilterItem3" key="1"
                        text="{i18n|
sap.suite.ui.generic.template.ListReport|
SEPMRA_C_PD_Product>xtit.Price_GE100}"/>
            </ComboBox>
        </smartfilterbar:customControl>
    </smartfilterbar:ControlConfiguration>
</core:FragmentDefinition>
```

> **Note**
>
> You can use the index property to define the position of the filterable field. For more information, see also Smart Filter Bar [page 2413] and https://sapui5.hana.ondemand.com/#docs/api/symbols/sap.ui.comp.smartfilterbar.ControlConfiguration.html.

6. To generate the additional filter logic, implement the logic in the controller as shown in the example below.

Note that if a user changes a filter field in the filter bar, the table shows an overlay to indicate that the state of the filter bar differs from the date currently being displayed. If the control used in your app does not trigger a change event, you have to set this up using `.fireChange()` so that the filter bar will recognize the change and display the overlay.

Sample Code

```
sap.ui.controller("ManageProducts.ext.controller.CustomFilter", {
        onBeforeRebindTableExtension: function(oEvent) {
        var oBindingParams = oEvent.getParameter("bindingParams");
        oBindingParams.parameters = oBindingParams.parameters || {};

        var oSmartTable = oEvent.getSource();
        var oSmartFilterBar = this.byId(oSmartTable.getSmartFilterId());
        var vCategory;
        if (oSmartFilterBar instanceof
sap.ui.comp.smartfilterbar.SmartFilterBar) {
            //Custom price filter
            var oCustomControl =
oSmartFilterBar.getControlByKey("CustomPriceFilter");
            if (oCustomControl instanceof sap.m.ComboBox) {
                vCategory = oCustomControl.getSelectedKey();
                switch (vCategory) {
                    case "0":
                        oBindingParams.filters.push(new
sap.ui.model.Filter("Price", "LE", "100"));
                        break;
                    case "1":
                        oBindingParams.filters.push(new
sap.ui.model.Filter("Price", "GT", "100"));
                        break;
                    default:
                        break;
                }
            }
        }
    }
});
```

Step 2: Add field name and filter option texts to the i18n file

To make the field name and the filter options translatable, add the texts to the `i18n` file as follows:

Sample Code

```
#XFLD: Custom filter breakout label
xfld.Price=Price
#XTIT: Price range 0-100
xtit.Price_0-100=Price between 0-100
#XTIT: Price range Over 100
xtit.Price_GE100=Price: Over 100
```

Step 3: Define a view and a controller extension in the manifest.json file

To integrate the logic as an extension, define a view and controller extension to load the files you created in Step 1 (`Custom.Filter.fragment.xml` and `Custom.Filter.controller.js`).

The logic is added to the `ListReport` section of the Manage Products app.

Sample Code

```
"extends": {
    "extensions": {
        "sap.ui.controllerExtensions": {
            "sap.suite.ui.generic.template.ListReport.view.ListReport": {
                "controllerName": "ManageProducts.ext.controller.CustomFilter"
            }
        },
        "sap.ui.viewExtensions": {
            "sap.suite.ui.generic.template.ListReport.view.ListReport": {
                "SmartFilterBarControlConfigurationExtension|
SEPMRA_C_PD_Product": {
                    "className": "sap.ui.core.Fragment",
                    "fragmentName":
"ManageProducts.ext.fragment.CustomFilter",
                    "type": "XML"
                }
            }
        }
    }
},
```

Results

The list report of the Manage Products app displays the new *Price* field with filter options.

Adapting Texts in the Delete Dialog Box (List Report)

You can adapt the text of the *Delete* dialog box that is displayed when you delete list report items.

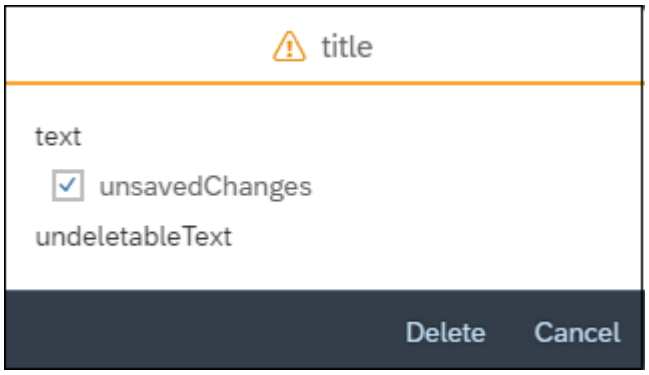
Context

⚠ Caution

Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

The *Delete* dialog looks as follows:



It contains the following text:

- *title*: always appears
- *text*: always appears
- *unsavedChanges*: appears in the following cases:
  - If not, only list report items with the editing status `unsavedChanges` are selected
  - If not, only list report items with the editing status `unsavedChanges` and `locked` are selected
- *undeletableText*: appears only if one or more selected items cannot be deleted using the `deletablePath` feature.

> **i Note**
>
> For information about how to adapt texts on the object page, see Adapting Texts in the Delete Dialog Box (Object Page Header) [page 1846] and Adapting Texts in the Delete Dialog Box (Object Page with Nested Smart Table) [page 1848].

Procedure

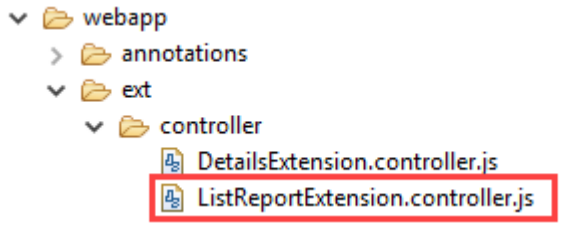
1. Register your extension at the app descriptor (manifest.json) for the list report.

   Sample Code

```
"extends": {
    "extensions": {
        ...
         "sap.ui.controllerExtensions": {
            ...
            "sap.suite.ui.generic.template.ListReport.view.ListReport": {
                ...
                "controllerName":
"MY_APP.ext.controller.ListReportExtension",
                ...
            }
        }
```

```
...
```

2. Maintain the controller extension files in your app.



3. Implement the function `'beforeDeleteExtension'` in the controller extension file of the list report. You have several options to determine the delete dialog box:
   - The easy way

     Sample Code

     ```
     beforeDeleteExtension: function(oBeforeDeleteProperties) {
         var oMessageText = {
             title: "My title";
             text: "My text",
             unsavedChanges: "My unsaved changes",
             undeletableText: "My undeletable text"
         };
         return oMessageText;
     }
     ```

   - Using promises

     Sample Code

     ```
     beforeDeleteExtension: function(oBeforeDeleteProperties) {
         var oMessageText = {
             title: "My title";
             text: "My text",
             unsavedChanges: "My unsaved changes",
             undeletableText: "My undeletable text"
         };
         return Promise.resolve(oMessageText);
     }
     ```

   - Using the `extensionAPI.SecuredExecution` (see also Using the SecuredExecution Method [page 1590])

     Sample Code

     ```
     beforeDeleteExtension: function(oBeforeDeleteProperties) {
         var oMessageText = {
             title: "My title";
             text: "My text",
             unsavedChanges: "My unsaved changes",
             undeletableText: "My undeletable text"
         };
         return this.extenionAPI.securedExecution(function() {
             return new Promise(function(resolve) {
                 ...
                 resolve(oMessageText);
                 ...
             });
         });
     ```

```
}
```

> **i Note**
>
> The property `oBeforeDeleteProperties` contains information about the selected items for deletion of the list report.

Adapting Texts in the Delete Dialog Box (Object Page Header)

You can adapt the text of the *Delete* dialog box that is displayed when you delete the entire object page.

Context

> **⚠ Caution**
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

In the object page header, you can display the *Delete* dialog box by choosing the *Delete* button to delete the entire content of the object page.

Perform these steps to be able to use the extension:

Procedure

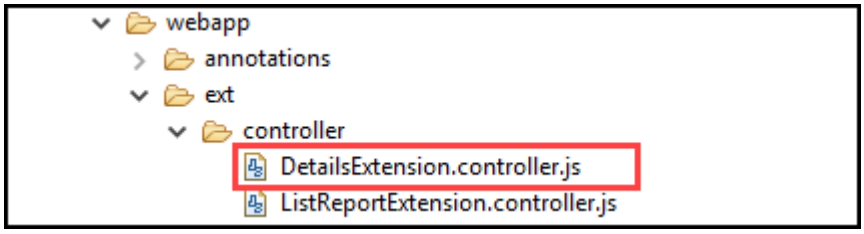
1. Register your extension at the app descriptor (manifest.json):

> **Sample Code**

```
"extends": {
   "extensions": {
      ...
      "sap.ui.controllerExtensions": {
         ...
         "sap.suite.ui.generic.template.ObjectPage.view.Details": {
            ...
            "controllerName": "MY_APP.ext.controller.DetailsExtension",
            ...
         }
```

```
        }
        ...
```

2. Maintain the controller extension files in your app:



3. The *Delete* dialog box contains the following text:
   - title: always appears
   - text: always appears

   Implement the `beforeDeleteExtension` function in the controller extension file of the object page. You have several options to determine the delete dialog:
   - The easy way

     Sample Code

     ```
     beforeDeleteExtension: function() {
         var oMessageText = {
             title: "My title";
             text: "My text"
         };
         return oMessageText;
     }
     ```

   - Using promises

     Sample Code

     ```
     beforeDeleteExtension: function() {
         var oMessageText = {
             title: "My title";
             text: "My text"
         };
         return Promise.resolve(oMessageText);
     }
     ```

   - Using the `extensionAPI.SecuredExecution` (see also Using the SecuredExecution Method [page 1590])

     Sample Code

     ```
     beforeDeleteExtension: function() {
         var oMessageText = {
             title: "My title";
             text: "My text"
         };
         return this.extenionAPI.securedExecution(function() {
             return new Promise(function(resolve) {
                 ...
                 resolve(oMessageText);
                 ...
             });
         });
     ```

```
}
```

Adapting Texts in the Delete Dialog Box (Object Page with Nested Smart Table)

You can adapt the text of the *Delete* dialog box that is displayed when you delete items from nested smart tables on the object page.

Context

> ⚠ Caution
>
> Use app extensions with caution and only if you cannot produce the required behavior by other means, such as manifest settings or annotations. To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].
>
> After you've created an app extension, its display (for example, control placing, CSS) and system behavior (for example, model and binding usage, busy handling) of the app extension lies within the application's responsibility. SAP Fiori elements provides support only for the official extensionAPI functions. Don't access or manipulate SAP Fiori elements' internal coding.

On an object page with a nested smart table, you can choose the *Delete* button in the table toolbar to delete the selected table items.

Perform these steps to be able to use the extension:

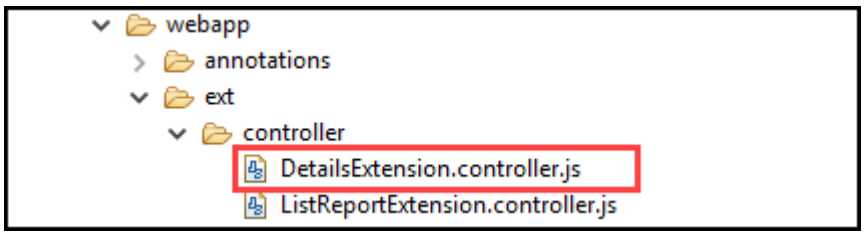
Procedure

1. Register your extension with the app descriptor (manifest.json):

> Sample Code

```
"extends": {
   "extensions": {
      ...
      "sap.ui.controllerExtensions": {
         ...
         "sap.suite.ui.generic.template.ObjectPage.view.Details": {
            ...
            "controllerName": "MY_APP.ext.controller.DetailsExtension",
            ...
         }
      }
      ...
```

2. Maintain the controller extension files in your app:



3. The *Delete* dialog box contains the following text:
   - title: always appears
   - text: always appears
   - undeletableText: appears only if one or more selected table items cannot be deleted using the `deletablePath` feature.

   Implement the `beforeLineItemDeleteExtension` function in the controller extension file of the object page. You have several options to determine the delete dialog:
   - The easy way

     Sample Code

     ```
     beforeLineItemDeleteExtension:
     function(oBeforeLineItemDeleteProperties) {
         if (oBeforeLineItemDeleteProperties.sUiElementId !==
     "My_APP::sap.suite.ui.generic.template.ObjectPage.view.Details::MY_Entit
     ySet--to_NavProperty::com.sap.vocabularies.UI.v1.LineItem::Table") {
             return;
         }
         var oMessageText = {
             title: "My title";
             text: "My text",
             undeletableText: "My undeletable text"
         };
         return oMessageText;
     }
     ```

   - Using promises

     Sample Code

     ```
     beforeLineItemDeleteExtension:
     function(oBeforeLineItemDeleteProperties) {
         if (oBeforeLineItemDeleteProperties.sUiElementId !==
     "My_APP::sap.suite.ui.generic.template.ObjectPage.view.Details::MY_Entit
     ySet--to_NavProperty::com.sap.vocabularies.UI.v1.LineItem::Table") {
             return;
         }
         var oMessageText = {
             title: "My title";
             text: "My text",
             undeletableText: "My undeletable text"
         };
         return Promise.resolve(oMessageText);
     }
     ```

   - Using the `extensionAPI.SecuredExecution` (see also Using the SecuredExecution Method [page 1590])

Sample Code

```
beforeLineItemDeleteExtension:
function(oBeforeLineItemDeleteProperties) {
    if (oBeforeLineItemDeleteProperties.sUiElementId !==
"My_APP::sap.suite.ui.generic.template.ObjectPage.view.Details::MY_Entit
ySet--to_NavProperty::com.sap.vocabularies.UI.v1.LineItem::Table") {
        return;
    }
    var oMessageText = {
        title: "My title";
        text: "My text",
        undeletableText: "My undeletable text"
    };
    return this.extenionAPI.securedExecution(function() {
        return new Promise(function(resolve) {
            ...
            resolve(oMessageText);
            ...
        });
    });
}
```

Note

The property `oBeforeLineItemDeleteProperties` contains information about the selected items that are to be deleted from the nested smart table. For example, `sUiElementId` identifies the UI element (here, the nested smart table) from which something is to be deleted.

Prefilling Fields When Creating a New Entity Using an Extension Point

When a user creates a new entity, it is possible to prefill fields with specific values.

There are two ways you can implement this system behavior:

- Creation using cross-app navigation
  For more information, see Prefilling Fields When Creating a New Entity [page 1783].
- Passing values entered by the user into the filter bar

Passing Values Entered by the User into the Filter Bar

In the list report, if users enter a value into the filter bar, perform a search, and then create a new object, this can mean that they want to create an instance that was not found with the previous search.You can support this scenario by providing the additional option "Create with filters" which passes filter values entered by the user to the newly created instance.

To enable this feature, perform these steps:

1. Incorporate the following snippet into the manifest.json:

```
...
"sap.ui.generic.app": {
    "_version": "???",
    "settings": {   ...   },
    "pages": {
        "ListReport|myEntitySet": {
            "entitySet": "myEntitySet",
            "component": {
                "name": "sap.suite.ui.generic.template.ListReport",
                "list": true,
                "settings": {
                    "createWithFilters": {
                        "strategy": "extension"
                    },
            ...
```

Result: The standard *Create* button in the list report is replaced by a menu button that gives you two options:

- The text for the first option is given by the i18n key `CREATE_NEW_OBJECT`. The default text is *Create Object*. We recommend overwriting this text with a more specific text, for example, *Create New Product*.
- The text for the second option is *Create with Filters*. We do not recommend overwriting this text. Note that this option is enabled only if the data displayed in the table corresponds to the displayed filter values. This means the user might have to choose *Go* to enable the option if the selection is not triggered automatically.

2. Implement the logic that transfers information from the filter bar to the creation process. To do so, you have to override the extension function `getPredefinedValuesForCreateExtension(oSmartFilterBar)`. This function receives an instance of `sap.ui.comp.smartfilterbar.SmartFilterBar` and must return an object that represents the name/value pairs that should be used in the creation process.
This is shown in the following code sample which extracts the filter values for `ProductCategory` and `Supplier` (if one exists and is unique):

Sample Code

```
getPredefinedValuesForCreateExtension: function(oSmartFilterBar){
    var oRet = {};
    var oSelectionVariant =
oSmartFilterBar.getUiState().getSelectionVariant();
    var aSelectOptions = oSelectionVariant.SelectOptions;
    var fnTransfer = function(sFieldname){
        for (var i = 0; i < aSelectOptions.length; i++){
            var oSelectOption = aSelectOptions[i];
            if (oSelectOption.PropertyName === sFieldname){
                if (oSelectOption.Ranges.length === 1){
                    var oFilter = oSelectOption.Ranges[0];
                    if (oFilter.Sign === "I" && oFilter.Option === "EQ")
{
                        oRet[sFieldname] = oFilter.Low;
                    }
                }
                break;
            }
        }
    };
    fnTransfer("ProductCategory");
    fnTransfer("Supplier");
    return oRet;
},
```

> **i Note**
>
> We recommend using this option only for fields that are available and editable on the object page, and that will not be changed via UI adaptation. Otherwise, users might potentially save values they have never seen or cannot change.

Custom State Handling for Extended Apps

You can perform inner app state handling for custom UI elements.

To do so, three methods need to be adapted by the relevant implementation:

- **`onCustomStateChange`**
  Method of the extensionAPI of the object page. This method should be called whenever the (persistable) state of the custom UI changes. The method does not have any parameters. For more information, see ExtensionAPI for object page extensions.
- **`provideCustomStateExtension`**
  Method of the object page controller you need to override if you want to handle custom states.
  An empty javascript object `oState` is passed to this method. The method adds any state information to this object.
  You can add properties to the object. Note that the value of the properties needs to have a predefined structure. This means that the corresponding value for each property has to be an object containing two properties, like this:
  - First property: data
    The value of this property can be any javascript object. The only restriction is that method `JSON.stringify` must be applicable to this object and the state of the object must be restorable from the result of this operation.
  - Second property: lifecycle
    The value of this property must be an object specifying the lifecycle of the corresponding state. For more information, see Lifecycle [page 1853].
- **`applyCustomStateExtension(oState, bIsSameAsLast)`**
  Method of the object page controller which must be overridden if you want to perform custom state handling.
  The object `oState` passed to this method contains properties according to the applicable states that have been added to the state object in a suitable `provideCustomStateExtension` call. Note that the value of this property is the value of the corresponding `data` property and the information may have been serialized and deserialized in the meantime.
  The content of the `lifecycle` property is not passed to the `applyCustomStateExtension` method. The lifecycle information determines only which information is passed to the `applyCustomStateExtension` method.
  You have to evaluate the state and apply it to the custom UI elements accordingly.

> **i Note**
>
> If parameter `bIsSameAsLast` is `truthy`, users reach the page for the same instance they visited the last time. In this case, you do not need to adapt the UI state, since the whole page should still be in the same state as when the users left it.

In draft scenarios, the parameter `bIsSameAsLast` is also `truthy` if the instance that is currently displayed and the instance that was visited previously are semantically the same but differ in their draft status (for example, one is the active version, the other one the draft). However, due to technical restrictions, this might fail in some cases (which means that `bIsSameAsLast` would be `faulty`, although the two instances are semantically identical).

Lifecycle

When overriding the `provideCustomStateExtension` method, you need to define the lifecycle attached to the different parts of the state.

The lifecycle object has the following potential properties. Each of them has boolean values, with standard javascript logic for truthy and faulty values.

- `session`: Setting this property to `true` indicates that the lifecycle of this state should correspond to the whole session.
  The exact definition of a session's lifetime depends on the underlying SAP Fiori launchpad (FLP) infrastructure.
  These boundary conditions apply:
  - The session survives any FLP cross-app navigation.
  - The session ends when the user closes the browser.

  In particular, the lifetime is sufficient to ensure that the state can be restored after navigating to another SAP Fiori application (via cross-app navigation) and returning by back navigation.
  Note that a hash of this information is stored in the URL.
- `permanent`: Setting this property to `true` indicates that the lifecycle of this state survives the session.
  A hash of the state is stored in the URL. You can store the mapping information needed to resolve the hash to the real state on the frontend server.
  Note that you need to make specific configuration settings to allow this persistency.
  If persistency is enabled, the state is part of the URL, even if the user bookmarks it or sends it with e-mail. Otherwise, it corresponds to the session.

  > **i Note**
  >
  > Choose this lifecycle only if the corresponding state is also relevant for other user sessions. For example, it is not relevant for a scroll position.
- `page`: Setting this property to `true` indicates that the state should also be applied when the user navigates to the same page for a different instance.
  Handle this property with care since some time might have passed since the user visited this page for another instance. The user might be surprised to find the page for the new instance in the given state.
- `pagination`: This property has the same semantics as the `page` property. However, the state is preserved only when you use pagination to change instances . Pagination means that the content of one instance is immediately replaced by another instance on the UI. Typical examples: Using paginator buttons or browsing through different instances in a flexible column layout scenario.

> **i Note**
>
> The `session` and `permanent` properties can be used to define a lifecycle that extends the lifetime of the current application. In this case, the state is stored as a hash within the URL. This means that the state can

only be recovered if the same URL is called again. This happens if the user navigates to the corresponding page with backward or forward navigation in the history, uses a bookmark, or a URL received by other means, for example, e-mail.

The state information is not available if the user navigates to the same app using normal forward navigation. Example: The user chooses the FLP home button to navigate to the FLP and then selects the tile representing the app a second time.

Reuse Components

Reuse components may also want to keep a specific state. However, they cannot override the controller's extension functions.

For a reuse component to keep a state, you need to implement the methods `stGetCurrentState()`, and `stApplyState(oState, bIsSameAsLast)`.

Note that the signature of `stGetCurrentState()` differs from the signature of the corresponding extension method `provideCustomStateExtension(oState)`. While `stGetCurrentState()` is only responsible for providing a state object with the structure defined above, `provideCustomStateExtension(oState)` enters the state into a given empty state object.

Example: Custom State Handling

This example shows how to implement inner app state handling for custom UI elements.

The custom UI shows a map containing two state information items:

- The `zoomFactor` is a number between 1 and 100. The `zoomFactor` should be passed to other instances on pagination. The `zoomFactor` should also remain valid for the whole session.
- The `selectedCity` is information consisting of a country and a city. The selected state should not be passed to other instances. However, it should be bookmarkable for the current instance, when customers enable storage on the frontend server.

The following sample code shows this:

Sample Code

```
onZoomFactorChange: function(iZoomFactor){
    if (!this.isAdoptingState){ // ignore case where we are just adapting to
the given state
        this.zoomFactor = iZoomFactor;
        // Inform framework that state of custom ui area has changed
        this.extensionAPI.onCustomStateChange();
    }
},

onSelectedCityChange: function(sCountry, sCity){
    if (!this.isAdoptingState){ // ignore case where we are just adapting to
the given state
        this.city = sCity;
        this.country = sCountry;
```

```
            // Inform framework that state of custom ui area has changed
            this.extensionAPI.onCustomAppStateChange();
        }
    },

    provideCustomStateExtension: function(oState){
        oState.zoomFactor = {
            data: this.zoomFactor,
            lifecycle: {
                session: true,
                pagination: true
            }
        };
        oState.selectedCity = {
            data: {
                country: this.country,
                city: this.city
            },
            lifecycle: {
                permanent: true
            }
        };
    },

    applyCustomStateExtension: function(oState, bIsSameAsLast){
        if (bIsSameasLast){
            return; // all controls are still in the correct state
        }
        this.isAdoptingState = true;
        this.setZoomFactor(oState.zoomFactor);
        this.setSelectedCity(oState.selectedCity.country,
    oState.selectedCity.city);
        this.isAdoptingState = false;
    },

    setZoomFactor: function(iZoomFactor){
        this.zoomFactor = iZoomFactor;
        //... custom code which brings the map to the given zoom factor
    },

    setSelectedCity: function(sCountry, sCity){
        this.city = sCity;
        this.country = sCountry;
        //... custom code which selects the specified city
    }
```

Note

We recommend introducing the controller property `isAdoptingState` if `setZoomFactor` and/or `setSelectedCity` trigger the event handlers calls `onZoomFactorChange` or `onSelectedCityChange`, respectively.

Adaptation Extension Example: Adding a Button to the Table Toolbar in the List Report

In this example, you add a button to the table toolbar in the list report and extend it to filter only the records which have a price that is greater than or equal to 1000.

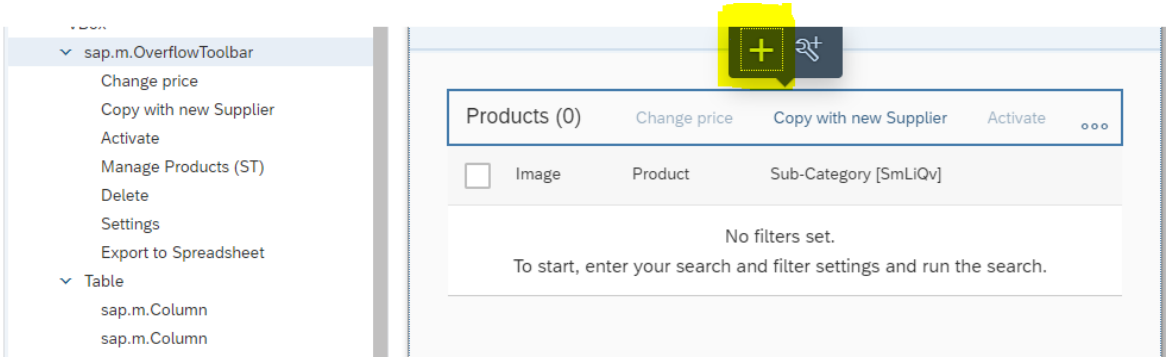
Prerequisites

You have performed the steps described in Extending Delivered Apps Using Adaptation Extensions [page 1596].

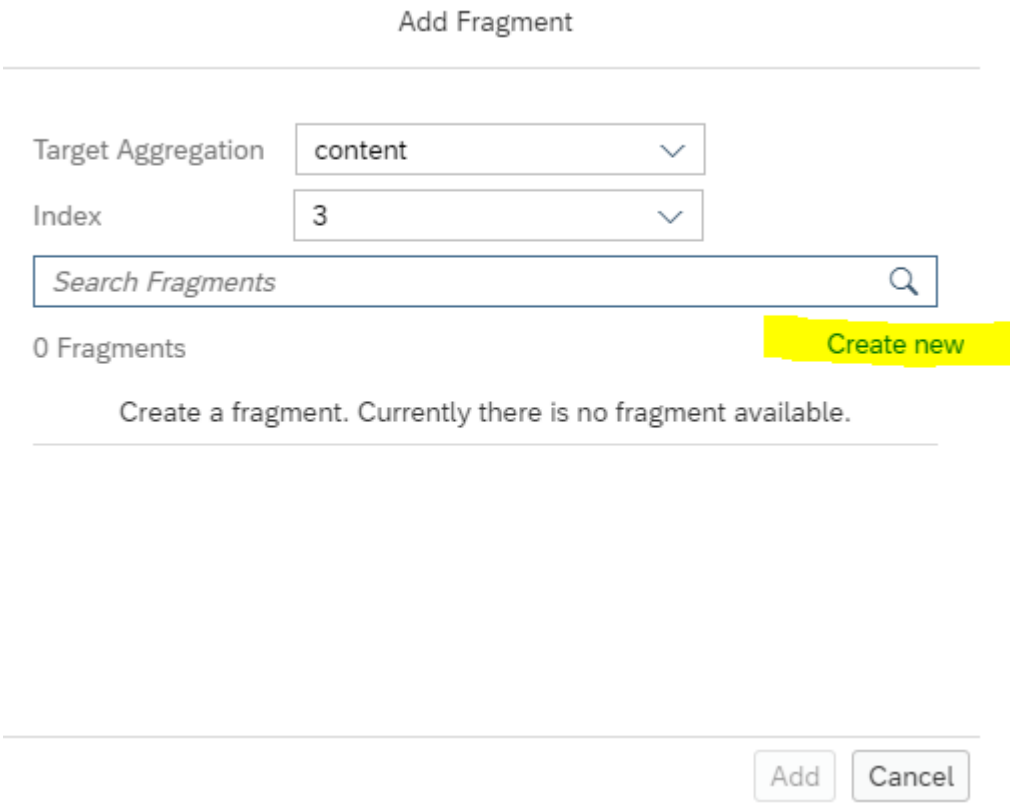
Context

Procedure

1. Select the table tool bar and choose *Add Fragment*.



2. Under *Target Aggregation*, select *content* and choose your preferred index value. Choose *Create New* to create the fragment. If the fragment is already there, you can search for it.

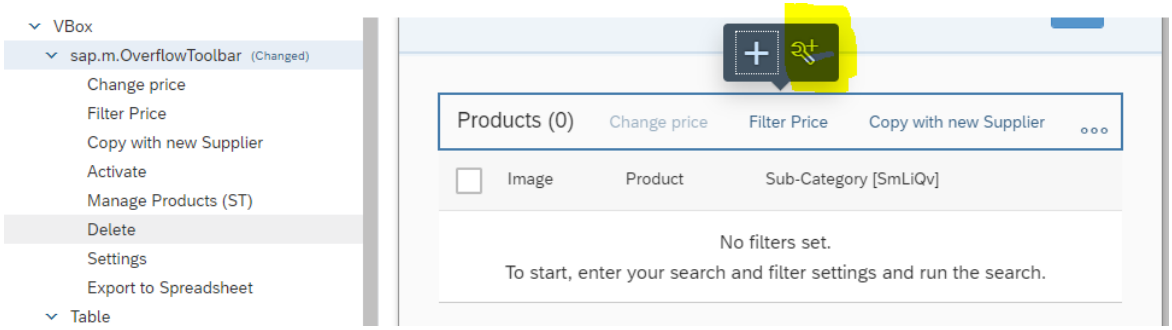


3. Enter a fragment name, for example, `FilterPriceButton`, and choose *Create*.
4. Write the following code in the auto-generated `FilterPriceButton.fragment.xml` file. Note that the bold code needs a supporting function in the extension controller. This is described in the next step.

> ### Sample Code
>
> ```
> <core:FragmentDefinition xmlns:core='sap.ui.core' xmlns='sap.m'>
> <Button id="PriceBtnID" text="Filter Price"
> press=".extension.ProdMan.AdaptProject.ListReportExtension.handleFilterPricePress"></Button>
> </core:FragmentDefinition>
> ```

> ### Note
>
> `".extension.ProdMan.AdaptProject.ListReportExtension.handleFilterPricePress"` consists of the following elements that are connected with a dot (.).
>
> `.extension`: Required according to the UI adaptation tool guidelines
>
> `ProdMan.AdaptProject`: Name of the project
>
> `ListReportExtension`: Controller extension file name
>
> `handleFilterPricePress`: Function in the controller extension file

5. Create a controller extension. Select the table toolbar as described under step 1 and choose *Extend with Controller*.



6. Enter the controller name, for example, `ListReportExtension`, and choose *Extend*.

   The system generates the controller extension file `ListReportExtension.js`.

   This auto-generated file contains predefined life-cycle functions under the *Override* block. All extensibility functions provided by SAP Fiori elements should be consumed inside *Override*. The custom-defined functions should be consumed outside the *Override* block.

   To complete the example, copy and paste the code shown below to `ListReportExtension.js`.

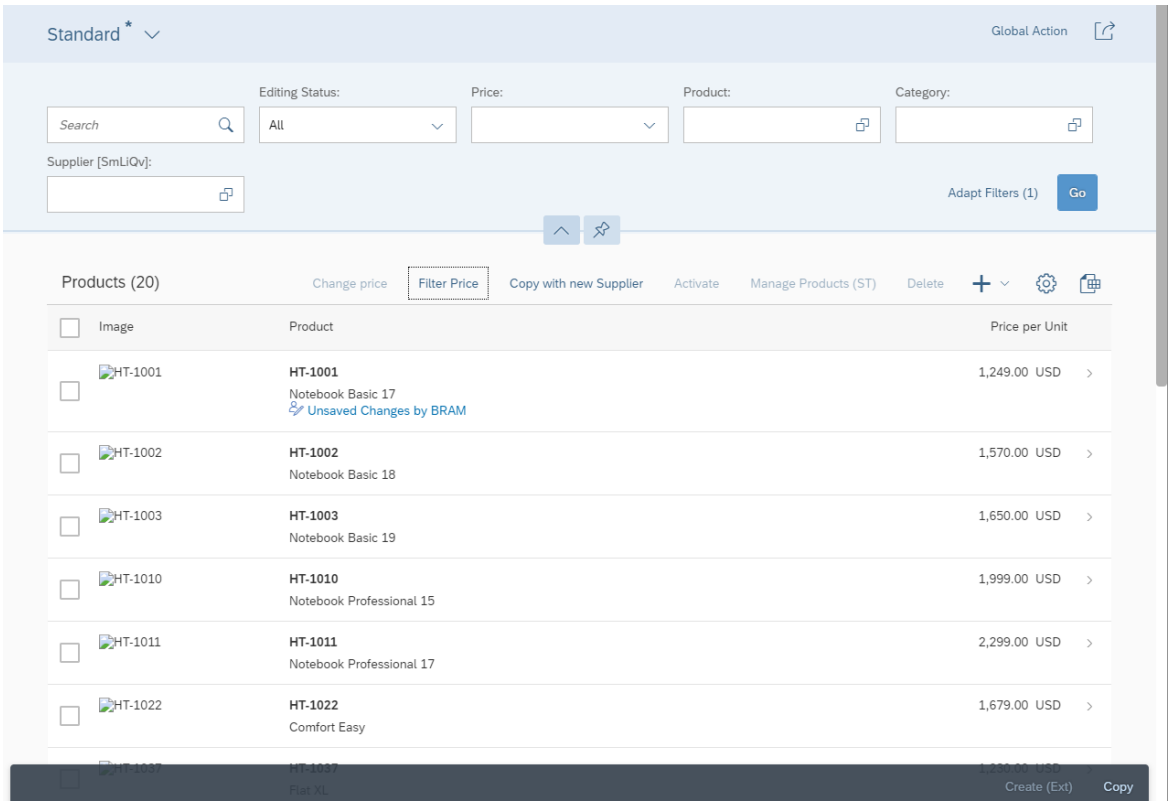
> **i Note**
>
> `handleFilterPricePress` is a custom-defined function and `addFilters` is the extensibility function provided by SAP Fiori elements.

> **Sample Code**
>
> In this example, there is a property named *Price* named property in your list report entity type.

```
/**
*@controller Name:
sap.suite.ui.generic.template.ListReport.view.ListReport,
*@viewId:STTA_MP::sap.suite.ui.generic.template.ListReport.view.ListReport:
:STTA_C_MP_Product
*/
sap.ui.define(["sap/ui/core/mvc/Controller", "sap/ui/core/mvc/
ControllerExtension"], function (Controller, ControllerExtension) {
return
ControllerExtension.extend("ProdMan.AdaptProject.ListReportExtension", {

handleFilterPricePress : function() {
     this.iPriceRestriction = "1000";
     var oExtensionAPI = this.base.templateBaseExtension.getExtensionAPI();
     oExtensionAPI.rebindTable();
},
//override an existing method of the Main.controller
override: {
//adding a life cycle method
/**
* Called when a controller is instantiated and its View controls (if
available) are already created.
* Can be used to modify the View before it is displayed, to bind event
handlers and do other one-time initialization.
* @memberOf src.client.uiadaptation
*/
onInit: function () {
},
/**
* Similar to onAfterRendering, but this hook is invoked before the
controller's View is re-rendered
* (NOT before the first rendering! onInit() is used for that one!).
* @memberOf src.client.uiadaptation
```

```
*/
onBeforeRendering: function () {
},
/**
* Called when the View has been rendered (so its HTML is part of the
document). Post-rendering manipulations of the HTML could be done here.
* This hook is the same one that SAPUI5 controls get after being rendered.
* @memberOf src.client.uiadaptation
*/
onAfterRendering: function () {
},
/**
* Called when the Controller is destroyed. Use this one to free resources
and finalize activities.
* @memberOf src.client.uiadaptation
*/
onExit: function () {
},
"templateBaseExtension": {
addFilters : function(fnAddFilter, sControlId){
if(this.iPriceRestriction){
fnAddFilter(this, new sap.ui.model.Filter("Price", "GE",
this.iPriceRestriction));
}
}
}
}
});
});
```

7. In the adaptation editor, choose the *Preview* tab page and choose *Go*. Check the value in the *Price* field for all records.
8. Choose *Filter Price* and check the value in the *Price* field for the filtered records.



Adapting the UI: List Report and Object Page

You can use the SAPUI5 Visual Editor in the SAP Web IDE to extend and customize specific features in the list report and on the object page.

> **i Note**
>
> Adapt the UI only for the use cases described here. Otherwise, issues regarding consistency, compatibility, or other problems may occur immediately or in future releases.

Adapting the UI: List Report

Feature	Setting
Display the *Export to Excel* button in the table toolbar	For `SmartTable`, set the *Use Export to Excel* property to true.
Combine buttons (actions) in the toolbar	Select the buttons you want to combine by holding the `ctrl` key and left-clicking them in the required order. Then, release the `ctrl` key, right-click one of the selected buttons and choose *Combine* from the context menu. **i Note** If the buttons don't all fit because the preview size in the UI adaptation editor is too small, you can expand the editor tab by double-clicking it and collapsing the outline and property panels.
Change the column width of `sap.ui.table.Table`,`sap.ui.table.AnalyticalTable`, or `sap.m.Table`.	Choose the column header to select the corresponding `sap.ui.table.Column`. Change the *Width* property as needed.
Table for retrieving data without using the *Go* button Collapse the smart filter bar by default when the app is launched	For the `SmartTable` control, set the `enableAutoBinding` property to `True`.
Center-align status columns	For `sap.m.Table`, select a column and set the *H Align* property to `Center`. For `sap.ui.table`, click the column header and set the *H Align* property to `Center`. For vertical alignment of whole responsive table, see Smart Tables [page 1628].

Feature	Setting
Disable sticky column header and sticky table toolbar	By default, the sticky column header and the sticky table toolbar in the list report are enabled. To disable them, change the *Sticky* property value under *Table*.
Hide a toolbar action for a specific table (if you use multiple views on list report tables)	Set the `visible` property to `false`.
Disable standard system behavior for list report tables (analytical, grid, tree tables): Table should not occupy the entire space available in the container.	In the Dynamic Page, set the `Fit Content` property to `false`.
Change the layout of the list report table for better readability in case of a high number of columns.	For `sap.m.Table`, set the `PopinLayout` property to one of the following values: • `Block` • `GridSmall` • `GridLarge` The default layout is `Block`.

Adapting the UI: Object Page

Feature	Setting
Change the image shape from square to circle	Switch to preview mode, navigate to the object page, switch back to *Adapt the UI*. Select the object page header and set the *Object Image Shape* property to `Circle`.
Change the avatar shape in the object page dynamic header from a square to a circle	Switch to preview mode, navigate to the object page, switch back to *Adapt the UI*. Select the avatar in the object page header and set the *displayShape* property to `Circle`.
Hide anchor bar	Switch to preview mode, navigate to the object page, switch back to *Adapt the UI*. Select the object page layout and set the *Show Anchor Bar* property to `false`.
Switch to tabs	Switch to preview mode, navigate to the object page, switch back to *Adapt the UI*. Select the object page layout and set the *Use Icon Tab Bar* property `"useIconTabBar"` to `true`.
Display the *Export to Excel* button in the table toolbar	For `SmartTable`, set the *Use Export to Excel* property to `true`.

Feature	Setting
Show header content in edit mode	By default, there is a binding at the *showHeaderContent* property of `sap.uxap.ObjectPageLayout` that the UI Adaptation editor cannot display. Change this property to `False` to get a change file. Then, change the `newValue` from `false` to `true`. Alternatively, you can make a binding change if you need to change the value according to a property or an expression. For more information, see Creating a Binding Change [page 1864].
Show content parts in the header according to a specific mode	Ensure that the header content is also displayed in edit/create mode, as described above. Then, search for the *sap.m.VBox* under *sap.uxap.ObjectPageLayout* > *headerContent* of the header facet you want to adjust. At the *sap.m.VBox*, set a binding change in the *Visible* property. See also Creating a Binding Change [page 1864].
Hide the *Share* button	Select the *Share* button in the header and set the `Visible` property to `false`.
Set widths of mixed content in sections When placing mixed content, such as forms or tables into one subsection, you may want to adjust the content blocks to display a table next to a form, for example.	Select the `sap.ui.layout.GridData` of the corresponding section and set the spans according to your requirements.
Don't collapse headers when scrolling down	Select the *sap.uxap.ObjectPageLayout* and set the `Always show content header` property to `true`. **i Note** On tablets and mobile phones, the header collapses automatically.
Show more contact information in the header facet	In the outline, choose *sap.m.Page* > *content* > *sap.uxap.ObjectPageLayout* > *headerContent*. For each header facet, an sap.m.VBox is displayed. In the contact header facet, several sap.m.HBoxes are displayed when icons and texts or links are available but invisible. Set the *Visible* properties of the required items to `true`.

Feature	Setting
Show paginator buttons (up and down arrows) on first object page By default, the paginator buttons for navigating to the previous object page or next object page are not displayed on the first object page. They are displayed from the second object page onwards.	From the outline, choose ▶ *sap.m.Page* ▶ *content* ▶ *sap.uxap.ObjectPageLayout* ▶ *headerTitle* ▶ *sap.uxap.ObjectPageHeader* ▶ *navigationBar* ▶ *sap.m.Bar* ▶ *contentRight* ▶ *sap.m.HBox* and set the property *Visible* to `true`.
Set object page tables with non-editable content only to not editable **i Note** This is relevant for tables with non-editable content only. By default, object page tables are automatically set to `editable` if the object page is in edit mode. This means that users can only access editable table content when they navigate through the table using the tab key. However, in a table with only non-editable content, there is no tab stop.	Set the `editable` property of the smart table to `false`. The inner table's `navigationMode` property is set to `Navigation`. Users can then navigate through the entire focusable table content, and not only through the editable fields.
Enable "Include Item In Selection" for tables. By setting this property to true, the item selection is displayed even if a user navigates away from a table.	For the SmartTable control, set the `includeItemInSelection` property to `True`.
Enable the *Save and Edit* button in non-draft applications. Users can choose this button to save the current changes. The object page stays in edit mode so that they can continue editing.	In the SAPUI5 Visual Editor, go to the object page of your app. Switch to edit mode and choose the `Save and Edit` button from the outline panel. Change the visibility to true. In the change file that's created, change the new value from `true` to `{ui>/editable}`.
Change the layout of the object table for better readability in case of a high number of columns.	For `sap.m.Table`, set the `PopinLayout` property to one of the following values: • `Block` • `GridSmall` • `GridLarge` The default layout is `Block`.

More Information

For information about adapting the UI in the SAP Web IDE, choose ▶ *Help* ▶ *Documentation* ▶ *Developing* ▶ *Developing Web Applications* ▶ *SAPUI5 Visual Editor*.

Creating a Binding Change

You can create property binding changes manually by using the UI Adaptation Editor. Note that you can use only those properties whose data has already be retrieved by the model.

Context

> **i Note**
>
> This procedure is only relevant for the following use cases described in Adapting the UI: List Report and Object Page [page 1860]:
>
> - Object page: Show header content in edit mode
> - Object page: Show content parts in the header according to a specific mode

Perform the following steps to create a property binding change:

Procedure

1. Open SAP Web IDE and choose the SAPUI5 Visual Editor for your app.
2. Change the property to which you want to apply a property binding, for example, the *Visible* property of a button. This is an example of a change file:

```
{
    "fileName": "id_1460988346969_256_propertyChange",
    "fileType": "change",
    "changeType": "propertyChange",
    "reference": "STTA_MP.Component",
    "packageName": "$TMP",
    "content": {
        "property": "visible",
        "oldValue": true,
        "newValue": false
    },
    "selector": {        "id":
"STTA_MP::sap.suite.ui.generic.template.ObjectPage.view.Details::STTA_C_MP_Pro
duct--
action::STTA_PROD_MAN.STTA_PROD_MAN_Entities::STTA_C_MP_ProductCopywithparams"
,
        "type": "sap.uxap.ObjectPageHeaderActionButton"
    },
    "layer": "VENDOR",
    "texts": {},
    "namespace": "apps/STTA_MP/changes/",
    "creation": "2016-04-18T14:05:47.149Z",
    "originalLanguage": "EN",
    "conditions": {},
    "context": "",
    "support": {
        "generator": "Change.createInitialFileContent",
        "service": "",
        "user": ""
    }
```

```
}
```

Make the following replacements in this change:

- Change the value of the `changeType` from `propertyChange` to `propertyBindingChange`.
- In the `content`, replace `newValue` with `newBinding`, and its value with your required binding, for example, `{myProperty}`. In this example, `myProperty` contains the values `true` or `false` to change the visibility.
  The result looks as follows:

```
{
    "fileName": "id_1460988346969_256_propertyChange",
    "fileType": "change",
    "changeType": "propertyBindingChange",
    "reference": "STTA_MP.Component",
    "packageName": "$TMP",
    "content": {
        "property": "visible",
        "oldValue": true,
        "newBinding": "{myProperty}"
    },
    "selector": {
        "id":
"STTA_MP::sap.suite.ui.generic.template.ObjectPage.view.Details::STTA_C_MP_
Product--
action::STTA_PROD_MAN.STTA_PROD_MAN_Entities::STTA_C_MP_ProductCopywithpara
ms",
        "type": "sap.uxap.ObjectPageHeaderActionButton"
    },
    "layer": "VENDOR",
    "texts": {},
    "namespace": "apps/STTA_MP/changes/",
    "creation": "2016-04-18T14:05:47.149Z",
    "originalLanguage": "EN",
    "conditions": {},
    "context": "",
    "support": {
        "generator": "Change.createInitialFileContent",
        "service": "",
        "user": ""
    }
}
```

i Note

You can also use an expression binding. For example, if you want to inverse your property you can use `newBinding: "{= !${myProperty}}"`.

The following expressions might be useful:

- Object Page: Edit mode: `{ui>/editable}`
- Object Page: Display mode: `{= !${ui>/editable}}`
- Object Page: Create mode: `{ui>/createMode}`
  Indicates whether the UI currently displays an entity that is about to be created (no active version exists yet).
- Object Page: Controls enabled: `{ui>/enabled}`
  Indicates whether active UI elements (such as buttons) should currently be enabled.

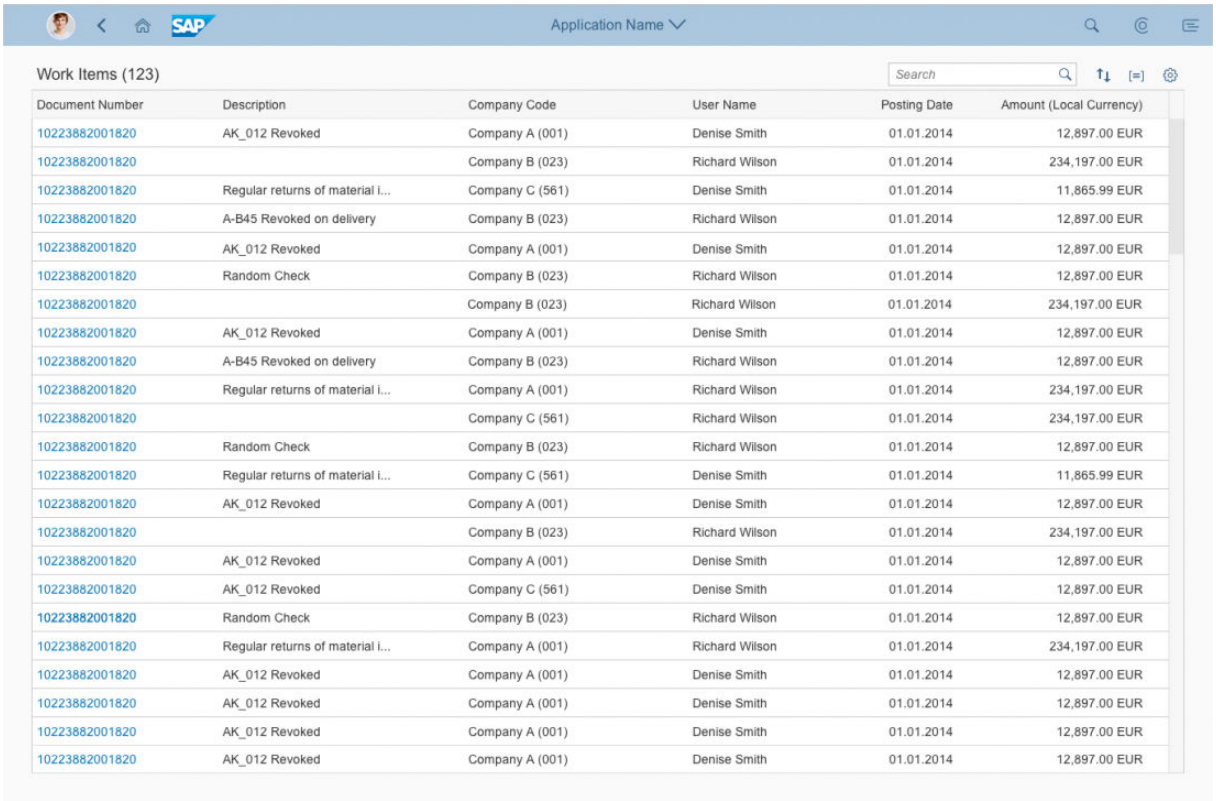
You can set a binding to an i18n text. To do so, add the i18n model in front of the property. Example: `newBinding: "{i18n>xtol.MoveDown}"`.

Worklist

A worklist displays a collection of items that are to be processed by the user.

Working through the item list usually involves reviewing details of the list items and taking action. In most cases, the user has to either complete or delegate a work item.

The focus of the worklist floorplan is on processing the items. This differs from the list report floorplan, which focuses on filtering content to create a list.



Document Number	Description	Company Code	User Name	Posting Date	Amount (Local Currency)
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820		Company B (023)	Richard Wilson	01.01.2014	234,197.00 EUR
10223882001820	Regular returns of material i...	Company C (561)	Denise Smith	01.01.2014	11,865.99 EUR
10223882001820	A-B45 Revoked on delivery	Company B (023)	Richard Wilson	01.01.2014	12,897.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	Random Check	Company B (023)	Richard Wilson	01.01.2014	12,897.00 EUR
10223882001820		Company B (023)	Richard Wilson	01.01.2014	234,197.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	A-B45 Revoked on delivery	Company B (023)	Richard Wilson	01.01.2014	12,897.00 EUR
10223882001820	Regular returns of material i...	Company A (001)	Richard Wilson	01.01.2014	234,197.00 EUR
10223882001820		Company C (561)	Richard Wilson	01.01.2014	234,197.00 EUR
10223882001820	Random Check	Company B (023)	Richard Wilson	01.01.2014	12,897.00 EUR
10223882001820	Regular returns of material i...	Company C (561)	Denise Smith	01.01.2014	11,865.99 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820		Company B (023)	Richard Wilson	01.01.2014	234,197.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	AK_012 Revoked	Company C (561)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	Random Check	Company B (023)	Richard Wilson	01.01.2014	12,897.00 EUR
10223882001820	Regular returns of material i...	Company A (001)	Richard Wilson	01.01.2014	234,197.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR
10223882001820	AK_012 Revoked	Company A (001)	Denise Smith	01.01.2014	12,897.00 EUR

- Adapting the Application Header [page 1615]
- Smart Tables [page 1628]
- Worklist [page 1866]

Worklist Features That Differ from List Report Features

From a technical perspective, a worklist is a simplified list report. The following aspects differ from the list report features:

- You create a worklist using SAP WebIDE. In the *Template Selection* step, choose *Worklist* as a template. For more information, see Building an App Using SAP Web IDE [page 1553].

- The worklist does not contain a smart filter bar. The search field is available in the table toolbar.
- You can choose an app-specific title by setting the corresponding value for the `subTitleIfVariantMgmtHidden` property.
  For more information, see Step 2 under Creating a List Report Without Variant Management [page 1639].
- Variant management:
  - By default, variant management is hidden. You can customize the worklist to provide variant management at table level. To do so, set the `variantManagementHidden` flag to `false` in the manifest.json. You can enable page level variant management by setting `smartVariantManagement` to `true` and the `variantManagementHidden` flag to `false` in the manifest.json. Variants can also be shared.
  - The *Execute on Select* action is not available.
- Smart table:
  - The multiselect function is enabled for all tables. If there are only line item actions, a no-selection table is enabled.
  - The *Export to Microsoft Excel* feature is not available.
  - The default table type is **responsive**. The table title contains the row count. A fixed layout and growing using the scrolling function is enabled.
  - The table header menu provides the following options: Sort, filter, group, and column settings.

Worklist Types

In the SAP WebIDE, you generate a simple worklist. You can create a category worklist by defining a view on a worklist, as described under Defining Multiple Views on a List Report Table - Single Table Mode [page 1649].

Configuring the Worklist

See the user assistance for List Report and Object Page [page 1622] for any worklist configuration options. Apart from the features listed above, you can also use the options described for the list report, for example:

- Configuring List Report Features [page 1637]
- Configuring Navigation [page 1563]
- Configuring Tables [page 1735]
- Configuring Further Common Features [page 1778]
- Extending List Reports and Object Pages Using App Extensions [page 1799]

For general information about the worklist floorplan, see SAP Fiori Design Guidelines.

Analytical List Page

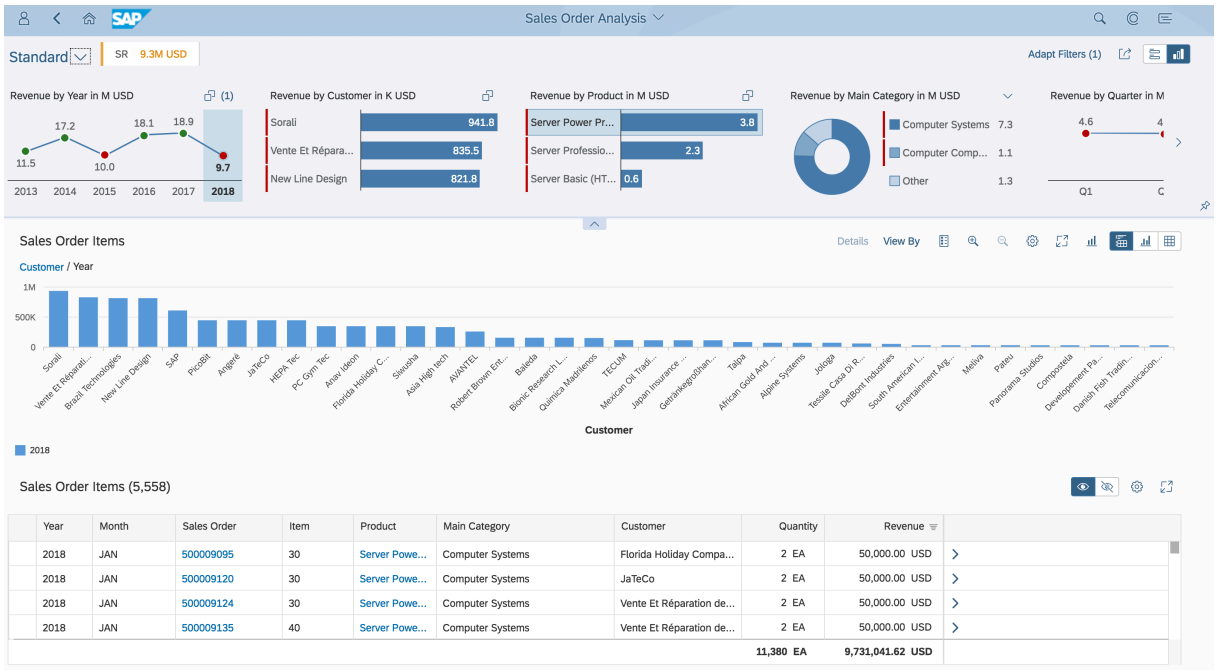
Analytical List Page (ALP) is an SAP Fiori elements application for detailed analytics.

ALP lets you analyze data from different perspectives, investigate a root cause, and to act on transactional content. You can identify relevant areas within data sets or significant single instances using data visualization and business intelligence. All this can be done seamlessly within one page.

The combination of transactional and analytical data using chart and table visualization lets you quickly view the data you need. This hybrid view allows an interesting interplay between the chart and table representations.

Configure ALP to include the following use cases seamlessly on one page:

- Related KPIs (key performance indicators) on the header area as KPI tags. These KPI tags also allow progressive disclosure and navigation through KPI cards.
- Filter data sets used for the main content area with different filter modes. For example, visual filters provide an intuitive way of choosing filter values from an associated measure value.
- Seamless navigation to applications from the content area and the KPI card area.
- Customizing and sharing ALP as a page variant with other users.



- Visual Filter Setup [page 1885]
- Chart-Only View [page 1910]
- Table-Only View [page 1902]
- Managing Variants [page 1616]
- Creating Key Performance Indicator Tags [page 1873]
- Choosing Filter Modes [page 1880]
- Configuring the Content Area [page 1901]

More Information

For more information about the analytical list page plugin, see Building an App Using SAP Web IDE [page 1553].

Related Information

Descriptor Configuration [page 1869]
Configuring the Title Area [page 1872]
Configuring the Filter Area [page 1881]
Configuring the Content Area [page 1901]
Configuring Analytical List Page App Extensions [page 1915]

Descriptor Configuration

The descriptor file (manifest.json) is an application configuration file that contains valid entries for initializing the analytical list page (ALP).

Generic App Configuration

> **i** Note
>
> Analytical List Page (ALP) works only for analytical.
>
> Sample Code

```
{
<EntityType Name="SEPMRA_C_ALP_SlsOrdItemCubeALPResult"
sap:semantics="aggregate" sap:label="Sales Analysis" sap:value-list="true"
sap:content-version="1">
                                                        <Key>

<PropertyRef Name="ID"/>
                                                        </Key>
                                                        <Property
Name="ID" Type="Edm.String" Nullable="false" sap:sortable="false"
sap:filterable="false"/>
                                                        <Property
Name="SalesOrder" Type="Edm.String" MaxLength="10" sap:aggregation-
role="dimension" sap:display-format="UpperCase" sap:label="Sales Order"
sap:quickinfo="Sales Order ID" sap:creatable="false"
sap:updatable="false"/>
                                                        <Property
Name="SalesOrderItem" Type="Edm.String" MaxLength="10" sap:aggregation-
role="dimension" sap:display-format="UpperCase" sap:label="Sales Order
Item" sap:quickinfo="Sales Order Item ID"/>
                                                .
                                                .
</EntityType>
```

```
}
```

The following code sample provides the descriptor configuration with the default values relevant for the ALP.

```
"sap.ui.generic.app": {
 "_version": "1.2.0",
      "settings":{
         "flexibleColumnLayout":{
            "defaultTwoColumnLayoutType":"TwoColumnsBeginExpanded"
         }
      },
 "pages": [{ // This can have multiple page definitions within it. ALP currently
only consumes the first page.
   "entitySet": "SEPMRA_C_ALP_SlsOrdItemCube",
/* Represents the entity set that is used to populate the main content area.
For parametrized entity set, use result entity set name instead of parametrized
entity set. */
   "component": {
    "name": "sap.suite.ui.generic.template.AnalyticalListPage", // Should not be
changed.
     "list":true,
    "settings": {
    "tableSettings": {
    "type": "GridTable or AnalyticalTable or ResponsiveTable",
/*Use these settings to select a table in ALP. Supports three table types:
"AnalyticalTable", "GridTable" & "ResponsiveTable".
By default, ALP determines the table type when undefined or if there is an
incorrect value.
Note: Do not set "tableType" to "AnalyticalTable" if the underlying service for
the main entity set
is not an aggregate service.*/
      "multiSelect": true,
/* When true, multiple records in the tabular display can be
selected. This setting takes effect only if the service has
defined Actions in annotation or inside ControllerExtension.
Else, the selection is always single.*/
      "selectAll": true
/*Use this setting to select all the rows that are available in the back end
with the current filters.
It triggers only one batch call to select all the records.*/
      "selectionLimit": 20
/*You can only select 20 rows at a time while selecting a range of rows in the
table.
If selectionLimit is not provided, then a default value of 200 is set to
selectionLimit.
Note: Select all and selection limit are applicable only for GridTable and
AnalyticalTable*/
}
      "qualifier": "DefaultPresentationVariant"
/* Represents the SelectionPresentationVariant qualifier.
ALP looks for SelectionPresentationVariant with this qualifier
and if not found, it looks for PresentationVariant with this qualifier.
This setting is optional, if this is not provided ALP looks for
Default LineItem and Chart without qualifiers. */
       "condensedTableLayout": true, // When set to false, compact style will be
used to render the table.
       "smartVariantManagement": true, // When false, control level variant
management is used instead of page level variant management
       "defaultContentView": "charttable", // Determines the visualization of
content area. Possible other values: chart or table.
       "defaultFilterMode": "visual", // Determines the filter mode that is
used. Possible other value: compact
       "showGoButtonOnFilterBar": false, // Go button is displayed for compact
filters when this is set to true
        "contentTitle": "{{contentAreaTitle}}", //Lets you define title for the
content area. Ensure that the contentAreaTitle is also defined in i18n.properties
```

```
        "autoHide":false,
       "filterDefaultsFromSelectionVariant":false, //Lets you to add default
values for FilterBar using SelectionVariant annotation.
        "condensedTableLayout":false,
       "keyPerformanceIndicators": { // The first 3 KPIs listed here show up in
the KPI tags
            "KPIRevenue": { // First KPI
                "model": "kpi",
/* Links to the item in "models" section which provides additional information,
for example, the data source for the KPI from which we could further obtain the
data source and annotation corresponding to this KPI. This property must not be
empty.  */
                "entitySet": "SEPMRA_C_ALP_TotalSalesKPI", // Entity set used
for bringing up the details displayed within the KPI tag/card.
                "qualifier": "KPIRevenue",
/* Refers to the UI.KPI annotation. */
                "detailNavigation": "OverviewPage",
/* Points to an element within "outbounds" property of the
"crossNavigation" section. The details there help us in determining
the target application as well as the parameters that need to be passed
upon navigation from the KPI card footer. */
                "groupId": "mainKpiGroup"
/* To achieve faster end-to-end response time within analytical list page, enable
batching of KPIs by defining groupId in the KPI section of the descriptor
configuration file. You can determine which KPIs should be grouped together in a
given batch call.
Note:
-> The KPIs with same data source are batched together. KPIs with same
groupId and different data source triggers different batch calls.
-> Batching of KPIs may increase the number of batch calls to the back end.
*/
            }
        } // End of KPIs
    } // End of settings
  },
  "pages": [{ //ALP does an inner app navigation to the smart template specified
below
            "entitySet": "SEPMRA_C_ALP_SlsOrdItemCube",
            "component": {
                "name": "<
<sap.suite.ui.generic.template.ObjectPage>>"
            }
            "navigation": { //Optional (If specified ALP navigates to the target
Application specified below through external app navigation)
            "display": {
               "path": "sap.apps.crossNavigation.outbounds",
               "target": "<

    <NavigateToCTRItem>>"
             }
            }
    }]
}] // End of outer pages
}
```

Related Information

Analytical List Page [page 1868]

Configuring the Title Area

The dynamic area of the analytical list page is the title area.

In the header area, you can view information related to the Key Performance Indicator (KPI) or choose any of the following built-in SAP Fiori elements features to:

- Define or manage page variants
- Choose filter modes (compact or visual)
- Customize the filter area
- Share the analytical list page

UI.KPI Annotation

Use the `UI.KPI` annotation to display KPIs on your analytical list page application. Define the descriptor configuration file with a qualifier and map it to the UI.KPI annotation. If this annotation is not configured, then the `UI.SelectionPresentationVariant` annotation with the same qualifier name takes effect.

We recommended using a single SAP-wide KPI gallery based on the KPIs generated by standard SAP KPI creation tools, such as the SAP Smart Business framework.

Sample Code

Descriptor setting

```
"KPIQuantity": {
                "model": "kpi",
                "entitySet": "SEPMRA_C_ALP_SlsOrdItemCubeALPResults",
                "qualifier": "KPIQuantity
}
```

To enable navigation for a KPI card, you need to define the semantic object, action, and KPI ID in the `UI.KPI` annotation. The KPI ID is passed to the target application as the `EvaluationId` for launching the smart business application.

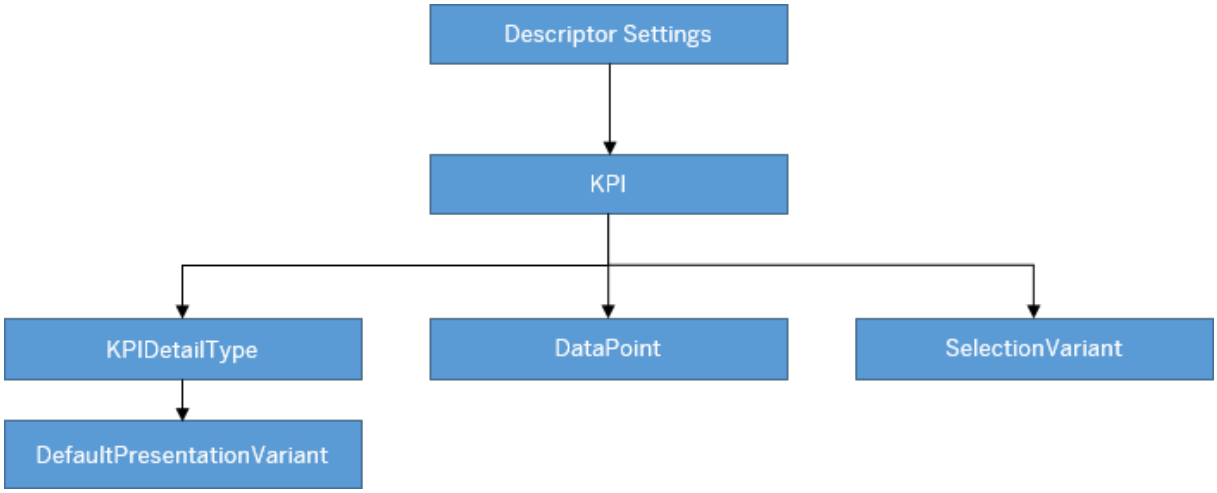
Related Information

Analytical List Page [page 1868]
Managing Variants [page 1616]
Creating Key Performance Indicator Tags [page 1873]
Choosing Filter Modes [page 1880]

Creating Key Performance Indicator Tags

The key performance indicator (KPI) tag is an abbreviated and clickable title with a KPI value.

Each KPI can have its own OData source, entity set, and annotation file. The KPI value changes if an action is executed on the transactional content. For example, releasing sales orders affects a related KPI and posting an accounting document affects certain financial KPIs.



- [page 1873]
- [page 1874]
- [page 1874]
- [page 1874]
- [page 1875]
- [page 1875]

Hover over each action for a description. Click the action for more information.

Descriptor Settings: KPI configuration

Property: `keyPerformanceIndicators`

Use the `UI.KPI` annotation to display KPIs on your analytical list page application (title area or below filter bar depending on the value of the `filterable`). Define the descriptor configuration file with a qualifier and map it to the UI.KPI annotation. If this annotation is not configured, then the `UI.SelectionPresentationVariant` annotation with the same qualifier name takes effect.

We recommend using a single SAP-wide KPI gallery based on the KPIs generated by standard SAP KPI creation tools, such as the SAP Smart Business framework.

Configuration Sample:

```
"sap.ui.generic.app":{
   "pages":[
      {
         "entitySet":"SEPMRA_C_ALP_SlsOrdItemCube",
         "component":{
            "name":"sap.suite.ui.generic.template.AnalyticalListPage",
            "list":true,
```

```
            "settings":{
               "keyPerformanceIndicators":{
                  "KPIRevenue":{
                     "model":"kpi",
                     "entitySet":"SEPMRA_C_ALP_TotalSalesKPI",
                     "qualifier":"KPIQuantity",
                     "detailNavigation":"EPMProduct"
                  }
               }
      }
},
```

Annotation: `UI.KPI` annotation with the qualifier `KPIQuantity`

```
<Annotation Term="UI.KPI" Qualifier="KPIQuantity">
   <Record>
      <PropertyValue Property="DataPoint" Path="@UI.DataPoint#DPForQuantity" />
      <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#SVForQuantity" />
      <PropertyValue Property="ID" String="ActualCostByGLAccountNameKPI" />
      <PropertyValue Property="Detail">
         <Record Type="UI.KPIDetailType">
            <PropertyValue Property="SemanticObject" String="EPMProduct" />
            <PropertyValue Property="Action" String="manage_stta" />
            <PropertyValue Property="DefaultPresentationVariant"
Path="@UI.PresentationVariant#PVForQuantity" />
         </Record>
      </PropertyValue>
   </Record>
</Annotation>
```

To enable navigation for a KPI card, you need to define the semantic object, action, and KPI ID in the `UI.KPI` annotation. The KPI ID is passed to the target application as an `EvaluationId` for launching the SAP Smart Business application.

Annotation: KPIDetailType

```
<PropertyValue Property="Detail">
   <Record Type="UI.KPIDetailType">
      <PropertyValue Property="SemanticObject" String="EPMProduct" />
      <PropertyValue Property="Action" String="manage_stta" />
      <PropertyValue Property="DefaultPresentationVariant"
Path="@UI.PresentationVariant#PVForQuantity" />
   </Record>
</PropertyValue>
```

Annotation: DataPoint

Sample Code

`DataPoint` annotation with the qualifier `DPForQuantity`

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint"
Qualifier="DPForQuantity">
   <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
      <PropertyValue Property="Title" String="Quantity by Customer Company
Name" />
      <PropertyValue Property="Description" String="About
NumberOfSalesOrders" />
      <PropertyValue Property="Value" Path="Quantity" />
      <PropertyValue Property="Criticality"
EnumMember="com.sap.vocabularies.UI.v1.CriticalityType/Neutral" />
   </Record>
```

```
</Annotation>
```

Annotation: `SelectionVariant` with the qualifier `SVForQuantity`

Description: Configure this annotation for filters and parameters to provide default values for the corresponding filter fields. This overrides the default values from the `Common.FilterDefaultValue` annotation. You get the filters from the `SelectionVariant.SelectOptions` and the parameters from the `SelectionVariant.Parameters`.

```
<Annotation Term="UI.SelectionVariant" Qualifier="SVForQuantity">
   <Record>
      <PropertyValue Property="Parameters">
         <Collection>
            <Record Type="UI.Parameter">
               <PropertyValue Property="PropertyName"
PropertyPath="CompanyCurrency" />
               <PropertyValue Property="PropertyValue" String="EUR" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="SelectOptions">
         <Collection>
            <Record Type="UI.SelectOptionType">
               <PropertyValue Property="PropertyName"
PropertyPath="MainProductCategory" />
               <PropertyValue Property="Ranges">
                  <Collection>
                     <Record Type="UI.SelectionRangeType">
                        <PropertyValue EnumMember="UI.SelectionRangeSignType/I"
Property="Sign" />
                        <PropertyValue EnumMember="UI.SelectionRangeOptionType/
EQ" Property="Option" />
                        <PropertyValue Property="Low" String="Computer
Systems" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Annotation: `PresentationVariant` annotation with the qualifier `PVForQuantity`

```
<Annotation Term="com.sap.vocabularies.UI.v1.PresentationVariant"
Qualifier="PVForQuantity">
   <Record>
      <PropertyValue Property="SortOrder">
         <Collection>
            <Record Type="Common.SortOrderType">
               <PropertyValue Property="Property" PropertyPath="Quantity" />
               <PropertyValue Property="Descending" Bool="true" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Visualizations">
         <Collection>
            <AnnotationPath>@com.sap.vocabularies.UI.v1.Chart#QuantityChart</
AnnotationPath>
         </Collection>
      </PropertyValue>
   </Record>
```

```
</Annotation>
```

Sample Code

Chart annotation with the qualifier `QuantityChart`

```
<Annotation Term="com.sap.vocabularies.UI.v1.Chart" Qualifier="QuantityChart">
   <Record>
      <PropertyValue Property="Title" String="NumberOfSalesOrders" />
      <PropertyValue Property="MeasureAttributes">
         <Collection>
            <Record
Type="com.sap.vocabularies.UI.v1.ChartMeasureAttributeType">
               <PropertyValue Property="Measure" PropertyPath="Quantity" />
               <PropertyValue Property="Role"
EnumMember="com.sap.vocabularies.UI.v1.ChartMeasureRoleType/Axis1" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="DimensionAttributes">
         <Collection>
            <Record
Type="com.sap.vocabularies.UI.v1.ChartDimensionAttributeType">
               <PropertyValue Property="Dimension"
PropertyPath="SoldToPartyCompanyName" />
               <PropertyValue Property="Role"
EnumMember="com.sap.vocabularies.UI.v1.ChartDimensionRoleType/Category" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Bar" />
      <PropertyValue Property="Description" String="NumberOfSalesOrders by
Customer Company Name" />
   </Record>
</Annotation>
C
```

KPI Titles

The abbreviation of the KPI title is based on this logic:

If a KPI name is	Abbreviation is based on	Example
One word	First three letters	KPI Name: TargetMargin KPI Title: *TAR*
Two words	First letter of each word	KPI Name: Actual Cost KPI Title: *AC*
Three words or more	First letter of first three words	KPI Name: Actual Margin Relative KPI Title: *AMR*

Note

The KPI name is taken from the `Title` property of the `DataPoint` annotation.

KPI Value and Color

Use the `UI.KPI` annotation to render KPI values and to determine the KPI's color. This annotation is associated with the `SelectionVariant` and `DefaultPresentationVariant`.

The first `DataPoint` visualization element in the `PresentationVariant` renders the KPI value and determines the KPI's color. If a KPI is percent-based, then the "%" sign appears along with the KPI value.

> **i Note**
>
> Click the KPI title or value to view more details on a KPI card.

The color of the KPI value depends on the threshold values. To define the threshold values, use these properties in the DataPoint annotation:

- CriticalityCalculation: Allows you to hard code values or to include the value from a property path
- Criticality: Allows complex back-end logic to specify the criticality values

> **i Note**
>
> If you use both properties in the annotation, then the criticality value overrides the `CriticalityCalculation` value.

The color logic depends on the measure type. You can choose a maximizing measure, minimizing measure, or range-based measure types based on the `"ImprovementDirection"` value of the measure.

- Target Measure: Hover over each item to view the conditions.

- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Maximizing Measure: Hover over each item to view the conditions.

- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Minimizing Measure: Hover over each item to view the conditions.

- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]
- Creating Key Performance Indicator Tags [page 1873]

If the threshold values are insufficient or incorrect, the ALP chooses the closest color match for a KPI value. For example, if a target KPI has the `DataPoint.Value` aggregate < `ThresholdValues.ToleranceRangeLowValue` and does not have a value for `ThresholdValues.DeviationRangeLowValue`, then the KPI value has the color for critical.

> **i Note**
>
> The criticality indicator line in a KPI tag also takes the same color as the KPI value.

Scale, Decimal Precision and Number Formatting

The SAPUI5 formatter returns the scale factor and decimal factor for a KPI value.

The NumberOfFractionalDigits information can be provided in com.sap.vocabularies.UI.v1.DataPoint [page 2010] term, using the ValueFormat property. The NumberOfFractionalDigits property is used to determine the number of fraction digits. These are the rules:

- Decimals are not shown by default.
- You can specify 1 or 2 decimal places by using the NumberOfFractionalDigits property in Annotations. If a value higher than 2 is provided, it is considered in addition to 2.

In the following example, the price property number of fractions digit value provided in the OData metadata - 3 is overridden with the value 1 as provided in the com.sap.vocabularies.UI.v1.DataPoint [page 2010] ValueFormat property

> **Sample Code**

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint" Qualifier="Price">
    <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
        <PropertyValue Property="Value" Path="Price"/>
        <PropertyValue Property="ValueFormat">
            <Record Type="com.sap.vocabularies.UI.v1.NumberFormat">
                <PropertyValue Property="NumberOfFractionalDigits" int="1"/>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

Display KPI Tags with Units of Measure

The global and filterable KPIs in the ALP can now display KPI values with a **Unit of Measure**. For example, to add a **Unit of Measure**, define a path or string value directly.

Sample Code

```
<Annotations Target="CZ_PROJECTKPIS_CDS.CZ_PROJECTKPISType/ActualCost">
        <Annotation Term="Org.OData.Measures.V1.ISOCurrency"
Path="CompanyCodeCurrency"/>
</Annotations>
<Annotations Target="CZ_PROJECTKPIS_CDS.CZ_PROJECTKPISType/TargetMargin">
        <Annotation Term="Org.OData.Measures.V1.ISOCurrency" String="EUR"/>
</Annotations>
<Annotations Target="CZ_PROJECTKPIS_CDS.CZ_PROJECTKPISType/
ActualMarginRelative">
        <Annotation Term="Org.OData.Measures.V1.Unit" String="%"/>
</Annotations>
<Annotations Target="CZ_PROJECTKPIS_CDS.CZ_PROJECTKPISType/NetWeight">
        <Annotation Term="Org.OData.Measures.V1.Unit" Path="WeightUnit"/>
</Annotations>
```

You can override a unit of measure that comes from the back end. For example, the following annotation configuration overrides the unit of measure from the back end and changes it to a percentage-based unit of measure.

```
<Annotations Target="CZ_PROJECTKPIS_CDS.CZ_PROJECTKPISType/ActualCost">
    <Annotation Term="Org.OData.Measures.V1.Unit" String="%"/>
</Annotations>
```

Filterable KPI

The filterable KPIs react to filter bar changes when there is an exact match between the technical name, modified filter, and parameter field in the filter bar and in the KPI's entity set.

Note

- If the default filter values specified in the `SelectionVariant` annotation are also part of the filter bar fields, then the filter bar values override the `SelectionVariant` annotation default values. Otherwise they are applied in addition to the filter bar values.
- The ALP ignores the `UI.Hidden` fields on filter selection for filterable KPIs, if the filter field coming from filter bar is marked as `UI.Hidden` in the KPI entity set.

To ensure that KPIs show up as filterable, set `"filterable"=true` in the app's descriptor file.

Configuration Sample:

```
"sap.ui.generic.app":{
   "pages":[
      {
         "entitySet":"SEPMRA_C_ALP_SlsOrdItemCube",
         "component":{
            "name":"sap.suite.ui.generic.template.AnalyticalListPage",
            "list":true,
```

```
            "settings":{
               "keyPerformanceIndicators":{
                  "KPIRevenue":{
                     "model":"kpi",
                     "entitySet":"SEPMRA_C_ALP_TotalSalesKPI",
                     "qualifier":"KPIRevenue",
                     "filterable":true,
                     "detailNavigation":"OverviewPage"
                  }
               }
      }
},
```

Semantic coloring for Filterable KPI

If `CriticalityCalculation` is defined in the annotations, filterable KPIs are updated based on a change to the filter or parameter. If the values come from a path, then criticality indicators change based on the KPI value.

> **i Note**
>
> Only path-based is supported for filterable KPIs and semantic coloring depends on the changes to the filter bar.

Related Information

Configuring the Title Area [page 1872]
Managing Variants [page 1616]

Choosing Filter Modes

ALP offers compact and visual filter modes. You can choose to set filters from both modes.

> **i Note**
>
> Based on the annotation configuration, filter dimensions show up in either of the filter modes, or on both the filter modes. However, a filter field on the visual filter always shows up in compact filter mode.

Filter Bar Customization

You can customize filter fields or charts that show up on the filter bar. Use the *Adapt Filters (<number>)* option in the header area to customize both filter modes. The filter fields in the

- Compact filter configuration dialog box are the fields in the entity set that have `sap:filterable="true"` in their metadata. Select the filter fields that you want to use on the filter bar.
- Visual filter configuration dialog box are the fields in the entity set that have `sap:filterable="true"` in their metadata. The fields need to have a `ValueList` annotation with a valid presentation variant qualifier.

In the visual filter mode, you can change the chart type, sort order, and measure according to your needs. Visual filter supports three basic chart types (bar, donut, and line). When you select a chart filter, the other chart filters are refreshed to provide the appropriate fields. In some cases, some of the filters selected may no longer be visible in the chart area. If this is the case, choose the *Value help* or *Selected* icon, to view all of the visual filter conditions that have been set.

> **i** Note
>
> The filter selection count depends on the number of fields you select and is not based on the individual field values set within the field. The count also includes filters set from the filter dialog that are not seen on the filter bar.

Filter Modes

Analytical List Page lets you toggle between the visual filter and compact filter, even while filter conditions are applied. When you toggle, ALP ensures that the filter values in one mode sync with those in the other mode. However,

- Filter selection in fields without visual filter configuration are seen only in compact filters.
- Parameters do not show up in visual filters.

Despite the prior conditions, all selections in the compact filter are included for the main content area, even when visual filters are used.

Related Information

Configuring the Title Area [page 1872]
Managing Variants [page 1616]
Defining ValueList Annotation [page 1896]

Configuring the Filter Area

The section following the title area is the filter area. Set up filters to get the serach results you want in the main content area.

You can use either a compact filter or a visual filter to perform the search operation (when sap:filterable=true). To make sure the selection filter with incoming values, pass the filter context through the navigation context.

You can also configure default filter values by defining the `Common.FilterDefaultValue` annotation for a property type.

> **i** Note
>
> The default filter values from the `Common.FilterDefaultValue` property are overwritten by the variant management or navigation context in the specified order.

Sample Code

Default filter value configuration annotation sample

```
<Annotations Target="SEPMRA_SO_ANA.SEPMRA_C_ALP_SlsOrdItemCubeType/
DeliveryCalendarYear">
    <Annotation Term="Common.FilterDefaultValue" String="2018" />
</Annotations>
```

The filter dimensions shown in the filter dialog can belong to any of the following groups:

Basic Group

All filter dimensions listed within the SelectionField annotation property belong to the this filter group. In addition, all mandatory filters, parameters, and data fields marked with the `FieldGroup` annotation with qualifier `_BASIC` are included in this group.

Field Group

Filter dimensions listed under a FieldGroup annotation property but not listed within the SelectionField annotation property belong to this filter group. For example, a new group with name `<EntityTypeName>` is created. If there are multiple `entityTypes`, multiple `entityTypeName` groups are created.

This kind of filter dimension is a part of the filter dialog but does not show up in the filter bar unless it is explicitly added by means of a filter dialog or is part of a chosen variant.

Entity Type Name Group

Filter dimensions that are not part of the `SelectionField` or do not have `FieldGroup` annotation properties belong to this filter group. This kind of filter dimension is part of the filter dialog but does not show up in the filter bar unless it is explicitly added by means of a filter dialog or is part of a chosen variant.

```
<Annotation Term="UI.FieldGroup" Qualifier="Group Name">
    <Record>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataField">
                    <PropertyValue Property="Value" Path="Property Name"/>
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Label" String="Group Name"/>
    </Record>
</Annotation>
```

Go Button

The *Go* button appears in the ALP filter area. Use this button to refresh or load the main content area. By default, the *Go* button is disabled. To enable the *Go* button, set `"showGoButtonOnFilterBar": true` in the descriptor file `sap.ui.generic.app settings` property.

If you enable the *Go* button and modify the filter selection, the main content area is not refreshed until you choose *Go*.

```
"sap.ui.generic.app": {
```

```
        "_version": "1.1.0",
        "pages": [{
            "entitySet": "SEPMRA_C_ALP_SlsOrdItemCube",
            "component": {
                "name": "sap.suite.ui.generic.template.AnalyticalListPage",
                "list": true,
                "settings": {
                                    "showGoButtonOnFilterBar": true,
                                    }
                        }
            }]
               }
```

Default Values in Filter Bar

The new `filterDefaultsFromSelectionVariant` app descriptor setting, lets you to add default values for `FilterBar` using `SelectionVariant` annotation.

Sample Code

```
<Annotation Term="UI.SelectionVariant" Qualifier="Default">
    <Record>
        <PropertyValue Property="Parameters">
            <Collection>
                <Record Type="UI.Parameter">
                    <PropertyValue Property="PropertyName"
PropertyPath="P_CompanyCode" />
                    <PropertyValue Property="PropertyValue" String="EASI" />
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="SelectOptions">
            <Collection>
                <Record Type="UI.SelectOptionType">
                    <PropertyValue Property="PropertyName"
PropertyPath="Customer"/>
                    <PropertyValue Property="Ranges">
                        <Collection>
                            <Record Type="UI.SelectionRangeType">
                                <PropertyValue
EnumMember="UI.SelectionRangeSignType/I" Property="Sign"/>
                                <PropertyValue
EnumMember="UI.SelectionRangeOptionType/EQ" Property="Option"/>
                                <PropertyValue Property="Low" String="ABC"/>
                            </Record>
                        </Collection>
                    </PropertyValue>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Note

- The default values coming from the `SelectionVariant`:
  - Override the default values specified by the `Common.FilterDefaultValue`.
  - Are considered on application load for standard variant. The default values do not persist if you switch to another variant and return to the standard variant.

- SAP Fiori launch pad default values overrides the `SelectionVariant` default values.

Related Information

Analytical List Page [page 1868]
Compact Filter Setup [page 1884]
Visual Filter Setup [page 1885]
Defining ValueList Annotation [page 1896]

Compact Filter Setup

Lets you specify filter field values based on the configuration in the `sap:filter-restriction` of the respective field in the entity set.

In addition to the fields configured in the `SelectionFields` annotation, compact filters also display the mandatory parameters and filters you need to query the entity set. You can explicitly add the additional filter fields listed in the filter configuration dialog to the filter area, if needed.

The `ValueList` annotation handles the dependency of filter values based on the selection of filter values in other filter fields.

> **i Note**
>
> To enable value help in compact filters, you have to define at least one `ValueList` annotation configuration without the `PresentationVariantQualifier` property.

The `AND` operator groups multiple filter field selections and the `OR` operator groups multiple selections within the same field.

Annotation for Filter Field Selection

Annotation: `UI.SelectionFields`

```
</Annotation>
 <Annotation Term="UI.SelectionFields">
   <Collection>
     <PropertyPath>DeliveryCalendarYear</PropertyPath>
     <PropertyPath>SoldToParty</PropertyPath>
     <PropertyPath>Product</PropertyPath>
     <PropertyPath>MainProductCategory</PropertyPath>
     <PropertyPath>DeliveryCalendarQuarter</PropertyPath>
   </Collection>
</Annotation>
```

Related Information

Configuring the Filter Area [page 1881]
Defining ValueList Annotation [page 1896]

Visual Filter Setup

An intuitive way of choosing filter values from an associated measure value.This setup supports line, bar, and donut charts.

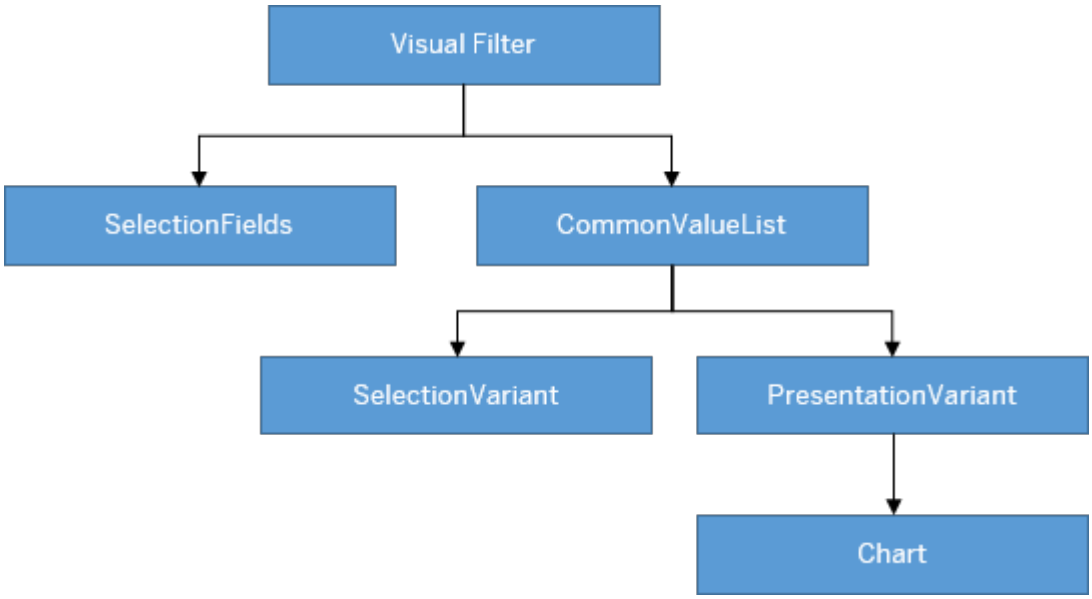
`SelectionFields` annotation for which a visual filter is defined.

> **i Note**
>
> The ALP ignores the `UI.Hidden` fields when you select filters if the IN mapping points to a field marked with `UI.Hidden` in the valuelist entity set. For example, the `Status_ID` from the main entity set points to `StatusCode` in the value help entity set (of the visual filter). If the `StatusCode` is marked as `UI.Hidden`, then the incoming value is ignored.

Configuring the `sap:value-list=fixed-values` property in the `annotation.xml` file lets you display visual filter values in a dropdown list. This allows users to select or deselect values that are not displayed in the chart. If the dropdown list is enabled, ensure that the records available in the collection path entity set of the visual filter and compact filter are the same for a smooth sync between the visual filter and the compact filter.

The visual filter includes only the first measure and dimension from the first chart annotation within the specified `PresentationVariantQualifier`. Make sure that the dimension you specify in the chart and the `ValueListProperty` of the `OUT` parameter is the same. You can also define a `SortOrder` property in the `PresentationVariant` annotation to control the sort order based on a sort field.

> **i Note**
>
> Sorting in visual filters is based on this logic:
>
> - For bar and donut chart types, sorting is always based on the measure displayed (the default is descending order). To change the sort order property, define the `SortOrder` property in the `PresentationVariant` annotation.
> - For line charts with time-based dimensions, sorting is always based on the dimension displayed in ascending order, however, only the last six time periods are displayed.
>   The sorting by the annotation is ignored for time-based dimensions for line-charts in the visual filter.
> - For line charts with non time-based dimensions, sorting is always based on the dimension (the default is ascending order). To change the sort order, define the `SortOrder` property in the `PresentationVariant` annotation.



- #unique_775/unique_775_Connect_42_subsection-im1 [page 1886]
- #unique_775/unique_775_Connect_42_subsection-im2 [page 1887]
- #unique_775/unique_775_Connect_42_subsection-im3 [page 1887]
- #unique_775/unique_775_Connect_42_subsection-im4 [page 1889]
- #unique_775/unique_775_Connect_42_subsection-im5 [page 1889]

Hover over each action for a description. Click the action for more information.

ValueList Annotation

This is an example of a code snippet for a value list annotation. You use it to configure visual filters. For more information, see *Defining ValueList Annotations*.

Sample Code

`ValueList` Annotation Sample

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="SEPMRA_ALP_SO_ANA_SRV.SEPMRA_C_ALP_SlsOrdItemCubeALPResult/
SalesOrderOverallStatus">
   <Annotation Term="Common.ValueList" Qualifier="VisualFilter">
      <Record>
         <PropertyValue Property="Label" String="Overall Status" />
         <PropertyValue Property="CollectionPath"
String="SEPMRA_C_ALP_SlsOrdItemCubeALPResults" />
         <PropertyValue Property="SearchSupported" Bool="false" />
         <PropertyValue Property="PresentationVariantQualifier"
String="FilterNumberOfSalesOrdersByStatus" />
         <PropertyValue Property="SelectionVariantQualifier"
String="SVForStatus" />
         <PropertyValue Property="Parameters">
            <Collection>
               <Record Type="Common.ValueListParameterInOut">
```

```
                  <PropertyValue Property="LocalDataProperty"
PropertyPath="SalesOrderOverallStatus" />
                  <PropertyValue Property="ValueListProperty"
String="SalesOrderOverallStatus" />
               </Record>
               <Record Type="Common.ValueListParameterInOut">
                  <PropertyValue Property="LocalDataProperty"
PropertyPath="MainProductCategory" />
                  <PropertyValue Property="ValueListProperty"
String="MainProductCategory" />
               </Record>
               <Record Type="Common.ValueListParameterInOut">
                  <PropertyValue Property="LocalDataProperty"
PropertyPath="ProductCategory" />
                  <PropertyValue Property="ValueListProperty"
String="ProductCategory" />
               </Record>
            </Collection>
         </PropertyValue>
      </Record>
   </Annotation>
</Annotations>
```

> **i** Note
>
> See the **Defining ValueList Annotations** section for information about the IN/OUT mapping of visual filters.

PresentationVariantQualifier

The `PresentationVariant` qualifier provides chart definitions for visual filters. The visual filter picks up the first chart annotation in the `PresentationVariant` annotation to render the chart. If the chart type is not supported, the ALP renders a bar chart (default chart type).

```
<Annotation Term="UI.PresentationVariant"
Qualifier="FilterNumberOfSalesOrdersByStatus">
   <Record>
      <PropertyValue Property="Text" String="Filter: Number of Sales Order by
Status" />
      <PropertyValue Property="SortOrder">
         <Collection>
            <Record Type="Common.SortOrderType">
               <PropertyValue Property="Property"
PropertyPath="NumberOfSalesOrders" />
               <PropertyValue Property="Descending" Bool="true" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Visualizations">
         <Collection>
            <AnnotationPath>@UI.Chart#FilterNumberOfSalesOrdersByStatus</
AnnotationPath>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Chart Annotation

```
<Annotation Term="UI.Chart" Qualifier="FilterNumberOfSalesOrdersByStatus">
   <Record Type="UI.ChartDefinitionType">
      <PropertyValue Property="Title" String="Sales Orders by Status" />
```

```
      <PropertyValue Property="Description" String="Number of Sales Orders by
Status" />
      <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Donut" />
      <PropertyValue Property="Dimensions">
         <Collection>
            <PropertyPath>SalesOrderOverallStatus</PropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="DimensionAttributes">
         <Collection>
            <Record Type="UI.ChartDimensionAttributeType">
               <PropertyValue Property="Dimension"
PropertyPath="SalesOrderOverallStatus" />
               <PropertyValue Property="Role"
EnumMember="UI.ChartDimensionRoleType/Category" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Measures">
         <Collection>
            <PropertyPath>NumberOfSalesOrders</PropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="MeasureAttributes">
         <Collection>
            <Record Type="UI.ChartMeasureAttributeType">
               <PropertyValue Property="Measure"
PropertyPath="NumberOfSalesOrders" />
               <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
               <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#NumberOfSalesOrders" />
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Chart Type

You can select chart data points and segments in a chart. These selections influence the other chart filters depending on the configuration of the value list annotation.

The Chart Type...	Displays...
Bar	Top or bottom three records
Line	First or last six data points
Donut	Top or bottom two records

You can enable the visual filter (donut chart type) to display an overlay message if there are measures with negative values. Set the `Analytics.AccumulativeMeasure` annotation to `false` as shown in the example. By default, the value of the `Analytics.AccumulativeMeasure` annotation is `true`.

```
<Annotations Target="SEPMRA_ALP_SO_ANA_SRV.Z_SEPMRA_C_ALP_QUARTERVHType/
DifferenceAmount">
    <Annotation Term="Analytics.Measure" Bool="true" />
    <Annotation Term="Analytics.AccumulativeMeasure" Bool="false" />
</Annotations>
```

Chart Title

ALP displays chart titles in the following order: <Measure Name> by <Dimension Name> in <Scale factor> <UoM>

- <Measure Name> indicates the measure associated with the chart. Use `sap:label()`
- <Dimension Name> indicates the dimension associated with the chart. Use `sap:label()`
- <Scale Factor> indicates the scale as specified using the `ScaleFactor` property of the `DataPoint` annotation associated with the measure displayed in the chart.

> **i Note**
>
> The scale factor in the chart and chart title are of the same scale.

Annotation: SelectionFields

Define the `SelectionFields` annotation for sorting the order of the fields displayed in the visual filters. If there are any mandatory filter fields that are not specified in the `SelectionFields`, then these fields appear first, followed by the other entries in the `SelectionFields`.

```
<Annotation Term="UI.SelectionFields">
  <Collection>
    <PropertyPath>DeliveryCalendarYear</PropertyPath>
    <PropertyPath>SoldToParty</PropertyPath>
    <PropertyPath>Product</PropertyPath>
    <PropertyPath>MainProductCategory</PropertyPath>
    <PropertyPath>DeliveryCalendarQuarter</PropertyPath>
  </Collection>
</Annotation>
```

Annotation: SelectionVariant

```
<Annotation Term="UI.SelectionVariant" Qualifier="SVForStatus">
   <Record>
      <PropertyValue Property="Parameters">
         <Collection>
            <Record Type="UI.Parameter">
               <PropertyValue Property="PropertyName"
PropertyPath="CompanyCurrency" />
               <PropertyValue Property="PropertyValue" String="EUR" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="SelectOptions">
         <Collection>
            <Record Type="UI.SelectOptionType">
               <PropertyValue Property="PropertyName"
PropertyPath="SalesOrderOverallStatus" />
               <PropertyValue Property="Ranges">
                  <Collection>
                     <Record Type="UI.SelectionRangeType">
                        <PropertyValue EnumMember="UI.SelectionRangeSignType/E"
Property="Sign" />
                        <PropertyValue EnumMember="UI.SelectionRangeOptionType/
EQ" Property="Option" />
                        <PropertyValue Property="Low" String="D" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
            <Record Type="UI.SelectOptionType">
               <PropertyValue Property="PropertyName" PropertyPath="Product" />
               <PropertyValue Property="Ranges">
                  <Collection>
```

```
                     <Record Type="UI.SelectionRangeType">
                        <PropertyValue EnumMember="UI.SelectionRangeSignType/E"
Property="Sign" />
                        <PropertyValue EnumMember="UI.SelectionRangeOptionType/
EQ" Property="Option" />
                        <PropertyValue Property="Low" String="HT-1502" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Text Arrangement

You can now change the formatting of the text that appears on the visual filter chart axis labels, legends, chart tooltips, and within the selected link (if a chart context is selected). The default view of the visual filter bar is based on the filter fields defined in the `TextArrangement` annotation and its type in the main entity set to change the text behavior like this:

Table 81:

Text Arrangement Type	Description
TextFirst	Use the visual filter to combine measures or item counts with filter values. The ChartDefault type that has the sap:text first, followed by the ID in brackets, for example, "Notebook (001)"
TextLast	ID followed by the sap:text in brackets, for example, "001 (Notebook)".
TextOnly	Shows only the sap:text, for example, "Notebook"
TextSeparate	Shows only the ID, for example, "002"

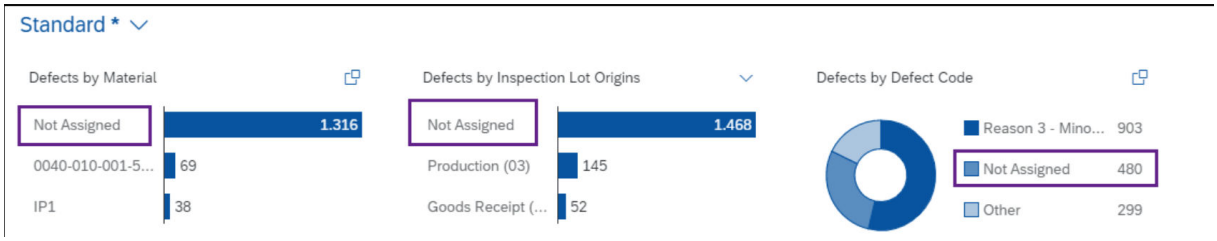
Sample Code

Text Arrangement Annotation

```
<Annotations Target="ProductType"> //Main EntitySet
  <Annotation Term="com.sap.vocabularies.UI.v1.TextArrangement"
EnumMember="com.sap.vocabularies.UI.v1.TextArrangementType/TextFirst"/>
</Annotations>
```

Display of Empty Values

The empty dimension value is displayed as *Not Assigned* in the visual filter chart. Note that this impacts the display only of visual filters but not of the value help, drop down, or compact filters. For the value help, drop down, or compact filter, it is displayed as *<empty>*.



Lazy Loading of Visual Filters

You enable lazy loading of visual filters by configuring the `lazyLoadVisualFilter` setting in the descriptor file. It is disabled by default.

If you enable lazy loading, then the batch call for loading of visual filters is deferred until the user switches to the visual filter bar.

Sample Code

Descriptor setting

```
"settings": {
              "qualifier": "MainContent",
              "defaultContentView": "charttable",
              "smartVariantManagement": true,
              "showGoButtonOnFilterBar": true,
              "multiSelect": true,
              "lazyLoadVisualFilter":true,
              "tableType": "AnalyticalTable",
              ....
}
```

Fixed Values on Visual Filters

To display default records on the visual filter chart, configure the SelectionVariant annotation with filter values (`SelectOptions` property) and link it with the `ValueList` annotation.

The in/out parameter values take precedence over the `SelectionVariant` value set, if the property is an in/out parameter and has a select option value without a chart dimension.

Table 82: In/out parameter taking precedence

Scenario	Description
Scenario 1: Annotation configuration Chart dimension = “Status”, Select Option in SV = [(“Status”, Values = “In Progress”, “New”), (“Project”, Values=”List Report”)]	• "Project" is not a chart dimension, it is an in/out parameter for the status • The chart displays records relating to `In Progress` and `New` status values for the project `List Report`.
Scenario 2: Overriding annotation configuration from filter bar Change the project value from `List Report` to `Analytical List Page` on the filter bar	Specifying **`Analytical List Page`** as a value for the project in the filter bar, re-renders charts to display records for *In Progress* and *New* status values of the Analytical List Page project. **i Note** The in/out parameter mapping values set for the other properties in the `SelectionVariant` annotation that are not part of the project field is considered as it is.

To render the visual filter with a parametrized entity set as the collection path, you need to provide parameters in the `SelectionVariant` annotation. Any values added in the smart filter bar take priority over the `SelectionVariant` annotation values.

Table 83:

Scenario	Description
Scenario 1: Annotation configuration Parameter in SelectionVariant = [(“P_DisplayCurrency”, Value=”USD”)]	The chart renders records with the currency unit USD.
Scenario 2: Overriding annotation configurations on the filter bar Change USD to EUR on the filter bar	If you specify EUR as a value for `P_DisplayCurrency` in the filter bar, the chart re-renders with records that have the currency unit EUR

Date Selection

Visual filters now support date-based, single selection fields in the Universal Time Coordinated (UTC) format. The date selection field appears on the visual filter if the annotations in the metadata file contain:

- Edm.DateTime and sap:display-format="Date"
- Edm.String and sap:semantics="yearmonthday"
- Edm.String and sap:semantics="yearmonth"
- Edm.String and sap:semantics="year"

Note

Displaying the value in the visual filter and its tooltip is impacted. Value help or the dropdown for selecting the values remains the same if sap:semantics="yearmonth" is set.

Note

You can see the date selection button on the visual filter for fields annotated with `sap:filter-restriction="single-value"`. For fields annotated with `sap:filter-restriction="multiple"`, you see the value help selection button.

Sample Code

Sample Metadata

```
<Property Name="StartDate" Type="Edm.DateTime" sap:display-format="Date"
sap:aggregation-role="dimension" sap:label="Date" sap:filter-
restriction="single-value"/>
<Property Name="StartDate" Type="Edm.String" sap:semantics="yearmonthday"
sap:aggregation-role="dimension" sap:label="Date" sap:filter-
restriction="single-value"/>
```

Unit of Measure with Multiple Units of Measure

Visual filter charts do not show up if the back end returns data with multiple units of measure (UoM). To achieve a single UoM, set the required UoM in the filter bar or change all the UoMs into one UoM in the back end.

For currency-based visual filter values, the currency value could come from another filter field in the main entity set (mapped to the value help currency field based entity set through the standard IN mapping). If this kind of mapping exists with an empty filter field in the main entity set, ALP uses the value set maintained in the *DisplayCurrency* field in the incoming navigation context for filtering the value help entity set to render the visual filter chart.

If there's a parameter in the main entity set with the exact technical name `P_DisplayCurrency`, the value is derived from the incoming `DisplayCurrency` field of the incoming navigation context.

Note

User preference for display currency type is stored in the SAP Fiori launchpad user settings. Applications pass this value to filters using the `DisplayCurrency` field.

Number Formatting

`NumberOfFractionalDigits` information can be provided in com.sap.vocabularies.UI.v1.DataPoint [page 2010] term, using the ValueFormat property. The `NumberOfFractionalDigits` property is used to determine the number of fraction digits. These are the rules:

- Decimals are not shown by default.
- You can specify 1 or 2 decimal places using the`NumberOfFractionalDigits` property in Annotations. If a value of more than 2 is provided, it is also included.

In the following example, the price property number of fractional digits provided in the OData metadata, 3 is overridden by the value 1 as provided in the com.sap.vocabularies.UI.v1.DataPoint [page 2010] ValueFormat property

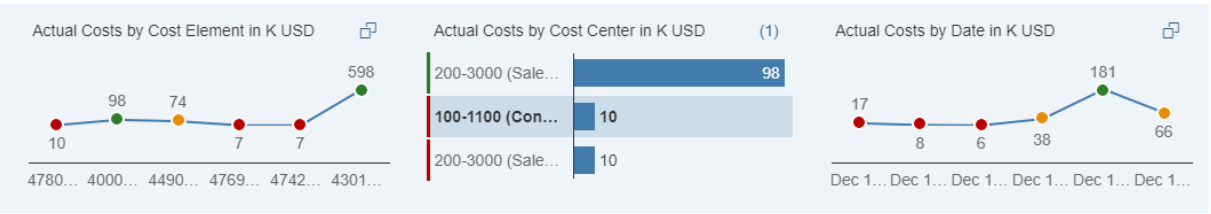
Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint" Qualifier="Price">
    <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
        <PropertyValue Property="Value" Path="Price"/>
        <PropertyValue Property="ValueFormat">
            <Record Type="com.sap.vocabularies.UI.v1.NumberFormat">
                <PropertyValue Property="NumberOfFractionalDigits" int="1"/>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

Semantic Coloring for Visual Filter Measure Values

Semantic coloring is based on the defined:

- Criticality in DataPoint annotations. The specified value, or the value returned from a path, determines the color
- CriticalityCalculation in DataPoint annotations, along with the improvement direction and various threshold values. This applies only when the criticality is not defined.



Note

No color is applied to the chart measure when

- A neutral value is returned
- Not enough threshold values are defined or when the improvement direction is missing

Grouping Visual Filter Calls (Optional)

Add a `groupId` for a set of visual filters to consolidate all group calls into one batch call. This helps you group fast-loading visual filters in one batch and group all the other slow loading visual filters into a separate batch call. This improves rendering of the fast-loading visual filters over the slow-loading visual filters.

Define the `onBeforeRebindVisualFilterExtension` extension controller method in the controller file. Ensure that the `groupId` is one of the keys in the `oContext` object which is passed to the extension as a parameter. Provide a valid string value as shown here:

Sample Code

```
onBeforeRebindVisualFilterExtension: function(sEntityType, sDimension,
sMeasure, oContext){
    'use strict';
    var Log = sap.ui.require("sap/base/Log");
    if (sDimension === "Product") {
        oContext.groupId = "Group1";
    }
    if (sDimension === "DeliveryCalendarMonth" || sDimension ===
"DeliveryCalendarQuarter") {
        oContext.groupId = "Group2";
    }
    Log.info("onBeforeRebindVisualFilterExtension called!");
}
```

Note

- The visual filter calls without a groupId are all combined in one batch.
- Visual filter calls assigned to a `groupId` reach the back end in one batch.

Guidelines

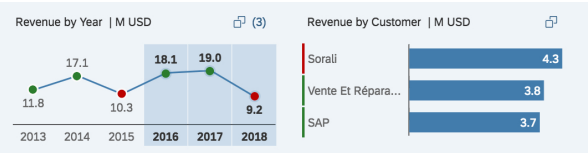
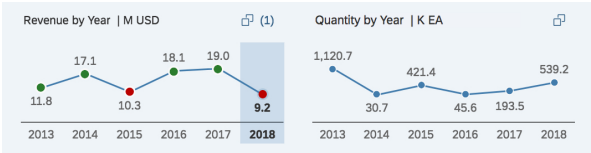
Show the filter dimension with one measure in the visual filter not with multiple measures.

Filter dimensions in the compact filters (filter bar) have exactly one representation in the visual filter bar.

Do not show the same filter dimension with two or more different measures at the same time in the visual filter bar. The example shows the filter `Dimension Year` with two different measures *Revenue* and *Quantity*. Showing the filter *dimensionYear* twice is not in sync with the compact filter, where it is shown only once. Furthermore, matching between the two filter types won't work.

If the use case requires you to show a dimension with different measures, consider using an overview page instead.

Table 84:

Do	Don't
	
For each dimension, display exactly one representation in the visual filter bar.	**Do not use the same filter dimension with different measures.**

Related Information

Configuring the Filter Area [page 1881]

Defining ValueList Annotation

Lets you map an entity to another entity that is associated with a different entity set. The value list annotation defines the relationship between filter fields of the main entity set and the fields in the value help entity set.

You can map an entity from a value help entity set to derive value help for an entity in the main entity set. Use this annotation to configure visual filters for ALP.

`Common.ValueList`

Property	Details
Label (Optional, String)	Enter label information
CollectionPath (Required, String)	Entity set for retrieving value help
SearchSupported (Required, Bool)	True or False

Parameters	Parameters to map a main entity set to a value help entity set.
	• Record Type: `Common.ValueListParameterOut` determines the
	  ○ Elements of a main entity set that make it to the filter query
	  ○ Field from a value help entity set that sets the field value in the main entity set
	• Record Type: `Common.ValueListParameterIn` determines the filtering data set from a value help entity set
	• Record Type: `Common.ValueListParameterInOut` combination of both in and out parameters.
	Parameter properties:
	• LocalDataProperty
	• ValueListProperty

PresentationVariantQualifier (Required, String)	Specify a qualifier name for the chart annotation

Example

Take a look at entity set "Z0020" (main entity set) with the three filters: region, country, and plant. The `SelectionFields` annotation has the following dimensions:

```
<Annotation Term="UI.SelectionFields">
    <Collection>
        <PropertyPath>RegionID</PropertyPath>
        <PropertyPath>CountryID</PropertyPath>
        <PropertyPath>PlantID</PropertyPath>
    </Collection>
</Annotation>
```

The value help for these dimensions display values present in the main entity set only. However, with the ValueList annotation you can retrieve records from a master list.

Consider the following table in which all of the entity sets refer to the same business object using different dimension names.

Table 85:

Business Object	Z0020 (Main Entity Set)	Z0021 (Entity Set)	Z0022 (Entity Set)	Z0023 (Entity Set)
Region	RegionID	RegionCode	RegionIdentifier	RegionNumber
Country	CountryID	CountryCode	CountryIdentifier	CountryNumber
Plant	PlantID	PlantCode	PlantIdentifier	PlantNumber

Region

`LocalDataProperty` maps to the `ValueListProperty`. The `RegionID` is mapped to the `RegionCode` and the `PlantID` is mapped to the `PlantCode`. In the following annotation example, the `PlantCode` value help entity set filters out regions based on the `PlantID` chosen from a main entity set.

Table 86:

Parameter	Z0020 (LocalDataProperty)	Z0021 (ValueListProperty)
ValueListParameterOut	RegionID	RegionCode
ValueListParameterIn	PlantID	PlantCode

The `PresentationVariantQualifier` is associated with a value help entity set. This means that values for chart properties (measure and dimension) are from a value help entity set. For example, if a `PresentationVariantQualifier` is a bar chart with the dimension `RegionCode` and the measure is `Net Sales`, then the ALP visual filter displays a bar chart that shows the top three regions.

Annotation Example

```
<Annotations Target="Z0020_CDS.Z0020Type/RegionID"> // specifies the ValueList
annotation is on the "RegionID" dimension in Z0020 entity set
<Annotation Term="Common.ValueList" Qualifier="0020_Region">
<Record>
        <PropertyValue Property="Label" String="Region"/>
        <PropertyValue Property="CollectionPath" String="Z0021"/> // Specifies
that the entity set for fetching the value help is Z0021
        <PropertyValue Bool="false" Property="SearchSupported"/>
        <PropertyValue Property="Parameters"> // In and Out parameters to map
the main entity set with the value help entity set
        <Collection>
            <Record Type="Common.ValueListParameterOut"> // OUT  parameter
determine elements in the main entity set that make it to the filter query and
also determine the entity from the value help entity set that influences their
value
                <PropertyValue Property="LocalDataProperty"
PropertyPath="RegionID"/> // LocalDataProperty points to an entity in the main
entity set - this is the entity used in forming the filter query
                <PropertyValue Property="ValueListProperty" String="RegionCode"/
> // ValueListProperty refers to the entity within the value help entity set
that provides the "value" for the entity of the main entity set which makes up
the filter query and is referred via the LocalDataProperty
            </Record>
            <Record Type="Common.ValueListParameterIn"> // IN parameters help in
filtering the data set of the value help entity set
                <PropertyValue Property="LocalDataProperty"
PropertyPath="PlantID"/>
                <PropertyValue Property="ValueListProperty" String="PlantCode"/>
            </Record>
        </Collection>
        </PropertyValue>
        <PropertyValue Property="PresentationVariantQualifier"
String="Bar_0021_Region"/> // The presence of a presentation variant means
Region comes up as a visual filter.
    </Record>
</Annotation>
</Annotations>
```

Country

The following annotation example defines the `CountryIDCountryIdentifier` entity present in the Z0022 entity set. If you query values for the value with the `RegionID` of the main entity set, then the value help for the `CountryID` field filters by the `RegionIdentifier`.

Table 87:

Parameter	Z0020 (LocalDataProperty)	Z0021 (ValueListProperty)
ValueListParameterOut	CountryID	CountryIdentifier
ValueListParameterIn	RegionID	RegionIdentifier

Annotation Example

```
<Annotations Target="Z0020_CDS.Z0020Type/CountryID">
<Annotation Term="Common.ValueList" Qualifier="0020_Country">
<Record>
        <PropertyValue Property="Label" String="Country"/>
        <PropertyValue Property="CollectionPath" String="Z0022"/> // Note that
country values come from Z0022 and is different from the value help entity set
of region (Z0021)
        <PropertyValue Bool="false" Property="SearchSupported"/>
        <PropertyValue Property="Parameters">
        <Collection>
            <Record Type="Common.ValueListParameterOut">
                <PropertyValue Property="LocalDataProperty"
PropertyPath="CountryID"/>
                <PropertyValue Property="ValueListProperty"
String="CountryIdentifier"/>
            </Record>
            <Record Type="Common.ValueListParameterIn">
                <PropertyValue Property="LocalDataProperty"
PropertyPath="RegionID"/>
                <PropertyValue Property="ValueListProperty"
String="RegionIdentifier"/>
            </Record>
        </Collection>
        </PropertyValue>
        <PropertyValue Property="PresentationVariantQualifier"
String="Country_Chart"/>
    </Record>
</Annotation>
</Annotations>
```

Plant

The following example annotation configures the `PlantID` value with the with the Z0023 value help entity set. Selecting a value for `PlantNumber` maps to `PlantID` of the main entity set. You can also configure multiple `Common.ValueListParameterIn` parameters to filter both region and country in the filter query.

Table 88:

Parameter	Z0020 (LocalDataProperty)	Z0021 (ValueListProperty)
ValueListParameterOut	PlantID	PlantNumber
ValueListParameterIn	RegionID	RegionNumber
ValueListParameterIn	CountryID	CountryNumber

Annotation Example

```
<Annotations Target="Z0020_CDS.Z0020Type/PlantID">
<Annotation Term="Common.ValueList" Qualifier="0020_Plant">
```

```
<Record>
        <PropertyValue Property="Label" String="Plant"/>
        <PropertyValue Property="CollectionPath" String="Z0023"/>
        <PropertyValue Bool="false" Property="SearchSupported"/>
        <PropertyValue Property="Parameters">
        <Collection>
            <Record Type="Common.ValueListParameterOut">
                <PropertyValue Property="LocalDataProperty"
PropertyPath="PlantID"/>
                <PropertyValue Property="ValueListProperty"
String="PlantNumber"/>
            </Record>
            <Record Type="Common.ValueListParameterIn">
                <PropertyValue Property="LocalDataProperty"
PropertyPath="RegionID"/>
                <PropertyValue Property="ValueListProperty"
String="RegionNumber"/>
            </Record>
            <Record Type="Common.ValueListParameterIn">
                <PropertyValue Property="LocalDataProperty"
PropertyPath="CountryID"/>
                <PropertyValue Property="ValueListProperty"
String="CountryNumber"/>
            </Record>
        </Collection>
        </PropertyValue>
        <PropertyValue Property="PresentationVariantQualifier"
String="Plant_Chart"/>
    </Record>
</Annotation>
</Annotations>
```

All three visual filters appear in the ALP header area if you have completed the chart configuration for the respective PresentationVariantQualifier.

If your configuration has *Net Sales* as a common measure in all three entity sets, then the first chart displays the top three regions (each bar = `RegionCode` from Z0021).- The second chart displays the top three countries (each bar = `CountryIdentifier` from Z0022). The last chart displays the top three plants (each bar = `PlantNumber` from Z0023) as measured against Net Sales.

Scenario 1: Selection in the Region Chart

If you select a bar in the region chart with this configuration, it then passes the `RegionID`=<chosen value from `RegionCode` of Z0021> to the filter query in the main entity set. As a result, ALP refreshes the data in the main content area and the other two charts (A `RegionID` is an IN parameter for the ValueList annotation of the `CountryID` and `PlantID`). The top three countries and plants are displayed for the chosen region.

Scenario 2: Selection in the Region Chart and Country Chart

After selecting the region chart, if you select a bar in the country chart then it passes the `CountryID`=<chosen value from `CountryIdentifier` of Z0022> to the filter query in the main entity set. As a result, the ALP refreshes data in the main content area and the third chart (A `CountryID` is an IN parameter only for "PlantID"). Plants within the chosen region and country are displayed.

Scenario 3: Selection in the Plant Chart

`PlantID` is an IN parameter used only for the `RegionID`. This means that along with the main content area only the first chart is refreshed and displays the top three countries for the chosen plant.

> **i Note**
>
> The IN mapping defined for a field is ignored when the same field has a `ValueList` annotation defined for it. For example, when `PlantID` is the IN mapping for the `ValueList` of the `PlantID`. However, the existing mapping values of the `PlantID` in the filter query would have no impact on a visual filter. The ignored value is considered for the main filter query and shows up in the visual filter *Selected* button.

Related Information

Configuring the Filter Area [page 1881]
Choosing Filter Modes [page 1880]
Compact Filter Setup [page 1884]

Configuring the Content Area

Visualize data from the main entity set and seamlessly navigate to an application. Define a valid `chart` or `LineItem` annotation to render content for the chart area and table area.

Context

Analytical list page determines the content of the main area as follows:

- If you have set the value for a qualifier in descriptor file, ALP looks for the `SelectionPresentationVariant` with the same qualifier. If the `SelectionPresentationVariant` is found, then the associated `PresentationVariant` is used.
  - If the `SelectionPresentationVariant` is not found, then the application looks for a `PresentationVariant` with the same qualifier.
  - If `PresentationVariant` is not found, then an error message is displayed.
- If you have not set the value for a qualifier in the descriptor file, ALP looks for a default `SelectionPresentationVariant` and uses the associated `PresentationVariant`
  - If the default `SelectionPresentationVariant` is not found, ALP looks for a default `PresentationVariant`
  - If default `PresentationVariant` is also not found, ALP looks for a default `chart` and `LineItem` annotations.
- If you have configured the content tile in the descriptor file, the `FilterableKPI` view appears along with the segmented buttons (Hybrid/Chart/Table) and the content area tile.

You can choose to view the main entity set data in the following view modes:

- Table only
- Chart only
- Hybrid (chart and table)

Configure `contentTitle` property in the descriptor configuration file, to add title for the content area.

Related Information

Analytical List Page [page 1868]
Table-Only View [page 1902]
Chart-Only View [page 1910]
Hybrid View [page 1914]

Table-Only View

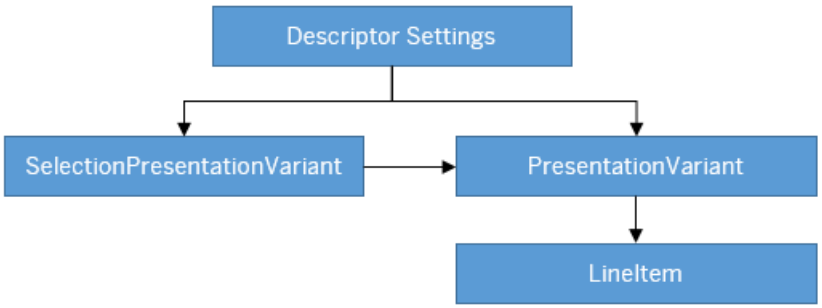
Displays transactional data in a tabular format. Use *table-only* mode to view individual records within the transactional data.

ALP provides the following table types:

- Analytical (sap.ui.table)
- Grid
- Responsive (sap.m.table)

Configure the `tableType` setting in the descriptor file to select any of the table types mentioned above.

> **i Note**
>
> Use the analytical table type with entity sets containing `sap:semantics` as aggregates.



- #unique_725/unique_725_Connect_42_subsection-im1 [page 1903]
- #unique_725/unique_725_Connect_42_subsection-im2 [page 1903]
- #unique_725/unique_725_Connect_42_subsection-im3 [page 1903]
- #unique_725/unique_725_Connect_42_subsection-im4 [page 1903]

Hover over each action for a description. Click the action for more information.

Descriptor Settings: Table-only view

Configuration Sample:

```
"sap.ui.generic.app":{
   "pages":[
      {
         "entitySet":"SEPMRA_C_ALP_SlsOrdItemCube",
         "component":{
            "name":"sap.suite.ui.generic.template.AnalyticalListPage",
            "list":true,
            "settings":{
                  "qualifier":"DefaultVariant",
                  }
            }
      }
},
```

Annotation: SelectionPresentationVariant with Qualifier="DefaultVariant"

Configuration Sample:

```
<Annotation Term="UI.SelectionPresentationVariant" Qualifier="DefaultVariant">
  <Record>
    <PropertyValue Property="Text" String="Product Financial Analysis"/>
    <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#DefaultSelectionVariant"/>
    <PropertyValue Property="PresentationVariant"
Path="@UI.PresentationVariant#DefaultPresentationVariant"/>
  </Record>
</Annotation>
```

Annotation: Presentation Variant

```
<Annotation Term="UI.PresentationVariant" Qualifier="DefaultPresentationVariant">
  <Record>
    <PropertyValue Property="Text" String="Default"/>
    <PropertyValue Property="SortOrder">
      <Collection>
        <Record Type="Common.SortOrderType">
          <PropertyValue Property="Property" PropertyPath="NetAmount"/>
          <PropertyValue Property="Descending" Bool="true"/>
        </Record>
      </Collection>
    </PropertyValue>
    <PropertyValue Property="Visualizations">
      <Collection>
         <AnnotationPath>@UI.LineItem#Default</AnnotationPath>
      </Collection>
   </PropertyValue>
  </Record>
</Annotation>Presentation Variant annotation example
```

Annotation: LineItem

```
<Annotation Term="UI.LineItem" Qualifier="Default">
  <Collection>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="DeliveryCalendarYear"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="DeliveryCalendarMonth"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
```

```
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="SalesOrder"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataFieldForIntentBasedNavigation">
      <PropertyValue Property="Label" String="Manage Sales Order"/>
      <PropertyValue Property="SemanticObject" String="EPMSalesOrder"/>
      <PropertyValue Property="Action" String="manage_st"/>
      <PropertyValue Property="RequiresContext" Bool="false"/>
        <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataFieldWithIntentBasedNavigation">
      <PropertyValue Property="SemanticObject" String="EPMSalesOrder"/>
      <PropertyValue Property="Action" String="manage_st"/>
      <PropertyValue Property="Value" Path="SalesOrder"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Label" String="Item"/>
      <PropertyValue Property="Value" Path="SalesOrderItem"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="Product"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataFieldWithIntentBasedNavigation">
      <PropertyValue Property="SemanticObject" String="EPMProduct"/>
      <PropertyValue Property="Action" String="manage_st"/>
      <PropertyValue Property="Value" Path="Product"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="ProductName"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="MainProductCategory"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Label" String="Customer"/>
      <PropertyValue Property="Value" Path="SoldToPartyCompanyName"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="Quantity"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="NetAmount"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
</Collection>
</Annotation>
```

The data set helps ALP decide which table to display when it is not configured in descriptor settings. If the data set

- Comes from an analytical OData service, then ALP renders data through an analytical table. This table provides grouping and aggregation of data at different levels.
- Doesn't come from an analytical OData service and grid table usage is enabled in your descriptor setting (`tabletype=GridTable`Presentation V), then ALP brings up the grid table (sap.ui.table) for the tabular display.

In all other cases, ALP uses the responsive table (sap.m.table) to render the data.

> **i Note**
>
> By default, ALP displays tables in condensed mode. You can modify the descriptor setting to use compact mode.

You can perform several table level operations using the table toolbar. The settings feature allows you to

- Add or remove dimensions and measures bound to a table
- Perform sort operations on a table
- Choose dimensions to group a table as needed

> **i Note**
>
> You can adjust the width of columns by modifying the `MaxLength` metadata for texts strings and `Precision` metadata for decimals.

Configure the descriptor file to provide table navigation. Choose *Details* to navigate from a table row to another application.

> **i Note**
>
> The selected table record context is passed to the target application as a navigation context. In case of:
>
> - Inner app navigation: The selected table record context is passed to the target application
> - External navigation: The selected table record and filter context is passed to the target application
> - To remove values from the selection variant passed in the navigation context, use the `adaptNavigationParameterExtension` breakout.

```
"sap.ui.generic.app": {
    "_version": "1.3.0",
    "pages": {
"page-1": {
            "entitySet": ….
            "component": {
                …
                …
            } // End of component
"pages": [{ //ALP does an inner app navigation to the smart template specified
below
                "entitySet": "<<ZCOSTCENTERCOSTSQUERY0020>>", //Ensure that the
Entity set specified here is the same entity set used in the main content area.
                "component": {
                    "name": "<<sap.suite.ui.generic.template.ObjectPage>>"
                }
"navigation": { //Optional (If specified ALP navigates to the target Application
specified below through external app navigation)
                "display": {
                                "path": "sap.app.crossNavigation.outbounds",
                                "target": "<<NavigateToCTRItem>>"
                            }
              }
            }]
        } // End of page-1
}
}
```

Filter Option

You can enable filtering in the smart table area by configuring the `enableTableFilterInPageVariant` setting in the descriptor file. By default, filtering is:

- Disabled in the page-level variant.
- Enabled in the control-level variant and cannot be disabled.

> **i Note**
>
> In SmartTable, you can view the filter option after selecting the column header or from the personalization settings. Whereas, in responsive tables, the filter option is available only from table personalization settings.

```
"settings": {
              "qualifier": "MainContent",
              "defaultContentView": "charttable",
              "smartVariantManagement": true,
              "showGoButtonOnFilterBar": true,
              "multiSelect": true,
              "enableTableFilterInPageVariant":true,
              "tableType": "AnalyticalTable",
              ....
}
```

Custom Navigation

The `DataFieldWithIntentBasedNavigation` record type within the `LineItem` annotation lets you include a navigation option in the smart table. Defining this annotation adds a new column to the table and provides a link to the target application based on the properties (semantic object and semantic action) configured in the annotation.

> **Sample Code**
>
> Annotation sample

```
<Annotation Term="UI.LineItem" Qualifier="ActualCosts">
    <Collection>
        <Record Type="UI.DataFieldWithIntentBasedNavigation">
            <PropertyValue Property="Label" String="Cost Center (IBN)"/>
            <PropertyValue Property="Value" Path="CostCenter" />
            <PropertyValue Property="SemanticObject" String="alpwp"/>
            <PropertyValue Property="Action" String="display"/>
        </Record>
    </Collection>
</Annotation>
```

The `DataFieldWithIntentBasedNavigation` record type within the `LineItem` annotation lets you include a navigation option within the smart table toolbar.

```
<Record Type="UI.DataFieldForIntentBasedNavigation">
<PropertyValue Property="Label" String="Manage"/>
<PropertyValue Property="SemanticObject" String="CompanyCode"/>
<PropertyValue Property="Action" String="display"/>
```

```
<PropertyValue Property="Inline" Bool="true"/>
</Record>
```

> **i Note**
>
> The selection column in the table is disabled if all of the following conditions are met:
>
> - There are no `DataFieldForAction` buttons defined by the application.
> - There are no `DataFieldForIntentBasedNavigation` buttons with `RequiresContext` set to true.
> - There are no manifest based custom actions with `RequiresSelection` set to true.

Actions by Annotation

Define custom action buttons on the table toolbar using the `DataFieldForAction` property associated to `LineItem` annotation.

> **i Note**
>
> To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

Inline Button

Include a custom buttons in a table by defining the inline property in the `DataFieldForAction` annotation. This button appears in the table column and the position of the column depends on its position in the `LineItem` annotation.

```
<Record Type="UI.DataFieldForIntentBasedNavigation">
<PropertyValue Property="Label" String="Manage"/>
<PropertyValue Property="SemanticObject" String="CompanyCode"/>
<PropertyValue Property="Action" String="display"/>
<PropertyValue Property="Inline" Bool="true"/>
</Record>
```

Determining Button

At the click of a button, the associated:

- `DataFieldForAction` annotation handles the action as defined. The user application must ensure that this points to the fully qualified action or function.
- `DataFieldForIntentBasedNavigation` annotation launches the SAP Fiori applications as specified in the action. If the action is configured to accept context, then the selected table and filter context is passed to the target application through the navigation context.

You can configure the table toolbar to contain buttons defined with the annotations `com.sap.vocabularies.UI.v1.DataFieldForAction` and `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`. The determining buttons for tables are in the footer bar, and chart buttons appear before determining buttons.

```
<PropertyValue Property="Actions">
   <Collection>
      <Record Type="UI.DataFieldForAction">
```

```
         <PropertyValue Property="Label" String="Copy"/>
         <PropertyValue Property="Action"
String="CZ_EASILINEITEMS_SADL_CDS.CZ_EASILINEITEMS_SADLType/Copy"/>
         <PropertyValue Property="InvocationGrouping"
EnumMember="UI.OperationGroupingType/Isolated"/>
         <PropertyValue Property="Determining" Bool="true"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
         <PropertyValue Property="Label" String="Manage Products (STTA)"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct"/>
         <PropertyValue Property="Action" String="manage_st"/>
         <PropertyValue Property="Determining" Bool="true"/>
      </Record>
   </Collection>
</PropertyValue>
```

```
<PropertyValue Property="Actions">
   <Collection>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy"/>
         <PropertyValue Property="Action"
String="CZ_EASILINEITEMS_SADL_CDS.CZ_EASILINEITEMS_SADLType/Copy"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
         <PropertyValue Property="Label" String="Manage Products (STTA)"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct"/>
         <PropertyValue Property="Action" String="manage_st"/>
         <PropertyValue Property="RequiresContext" />
      </Record>
   </Collection>
</PropertyValue>
```

Quick View in Table

You can configure the smart table column cells to display additional information in a quick view card. For more information, see Enabling Quick Views for Smart Link Navigation [page 1567]

Semantic Highlighting of Rows

Use a property or a dimension that represents the criticality of the record (row) in the back end to add record-level semantic coloring in the `SmartTable`.

To enable this feature, add the `criticalityProperty` to the `LineItem` annotation of the smartTable.

Sample Code

```
<Annotation Term="UI.LineItem">
  <Annotation Term="UI.Criticality"
Path="Element_transporting_criticality_of_complete_LineItem" />         //
LineItem Criticality annotation
  <Collection>
    <Record Type="UI.DataField">
      ...
    </Record>
  </Collection>
```

```
</Annotation>
```

The coloring of the criticality is defined as follows:

- 0 - None (No Color) - v1.CriticalityType/Neutral
- 1 - Error (Semantic Negative) - v1.CriticalityType/Negative
- 2 - Warning (Semantic Critical) - v1.CriticalityType/Critical
- 3 - Success (Semantic Positive) - v1.CriticalityType/Positive

The semantic coloring of table rows is supported for analytical tables, responsive tables, and grid tables.

> **i Note**
>
> For analytical tables, the field that returns the criticality must be a property and not a dimension.

Semantically Connected Fields

You can configure the table columns to view multiple field types. To enable this feature, configure `UI.LineItem`, `UI.DataFieldForAnnotation`, and `UI.FieldGroup` annotations as shown here:

> **‘≡› Sample Code**
>
> LineItem annotation

```
<Annotation Term="UI.LineItem">
    <Collection>
        ……
        ……
        <Record Type="UI.DataFieldForAnnotation">
            <PropertyValue Property="Target"
AnnotationPath="@UI.FieldGroup#ActionGroupTest" />
            <PropertyValue Property="Label" String="Multiple Actions"/>
        </Record>
        ……
        ……
    </Collection>
</Annotation>
```

> **‘≡› Sample Code**
>
> FieldGroup Annotation

```
<Annotation Term="UI.FieldGroup" Qualifier="ActionGroupTest">
    <Record Type="UI.FieldGroupType">
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="UI.DataFieldForAction">   // Can also be
DataField or ImageURL or DataFieldForIntentBasedNavigation or
DataFieldWithIntentBasedNavigation here
                    <PropertyValue Property="Action"
String="ZEPM_C_SALESORDERITEMQUERY_CDS.ZEPM_C_SALESORDERITEMQUERYResult/
update"/>
                    <PropertyValue Property="Label" String="Update Order"/>
                    <PropertyValue Property="Inline" Bool="true"/>
                </Record>
                <Record Type="UI.DataFieldForAction">
```

```
                    <PropertyValue Property="Action"
String="ZEPM_C_SALESORDERITEMQUERY_CDS.ZEPM_C_SALESORDERITEMQUERYResult/
delete"/>
                    <PropertyValue Property="Label" String="Delete Order"/>
                    <PropertyValue Property="Inline" Bool="true"/>
                </Record>
            </Collection>
    </Record>
</Annotation>
```

i Note

- You can see fields belonging to the same `entityType` in the table personalization.
- Grid and analytical tables do not support semantically connected fields for the annotation properties `ImageURL` and `UI.Chart`.

Related Information

Configuring the Content Area [page 1901]

Chart-Only View

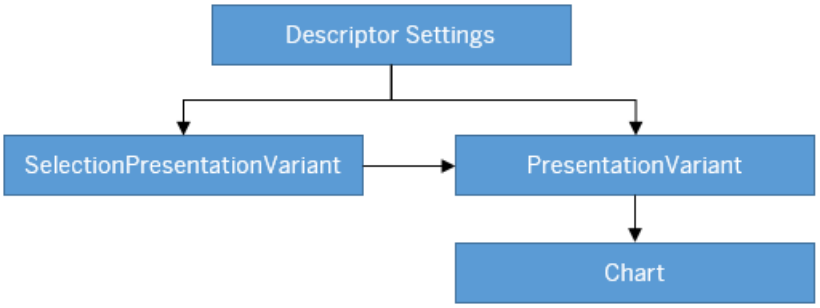
Uses a smart chart that provides visual representation of analytical data.

By selecting the chart context, users can navigate to other applications. A smart chart provides:

- Semantic object-based navigation
- Easy changing of chart types
- Customization of chart settings

The chart-only view provides a way to analyze data from different perspectives, step by step, to investigate a root causeby drilling down without direct access to transactional content. The smart chart control is used to provide the chart visualization.

You use this features to deal with analytical data that can be visually represented using charts, without needing to link them to the transactional data set.



- #unique_731/unique_731_Connect_42_subsection-im1 [page 1911]
- #unique_731/unique_731_Connect_42_subsection-im2 [page 1911]
- #unique_731/unique_731_Connect_42_subsection-im3 [page 1911]
- #unique_731/unique_731_Connect_42_subsection-im4 [page 1912]

Hover over each action for a description. Click the action for more information.

Descriptor Settings: Table-only view

Configuration Sample:

```
"sap.ui.generic.app":{
   "pages":[
      {
         "entitySet":"SEPMRA_C_ALP_SlsOrdItemCube",
         "component":{
            "name":"sap.suite.ui.generic.template.AnalyticalListPage",
            "list":true,
            "settings":{
                  "qualifier":"DefaultVariant",
                  }
            }
      }
},
```

Annotation: SelectionPresentationVariant with Qualifier="DefaultVariant"

Configuration Sample:

```
<Annotation Term="UI.SelectionPresentationVariant" Qualifier="DefaultVariant">
  <Record>
    <PropertyValue Property="Text" String="Product Financial Analysis"/>
    <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#DefaultSelectionVariant"/>
    <PropertyValue Property="PresentationVariant"
Path="@UI.PresentationVariant#DefaultPresentationVariant"/>
  </Record>
</Annotation>
```

Annotation: Presentation Variant

```
<Annotation Term="UI.PresentationVariant" Qualifier="DefaultPresentationVariant">
  <Record>
    <PropertyValue Property="Text" String="Default"/>
    <PropertyValue Property="SortOrder">
      <Collection>
        <Record Type="Common.SortOrderType">
          <PropertyValue Property="Property" PropertyPath="NetAmount"/>
          <PropertyValue Property="Descending" Bool="true"/>
        </Record>
      </Collection>
    </PropertyValue>
    <PropertyValue Property="Visualizations">
      <Collection>
         <AnnotationPath>@UI.Chart#Default</AnnotationPath>
      </Collection>
   </PropertyValue>
  </Record>
</Annotation>
```

Use the app-descriptor setting `chartPresentationQualifier` to render a chart based on a specific `PresentationVariant` annotation configuration.

```
"settings": {
"chartPresentationQualifier": "qualifier"}
```

Annotation: Chart

```
<Annotation Term="UI.Chart" Qualifier="Default">
  <Record Type="UI.ChartDefinitionType">
    <PropertyValue Property="Title" String="Revenue by Customer"/>
    <PropertyValue Property="Description" String="Net Revenue by Customer"/>
    <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Column"/>
    <PropertyValue Property="Dimensions">
      <Collection>
        <PropertyPath>SoldToParty</PropertyPath>
        <PropertyPath>DeliveryCalendarYear</PropertyPath>
      </Collection>
     </PropertyValue>
     <PropertyValue Property="DimensionAttributes">
       <Collection>
          <Record Type="UI.ChartDimensionAttributeType">
             <PropertyValue Property="Dimension" PropertyPath="SoldToParty"/>
             <PropertyValue Property="Role"
EnumMember="UI.ChartDimensionRoleType/Category"/>
          </Record>
          <Record Type="UI.ChartDimensionAttributeType">
             <PropertyValue Property="Dimension"
PropertyPath="DeliveryCalendarYear"/>
             <PropertyValue Property="Role"
EnumMember="UI.ChartDimensionRoleType/Series"/>
          </Record>
       </Collection>
     </PropertyValue>
     <PropertyValue Property="Measures">
       <Collection>
         <PropertyPath>NetAmount</PropertyPath>
       </Collection>
     </PropertyValue>
     <PropertyValue Property="MeasureAttributes">
       <Collection>
          <Record Type="UI.ChartMeasureAttributeType">
             <PropertyValue Property="Measure" PropertyPath="NetAmount"/>
             <PropertyValue Property="Role" EnumMember="UI.ChartMeasureRoleType/
Axis1"/>
             <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#NetAmount"/>
          </Record>
       </Collection>
     </PropertyValue>
   </Record>
</Annotation>
```

Semantic Navigation

If you select a data point or segment from a chart, the smart chart checks the annotation of any semantic object definition for these dimensions and uses it as a base to render the navigation links. For example,

Sample Code

Cost Center dimension

```
<Annotations xmlns=http://docs.oasis-open.org/odata/ns/edm
Target="Emp_Line_Item.Item/Cost_Center">
    <Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"
String="CostCenter"/>
</Annotations>
```

In the preceding example, the smart charts retrieve all the navigation parameters for which you have an authorization and that are defined for the `CostCenter` semantic object. The selected chart and filter context is passed to the target application through the navigation context.

Choose *Details* on the toolbar, to view navigation links that define actions associated with semantic objects.

Chart Operations

You can perform several chart level operations on the toolbar. The *Settings* option on the chart toolbar enables you to include additional filters on the chart, or to change:

- Dimensions and measures bound to a chart
- Roles for dimensions and measures
- Sort order in charts

The *Drilldown* option lets you change the chart grouping dimension. If you have already made a chart selection before selecting the *Drilldown* option and changing the grouping dimension, analytical list page uses the earlier chart dimension selection as a filter context for drilldown.

Actions by Annotation

Any action that you define in the `Actions` property of the chart annotation (`DataFieldForAction` or `DataFieldForIntentBasedNavigation`), is displayed as an additional button on the chart toolbar (when `determining=false`).

Note

To correctly integrate your app extension coding with SAP Fiori elements, use only the extensionAPI of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

Determining Button

You can configure the smart chart toolbar to contain action buttons defined with annotations `com.sap.vocabularies.UI.v1.DataFieldForAction` and `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`. Choosing `DataFieldForAction` executes the back-end function import as identified by the `Action` property. When you choose `DataFieldForIntentBasedNavigation`, the appropriate SAP Fiori app is launched. The

determining buttons for the chart are found in the footer bar and the chart buttons are ordered before the table buttons.

```
<PropertyValue Property="Actions">
   <Collection>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy"/>
         <PropertyValue Property="Action"
String="CZ_EASILINEITEMS_SADL_CDS.CZ_EASILINEITEMS_SADLType/Copy"/>
         <PropertyValue Property="InvocationGrouping"
EnumMember="UI.OperationGroupingType/Isolated"/>
         <PropertyValue Property="Determining" Bool="true"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
         <PropertyValue Property="Label" String="Manage Products (STTA)"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct"/>
         <PropertyValue Property="Action" String="manage_st"/>
         <PropertyValue Property="Determining" Bool="true"/>
      </Record>
   </Collection>
</PropertyValue>
```

With the click of a button, the associated:

- `DataFieldForAction` annotation handles the action as defined. The user application has to ensure that this points to the fully qualified action or function.
- `DataFieldForIntentBasedNavigation` annotation launches the SAP Fiori applications as specified in the action. If the action is configured to accept context, then the selected chart and filter context is passed to the target application through the navigation context.

```
<PropertyValue Property="Actions">
   <Collection>
      <Record Type="UI.DataFieldForAction">
         <PropertyValue Property="Label" String="Copy"/>
         <PropertyValue Property="Action"
String="CZ_EASILINEITEMS_SADL_CDS.CZ_EASILINEITEMS_SADLType/Copy"/>
      </Record>
      <Record Type="UI.DataFieldForIntentBasedNavigation">
         <PropertyValue Property="Label" String="Manage Products (STTA)"/>
         <PropertyValue Property="SemanticObject" String="EPMProduct"/>
         <PropertyValue Property="Action" String="manage_st"/>
      </Record>
   </Collection>
</PropertyValue>
```

Related Information

Configuring the Content Area [page 1901]

Hybrid View

The hybrid view is the default data display mode in the ALP content area. It lets you view both analytical data (chart format) and transactional data (table format).

Users can interact with both the chart and the table. The initial view of the chart, visualizes the most important aspects of the whole data set. Selecting a dimension within a chart area automatically filters all relevant

information in the table area. For example, if a chart selection is Country=ABC, then all records associated with this country selection are filtered in the table.

The *Auto-hide* feature is displayed in the hybrid (chart and table) view. When the feature is turned off, the table records that are not affected by the chart selection are also displayed in the table. If the feature is turned on, then only the records that are affected by the chart selection (for example, only the highlighted records) are displayed in the table, In this case chart selection acts as an additional filter on the table. The table records matching the selected chart context is highlighted.

> **i** Note
>
> Ensure that chart context dimensions are set as visible columns in the table for optimal accuracy of highlights.

> **→** Tip
>
> Note that the hybrid view is available in laptops with a screen height that is larger than 900px. This mode is not available for tabs or mobiles.

Related Information

Configuring the Content Area [page 1901]

Configuring Analytical List Page App Extensions

This section provides some of the advance configurations and extensions for your application.

For extensions common to all floorplans of SAP Fiori elements, see Extending SAP Fiori Elements-Based Apps [page 1585]

Related Information

Analytical List Page [page 1868]
Refresh API [page 1916]
Smart Table Extensions [page 1916]
Chart Extensions [page 1919]
Creating Custom Filter [page 1920]
Defining Custom Actions [page 1923]

Refresh API

Use the `refreshBinding` function (in `Component.js`) to refresh the header, filter, and content area elements.

This helps you to reload the data for all components configured in ALP without triggering the browser refresh. The selections in the filter bar (compact or visual filter) and the personalization settings (sorting, grouping, and so on) are retained even after the refresh.

> **i** Note
>
> The main chart selection is lost during the refresh. This may result in the change of table data when AutoHide is switched on for filter mode.

> **‹≡›** Sample Code
>
> Refreshing by using the controller extensions is defined by the application

```
//By calling refreshBinding function directly .
       var component = this.getOwnerComponent();
       component.refreshBinding();
```

Related Information

Configuring Analytical List Page App Extensions [page 1915]

Smart Table Extensions

Define custom actions for tables by configuring the descriptor and annotation files.

OnBeforeRebindTableExtension

Use onBeforeRebindTableExtension to define app-specific logic before the table is rendered. This allows you to bind additional parameters from custom filters to the table query.

In the descriptor file, define the sap.ui.controllerExtensions and extend ALP controller `sap.suite.ui.generic.template.AnalyticalListPage.view.AnalyticalListPage` as shown in this sample.

```
"sap.ui5": {
    "_version": "1.1.0",
    …
    …
"extends": {
        "extensions": {
```

```
        "sap.ui.controllerExtensions": {

"sap.suite.ui.generic.template.AnalyticalListPage.controller.AnalyticalListPage":
 {
                "controllerName":
"sap.poc.ftu.apps.alr.ext.controller.AnalyticalListPageExt",
                "sap.ui.generic.app": {
                    …
                    …
```

Define the extended behavior in the extension method. This enables ALP to pass the event object for extracting controls and other details.

```
sap.ui.controller("sap.poc.ftu.apps.alr.ext.controller.AnalyticalListPageExt",{
    onBeforeRebindTableExtension: function(oEvent) {
        alert('onBeforeRebindTableExtension called!');
},
}
```

Table Column Extensions

In the descriptor file, define `sap.ui.viewExtensions` to extend ALP view and to create custom columns. Configure `sap.suite.ui.generic.template.AnalyticalListPage.view.AnalyticalListPage` to extend the ALP view and `AnalyticalTableColumnsExtension` to extended columns and its fragments (define within the app namespace) .

```
"sap.ui5": {
    "_version": "1.1.0",
    "extends": {
        "extensions": {
            "sap.ui.viewExtensions": {

"sap.suite.ui.generic.template.AnalyticalListPage.view.AnalyticalListPage": {
                          "AnalyticalTableColumnsExtension|
CZ_EASILINEITEMS_SADL": { // The second part after the "|" operator is the
entity set
                              "className": "sap.ui.core.Fragment",
                              "fragmentName":
"sap.poc.ftu.apps.alr.ext.fragment.CustomColumn", // namespace of the
application having the custom fragment name
                              "type": "XML"
                          }
                      }
                } // End of viewExtensions
        }  // End of extensions
    },
        ....
        ....
}
```

Fragment extension code sample:

```
<core:FragmentDefinition xmlns:core="sap.ui.core"
          xmlns="sap.m" xmlns:table="sap.ui.table">
    <table:AnalyticalColumn width="150px" autoResizable="true">
        <Label text="My Extended Column"/> // Column header name
        <table:template>
            <Label text="data"/> // value in each cell of the table
        </table:template>
```

```
            <table:customData>
                <core:CustomData key="p13nData"
                      value='\{"columnKey": "Test", "columnIndex" : "1"}' />
            </table:customData>
    </table:AnalyticalColumn>
</core:FragmentDefinition>
```

Navigation Extension for Rows

ALP allows applications to do conditional navigation using the `onListNavigationExtension` API. ALP allows you to decide on the target application based on the context available in the selected table record. You can define different targets for each row in the table. You can also retain the standard ALP navigation mechanism (inner app navigation to object page or navigation to an external application) while enabling app-specific custom navigation to selected rows.

> **i** Note
>
> We recommend you use the `navigateExternal()` API as shown below to perform any external navigation.

> **Sample Code**

```
onListNavigationExtension: function(oEvent) {
    var oNavigationController = this.extensionAPI.getNavigationController();
    var oBindingContext = oEvent.getSource().getBindingContext();
    var oObject = oBindingContext.getObject();
    // for notebooks we trigger external navigation for all others we use
internal navigation
    if (oObject.CostCenter == "300-1000") {
        oNavigationController.navigateExternal("ActualCostsKPIDetails");
    } else {
        // return false to trigger the default internal navigation
        return false;
    }
    // return true is necessary to prevent further default navigation
    return true;
 }
```

Related Information

Configuring Analytical List Page App Extensions [page 1915]
Defining Custom Actions [page 1923]

Chart Extensions

Define custom actions for a chart by configuring the descriptor and annotation files.

onBeforeRebindChartExtension

Use the `onBeforeRebindChartExtension` chart support controller extension to define application-specific actions. This allows you to bind additional parameters, such as custom filters or chart queries.

In the app-descriptor file, define the `sap.ui.controllerExtensions` and extend the ALP controller `sap.suite.ui.generic.template.AnalyticalListPage.view.AnalyticalListPage` as shown here:

```
"sap.ui5": {
    "_version": "1.1.0",
    …
    …
"extends": {
        "extensions": {
        "sap.ui.controllerExtensions": {

"sap.suite.ui.generic.template.AnalyticalListPage.view.AnalyticalListPage": {
                "controllerName":
"sap.poc.ftu.apps.alr.ext.controller.AnalyticalListPageExt",
                "sap.ui.generic.app": {
                    …
                    …
```

Define the extended behavior in the extension method. This enables ALP to pass the event object for extracting controls and other details.

```
sap.ui.controller("sap.poc.ftu.apps.alr.ext.controller.AnalyticalListPageExt",{
    onBeforeRebindChartExtension: function(oEvent) {
        alert('onBeforeRebindChartExtension called!');
},
}
```

Related Information

Configuring Analytical List Page App Extensions [page 1915]

Creating Custom Filter

Define custom filters for compact filters and KPI tags.

Compact Filters

Define custom filter view fragments, use view extensions and define extended fragments and controllers in the application namespace as shown here:

```
"sap.ui5": {
    "_version": "1.1.0",
    "extends": {
        "extensions": {
         "sap.ui.controllerExtensions": { // Controller extension

"sap.suite.ui.generic.template.AnalyticalListPage.controller.AnalyticalListPage":
 { // ALP app view to be extended with controller
               "controllerName":
"analytics2.alr.ext.controller.CustomFiltersController", // extended Controller
declared using namespace
               ... // Other custom controllers
            ...
          } // End of ALP controller extensions
         }, // End of controller extensions
         "sap.ui.viewExtensions": { // View Extension

"sap.suite.ui.generic.template.AnalyticalListPage.view.AnalyticalListPage": { //
ALP app view to be extended with filter fragment
                   "SmartFilterBarControlConfigurationExtension|
CZ_EASILINEITEMS_SADL": { // <Filter Bar Extension>|<Entity Set>
            "className": "sap.ui.core.Fragment",
            "fragmentName": "analytics2.alr.ext.fragments.CustomFilters", //
extended Fragment declared using namespace
            "type": "XML"
        },
        ... // Other view extensions
        ...
        } // End of ALP view extensions
    } // End of view extensions
      }
   }
}
```

Sample of a custom view XML fragment:

```
<core:FragmentDefinition xmlns="sap.m"
xmlns:smartfilterbar="sap.ui.comp.smartfilterbar" xmlns:core="sap.ui.core">
    <smartfilterbar:ControlConfiguration key="CustomFilters" index="99"
visibleInAdvancedArea="true" label="Custom Filter" groupId="_BASIC">
        <smartfilterbar:customControl>
            <ComboBox id="CustomFilters-combobox">
                <core:Item key="0" text="Item1"/>
                <core:Item key="1" text="Item2"/>
                <core:Item key="2" text="Item3"/>
            </ComboBox>
        </smartfilterbar:customControl>
    </smartfilterbar:ControlConfiguration>
</core:FragmentDefinition>
```

Sample of a custom filter controller extension:

```
sap.ui.controller("analytics2.alr.ext.controller.CustomFiltersController", {
    onInitSmartFilterBarExtension: function(oEvent) {
       // the custom field in the filter bar might have to be bound to a custom
data model
       // if a value change in the field shall trigger a follow up action, this
method is the
       // place to define and bind an event handler to the field
       // Example:
       var oSmartFilterBar = oEvent.getSource();

oSmartFilterBar.getControlByKey("CustomFilters").attachSelectionChange(function(o
ChangeEvent){
    //code
       },this);
       Log.info("onInitSmartFilterBarExtension initialized");
    },
    onBeforeRebindTableExtension: function(oEvent) {
       // usually the value of the custom field should have an effect on the
selected data in the table.
       // So this is the place to add a binding parameter depending on the value
in the custom field.
    },
    onBeforeRebindChartExtension: function(oEvent) {
        // usually the value of the custom field should have an effect on the
selected data in the chart.
        // So this is the place to add a binding parameter depending on the
value in the custom field.
    },
    getCustomAppStateDataExtension : function(oCustomData) {
        // the content of the custom field shall be stored in the app state, so
that it can be restored
        // later. For example, after a back navigation. The developer has to
ensure that the content of the
        // field is stored in the object that is returned by this method.
        // Example:
        var oComboBox = this.byId("CustomFilters-combobox");
            if (oComboBox){
               oCustomData.CustomPriceFilter = oComboBox.getSelectedKey();
               }
        },
    restoreCustomAppStateDataExtension : function(oCustomData) {
        // in order to restore the content of the custom field in the filter
bar. For example, after a
        // back navigation, an object with the content is handed over to this
method and the developer
        // has to ensure, that the content of the custom field is set accordingly
        // also, empty properties have to be set
        // Example:
        if ( oCustomData.CustomPriceFilter !== undefined ){
           if ( this.byId("CustomFilters-combobox") ) {
               this.byId("CustomFilters-
combobox").setSelectedKey(oCustomData.CustomPriceFilter);
           }
        }
    }
});
```

→ Remember

- The custom filters do not show up in visual filters.
- If you define logic in the `onBeforeRebindChartExtension` or `onBeforeRebindTableExtension` to handle values that come from the custom filter fields, then these values are refreshed when the table

or chart area is refreshed. This ensures that custom filters are synchronized when the filter mode changes.

- When you choose *Clear*, ALP triggers `onClearFilterExtension`, which clears all filter dimensions. This means that you need to define the logic to handle the clear event for custom filters in the application controller extension . For example:

```
onClearFilterExtension: function(oEvent) {
        // Logic for clearing extended filters
        'use strict';
        if ( this.byId("CustomFilters-combobox") ) {
             this.byId("CustomFilters-combobox").setSelectedKey(null);
        }
}
```

Visual Filters

Use the `onBeforeRebindVisualFilterExtension` to customize the visual filter. Configure the extension to:

- Modify the visual filter or parameter values
- Add a custom query parameter to the visual filter call
- Influence the sorting order

In this extension, you can also access the incoming navigation context of the app with the `getNavigationContext` API.

```
onBeforeRebindVisualFilterExtension: function(sEntityType, sDimension, sMeasure,
oContext){     // oContext has filters, queryParameters, sorters,
entityParameters applicable for this specific visual filter
    'use strict';
    var oNavigationContext =
this.extensionAPI.getNavigationContext();                              //getting
incoming navigation context through extension API
    if (sDimension === "CostCenter") {
        oContext.queryParameters.Type =
"Cost";                                                  //adding custom query
parameter (It will be included in visual filter query as "?Type="Cost")

        if (oContext.entityParameters.P_DisplayCurrency === "USD")
{                          //Influencing applied parameters /
filters
            oContext.queryParameter.Country = "USA";
            oContext.filters.push(new sap.ui.model.Filter("Product", "EQ",
"HT-1000"));
        }
    }
}
```

Note

- The format for the date field is *YYYY-MM-DDT00:00:00Z*. For example, *2018-10-15T00:00:00Z*.
- The format for date and time values from the `SelectionVariant` annotation is *YYYY-MM-DDT00:00:00.000Z* (in UTC) or *YYYY-MM-DDTHH:MM:SS.fff-HH:MM* (local time with offset). For example, *2018-09-03T12:46:00.000Z* or *2018-09-03T12:46:12:123-7:00*

KPI Tags

You can add or modify the existing filters or parameter values using the extension API `onBeforeRebindFilterableKPIExtension`. The applications can also change depending on the KPI's entity type or KPI ID.

Sample Code

```
onBeforeRebindFilterableKPIExtension: function(oSelectionVariant,
sEntityType, sKPIId) {
        'use strict';
        // using this extension app can modify the existing filters and
parameters
        // and also add/remove/modify the custom filters applied to
FilterableKPIs
        if (sKPIId ===
"alp.tech.app::sap.suite.ui.generic.template.AnalyticalListPage.view.Analytica
lListPage::SEPMRA_C_ALP_SlsOrdItemCubeALPResults--
template::KPITag::kpi::KPINetProductPriceByCategory") {
            oSelectionVariant.addSelectOption("Product", "I", "EQ",
"HT-1502", null);
        }
        jQuery.sap.log.info("onBeforeRebindFilterableKPIExtension called!");
    }
             }
```

Related Information

Configuring Analytical List Page App Extensions [page 1915]

Defining Custom Actions

Define custom actions by using the extensions in the app-descriptor file. You can also define these custom actions so that they appear on charts, tables, or header toolbars based on the filter property value (chart/table/global).

Note

To correctly integrate your app extension coding with SAP Fiori elements, use only the `extensionAPI` of SAP Fiori elements. For more information, see Using the ExtensionAPI [page 1588].

Use the `requiresSelection` property to pass a row or record as a context. This ensures that the selection context is available with the `extensionAPI`. By default, `requiresSelection` property is `true`.

Use the `getSelectedContexts()` API in the `extensionAPI` class to get the selection context. For buttons in the chart toolbar, pass on the event ID as a parameter.

Set `"global": true` to define a button in the header area. This takes precedence over `determining` and `filter` settings that appear on the page header.

```
"sap.ui5": {
    "_version": "1.1.0",
    "extends": {
        "extensions": {
            "sap.ui.controllerExtensions": {
                "sap.suite.ui.generic.template.AnalyticalListPage.view.Analytical
ListPage": {
                    "controllerName": "analytics2.ext.controller.ALPExt",
                        "sap.ui.generic.app": {
                            "ZCostCenterCostQuery0020": {
                            "EntitySet": "ZCostCenterCostQuery0020",
                            "Actions": {
                                "Action A": {
                                    "id": "ActionA",
                                    "text": "{{Action A}}",
                                    "press": "onClickActionA",
                                    "global": true
                                },
                                "Action B": {
                                    "id": "ActionB_requiresSelection",
                                    "text": "{{Action B}}",
                                    "press": "onClickActionB",
                                    "filter": "table",
                                    "requiresSelection":false
                                }
                                "Action C: {
                                    "id": "ActionC_requiresSelection",
                                    "text": "{{Action C}}",
                                    "press": "onClickActionC",
                                    "filter": "chart",
                                    "requiresSelection":false
                                }
                                "Action D": {
                                    "id": "ActionD",
                                    "text": "{{Action D}}",
                                    "press": "onClickActionD",
                                    "filter": "table"
                                }
                                "Action E: {
                                    "id": "ActionE",
                                    "text": "{{Action E}}",
                                    "press": "onClickActionE",
                                    "filter": "chart"
                                }
                            } //End of Custom Actions
                        } // End of entity type ZCostCenterCostQuery0020
                    }
                } // End of ALP controllerExtensions
            } // End of controllerExtensions
        }
    },
        ....,
        ....
}
```

Custom actions defined in the application's custom controller:

```
sap.ui.controller("analytics2.ext.controller.ALPExt"){
    onBeforeRebindTableExtension: function(oEvent){
        console.log('onBeforeRebindTableExtension called!');
    },
    onBeforeRebindChartExtension: function(oEvent){
        console.log('onBeforeRebindChartExtension called!');
```

```
    },
    onClickActionA: function(oEvent){
        alert('Button A shows up only in table toolbar and is clicked toolbar!');
    },
    onClickActionB: function(oEvent){
        var contexts = this.extensionAPI.getSelectedContexts();
        alert('Button B which shows up in table toolbar only is clicked!');
    },
    onClickActionC: function(oEvent){
        var contexts = this.extensionAPI.getSelectedContexts(oEvent.ID);
        alert('Button C which shows up in chart toolbar only is clicked!');
    },
    onClickActionD: function(oEvent){
        alert('Button D which shows up in table toolbar only is clicked!');
    },
    onClickActionE: function(oEvent){
        alert('Button E which shows up in chart toolbar only is clicked!');
    }
}
```

Invoke Actions

This extension API lets you invoke any back-end action from the controller extensions (standard SAPUI5 API methods). For example:

```
onClickActionSTTA_C_SO_SalesOrder_ND1: function(oEvent) {
  var oApi = this.extensionAPI;
  var mParameters = {
        "SalesOrderID": "500000052"
    };
  oApi.invokeActions("STTA_SALES_ORDER_ND_SRV_01/
AFF8CCF97ACESave_stta_i_so_salesorder_nd", [], mParameters);
}
```

Related Information

Configuring Analytical List Page App Extensions [page 1915]
Smart Table Extensions [page 1916]

Adapting the UI: Analytical List Page

You can extend and customize specific features on the analytical list page by using the UI adaption editor in the SAP Web IDE.

> **i Note**
>
> Adapt the UI only for the use cases described here. Otherwise, control replacements might lead to changes that can no longer be applied.

Table 89:

Feature	Setting
Collapse smart filter bar	Configure the `Header Expanded` property for the template (ID is `template:Page`) to collapse and hide the filter bar.
Freeze filter bar on scroll	Configure the `Preserve header state on scroll` property for the template (ID is `template:Page`) to freeze the filter bar.
Variant management options	Configure the following page variant (ID is `template:PageVariant`) properties to enable the variant management options: • `Set as default`: This property lets you set the default variant for a user • `Public`: This property lets you make the variant as visible to users
Share icon	Configure the `Visibility` property to hide or unhide the share icon (ID is `template:Share`).
Clear button	The clear button may appear on the smart filter bar or on the header area based on the enabling of *Go* button. If the *Clear* button is on the: • Smart filer bar: Configure the `Visibility` property of the clear button (ID is `template:SmartFilterBar-btnClear`). • Header bar: Configure the `Visibility` property of the *Clear* button (ID is `template:ClearButton`).
Table toolbar	Configure the `Visibility` property of the table toolbar (ID is `template:TableToolbar`) to hide or unhide table toolbar.
Page footer	Configure the `Visibility` property of the page footer (ID is `template:FooterToolbar`) to hide or unhide page footer.
Chart	Use the following smart chart (ID is `chart`, example C_ContrItemMonitoringResults--chart) properties on the UI Adaptation layer: • Ignored Chart Types • Selection Mode • Chart tool tip • Chart type selection button

Adapting the UI: Chart

Table 90:

Property	Description
Ignored chart types	Comma-separated value fields do not showup in the list of available chart types.
No data	Allows you to override the default text when chart has no data to display.
Selection mode	Lets you select the mode (single, multiple, or none).
Show chart type selection button	Controls the visibility of the chart type button (users could change the chart type).
Show download button	Controls the visibility of the chart download button (users can download the chart as an image)
Show legend button	Controls the visibility of the legend button (users can toggle the visibility of the chart legends)

Values for Ignored Chart Type

Table 91:

Chart Type	ignoredChartTypes value
Bar Chart	bar
Column Chart	column
Line Chart	line
Combined Column Line Chart	combination
Pie Chart	pie
Doughnut Chart	donut
Scatter Plot	scatter
Bubble Chart	bubble
Heat Map	heatmap
Bullet Chart	bullet
Vertical Bullet Chart	vertical_bullet
Stacked Bar Chart	stacked_bar
Stacked Column Chart	stacked_column

Chart Type	ignoredChartTypes value
Combined Stacked Line Chart	stacked_combination
Horizontal Combined Stacked Line Chart	horizontal_stacked_combination
Bar Chart with 2 X-Axes	dual_bar
Column Chart with 2 Y-Axes	dual_column
Line Chart with 2 Y-Axes	dual_line
Stacked Bar Chart with 2 X-Axes	dual_stacked_bar
Stacked Column Chart with 2 Y-Axes	dual_stacked_column
Combined Column Line Chart with 2 Y-Axes	dual_combination
Combined Bar Line Chart with 2 X-Axes	dual_stacked_combination
Combined Stacked Line Chart with 2 Y-Axes	dual_horizontal_combination
Horizontal Combined Stacked Line Chart with 2 X-Axes	dual_horizontal_stacked_combination
100% Stacked Bar Chart	100_stacked_bar
100% Stacked Column Chart	100_stacked_column
100% Stacked Bar Chart with 2 X-Axes	100_dual_stacked_bar
100% Stacked Column Chart with 2 Y-Axes	100_dual_stacked_column
Waterfall Chart	waterfall
Horizontal Waterfall Chart	horizontal_waterfall

Adapting the UI: Chart Toolbar

Table 92:

Sample ID	Setting
C_ContrItemMonitoringResults--chart-btnPersonalisation	Icon: For chart personalization
C_ContrItemMonitoringResults--chart-btnDrillDownText	Visibility: To drill down within the chart toolbar

Adapting the UI: Table

Table 93:

Sample ID	Setting
C_ContrItemMonitoringResults--table-nodata	Allows you to override the default text when chart has no data to display
C_ContrItemMonitoringResults--table-btnPersonalisation	Icon: Lets you configure the table personalization
C_ContrItemMonitoringResults--table-btnExcelExport	Icon: To export content to excel
C_ContrItemMonitoringResults--table	Show Row Count

Adapting the UI: Table Column

Each column of the smart table can also be tweaked in UI Adaptation (sample ID is `C_ContrItemMonitoringResults--PurchaseContract`). The smart table columns expose several properties for this purpose. Some of them are listed here:

Table 94:

Property	Setting
Visible	Hide or unhide a column
Width	Adjust the column width
Auto Resizable	Expand column to the maximum width of the content
InResult	Ensures the column is always used in the query, which affects the aggregation level
Resizable	Resize column width
Show Sort Menu Entry	Displays column with a sort tab for personalization
Sort Order	Allows sorting
Sorted	Displays sorted items when sort is available in column header
Summed	Displays the total sum of a column

Overview Pages

An overview page is a data-driven SAP Fiori application built using SAPUI5 technology, OData services, and annotations for organizing large amounts of information.

Overview pages provide quick access to vital business information at a glance, in the form of visual, actionable cards. The user-friendly experience makes viewing, filtering, and acting upon data quick and simple. Business users can see the big picture at a glance, and also focus on the most important tasks, enabling faster decision making as well as immediate action.

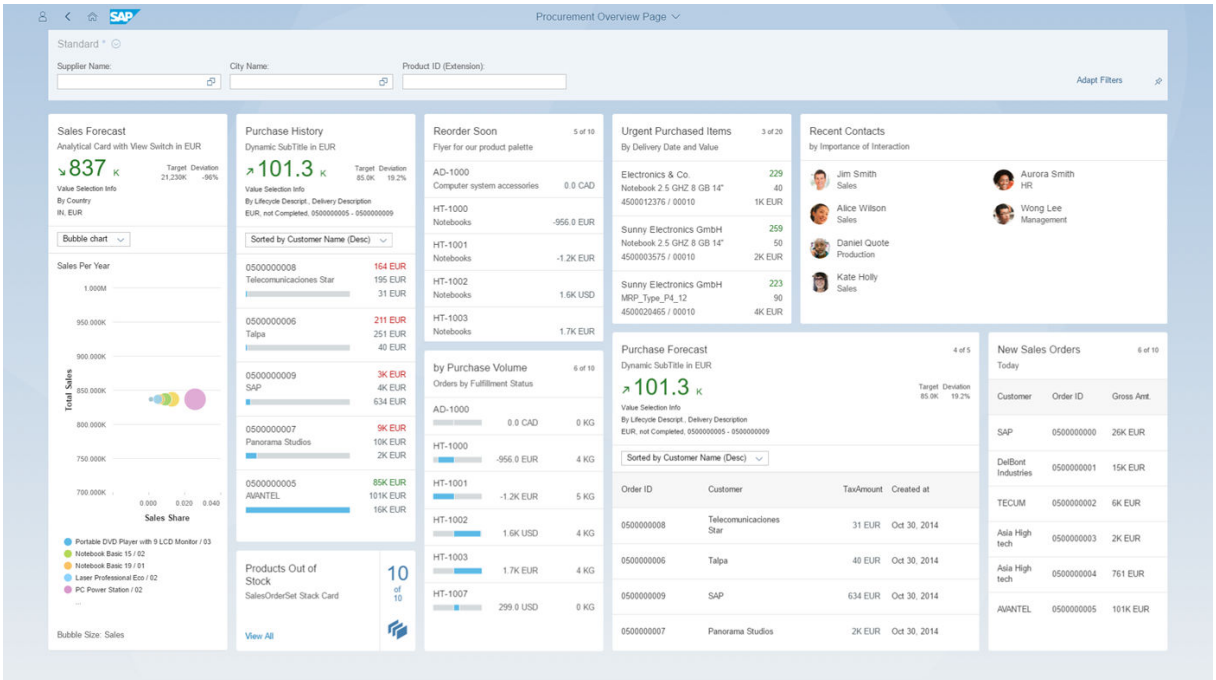
The application lets you create several cards for different types of content that helps in visualizing information in an attractive and efficient way. You can create overview pages and add cards to the page using the overview page wizard in SAP Web IDE.

The displayed data is fully interactive, with clickable areas for easy navigation to relevant applications. Based on SAP Fiori, overview pages organize action items with a fully responsive user interface. Users can access overview pages from SAP Fiori launchpad and narrow down the information displayed, or opt to hide cards to focus on a particular topic.

The overview page application contains the following main components:

- Application header: Provides a description of the area for which this application provides an overview (for example, procurement or sales). From the header area, users can change user account settings and manage cards.
- Smart filter: Provides application-level filters for changing the level of data displayed in the cards. For example, you could use the filter to display only transactions larger than $10,000, only items lighter than 50kg, and so on. .
- Cards: A card is a smart component that uses UI annotation to render its content. Each card is bound to a single entity set in a data source. A card may display a donut or bar chart, or a table. Stack cards contain a set of quick view cards, which can be viewed in an object stream. Cards are displayed on the overview page in up to five responsive columns and can be rearranged by dragging and dropping.

Overview page application instances consist of a UI component that extends the overview page application component and a manifest file that contains the application configuration.



- Analytical Cards [page 1976]
- Stack Cards [page 1970]
- List Cards [page 1953]
- List Cards [page 1953]
- List Cards [page 1953]
- Table Cards [page 1937]
- Stack Cards [page 1970]
- Link List Cards [page 1963]
- Overview Pages [page 1930]

More Information

For more information about the overview page plugin, see Building an App Using SAP Web IDE [page 1553].

Descriptor Configuration

The descriptor file (`manifest.json`) is an application configuration file that contains valid entries for initializing an application.

The manifest.json file defines static information about the application, such as the name of the application or the location of various files. It is written in JavaScript Object Notation (JSON) format.

Sample Code

Descriptor Sample Settings

```
"sap.ovp": {//section for ovp-specific app descriptor settings
    "globalFilterModel": "ZModelName", //OData model that contains entity
definitions relevant for global filters
    "globalFilterEntityType": "ZFilterEntityType", //Represents the entity to
use as a global filter in the smart filter bar control
    “globalFilterEntitySet”: “ZFilterEntitySet”, //Represents the entity set
to use as a global filter in the smart filter bar control
    "containerLayout": "resizable", //Represents the layout of the card
container, as fixed or resizable. The default value is fixed.
    “smartVariantRequired”: true, //Represents a switch to activate smart
variant management in the global filters. The default value is true.
    “showDateInRelativeFormat”: false, //Represents a switch to enable or
disable relative/normal date formatting in OVP applications
    "enableLiveFilter": false, //Represents the switch to activate live
update in the global filters, else manual update is required by clicking the
Go button
    ”imageSupported": true, //Allows the condensed list card to show images
or icons
    "considerAnalyticalParameters": true, //Flag to enable/disable analytical
parameter support for smart filter bar
    “refreshIntervalInMinutes”: 2, //Time interval in minutes to auto refresh
the card models
    “useDateRangeType”: true, //Flag to enable or disable semantic date range
control for the Smart filter bar. The default value is false.
    “cards”: {  //An object of cards
    “card01”: { //each card will contain the following
    "model": “ZCard1Model”, //Model for the card
    "template": “sap.ovp.cards.list”,//Card component path to use for this
card
    “settings”: {
    “title”: “card title”, //Language-dependent title of the card - used in
the card header
    “subTitle”: “sub title”, //Language-dependent subtitle of the card - used
in the card header
    “entitySet”: “zCard1EntitySet”, //Entity set displayed in this card
    “"valueSelectionInfo": “text for KPI Header”, //Additional information
relevant for the KPI Header
    "listFlavor": “Standard”, //Represents the flavor of the list to use in
this card. The flavor can be bar or standard.
    “listType”: “Extended”, //Represents the type of list to use for this
card. The list can be extended to display more information or condensed to
take up less space on the card
    “sortBy”: “zPropertyForSort”,//Property name to sort by this entity set
    “sortOrder“: “Ascending”//Sort order (ascending or descending)
    "staticContent": {}, //Applicable for static link list cards
    "annotationPath": “”, // Represents the annotation path
    "kpiAnnotationPath":"com.sap.vocabularies.UI.v1.KPI#AllActualCosts", //
Represents the KPI annotation path"selectionAnnotationPath": “”, //
Represents the selection annotation path
                "chartAnnotationPath": “”,//Represents the chart annotation
path
                "presentationAnnotationPath": “”, //Represents the
presentation annotation path
                "dataPointAnnotationPath": “”, //Represents the data point
annotation path
                "identificationAnnotationPath": “”, //Represents the
identification annotation path
                "dynamicSubtitleAnnotationPath": “”, //Represents the dynamic
subtitle annotation path
                "requireAppAuthorization": “” //Represents the cards for
which authorization is required
    "chartAnnotationPath":
"com.sap.vocabularies.UI.v1.Chart#SalesShareBubble",
```

```
                "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#SalesShareBubble",
                "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#Eval_by_Currency_Scatter",
                "selectionAnnotationPath" :
"com.sap.vocabularies.UI.v1.SelectionVariant#Eval_by_Currency_ColumnStacked",
                "navigation": "noHeaderNav" //Allows you to disable
navigation from the analytical list card header area

  }
}
}
}
```

Users can resize the card area (increase or decrease) in the resizable card layout. It is a grid-based layout that allows you to resize a card in horizontal and vertical directions. Configure the OVP section in the descriptor file with the `"containerLayout": "resizable"` property.

To load cards with specific requirements, define a default size as part of the card definition in the descriptor file.

```
"defaultSpan": {
  "rows": 7, \\Number of line items to display for table and list cards
  "cols": 2  \\Card width calculated based on window width
  "showOnlyHeader": true/false, \\To render only the card header
  "minimumTitleRow": 1/2/3, \\Display more text in the title, number of lines to
be shown
  "minimumSubTitleRow": 1/2 \\Display more text in the subtitle, number of lines
to be shown
}
```

Configuring Dependencies to SAPUI5 Libraries

To improve loading time, you can define dependencies to SAPUI5 libraries required by your application.

Overview page application instances are dependent on the `sap.ovp` SAPUI5 library, and the dependencies are configured in the `"sap.ui5"` dependencies section.

Sample Code

```
"sap.ui5": {
    ...
    "dependencies": {
        "minUI5Version": "1.32.0",
        "libs": {
            "sap.ovp": {
                "minVersion": "1.32.0"
            }
        }
    },
    ...
}
```

Configuring the Global Filter

The global filter provides the ability for end users to filter the data displayed in one or more cards.

The global filter is implemented using the `sap.ui.comp.smartfilterbar.SmartFilterBar` control. This control enables end users to persist their preferred filters, and share them with other users. The filter presents filterable properties according to the configured entity type, and is applied to all cards that have the same property name in their entity type. Also, you can define the filter you want to add to the filter bar by default by using the `UI.SelectionFields` configuration in the annotations file.

You configure the global filter in the `"sap.ovp"` section using the following properties:

- `globalFilterModel`: the OData model to use for the global filter
- `globalFilterEntityType`: the entity type that contains the filterable properties

Sample Code

```
"sap.ovp": {
    "_version": "1.1.0",
    "globalFilterModel": "ZCD204_EPM_DEMO_SRV",
    "globalFilterEntityType": "SalesOrder",
    "cards": {
        ...
    }
}
```

Enabling Basic Search

The search field on the smart filter bar lets you search for a value across all searchable entity sets. To enable the search field, set the property `"showBasicSearch":"true"` in the descriptor file.

The search functionality is applicable for entity types that has:

- `sap:searchable="true"` in the metadata file
- `SearchRestrictions` annotation present for the entity set

Overview Page Card

A card is a smart component that uses UI annotation to render its content. It contains a header area and a footer area.

Each card is bound to a single entity set in a data source and configuration is provided in the `"sap.ovp"..."cards"` object. The cards object contains the list of cards to display in the application.

The card ID is the property name and the card configuration is provided in an object as the value. At runtime, cards are displayed in the order that they appear in the application descriptor.

Card Header

All cards have a static header section that can be configured in the descriptor configuration file. The card header includes the properties: `category`, `title`, and `subTitle`.

The title of a card is mandatory. The subtitle is only mandatory if the card contains data point annotations (such as a KPI header). The title and subtitle can contain a maximum of two lines. The header also contains counter information that displays how many records are being presented in the card out of the total existing records, according to the current filter.

> **i Note**
>
> The count information displays only in table and list cards. For more information, see Table Cards [page 1937] and List Cards [page 1953].

Sample Code

```
"sap.ovp": {
    "_version": "1.1.0",
    "globalFilterModel": "ZCD204_EPM_DEMO_SRV",
    "globalFilterEntityType": "SalesOrder",
    "cards": {
        "card00": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.stack",
            "settings": {
                ...
                "category": "{{card00_category}}",
                "title": "{{card00_title}}",
                "subTitle ": "{{card00_subTitle}}",
                ...
            }
        },
        "card01": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.table",
            "settings": {
                "title": "{{card01_category}}",
                ...
            }
        },
        ...
    }
}
```

KPI Headers

The generic card provides a dynamic section that can display a key performance indicator (KPI), and related information, in the header. The KPI is an aggregated value, as defined in the annotation file. To display a KPI header in a card, make sure that your OData service supports aggregation of values and the data is coming from the backend only. The KPI header can contain a KPI value, including its unit of measure, its trend, and percentage of change, and KPI header description.

The KPI extension uses the following annotation terms:

- `com.sap.vocabularies.UI.v1.DataPoint`: Used to retrieve information about the title and the value of the KPI.
- `com.sap.vocabularies.UI.v1.PresentationVariant`: Used to retrieve information about the fields by which to group, and sorting information.

- `com.sap.vocabularies.UI.v1.SelectionVariant`: Used to retrieve information about the filters.

> **Note**
>
> For more information about these annotations, see Annotations Used in Overview Pages [page 2010].

These annotation terms can be configured in the application manifest file, as shown in the following example:

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
        ...
        "card02": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.charts.bubble",
            "settings": {
                "entitySet": "SalesOrders",
                "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#bubble",
                "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#bubble",
                "chartAnnotationPath":
"com.sap.vocabularies.UI.v1.Chart#bubble",
                "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#bubble",
                "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#bubble"
            }
        },
        ...
    }
}
```

Types of Cards

Overview pages may contain transactional cards, such as table, list, stack, quick view cards, and analytical chart cards such as line, bubble, donut, column, and bullet chart cards.

The following is a list of available cards:

Card Type	Description
Table	Table cards display a list of records according to the configuration in the `com.sap.vocabularies.UI.v1.LineItem` term. Table cards displays data in a 3-column table layout.
List	List cards display lists of records according to the configuration in the `com.sap.vocabularies.UI.v1.LineItem` term. List cards display up to six fields of data in each list item.
Link List	Link list cards display a list of links with an image or icon and a (optional) subtitle for each of the links.

Card Type	Description
Stack	Stack cards aggregate a set of cards of the same type, which are based on a common topic or action. When clicked, stack cards can display up to 20 cards in an object stream.
Quick View	Quick view cards display detailed information about a single record, in greater depth than would be displayed in a table or list.
Analytical Chart	Analytical chart cards show data in a variety of chart formats. They can be line, bubble, donut, column, stacked column, vertical column, combination, or scatter chart cards. The value of the template property points to the generic card component `sap.ovp.cards.charts.analytical`.

You can set all cards to refresh automatically for a given interval. The minimum and default refresh time is one minute. To enable auto refresh, configure the `refreshIntervalInMinutes` property in the descriptor configuration file.

```
"sap.ovp": {
        "globalFilterModel": "salesOrder",
        "globalFilterEntityType": "GlobalFilters",
        ...
        ...
        "refreshIntervalInMinutes": 12,
        "disableTableCardFlexibility": false,
               "cards": {
```

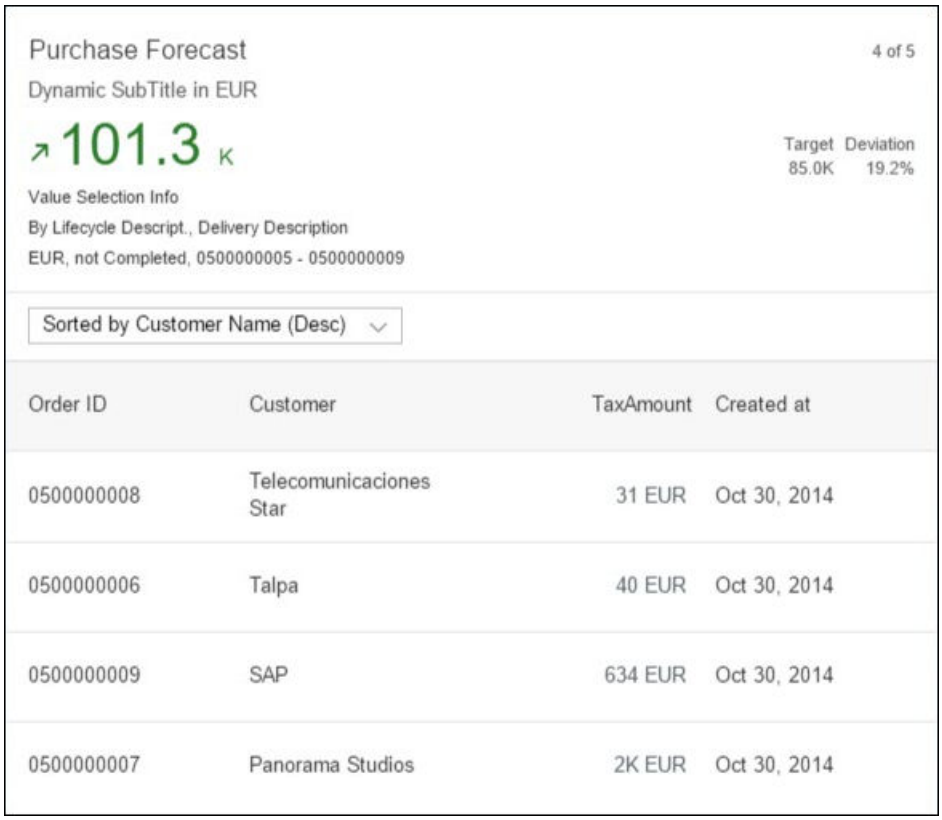
Table Cards

A table card displays a list of records in a 3-column table layout.

To create a table card, you need these annotations:

> **i** Note
>
> To add annotations, use the SAP WebIDE annotation modular or the SAP WebIDE code editor. For more information, see Building an App Using SAP WebIDE

> **i** Note
>
> (Optional) You can configure smart links in table cards to access quick links.



- #unique_724/unique_724_Connect_42_subsection-im1 [page 1938]
- #unique_724/unique_724_Connect_42_subsection-im2 [page 1939]
- #unique_724/unique_724_Connect_42_subsection-im3 [page 1940]
- #unique_724/unique_724_Connect_42_subsection-im4 [page 1942]
- #unique_724/unique_724_Connect_42_subsection-im5 [page 1943]

Hover over each action for a description. Click the action for more information.

Descriptor Settings: Title

Property: `title`

Description: Configuring this property displays the card title at the top of the table card.

Configuration Sample:

```
    "sap.ovp": {
        "globalFilterModel": "salesOrder",
        "globalFilterEntityType": "GlobalFilters",
              ...
              ...
        "cards": {
          "card014": {
            "model": "salesOrder",
            "template": "sap.ovp.cards.table",
            "settings": {
```

```
              "title": "Purchase Forecast",
              "entitySet": "SalesShare",
              ...
              ...
              ...
                }
              ]
            }
          },
```

Descriptor Settings: Subtitle

Property: `subTitle`

Description: Configuring this property displays the card subtitle below the title of the table card.

Configuration Sample:

```
    "sap.ovp": {
        "globalFilterModel": "salesOrder",
        "globalFilterEntityType": "GlobalFilters",
              ...
              ...
        "cards": {
          "card014": {
            "model": "salesOrder",
            "template": "sap.ovp.cards.table",
            "settings": {
              "title": "Sales Forecast",
              "subTitle": "per Supplier",
              "entitySet": "SalesShare",
              ...
              ...
                }
              ]
            }
          },
```

You can display the unit of measure next to numeric values by providing the `sap:unit` attribute in the OData metadata file or by annotating the unit in the annotation document. For example, if you have the following data, and want to display 850 kg after the subtitle text

```
{
   CurrencyCode:"KG"
   GrossAmount:850
}
```

Use one of the following options:

Option 1: Define `sap:unit` in the metadata

```
<Property Name="CurrencyCode" Type="Edm.String" MaxLength="5"
sap:label="Currency" sap:updatable="false" sap:semantics="currency-code"/>
<Property Name="GrossAmount" Type="Edm.Decimal" Precision="16" Scale="3"
sap:unit="CurrencyCode" sap:label="Gross Amt." sap:creatable="false"
sap:updatable="false"/>
```

Option 2: Define `Org.OData.Measures.V1.ISOCurrency` in annotations

```
<Annotations Target="GWSAMPLE_BASIC.SalesOrder/GrossAmount">
<Annotation Term="Org.OData.Measures.V1.ISOCurrency" Path="CurrencyCode"/>
</Annotations>
```

KPI annotation

Descriptor Settings

```
"sap.ovp": {//section for ovp-specific app descriptor settings
    ...
    "kpiAnnotationPath":"com.sap.vocabularies.UI.v1.KPI#AllActualCosts", //
Represents the KPI annotation path

     ...
}
}
```

Annotation Sample

```
<Annotation Term="UI.KPI" Qualifier="AllActualCosts">
   <Record Type="UI.KPIType">
      <PropertyValue Property="Detail">
         <Record Type="UI.KPIDetailType">
            <PropertyValue Property="DefaultPresentationVariant"
Path="@UI.PresentationVariant#Eval_by_Currency1" />
            <PropertyValue Property="AlternativePresentationVariants">
               <Collection>
                  <Path>@UI.PresentationVariant#Eval_by_Currency_Column</Path>
               </Collection>
            </PropertyValue>
            <PropertyValue Property="SemanticObject" String="Action" />
            <PropertyValue Property="Action" String="toappnavsample" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#Eval_by_Currency_1" />
      <PropertyValue Property="DataPoint" Path="@UI.DataPoint#Eval_by_Country-
Generic" />
      <PropertyValue Property="ID" String="String for KPI Annotation" />
   </Record>
</Annotation>
```

Sample Code

`PresentationVariant` annotation

```
 <Annotation Term="UI.PresentationVariant" Qualifier="Eval_by_Currency1">
    <Record>
       <PropertyValue Property="MaxItems" Int="5" />
       <PropertyValue Property="GroupBy">
          <Collection>
             <PropertyPath>Country</PropertyPath>
             <PropertyPath>Currency</PropertyPath>
          </Collection>
       </PropertyValue>
       <PropertyValue Property="SortOrder">
          <Collection>
             <Record>
                <PropertyValue Property="Property" PropertyPath="TotalSales" />
                <PropertyValue Property="Descending" Boolean="true" />
             </Record>
          </Collection>
       </PropertyValue>
       <PropertyValue Property="Visualizations">
          <Collection>
             <AnnotationPath>@UI.Chart#Eval_by_Currency_Donut</AnnotationPath>
          </Collection>
       </PropertyValue>
    </Record>
```

```
</Annotation>
```

Sample Code

`SelectionVariant` annotation

```
<Annotation Term="UI.SelectionVariant" Qualifier="Eval_by_Currency_1">
   <Record>
      <PropertyValue Property="SelectOptions">
         <Collection>
            <Record>
               <PropertyValue Property="PropertyName"
PropertyPath="Country" />
               <PropertyValue Property="Ranges">
                  <Collection>
                     <Record>
                        <PropertyValue Property="Sign"
EnumMember="UI.SelectionRangeSignType/I" />
                        <PropertyValue Property="Option"
EnumMember="UI.SelectionRangeOptionType/EQ" />
                        <PropertyValue Property="Low" String="IN" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Parameters">
         <Collection>
            <Record Type="UI.Parameter">
               <PropertyValue Property="PropertyName"
PropertyPath="Currency_Target" />
               <PropertyValue Property="PropertyValue" String="EUR" />
            </Record>
            <Record Type="UI.Parameter">
               <PropertyValue Property="PropertyName"
PropertyPath="UoM_Target" />
               <PropertyValue Property="PropertyValue" String="KGM" />
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Sample Code

`DataPoint` annotation

```
<Annotation Term="UI.DataPoint" Qualifier="Eval_by_Country-Generic">
   <Record Type="UI.DataPointType">
      <PropertyValue Property="Title" String="Sales India - Generic Card" />
      <PropertyValue Property="Value" Path="Sales" />
      <PropertyValue Property="ValueFormat">
         <Record>
            <PropertyValue Property="ScaleFactor" Int="2" />
            <PropertyValue Property="NumberOfFractionalDigits" Int="1" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="CriticalityCalculation">
         <Record>
            <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Minimizing" />
            <PropertyValue Property="DeviationRangeHighValue" String="7300" />
            <PropertyValue Property="ToleranceRangeHighValue" String="7200" />
```

```
            </Record>
         </PropertyValue>
         <PropertyValue Property="TargetValue" String="2.000 " />
         <PropertyValue Property="TrendCalculation">
            <Record>
               <PropertyValue Property="ReferenceValue" String="5201680" />
               <PropertyValue Property="DownDifference" Int="10000000.0" />
            </Record>
         </PropertyValue>
      </Record>
   </Annotation>
```

Configuring View Switch

Property: `valueSelectionInfo`

Description: Configuring this property allows you to define a dropdown list to filter/view data at the card level.

Descriptor Settings

```
    "sap.ovp": {
        "globalFilterModel": "salesOrder",
        "globalFilterEntityType": "GlobalFilters",
        "showDateInRelativeFormat": false,
        "disableTableCardFlexibility": false,
        "considerAnalyticalParameters": true,
        "useDateRangeType": false,
        "cards": {
          "card014": {
            "model": "salesOrder",
            "template": "sap.ovp.cards.table",
            "settings": {
              "title": "Sales Forecast",
              "subTitle": "per Supplier",
              "valueSelectionInfo": "Value Selection Info",
              "entitySet": "SalesShare",
              "tabs": [
                {
                  "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
                  "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View1",
                  "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#line1",
                  "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#line",
                  "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification",
                  "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
                  "value": "{{dropdown_value2}}"
                },
                {
                  "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
                  "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View4",
                  "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#item2",
                  "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
                  "value": "{{dropdown_value4}}"
                }
              ]
            }
          },
```

Configuring the Table Area

Annotation: `LineItem`

Description: Configuring this annotation displays the table header title (`Label` property) and the corresponding values (`Value` property) for the row items.

Descriptor Settings

```
    "sap.ovp": {
        "globalFilterModel": "salesOrder",
        "globalFilterEntityType": "GlobalFilters",
        "showDateInRelativeFormat": false,
        "disableTableCardFlexibility": false,
        "considerAnalyticalParameters": true,
        "useDateRangeType": false,
        "cards": {
          "card014": {
            "model": "salesOrder",
            "template": "sap.ovp.cards.table",
            "settings": {
              "title": "Sales Forecast",
              "subTitle": "per Supplier",
              "valueSelectionInfo": "Value Selection Info",
              "entitySet": "SalesShare",
              "tabs": [
                {
                  "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
                  "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View1",
                  "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#line1",
                  "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#line",
                  "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification",
                  "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
                  "value": "{{dropdown_value2}}"
                },
                {
                  "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
                  "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View4",
                  "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#item2",
                  "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
                  "value": "{{dropdown_value4}}"
                }
              ]
            }
          },
```

Annotation Sample for `LineItem#View4`

```
                <Annotation Term="com.sap.vocabularies.UI.v1.LineItem"
Qualifier="View4">
                    <Collection>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Label" String="Order ID"/>
                            <PropertyValue Property="Value" Path="SalesOrderID"/>
                        </Record>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Label" String="Customer"/>
```

```
                            <PropertyValue Property="Value"
Path="ToBusinessPartner/EmailAddress"/>
                        </Record>
                        <Record Type="com.sap.vocabularies.UI.v1.DataField">
                            <PropertyValue Property="Label" String="Customer"/>
                            <PropertyValue Property="Value" Path="CustomerName"/>
                        </Record>
                        <Record
Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation">
                            <PropertyValue Property="Label" String="TaxAmount"/>
                            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#TaxAmount"/>
                        </Record>
                    </Collection>
                </Annotation>
```

Annotation Sample for `DataPoint#TaxAmount`

```
                <Annotation Term="com.sap.vocabularies.UI.v1.DataPoint"
Qualifier="TaxAmount">
                    <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
                        <PropertyValue Property="Title" String="TaxAmount"/>
                        <PropertyValue Property="Value" Path="TaxAmount"/>
                    </Record>
                </Annotation>
```

The `com.sap.vocabularies.UI.v1.LineItem` term can be configured in the application manifest file by setting the `annotationPath` property with a qualifier, as shown in the example below. If the `annotationPath` property is not configured, the `com.sap.vocabularies.UI.v1.LineItem` term, without a qualifier, is used.

Sample Code

```
"sap.ovp": {
    ...
    "disableTableCardFlexibility": false,
    ...
    "card02": {
    "model": "salesOrder",
    "template": "sap.ovp.cards.table",
    "settings": {
        "title": "Sales Orders With Analytical Header",
        "subTitle": "Sales Orders With Analytical Header",
        "listFlavor": "bar",
        "listType": "extended",
        "entitySet": "SalesOrderSet",
        "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line_without_trend",
        "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#line1",
          "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View1",
        "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#line"
      }
    },
      ...
  }
}
```

You can use different `com.sap.vocabularies.UI.v1.LineItem` annotations for different card instances of the same entity type by using different qualifiers and setting the `annotationPath` property with the qualifier in the card configuration. For example `com.sap.vocabularies.UI.v1.LineItem#Qualifier1`.

If you set `"disableTableCardFlexibility": true`, then at runtime, `DataField` records are sorted according to importance set in the `com.sap.vocabularies.UI.v1.ImportanceType` annotation, and their order of entry. The first two `DataField` records are displayed in the first two columns of the table. If there is a `DataFieldForAnnotation` record that has a `DataPoint` target, it is used for the third column and its value is highlighted according to the criticality of the `DataPoint`. If no `DataFieldForAnnotation` is defined, the next `DataField` record is displayed in the third column.

If you set `"disableTableCardFlexibility": false`, then at runtime, table columns are sorted according to the importance set in the `com.sap.vocabularies.UI.v1.ImportanceType` annotation, and their order of entry.

> **i** Note
>
> If a `DataField` record points to a path that exists in a `DataPoint` target, it is skipped so that the property is not displayed more than once in the same table.

Example

In this example, the first column in the table displays **Product Name**, the second column displays **Supplier**, and the third column displays **Weight Measure** (according to the `DataPoint` target).

> Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint"
Qualifier="WeightMeasure">
    <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
        <PropertyValue Property="Title" String="Weight"/>
        <PropertyValue Property="Description" Path="Name"/>
        <PropertyValue Property="Value" Path="WeightMeasure"/>
        <PropertyValue Property="CriticalityCalculation">
            <Record
Type="com.sap.vocabularies.UI.v1.CriticalityCalculationType">
                <PropertyValue Property="ImprovementDirection"
EnumMember="com.sap.vocabularies.UI.v1.CriticalityCalculationType/Maximize"/>
                <PropertyValue Property="ToleranceRangeLowValue" Int="2.5"/>
                <PropertyValue Property="DeviationRangeLowValue" Int="4.3"/>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
<Annotation Term="com.sap.vocabularies.UI.v1.LineItem">
    <Collection>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Product ID"/>
            <PropertyValue Property="Value" Path="ProductID"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Category"/>
            <PropertyValue Property="Value" Path="Category"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Product Name"/>
            <PropertyValue Property="Value" Path="Name"/>
            <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/High"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
```

```
            <PropertyValue Property="Label" String="Supplier"/>
            <PropertyValue Property="Value" Path="SupplierName"/>
            <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/High"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Unit Price"/>
            <PropertyValue Property="Value" Path="Price"/>
            <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/High"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation"
Qualifier="WeightMeasure">
            <PropertyValue Property="Label" String="Weight Measure"/>
            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#WeightMeasure"/>
        </Record>
    </Collection>
</Annotation>
```

Smart Links

Define a semantic object for the entity set and its property using the annotation target to enable smart links in a table card. For example:

```
<Annotations Target="GWSAMPLE_BASIC.SalesOrder/SalesOrderID">
    <Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject"
String="OVP" />
</Annotations>
```

Table cards also let you view contact information if you have defined the `com.sap.vocabularies.Communication.v1.Contact` annotation.

```
<Annotation Term="com.sap.vocabularies.UI.v1.LineItem"
Qualifier="NewSalesOrders">
    <Collection>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Order ID (Company)"/>
            <PropertyValue Property="Value" Path="SalesOrderID"/>
        </Record>
          <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation">
               <PropertyValue Property="Label" String="Created by (Role)" />
               <PropertyValue Property="Target"
AnnotationPath="ToBusinessPartner/
@com.sap.vocabularies.Communication.v1.Contact" />
          </Record>
    </Collection>
</Annotation>
```

```
<Annotation Term="com.sap.vocabularies.Communication.v1.Contact">
   <Record>
      <PropertyValue Property="tel">
         <Collection>
            <Record>
               <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/fax" />
               <PropertyValue Property="uri" Path="FaxNumber" />
            </Record>
            <Record>
```

```
               <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/work
com.sap.vocabularies.Communication.v1.PhoneType/pref" />
               <PropertyValue Property="uri" Path="PhoneNumber" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="email">
         <Collection>
            <Record>
               <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.ContactInformationType/pref
com.sap.vocabularies.Communication.v1.ContactInformationType/work" />
               <PropertyValue Property="address" Path="EmailAddress" />
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Text Arrangement

The text arrangement annotation lets you to format of text.

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="GWSAMPLE_BASIC.SalesOrder/CustomerID">
    <Annotation Term="com.sap.vocabularies.Common.v1.Text" Path="Supplier_Name"/>
    <Annotation Term="com.sap.vocabularies.UI.v1.TextArrangement"
EnumMember="com.sap.vocabularies.UI.v1.TextArrangementType/TextLast" />
</Annotations>
```

In the preceding example, the text `Customer` is bound to the `ContactID` property and appears as shown in the table:

Text Arrangement Type	Result
`TextLast`	ContractID (Customer)
`TextFirst`	Customer (ContractID)
`TextOnly`	Customer

Text Alignment

The DataPoint or DataField is aligned as shown in the table below:

Annotation Field	Table Alignment
DataField, Contact annotation	Left Aligned

Annotation Field	Table Alignment
DataPoint with or without Criticality/CriticalityCalculation OR property (either DataField or DataPoint) of type Edm.DateTime or Edm.DateTimeOffset	Right Aligned
DataPoint with Criticality and property of type Edm.String	Center Aligned

Configuring the Table Area

The `LineItem` annotation lets you configure the table header title (label property) and the corresponding values (value property) for the column items.

1. Define `DataField` property to display values (text) for the table column. For example:

```
<Annotation Term="com.sap.vocabularies.UI.v1.LineItem" Qualifier="View4">
   <Collection>
      <Record Type="com.sap.vocabularies.UI.v1.DataField">
         <PropertyValue Property="Label" String="Order ID" />
         <PropertyValue Property="Value" Path="SalesOrderID" />
      </Record>
      <Record Type="com.sap.vocabularies.UI.v1.DataField">
         <PropertyValue Property="Label" String="Customer" />
         <PropertyValue Property="Value" Path="ToBusinessPartner/
EmailAddress" />
      </Record>
      <Record Type="com.sap.vocabularies.UI.v1.DataField">
         <PropertyValue Property="Label" String="Customer" />
         <PropertyValue Property="Value" Path="CustomerName" />
      </Record>
      <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation">
         <PropertyValue Property="Label" String="TaxAmount" />
         <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#TaxAmount" />
      </Record>
   </Collection>
</Annotation>
```

2. (Optional) Define `DataFieldforAnnotation` property using these annotations:
   - `DataPoint`: To view numeric values in the table. For example:

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint"
Qualifier="TaxAmount">
   <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
      <PropertyValue Property="Title" String="TaxAmount" />
      <PropertyValue Property="Value" Path="TaxAmount" />
   </Record>
</Annotation>
```

   - `Contact`: To view quick view information in the table. For example, you can configure the table card to display contact information as shown here:

```
<Annotation Term="com.sap.vocabularies.Communication.v1.Contact">
   <Record>
      <PropertyValue Property="tel">
         <Collection>
            <Record>
```

```
                <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/fax" />
                <PropertyValue Property="uri" Path="FaxNumber" />
             </Record>
             <Record>
                <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/work
com.sap.vocabularies.Communication.v1.PhoneType/pref" />
                <PropertyValue Property="uri" Path="PhoneNumber" />
             </Record>
          </Collection>
       </PropertyValue>
       <PropertyValue Property="email">
          <Collection>
             <Record>
                <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.ContactInformationType/
pref com.sap.vocabularies.Communication.v1.ContactInformationType/work" />
                <PropertyValue Property="address" Path="EmailAddress" />
             </Record>
          </Collection>
       </PropertyValue>
    </Record>
</Annotation>
```

3. (Optional) Define navigation properties for table area.
   For configuring navigation information:
   - Use com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation to define intent based navigation to SAP Fiori application.
   - Use com.sap.vocabularies.UI.v1.DataFieldWithUrl term to configure navigation to external apps and websites.

> **i Note**
>
> The recommended way to configure intent-based navigation using `DataFieldForIntentBasedNavigation` property. However, you can also use `DataFieldWithUrl` for navigation to a specific application route that is not configured as target map. The overview page identify this as an intent-based navigation and opens the application in the same tab with relevant context.

Configuring the Table Card Header Area (Optional)

You can configure the header area from the app descriptor file and annotations.

1. Configure card header properties (card title, subtitle, KPI value, and view switch settings in the descriptor file as shown here):
   - Card `title` and `subTitle`

```
"sap.ovp": {
"globalFilterModel": "salesOrder",
"globalFilterEntityType": "GlobalFilters",
      ...
      ...
"cards": {
  "card014": {
    "model": "salesOrder",
    "template": "sap.ovp.cards.table",
    "settings": {
       "title": "Sales Forecast",
```

```
        "subTitle": "per Supplier",
        "entitySet": "SalesShare",
      ...
      ...
        }
      ]
    }
},
```

- KPI information

  Annotation Sample

  Descriptor Settings

```
"sap.ovp": {//section for ovp-specific app descriptor settings
    ...

"kpiAnnotationPath":"com.sap.vocabularies.UI.v1.KPI#AllActualCosts", //
Represents the KPI annotation path

     ...
}
}
```

Sample Code

`DataPoint` annotation

```
<Annotation Term="UI.DataPoint" Qualifier="Eval_by_Country-Generic">
   <Record Type="UI.DataPointType">
      <PropertyValue Property="Title" String="Sales India - Generic
Card" />
      <PropertyValue Property="Value" Path="Sales" />
      <PropertyValue Property="ValueFormat">
         <Record>
            <PropertyValue Property="ScaleFactor" Int="2" />
            <PropertyValue Property="NumberOfFractionalDigits"
Int="1" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="CriticalityCalculation">
         <Record>
            <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Minimizing" />
            <PropertyValue Property="DeviationRangeHighValue"
String="7300" />
            <PropertyValue Property="ToleranceRangeHighValue"
String="7200" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="TargetValue" String="2.000 " />
      <PropertyValue Property="TrendCalculation">
         <Record>
            <PropertyValue Property="ReferenceValue" String="5201680" />
            <PropertyValue Property="DownDifference" Int="10000000.0" />
         </Record>
      </PropertyValue>
   </Record>
</Annotation>
```

- View switch: Configuring this property allows you to define a dropdown list to filter/view data at the card level.

```
"sap.ovp": {
    "globalFilterModel": "salesOrder",
```

```
    "globalFilterEntityType": "GlobalFilters",
    "showDateInRelativeFormat": false,
    "disableTableCardFlexibility": false,
    "considerAnalyticalParameters": true,
    "useDateRangeType": false,
    "cards": {
     "card014": {
         "model": "salesOrder",
         "template": "sap.ovp.cards.table",
         "settings": {
     "title": "Sales Forecast",
     "subTitle": "per Supplier",
     "valueSelectionInfo": "Value Selection Info",
     "entitySet": "SalesShare",
     "tabs": [
           {
     "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
     "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View1",
     "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#line1",
     "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#line",
     "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification",
     "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
     "value": "{{dropdown_value2}}"
           },
           {
     "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
     "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View4",
     "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#item2",
     "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
     "value": "{{dropdown_value4}}"
    }
   ]
  }
},
```

2. Define the `DataPoint` annotation to complete configuring KPI information on the card header area. The following are the annotation properties:
   - Add `Title` property to configure the table column name, and also to display as a title on the KPI header
   - Add `Value` property to display KPI measure
   - Add `ValueFormat` to define number format
   - Add criticality to highlight the KPI measure value. You can define criticality as a path or enum value. The supported enum values are:
     - `com.sap.vocabularies.UI.v1.CriticalityType/Neutral` - default neutral color is considered
     - `com.sap.vocabularies.UI.v1.CriticalityType/Negative` - Red is considered
     - `com.sap.vocabularies.UI.v1.CriticalityType/Critical` - Orange is considered
     - `com.sap.vocabularies.UI.v1.CriticalityType/Positive` - Green is considered

   You can also define criticality using a path property that returns value:
   - 0 for Neutral- default neutral colour is considered
   - 1 for Negative – Red is considered
   - 2 for Critical – Orange is considered

- 3 for Positive – Green is considered

Configuring the Table Header Navigation (Optional)

The identification annotation lets you configure navigation (from header and table area) within an application.

> **i** Note
>
> In case of table area navigation, the navigation settings configured using the `LineItem` annotation takes precedence over the navigation settings configured using the identification annotation.

Customizing Table Card

You can customize the content on the table area with smart links, text alignment, filtering or grouping information.

Adding Smart Links

Define a semantic object for the entity set and its property using the annotation target to add smart links. For example:

```
<Annotations Target="GWSAMPLE_BASIC.SalesOrder/SalesOrderID">
<Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject" String="OVP" />
</Annotations>
```

Text Arrangement in Table Area

Define `Text` arrangement annotation to format content on the table area.

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="GWSAMPLE_BASIC.SalesOrder/CustomerID">
<Annotation Term="com.sap.vocabularies.Common.v1.Text" Path="Supplier_Name"/>
<Annotation Term="com.sap.vocabularies.UI.v1.TextArrangement"
EnumMember="com.sap.vocabularies.UI.v1.TextArrangementType/TextLast" />
</Annotations>
```

In the preceding example, the text `Customer` is bound to the `ContactID` property and appears as shown in the table:

Text Arrangement Type	Result
`TextLast`	`ContractID (Customer)`
`TextFirst`	`Customer (ContractID)`
`TextOnly`	`Customer`

Filtering

You can add filters to table card by using the `com.sap.vocabularies.UI.v1.SelectionVariant` annotation term, or by passing filter parameter in the URL. For more information, see Configuring Card Filters [page 2000]

Sorting or Grouping

You can sort information in the table card by using `sortBy` and `sortOrder` properties in the application descriptor file. For more information, see Configuring Sort Properties [page 2001]

List Cards

List cards display lists of records according to the configuration in the `com.sap.vocabularies.UI.v1.LineItem` term. List cards display up to six fields of data in each list item.

When creating a list card, you can choose from a number of different types of lists. The number of items displayed depends on the type of list. You can choose from two types of list cards:

- Condensed
- Extended

For each of these types you can choose from two flavors:

- Standard
- Bar

The `com.sap.vocabularies.UI.v1.LineItem` term can be configured in the application manifest file by setting the `annotationPath` property with a qualifier, as shown in the example below. If the `annotationPath` property is not configured, the `com.sap.vocabularies.UI.v1.LineItem` term, without a qualifier, is used.

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
```

```
        ...
        "card04": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.list",
            "settings": {
                "sortBy": "Price",
                "sortOrder": "descending",
                "listFlavor": "bar",
                "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#bar",
                "category": "{{card04_category}}",
                "entitySet": "Products"
            }
        },
        ...
    }
}
```

By default, the fields in the list card are mapped to the `com.sap.vocabularies.UI.v1.LineItem` annotation. Any other collection of `DataFieldAbstract` can be used by setting the `annotationPath` property. `LineItem` is a collection of `DataFieldAbstract` records. You can use different `com.sap.vocabularies.UI.v1.LineItem` annotations for different card instances of the same entity type by using different qualifiers and setting the `annotationPath` property with the qualifier in the card configuration. For example `com.sap.vocabularies.UI.v1.LineItem#Qualifier1`.

At runtime, the `DataField` records are sorted according to the optional `Importance (com.sap.vocabularies.UI.v1.ImportanceType)` annotation. `DataField` entries are sorted according to importance and their order of entry.

Condensed List Type - Standard Flavor

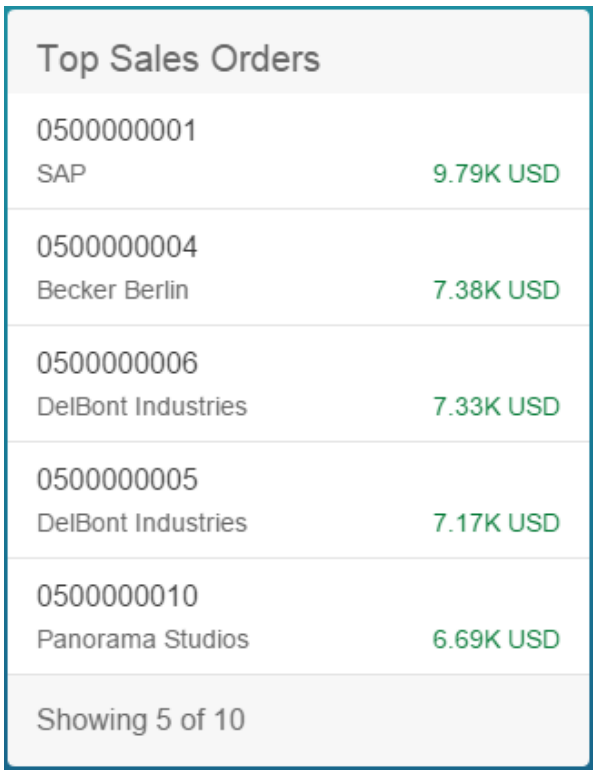
In this type of list, each list item displays up to three fields. The first two `DataField` records are displayed at the top left and bottom left of the list item. If there is a `DataFieldForAnnotation` record that has a `DataPoint` target, it is displayed at the top right of the list item, and its value can be highlighted according to the criticality of the datapoint. If no `DataFieldForAnnotation` record is defined, the next `DataField` record is displayed instead.

Number of list items displayed: up to 3.

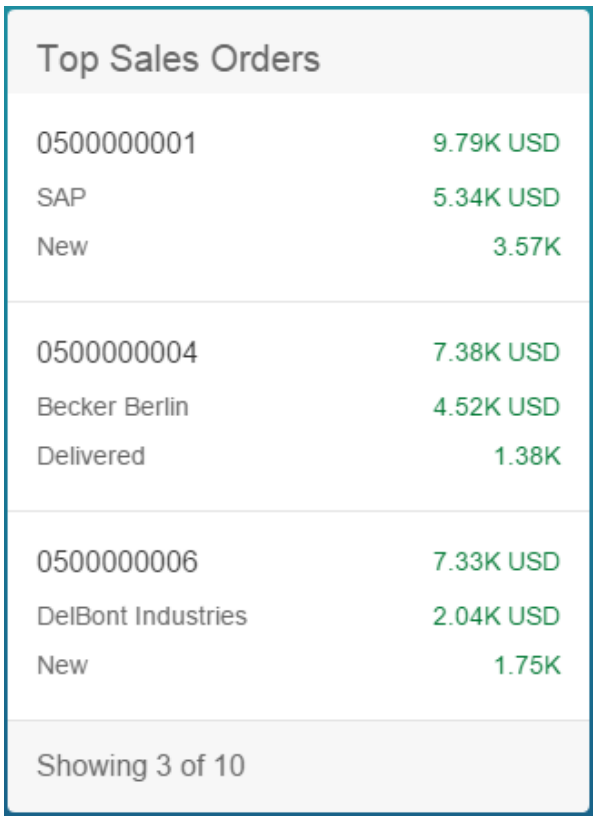
Extended List Type - Standard Flavor

In this type of list, each list item displays up to six fields. `DataField` records are displayed on the left side of the line item, and `DataPoint` records are displayed on the right. If no `DataPoint` record is defined, or less than three `DataPoint` records are available, the right side of the line item displays `DataField` records instead.

Number of list items displayed: up to 6.



Condensed List Type - Standard Flavor



Extended List Type - Standard Flavor

Condensed List Type - Bar Flavor

In this type of list, each list item displays up to three fields. Only use this kind of list if you want to display a `DataPoint` record. The first `DataField` record is displayed as a title for the line item in the top-left field. The first `DataPoint` record is displayed as a bar beneath the title, and as a numeric value to the right of the bar. You can also display an additional `DataPoint` record to the right of the first `DataPoint` record. The bar can display values as a percentage or as any numeric value.

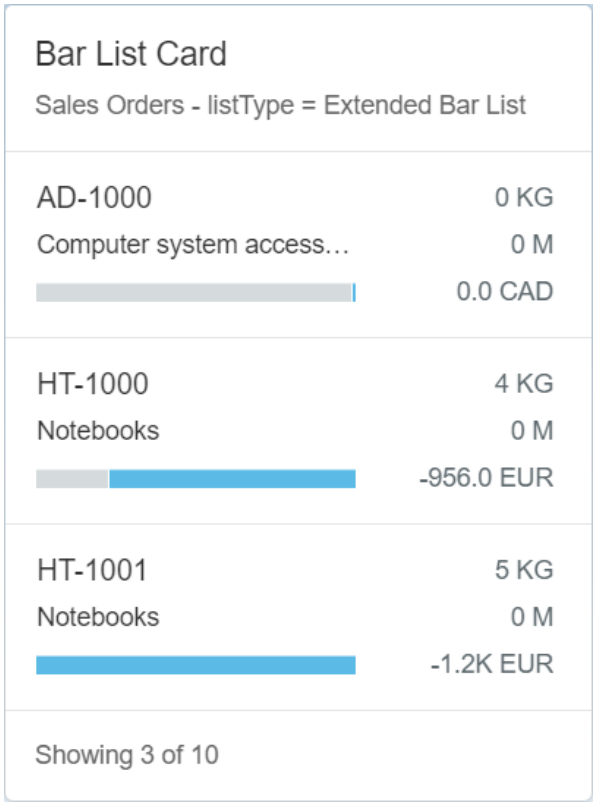
Number of list items displayed: up to 3.

Extended List Type - Bar Flavor

In this type of list, each list item displays up to five fields. The first `DataField` record is displayed as a title for the line item in the top-left field. An additional `DataField` record can be displayed beneath the title. The first `DataPoint` record is displayed as a bar beneath the `DataField` records, and as a numeric value to the right of the bar at the bottom of the line item. You can also display two additional `DataPoint` records to the right of line item, one above the other.

> **i Note**
>
> If highlighting (criticality) is defined in the annotation of the `DataPoint` records, only one will be highlighted in the line item according to the order in which they are displayed.
>
> Number of list items displayed: up to 5.

Condensed List Type - Bar Flavor



Extended List Type - Bar Flavor

Condensed List Card with Images or Icons

To display images or icons in the condensed list card, set the property `"imageSupported": true,` in the descriptor settings and configure the `DataField` property in the `LineItem` annotation. For example:

```
<Annotation Term="com.sap.vocabularies.UI.v1.LineItem">
 <Collection>
 <Record Type="com.sap.vocabularies.UI.v1.DataField">
 <PropertyValue Property="IconUrl" Path="web_address"/>
 <PropertyValue Property="Value" Path="MaterialName"/>
 </Record>
</Collection>
<Annotation>
```

> **i Note**
>
> In list card, an image control is used instead of avatar. For more information, see Using Images, Initials, and Icons [page 1618].

Configuring the List Area

1. Define `DataField` property to display values (text) for the list. For example:

```
<Annotation Term="com.sap.vocabularies.UI.v1.LineItem" Qualifier="View4">
   <Collection>
      <Record Type="com.sap.vocabularies.UI.v1.DataField">
         <PropertyValue Property="Label" String="Order ID" />
         <PropertyValue Property="Value" Path="SalesOrderID" />
      </Record>
      <Record Type="com.sap.vocabularies.UI.v1.DataField">
         <PropertyValue Property="Label" String="Customer" />
         <PropertyValue Property="Value" Path="ToBusinessPartner/
EmailAddress" />
      </Record>
      <Record Type="com.sap.vocabularies.UI.v1.DataField">
         <PropertyValue Property="Label" String="Customer" />
         <PropertyValue Property="Value" Path="CustomerName" />
      </Record>
      <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation">
         <PropertyValue Property="Label" String="TaxAmount" />
         <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#TaxAmount" />
      </Record>
   </Collection>
</Annotation>
```

2. (Optional) Define `DataFieldforAnnotation` property using these annotations:
   - `DataPoint`: To view numeric values in the list. For example:

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint"
Qualifier="TaxAmount">
   <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
      <PropertyValue Property="Title" String="TaxAmount" />
      <PropertyValue Property="Value" Path="TaxAmount" />
   </Record>
</Annotation>
```

   - `Contact`: To view quick view information in the list. For example, you can configure the list card to display contact information as shown here:

```
<Annotation Term="com.sap.vocabularies.Communication.v1.Contact">
   <Record>
      <PropertyValue Property="tel">
         <Collection>
            <Record>
               <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/fax" />
               <PropertyValue Property="uri" Path="FaxNumber" />
            </Record>
            <Record>
               <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/work
com.sap.vocabularies.Communication.v1.PhoneType/pref" />
```

```
                <PropertyValue Property="uri" Path="PhoneNumber" />
             </Record>
          </Collection>
       </PropertyValue>
       <PropertyValue Property="email">
          <Collection>
             <Record>
                <PropertyValue Property="type"
EnumMember="com.sap.vocabularies.Communication.v1.ContactInformationType/
pref com.sap.vocabularies.Communication.v1.ContactInformationType/work" />
                <PropertyValue Property="address" Path="EmailAddress" />
             </Record>
          </Collection>
       </PropertyValue>
    </Record>
</Annotation>
```

3. (Optional) Define navigation properties for list area.
   For configuring navigation information:
   - Use com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation to define intent based navigation to SAP Fiori application.
   - Use com.sap.vocabularies.UI.v1.DataFieldWithUrl term to configure navigation to external apps and websites.

> **i Note**
>
> The recommended way to configure intent-based navigation using `DataFieldForIntentBasedNavigation` property. However, you can also use `DataFieldWithUrl` for navigation to a specific application route that is not configured as target map. The overview page identify this as an intent-based navigation and opens the application in the same tab with relevant context.

Configuring the List Card Header Area

Configure card header properties.

1. Configure card title and subtitle in the descriptor file as shown here:
   Card `title` and `subTitle`

```
"sap.ovp": {
    "globalFilterModel": "salesOrder",
    "globalFilterEntityType": "GlobalFilters",
          ...
          ...
    "cards": {
      "sap.ovp.test_card.card002": {
        "model": "salesOrder",
        "template": "sap.ovp.cards.list",
        "settings": {
          "title": "Purchase History",
          "subTitle": "SubTitle",
          "entitySet": "SalesOrderSet",
          ...
          ...
            }
        ]
      }
    }
```

2. Configure KPI value
   Descriptor Settings

```
"sap.ovp": {//section for ovp-specific app descriptor settings
    ...
    "kpiAnnotationPath":"com.sap.vocabularies.UI.v1.KPI#AllActualCosts", //
Represents the KPI annotation path

     ...
}
```

Sample Code

KPI Annotation

```
<Annotation Term="UI.KPI" Qualifier="AllActualCosts">
   <Record Type="UI.KPIType">
      <PropertyValue Property="Detail">
         <Record Type="UI.KPIDetailType">
            <PropertyValue Property="DefaultPresentationVariant"
Path="@UI.PresentationVariant#Eval_by_Currency1" />
            <PropertyValue Property="AlternativePresentationVariants">
               <Collection>
                  <Path>@UI.PresentationVariant#Eval_by_Currency_Column</
Path>
               </Collection>
            </PropertyValue>
            <PropertyValue Property="SemanticObject" String="Action" />
            <PropertyValue Property="Action" String="toappnavsample" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#Eval_by_Currency_1" />
      <PropertyValue Property="DataPoint"
Path="@UI.DataPoint#Eval_by_Country-Generic" />
      <PropertyValue Property="ID" String="String for KPI Annotation" />
   </Record>
</Annotation>
```

Sample Code

Associated Presentation Variant Annotation

```
<Annotation Term="UI.PresentationVariant" Qualifier="Eval_by_Currency1">
   <Record>
      <PropertyValue Property="MaxItems" Int="5" />
      <PropertyValue Property="GroupBy">
         <Collection>
            <PropertyPath>Country</PropertyPath>
            <PropertyPath>Currency</PropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="SortOrder">
         <Collection>
            <Record>
               <PropertyValue Property="Property"
PropertyPath="TotalSales" />
               <PropertyValue Property="Descending" Boolean="true" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Visualizations">
         <Collection>
```

```
            <AnnotationPath>@UI.Chart#Eval_by_Currency_Donut</
AnnotationPath>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Sample Code

Associated Selection Variant Annotation

```
<Annotation Term="UI.SelectionVariant" Qualifier="Eval_by_Currency_1">
   <Record>
      <PropertyValue Property="SelectOptions">
         <Collection>
            <Record>
               <PropertyValue Property="PropertyName"
PropertyPath="Country" />
               <PropertyValue Property="Ranges">
                  <Collection>
                     <Record>
                        <PropertyValue Property="Sign"
EnumMember="UI.SelectionRangeSignType/I" />
                        <PropertyValue Property="Option"
EnumMember="UI.SelectionRangeOptionType/EQ" />
                        <PropertyValue Property="Low" String="IN" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Parameters">
         <Collection>
            <Record Type="UI.Parameter">
               <PropertyValue Property="PropertyName"
PropertyPath="Currency_Target" />
               <PropertyValue Property="PropertyValue" String="EUR" />
            </Record>
            <Record Type="UI.Parameter">
               <PropertyValue Property="PropertyName"
PropertyPath="UoM_Target" />
               <PropertyValue Property="PropertyValue" String="KGM" />
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Define the `DataPoint` annotation to complete configuring KPI information on the card header area. The following are the annotation properties:

- Add `Title` property to display as a title on the KPI header
- Add `Value` property to display KPI measure
- Add `ValueFormat` to define number format
- Add criticality to highlight the KPI measure value. You can define criticality as a path or enum value. The supported enum values are:
  - `com.sap.vocabularies.UI.v1.CriticalityType/Neutral` - default neutral color is considered
  - `com.sap.vocabularies.UI.v1.CriticalityType/Negative` - Red is considered

- `com.sap.vocabularies.UI.v1.CriticalityType/Critical` - Orange is considered
- `com.sap.vocabularies.UI.v1.CriticalityType/Positive` - Green is considered

You can also define criticality using a path property that returns value:

- 0 for Neutral- default neutral colour is considered
- 1 for Negative – Red is considered
- 2 for Critical – Orange is considered
- 3 for Positive – Green is considered

Sample Code

Associated DataPoint annotation

```
<Annotation Term="UI.DataPoint" Qualifier="Eval_by_Country-Generic">
   <Record Type="UI.DataPointType">
      <PropertyValue Property="Title" String="Sales India - Generic
Card" />
      <PropertyValue Property="Value" Path="Sales" />
      <PropertyValue Property="ValueFormat">
         <Record>
            <PropertyValue Property="ScaleFactor" Int="2" />
            <PropertyValue Property="NumberOfFractionalDigits" Int="1" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="CriticalityCalculation">
         <Record>
            <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Minimizing" />
            <PropertyValue Property="DeviationRangeHighValue"
String="7300" />
            <PropertyValue Property="ToleranceRangeHighValue"
String="7200" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="TargetValue" String="2.000 " />
      <PropertyValue Property="TrendCalculation">
         <Record>
            <PropertyValue Property="ReferenceValue" String="5201680" />
            <PropertyValue Property="DownDifference" Int="10000000.0" />
         </Record>
      </PropertyValue>
   </Record>
</Annotation>
```

3. Configure `tabs` property (view switch) in the descriptor settings to filter/view data at the card level.

```
"cards": {
      "card00": {
        "model": "salesOrder",
        "template": "sap.ovp.cards.list",
        "settings": {
          "category" : "Sales Orders With Analytical Header",
          "listFlavor": "bar",
          "listType": "extended",
          "entitySet": "SalesOrderSet",
          "tabs": [
                {
                    "annotationPath":
"com.sap.vocabularies.UI.v1.LineItem#View1",
                    "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#line1",
                    "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#line",
```

```
                    "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification",
                    "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
                    "value": "{{dropdown_value1}}"
                },
                {
                    "annotationPath":
"com.sap.vocabularies.UI.v1.LineItem#View2",
                    "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#customer",
                    "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#SP2",
                    "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#item2",
                    "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
                    "value": "{{dropdown_value2}}"
                },
```

4. Configure list header navigation.
   The identification annotation lets you configure navigation (from header and list area) within an application.

> **i** Note
>
> In case of list area navigation, the navigation settings configured using the `LineItem` annotation takes precedence over the navigation settings configured using the identification annotation.

Customizing List Card

Adding Smart Links

Define a semantic object for the entity set and its property using the annotation target to add smart links. For example:

```
<Annotations Target="GWSAMPLE_BASIC.SalesOrder/SalesOrderID">
<Annotation Term="com.sap.vocabularies.Common.v1.SemanticObject" String="OVP" />
</Annotations>
```

Text Arrangement in List Area

Define `Text` arrangement annotation to format content on the list area.

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="GWSAMPLE_BASIC.SalesOrder/CustomerID">
<Annotation Term="com.sap.vocabularies.Common.v1.Text" Path="Supplier_Name"/>
```

```
<Annotation Term="com.sap.vocabularies.UI.v1.TextArrangement"
EnumMember="com.sap.vocabularies.UI.v1.TextArrangementType/TextLast" />
</Annotations>
```

In the preceding example, the text `Customer` is bound to the `ContactID` property and appears as shown in the table:

Text Arrangement Type	Result
`TextLast`	`ContractID (Customer)`
`TextFirst`	`Customer (ContractID)`
`TextOnly`	`Customer`

Filtering

You can add filters to list card by using the `com.sap.vocabularies.UI.v1.SelectionVariant` annotation term, or by passing filter parameter in the URL. For more information, see Configuring Card Filters [page 2000]

Sorting or Grouping

You can sort information in the list card by using `sortBy` and `sortOrder` properties in the application descriptor file. For more information, see Configuring Sort Properties [page 2001]

Link List Cards

Allows you to view a list of links with title, picture, icon, or subtitle.

Link List card supports the following navigation types:

- QuickView link: To view contact information from a collection. Example, Recent contact list
- Cross App link: To access related applications. Example, intent-based navigation
- External URL link

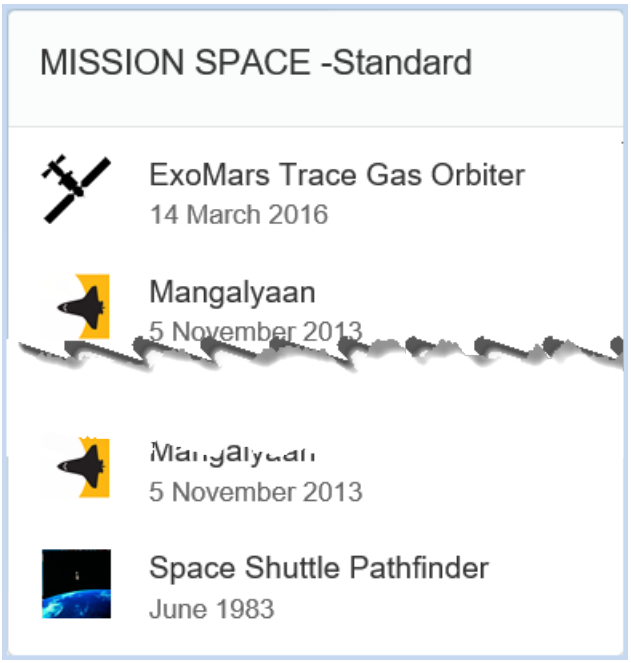
Link List Card Types

Supports standard and carousel link list card types.

Standard

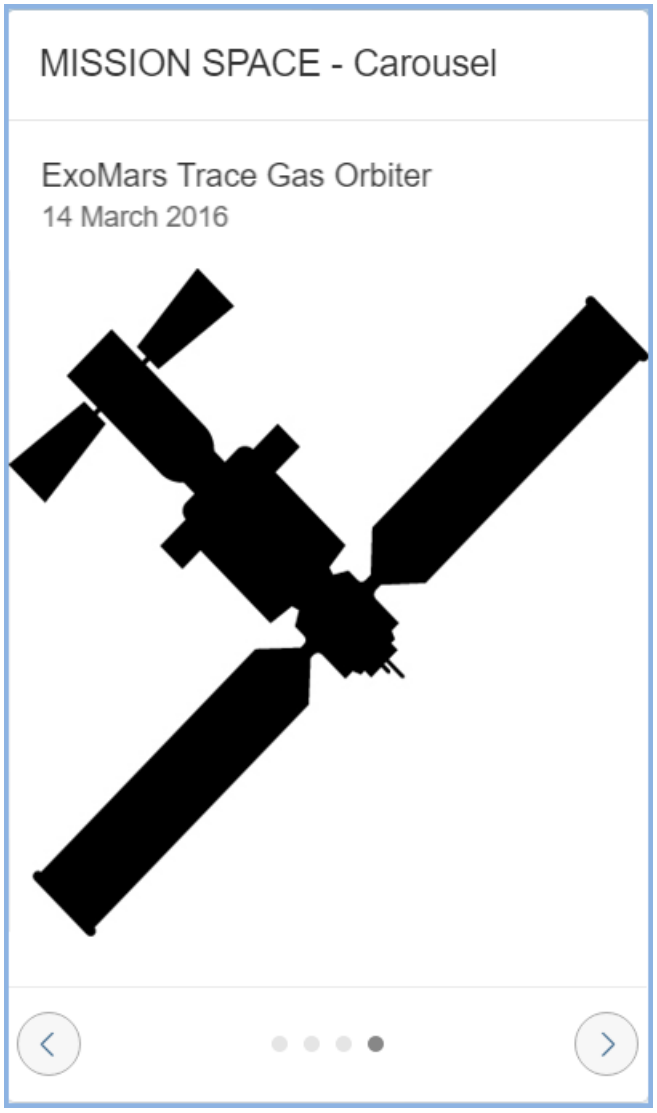
Provides an ordered list items. (Optional) You can

- Add a picture and/or subtitle to the list item.
- Configure the card size to view multiple columns, if the dashboard layout has multiple columns.



Carousel

Provides carousel based view. Title and subtitle appears on top of the card, allowing more space for image. You can also configure this list type with multiple columns.



You can customize the information in the link list to appear in the following ways:

- One or two line(s) of
  - Text only
  - Text with icon supports image control
  - Text with image supports avatar control
- Interaction States: You can add more interactions to the link list.

Annotation Examples

Standard Link List

```
"card20": {
    "model": "salesOrder",
    "template": "sap.ovp.cards.linklist",
    "settings": {
        "title": "Dynamic Linklist Card",
```

```
        "targetUri": "https://en.abc.org/wiki/A_2",
        "subtitle": "NEW CARD",
        "listFlavor": "standard",
        "sortBy": "Name",
        "entitySet": "ProductSet",
        "sortOrder": "ascending",
        "headerAnnotationPath": "com.sap.vocabularies.UI.v1.HeaderInfo#header1",
        "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#identify1",
    }
}
```

Carousel

```
"card_20": {
    "model": "HEPM_OVP_TECH_VAL",
    "template": "sap.ovp.cards.linklist",
    "settings": {
        "title": "{{card20_title}}",
        "entitySet": "Hepmra_C_OFT_Employee",
        "listFlavor": "carousel",
        "sortBy": "Employee",
        "sortOrder": "ascending",
    }
},
```

Configuring Static Link List Card

Lets you display static data (static links and images/icons) in the form of list items.

You can set the `listFlavor` property in the card settings to display information in standard link list type or as a carousel link list type.

> **i Note**
>
> To add annotations, use the SAP WebIDE annotation modular or code editor. For more information, see Building an App Using SAP Web IDE [page 1553].

1. Configure card `title`, `subtitle`, and `listFlavor`.
2. Configure staticContent properties such as `title`, `subtitle`, `imageUri`, `imageAltText`
3. Configure navigation properties
   - Url based navigation: Define `targetUri` and `openInNewWindow` properties

> **Sample Code**
>
> Target Uri Based Navigation

```
"card010_QuickLinks": {
  "model": "salesOrder",
  "template": "sap.ovp.cards.linklist",
  "settings": {
    "title": "Quick Links",
    "subTitle": "Standard Link List With Static Data",
    "listFlavor": "standard",
    "staticContent": [
      {
        "title": "Jim Smith",
        "subTitle": "Sales",
```

```
        "imageUri": "img/JD.png",
        "imageAltText": "Jim Smith",
        "targetUri": "https://abc.com",
        "openInNewWindow": true
      },
      {
        "title": "Alice Wilson",
        "subTitle": "Sales",
        "imageUri": "img/AW.png",
        "imageAltText": "Jim Smith",
        "targetUri": "https://abc.com",
        "openInNewWindow": true
      }
    ]
  }
}
```

- IntentBasedNavigation: Define `semanticObject` and `action` properties

Sample Code

IntentBasedNavigation

```
"card010_QuickLinks": {
  "model": "salesOrder",
  "template": "sap.ovp.cards.linklist",
  "settings": {
    "title": "Quick Links",
    "subTitle": "Standard Link List With Static Data",
    "listFlavor": "standard",
    "defaultSpan": {
      "rows": 15,
      "cols": 1
    },
    "staticContent": [
      {
        "title": "Create Purchase Order",
        "imageUri": "sap-icon://Fiori6/F0865",
        "imageAltText": "{{card30_icon_prod_man}}",
        "semanticObject": "Action",
        "action": "toappnavsample",
        "disableInCard": true
      },
      {
        "title": "Create Supplier",
        "imageUri": "sap-icon://Fiori2/F0246",
        "imageAltText": "{{card30_icon_so_man}}",
        "semanticObject": "Action",
        "action": "toappnavsample"
      },
      {
        "title": "Create Contact",
        "imageUri": "sap-icon://Fiori6/F0866",
        "imageAltText": "{{card30_icon_so_man}}",
        "semanticObject": "Action",
        "action": "toappnavsample"
      }
    ]
  }
},
```

Configuring Dynamic Link List Card

Lets you display data (links and images/icons) in the form of list items.

You can set the `listFlavor` property in the card settings to display information in standard link list type or as a carousel link list type.

Additionally, dynamic link list card supports contact annotation as default annotation without qualifier to enable quick view information.

> **i Note**
>
> To add annotations, use the SAP WebIDE annotation modular or code editor. For more information, see Building an App Using SAP Web IDE [page 1553].

1. 1. Configure descriptor settings

```
        "card017": {
          "model": "salesOrder",
          "template": "sap.ovp.cards.linklist",
          "settings": {
            "title": "Standard Dynamic Linklist Card",
            "targetUri": "https://en.abc.org/wiki/xyz _2",
            "subTitle": "Smartlink Feature Test",
            "listFlavor": "standard",
            "entitySet": "ProductSet",
            "sortBy": "Name",
            "sortOrder": "ascending",
            "headerAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#header1",
            "defaultSpan": {
              "rows": 20,
              "cols": 2
            }
          }
        },
```

2. Configure list information in the `UI.HeaderInfo` annotation.
   - List title: Set the `Title` property.
   - List subtitle: Set the `Description` property.
   - List item picture: Set the `ImageUrl` property to display list image. Or, set the `typeImageUrl` property to display list icon.
   - Carousel picture: Configuration is similar to the list item picture. Additionally, the carousel picture reacts to a click event similar to the list title.

```
<Annotation Term="com.sap.vocabularies.UI.v1.HeaderInfo" Qualifier="header1">
   <Record Type="com.sap.vocabularies.UI.v1.HeaderInfoType">
      <PropertyValue Property="TypeName" String="Product" />
      <PropertyValue Property="TypeNamePlural" String="Products" />
      <PropertyValue Property="Title">
         <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Product Name" />
            <PropertyValue Property="Value" Path="Name" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="Description">
         <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Product Description" />
            <PropertyValue Property="Value" Path="Description" />
         </Record>
```

```
      </PropertyValue>
      <PropertyValue Property="TypeImageUrl" Path="ImageUrl" />
   </Record>
</Annotation>
```

3. Configure contact annotation
   A contact card is displayed as a popover. The data shown on the contact card are taken from the communication contact annotation of the card. Currently the following elements of the contact annotation are evaluated:
   - `fn` (Full name - used as headline of the contact card header)
     `photo` (URL for a picture – used in the contact card header)
     `role` (used as the description text in the contact card header)
     `tel` (with property type “work” and “pref” the contact card shows it as “Phone”, with property type “cell” and “work” it is shown as “Mobile”
     `email` (with property type “work” and “pref” the contact card shows it as “E-Mail”)

```
<Annotation Term="com.sap.vocabularies.Communication.v1.Contact">
   <Record>
       <PropertyValue Property="fn" Path="FullName" />
      <PropertyValue Property="title" Path="Title" />
      <PropertyValue Property="org" Path="CompanyName" />
      <PropertyValue Property="role" Path="OrganizationRole" />
      <PropertyValue Property="tel">
         <Collection>
            <Record>
               <PropertyValue Property="type"
                  EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/
fax" />
               <PropertyValue Property="uri" Path="FaxNumber" />
            </Record>
            <Record>
               <PropertyValue Property="type"
                  EnumMember="com.sap.vocabularies.Communication.v1.PhoneType/
work com.sap.vocabularies.Communication.v1.PhoneType/pref" />
               <PropertyValue Property="uri" Path="PhoneNumber" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="email">
         <Collection>
            <Record>
               <PropertyValue Property="type"

EnumMember="com.sap.vocabularies.Communication.v1.ContactInformationType/pref
com.sap.vocabularies.Communication.v1.ContactInformationType/work" />
               <PropertyValue Property="address" Path="EmailAddress" />
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

4. Configure navigation type.
   Link list card supports the following navigation types:
   - Intent based navigation: Define `SemanticObject` and `Action` properties in the `UI.Identification` annotation to set the navigation target.
   - Function import (`DataFieldWithAction`): Define `Action` property in the `LineItem` annotation to trigger a function import that performs an OData action for an entity.
   - External navigation: Define `UI.LineItem` of type `WITH_URL`. The URL for this navigation is taken from the entity type field that is named in the line item’s `url` property.

Table 95: Navigation Behavior

Annotation	Behavior
For any of the these annotation configuration: ○ `UI.Identification` annotation with property `Action` ○ `com.sap.vocabularies.UI.v1.LineItem` annotation with property `Action` ○ `com.sap.vocabularies.UI.v1.LineItem` annotation with property `Url`	Navigation is available from line item
For any of the these annotation configuration + default contact annotation: ○ `UI.Identification` annotation with property `Action` ○ `com.sap.vocabularies.UI.v1.LineItem` annotation with property `Action` ○ `com.sap.vocabularies.UI.v1.LineItem` annotation with property `Url`	Navigation is available from line item and you can see quick view information on click of the title.
For only default contact annotation configuration	Quick view information is available on click of title
If no navigation based annotation or contact annotation	Only label is displayed

Stack Cards

Stack cards aggregate a set of cards of the same type, which are based on a common topic or action. When clicked, up to 20 stacked cards can be displayed in the object stream.

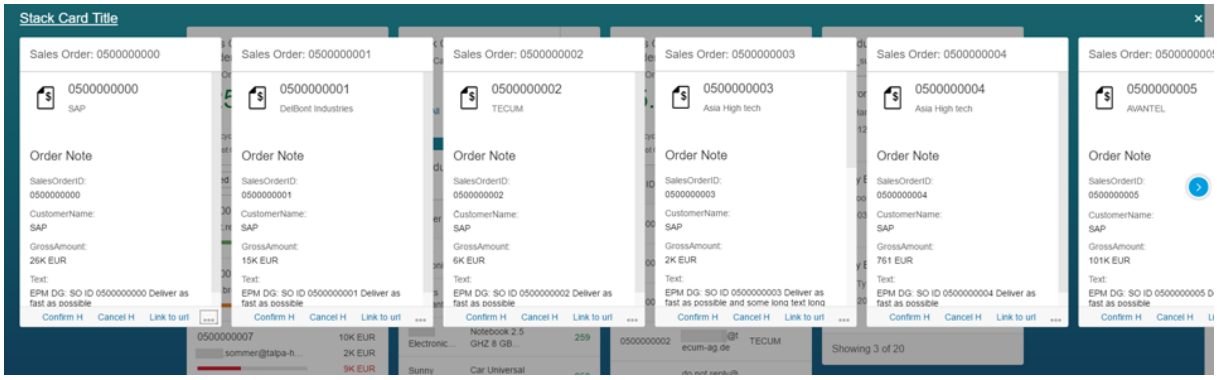
The left-hand side of the card contains the application title (which is also the title of the object stream) and stack description (optional). Click this section of the card or *View All* to open the application.

Object Stream

On the right-hand side of the card you can view the number of items in the stack. Click this section to view the object stream (up to 20 quick view cards) excluding the placeholder card that appears as the last card in the object stream).

> **i** Note
>
> The placeholder card provides additional information and appears only when the object stream has 20 quick view cards.

You can configure the quick view cards to provide actions (such as confirm or reject) and a navigation link. The object stream's header is also navigable and navigates to the same destination as the navigation from the header of the stack card.



Because of the relationship between the stack card and its object stream, some of the configurations for the object stream cards are included in the stack card definitions.

- `entitySet` – the dominant entity set (for example, `SalesOrderSet`).
- `objectStreamCardsNavigationProperty` – the navigation property used to display information from a secondary entity set. For example, to display sales orders by business partner, the entity set would be `BusinessPartnersSet` and the navigation property `SalesOrder`.

  > **i Note**
  >
  > This definition is not relevant for quick view cards.

- `objectStreamCardsSettings` – an optional configuration of additional settings for the cards displayed in the object stream.
  A `showFirstActionInFooter` flag is added to the `objectStreamCardsSettings` object in the stack card. The default value of the flag is `false`. If this flag is set to *true*, the first action on the footer of the quick view card will be a navigation action. If the flag is set to *false*, the navigation action on the footer of the quick view card won't be displayed. Other actions will be shown instead.
- `itemText` - lets you include custom text along with the existing message in the placeholder card.

Example

In the following example, the stack card displays information about business partners in a quick view card, using the information configured in `com.sap.vocabularies.UI.v1.Identification` with a qualifier.

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
        "card00": {
            "model": "salesOrder",
                "template": "sap.ovp.cards.stack",
                "settings": {
                    "title": "Stack Card Title",
                    "subTitle": "Stack Card",
                    "requireAppAuthorization": "#Action-toappnavsample",
                    "itemText": "items awaiting approval",
                    "entitySet": "SalesOrderSet",
                    "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification,com.sap.vocabularies.UI.v1.Identifi
cation#item2",
```

```
                    "objectStreamCardsSettings": {
                        "showFirstActionInFooter": false
                    }
                }
        },
        ...
    }
}
```

Continuing the example above, the following metadata demonstrates the relationship between the `BusinessPartner` and `SalesOrder` entity sets. The `BusinessPartner` entity type contains the configured navigation `ToSalesOrders` property to the `SalesOrder` using the `Assoc_BusinessPartner_SalesOrders` association and the `Assoc_BusinessPartner_SalesOrders_AssocS` association set.

Sample Code

```
<EntityType Name="BusinessPartner" sap:content-version="1">
    <Key>
        <PropertyRef Name="BusinessPartnerID"/>
    </Key>
    <Property Name="Address" Type="GWSAMPLE_BASIC.CT_Address"
Nullable="false"/>
    <Property Name="BusinessPartnerID" Type="Edm.String" Nullable="false"
MaxLength="10" sap:label="Bus. Part. ID" sap:creatable="false"
sap:updatable="false"/>
    <Property Name="CompanyName" Type="Edm.String" MaxLength="80"
sap:label="Company Name"/>
    <Property Name="WebAddress" Type="Edm.String" sap:label="Web Address"
sap:sortable="false" sap:filterable="false" sap:semantics="url"/>
    <Property Name="EmailAddress" Type="Edm.String" MaxLength="255"
sap:label="E-Mail Address" sap:semantics="email"/>
    <Property Name="PhoneNumber" Type="Edm.String" MaxLength="30"
sap:label="Phone No." sap:semantics="tel"/>
    <Property Name="FaxNumber" Type="Edm.String" MaxLength="30"
sap:label="Fax Number"/>
    <Property Name="LegalForm" Type="Edm.String" MaxLength="10"
sap:label="Legal Form"/>
    <Property Name="CurrencyCode" Type="Edm.String" MaxLength="5"
sap:label="Currency" sap:semantics="currency-code"/>
    <Property Name="BusinessPartnerRole" Type="Edm.String" MaxLength="3"
sap:label="Bus. Part. Role"/>
    <Property Name="CreatedAt" Type="Edm.DateTime" Precision="7"
sap:label="Time Stamp" sap:creatable="false" sap:updatable="false"/>
    <Property Name="ChangedAt" Type="Edm.DateTime" Precision="7"
ConcurrencyMode="Fixed" sap:label="Time Stamp" sap:creatable="false"
sap:updatable="false"/>
    <NavigationProperty Name="ToSalesOrders"
Relationship="GWSAMPLE_BASIC.Assoc_BusinessPartner_SalesOrders"
FromRole="FromRole_Assoc_BusinessPartner_SalesOrders"
ToRole="ToRole_Assoc_BusinessPartner_SalesOrders"/>
    <NavigationProperty Name="ToContacts"
Relationship="GWSAMPLE_BASIC.Assoc_BusinessPartner_Contacts"
FromRole="FromRole_Assoc_BusinessPartner_Contacts"
ToRole="ToRole_Assoc_BusinessPartner_Contacts"/>
    <NavigationProperty Name="ToProducts"
Relationship="GWSAMPLE_BASIC.Assoc_BusinessPartner_Products"
FromRole="FromRole_Assoc_BusinessPartner_Products"
ToRole="ToRole_Assoc_BusinessPartner_Products"/>
</EntityType>
<EntityType Name="SalesOrder" sap:content-version="1">
    <Key>
        <PropertyRef Name="SalesOrderID"/>
    </Key>
```

```
    <Property Name="SalesOrderID" Type="Edm.String" Nullable="false"
MaxLength="10" sap:label="Sa. Ord. ID" sap:creatable="false"
sap:updatable="false"/>
    <Property Name="SalesOrderGuid" Type="Edm.Guid" Nullable="false"
sap:label="SalesOrder GUID" sap:creatable="false" sap:updatable="false"/>
    <Property Name="Note" Type="Edm.String" MaxLength="255"
sap:label="Description" sap:updatable="false" sap:sortable="false"
sap:filterable="false"/>
    <Property Name="NoteLanguage" Type="Edm.String" MaxLength="1"
sap:label="Language" sap:creatable="false" sap:updatable="false"
sap:sortable="false" sap:filterable="false"/>
    <Property Name="CustomerID" Type="Edm.String" MaxLength="10"
sap:label="Bus. Part. ID" sap:updatable="false"/>
    <Property Name="CustomerName" Type="Edm.String" MaxLength="80"
sap:label="Company Name" sap:creatable="false" sap:updatable="false"/>
    <Property Name="CurrencyCode" Type="Edm.String" MaxLength="5"
sap:label="Currency" sap:updatable="false" sap:semantics="currency-code"/>
    <Property Name="GrossAmount" Type="Edm.Decimal" Precision="16" Scale="3"
sap:unit="CurrencyCode" sap:label="Gross Amt." sap:creatable="false"
sap:updatable="false"/>
    <Property Name="NetAmount" Type="Edm.Decimal" Precision="16" Scale="3"
sap:unit="CurrencyCode" sap:label="Net Amt." sap:creatable="false"
sap:updatable="false"/>
    <Property Name="TaxAmount" Type="Edm.Decimal" Precision="16" Scale="3"
sap:unit="CurrencyCode" sap:label="Tax Amt." sap:creatable="false"
sap:updatable="false"/>
    <Property Name="LifecycleStatus" Type="Edm.String" MaxLength="1"
sap:label="PO Lifecycle" sap:creatable="false" sap:updatable="false"/>
    <Property Name="LifecycleStatusDescription" Type="Edm.String"
MaxLength="60" sap:label="Lifecycle Descript." sap:creatable="false"
sap:updatable="false" sap:sortable="false" sap:filterable="false"/>
    <Property Name="BillingStatus" Type="Edm.String" MaxLength="1"
sap:label="PO Confirmation" sap:creatable="false" sap:updatable="false"/>
    <Property Name="BillingStatusDescription" Type="Edm.String"
MaxLength="60" sap:label="Billing Description" sap:creatable="false"
sap:updatable="false" sap:sortable="false" sap:filterable="false"/>
    <Property Name="DeliveryStatus" Type="Edm.String" MaxLength="1"
sap:label="PO Ordering" sap:creatable="false" sap:updatable="false"/>
    <Property Name="DeliveryStatusDescription" Type="Edm.String"
MaxLength="60" sap:label="Delivery Description" sap:creatable="false"
sap:updatable="false" sap:sortable="false" sap:filterable="false"/>
    <Property Name="CreatedAt" Type="Edm.DateTime" Precision="7"
sap:label="Time Stamp" sap:creatable="false" sap:updatable="false"/>
    <Property Name="ChangedAt" Type="Edm.DateTime" Precision="7"
sap:label="Time Stamp" sap:creatable="false" sap:updatable="false"/>
    <NavigationProperty Name="ToBusinessPartner"
Relationship="GWSAMPLE_BASIC.Assoc_BusinessPartner_SalesOrders"
FromRole="ToRole_Assoc_BusinessPartner_SalesOrders"
ToRole="FromRole_Assoc_BusinessPartner_SalesOrders"/>
    <NavigationProperty Name="ToLineItems"
Relationship="GWSAMPLE_BASIC.Assoc_SalesOrder_SalesOrderLineItems"
FromRole="FromRole_Assoc_SalesOrder_SalesOrderLineItems"
ToRole="ToRole_Assoc_SalesOrder_SalesOrderLineItems"/>
</EntityType>
<Association Name="Assoc_BusinessPartner_SalesOrders" sap:content-version="1">
    <End Type="GWSAMPLE_BASIC.BusinessPartner" Multiplicity="1"
Role="FromRole_Assoc_BusinessPartner_SalesOrders"/>
    <End Type="GWSAMPLE_BASIC.SalesOrder" Multiplicity="*"
Role="ToRole_Assoc_BusinessPartner_SalesOrders"/>
    <ReferentialConstraint>
        <Principal Role="FromRole_Assoc_BusinessPartner_SalesOrders">
            <PropertyRef Name="BusinessPartnerID"/>
        </Principal>
        <Dependent Role="ToRole_Assoc_BusinessPartner_SalesOrders">
            <PropertyRef Name="CustomerID"/>
        </Dependent>
    </ReferentialConstraint>
</Association>
```

```
<EntityContainer Name="GWSAMPLE_BASIC_Entities"
m:IsDefaultEntityContainer="true">
    <EntitySet Name="BusinessPartnerSet"
EntityType="GWSAMPLE_BASIC.BusinessPartner" sap:content-version="1"/>
    <EntitySet Name="SalesOrderSet" EntityType="GWSAMPLE_BASIC.SalesOrder"
sap:updatable="false" sap:content-version="1"/>
    <AssociationSet Name="Assoc_BusinessPartner_SalesOrders_AssocS"
Association="GWSAMPLE_BASIC.Assoc_BusinessPartner_SalesOrders"
sap:creatable="false" sap:updatable="false" sap:deletable="false" sap:content-
version="1">
        <End EntitySet="BusinessPartnerSet"
Role="FromRole_Assoc_BusinessPartner_SalesOrders"/>
        <End EntitySet="SalesOrderSet"
Role="ToRole_Assoc_BusinessPartner_SalesOrders"/>
    </AssociationSet>
</EntityContainer>
```

Quick View Cards

Quick view cards display detailed information about a single record, in greater depth than would be displayed in a table or list.

The information displayed in quick view cards is configured in the `com.sap.vocabularies.UI.v1.HeaderInfo` and `com.sap.vocabularies.UI.v1.Facets` terms. The `com.sap.vocabularies.UI.v1.HeaderInfo` term is used to present the entity header information, and the `com.sap.vocabularies.UI.v1.Facets` term is used to present detailed information about the record. Quick view card supports the following properties:

- `TypeNamePlural`
- `ImageUrl`
- `Title`
- `Description`

The content area shows `FieldGroup` records from `referenceFacet` items that have been tagged using the `IsSummary` annotation in the `com.sap.vocabularies.UI.v1.Facets` annotation.

The `com.sap.vocabularies.UI.v1.Facets` term can be configured in the application manifest file by setting the `annotationPath` property with a qualifier, as shown in the example below. If the `annotationPath` property is not configured, the `com.sap.vocabularies.UI.v1.Facets` term, without a qualifier, is used.

Card Footer

You can assign actions to the quick view cards that open in the object stream of the stack card. The actions are displayed as buttons in the card footer.

From `com.sap.vocabularies.UI.v1.Identification`, there are two kinds of actions that you can display in the card footer:

- `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`
- `com.sap.vocabularies.UI.v1.DataFieldForAction`

Depending on the number of assigned actions, they are displayed in the footer as follows:

- If there is only one action, the button is aligned to the right of the footer area.
- If there are two actions, such as *OK* and *Cancel*, they are presented according to their importance and order of entry. We recommend providing the annotation in a such a way that the positive action is on the left and the negative action is on the right.
- If there are more than two actions, an *Actions* button is displayed. Clicking the *Actions* button opens an `ActionSheet` control with all of the defined actions in the order that they appear in the metadata.
- If no actions are assigned to the card, the footer area is not displayed.

Each OData action is translated into a request according to the annotation. Clicking an action invokes the OData service request. Complex actions open a dialog box with the action parameters or a confirmation message.

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
        ...
        "card01": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.table",
            "settings": {
                ...
                "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#Qualifier1",
                ...
            }
        },
        ...
    }
}
```

Interaction buttons in the footer area are part of `sap.m.OverflowToolbar` so that quick view cards can display action buttons based on the width of the card, and if more actions are necessary, the remaining actions are shown in the overflow toolbar.

Note

Cards can have different sizes because the height of each quick view card is aligned with the content of the card. If there is more content than can be shown in the card, you'll be able to scroll vertically, but only within the content area itself. The header and footers stay fixed.

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
        "card00": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.stack",
            "settings": {
                "category": "{{card01_category}}",
                "entitySet": "BusinessPartners",
                "objectStreamCardsSettings": {
                    "annotationPath":
"com.sap.vocabularies.UI.v1.Facets#Qualifier2"
                }
            }
        },
```

```
            ...
        }
    }
```

Analytical Cards

Analytical cards let you view data in a variety of chart formats. The card is divided into two areas (header and chart).

In the header area, you can view the aggregated value of a key performance indicator (KPI), the trend, and percentage of change. In the chart area, you can view a graphical representation of the data.

> **i** Note
>
> If you configure the chart title, chart descriptions (x- and y-axis) are not displayed in the card (except for bubble charts). For example, the chart title *Net Sales by Days Payable*, already conveys that the y-axis is *Net Sales* and the x-axis represents *Days Payable*.

Chart Responsiveness

To improve the responsiveness of charts, you can use `UI.PresentationVariant.MaxItems` to limit the number of records fetched from the backend that are rendered on the screen. For example:

> Sample Code

```
<Annotation Term="UI.PresentationVariant"
Qualifier="Column_Eval_by_Country_123">
    <Record>
          <PropertyValue Property="MaxItems" Int="5" />
    </Record>
</Annotation>
```

> → Tip
>
> If there are too many data records displayed in the chart, it is difficult to select a data point. If you are only interested in checking the top records, then use `UI.PresentationVariant.SortOrder`. For more information, see Configuring Card Properties [page 1995].

Chart Interactivity

Users can choose specific data points to pass data in URL parameters to the target application. The target application can read these URL parameters and use them as required (typically to filter the data set that they are displaying).

When a user select a particular data point, the system passes a technical ID (`"RegionID"="001"`) instead of the display name (`"Region"="EMEA"`).

Axis Scaling

Axis scaling lets you scale and display data for line, bubble, or scatter charts in the analytics card. You can choose any of the following axis scaling types:

Type 1: Default property

The default behavior displays data in the respective chart format including the zero value. The graph is adjusted according to the available data range. Set the following property in the `UI.Chart` annotation:

```
<PropertyValue Property="AxisScaling" EnumMember="UI.ChartAxisScalingType/
AdjustToDataIncluding0"/>
```

Type 2: Adjust scale property

The adjust scale property displays data in the respective chart format based on the available data range. Set the following property in the `UI.Chart` annotation:

```
<PropertyValue Property="AxisScaling" EnumMember="UI.ChartAxisScalingType/
AdjustToData"/>
```

Type 3: Min-Max property

The min-max property lets you define the minimum and maximum data range to display data in the respective chart format. Set the following property in the `UI.Chart` annotation:

```
<PropertyValue Property="AxisScaling" EnumMember="UI.ChartAxisScalingType/
MinMaxValues"/>
```

You must define the `DataPoint` annotation to set the minimum and maximum values.

```
<Annotation Term="UI.DataPoint" Qualifier="<DataPoint Qualifier>">
  <Record Type="UI.DataPointType">
    <PropertyValue Property="MinimumValue" Decimal="<Minimum Value>" />
    <PropertyValue Property="MaximumValue" Decimal="<Maximum Value>" />
    <PropertyValue Property="NumberFormat">
     <Record>
     .......
     </Record>
     </PropertyValue>
   </Record>
</Annotation>
```

Descriptor Configuration Sample

Sample Code

The following is a snippet of a sample descriptor file:

```
"sap.app": {
   "_version": "1.1.0",
   "id": "sap.ovp.demo",
   "type": "application",
   "i18n": "i18n/i18n.properties",
   "applicationVersion": {
                           "version": "1.2.2"
                         },
   "title": "{{app_title}}",
   "description": "{{app_description}}",
   "dataSources": {
      "salesShare": {
                        "uri": "https://abc.com/SalesShare.xsodata",
```

```
                        "type": "OData",
                        "settings": {
                                     "odataVersion": "2.0",
                                     "annotations": [
                                     "salesShareAnno"
                                     ]
                                    }
                       },
       "salesShareAnno": {
                        "uri": "data/salesshare/annotations.xml",
                        "type": "ODataAnnotation",
                        "settings": {
                                    }
                          }
                    }
                }
"sap.ovp": {
    "globalFilterModel": "salesShare",
    "globalFilterEntityType": "GlobalFilters",
    "cards": {
               "cardBubble": {
                               "model": "salesShare",
                               "template": "sap.ovp.cards.charts.analytical",
                               "settings": {
                                            "entitySet": "SalesShare",
                                            "selectionAnnotationPath" :
"com.sap.vocabularies.UI.v1.SelectionVariant#Eval_by_CtryCurr",
                                            "chartAnnotationPath" :
"com.sap.vocabularies.UI.v1.Chart#Eval_by_CtryCurr",
                                            "presentationAnnotationPath" :
"com.sap.vocabularies.UI.v1.PresentationVariant#Eval_by_CtryCurr",
                                            "dataPointAnnotationPath" :
"com.sap.vocabularies.UI.v1.DataPoint#Eval_by_CtryCurr",
                                            "identificationAnnotationPath" :
"com.sap.vocabularies.UI.v1.Identification#Eval_by_CtryCurr"
                                           }
                              },
                "cardchartsline": {
                                   "model": "salesShare",
                                   "template":
"sap.ovp.cards.charts.analytical",
                                   "settings": {
                                                "entitySet": "SalesShare",
                                                "selectionAnnotationPath" :
"com.sap.vocabularies.UI.v1.SelectionVariant#Eval_by_Country",
                                                "chartAnnotationPath" :
"com.sap.vocabularies.UI.v1.Chart#Eval_by_Country",

"presentationAnnotationPath" :
"com.sap.vocabularies.UI.v1.PresentationVariant#Eval_by_Country",
                                                "dataPointAnnotationPath" :
"com.sap.vocabularies.UI.v1.DataPoint#Eval_by_Country",

"idenfiticationAnnotationPath" :
"com.sap.vocabularies.UI.v1.Identification#Eval_by_Country"
                                                }
                                          },
                 "cardchartsdonut": {
                                     "model": "salesShare",
                                     "template":
"sap.ovp.cards.charts.analytical",
                                     "settings": {
                                                  "entitySet": "SalesShare",

"selectionAnnotationPath" :
"com.sap.vocabularies.UI.v1.SelectionVariant#Eval_by_Currency",
                                                  "chartAnnotationPath" :
"com.sap.vocabularies.UI.v1.Chart#Eval_by_Currency",
```

```
"presentationAnnotationPath" :
"com.sap.vocabularies.UI.v1.PresentationVariant#Eval_by_Currency",

"dataPointAnnotationPath" :
"com.sap.vocabularies.UI.v1.DataPoint#Eval_by_Currency",

"idenfiticationAnnotationPath" :
"com.sap.vocabularies.UI.v1.Identification#Eval_by_Currency"
                                                    }
                                        }
                                    }
                }
```

Related Information

For more information about configuring charts, see Configuring Charts [page 1990].

Chart Cards Used in Overview Pages

This section describes the analytic chart cards you can use in overview pages.

> **i Note**
>
> Analytic cards don't have a fixed height. The height is adjusted automatically to accommodate the data points and legends. Legends are created automatically based on the defined measures and dimensions.

Line Chart

Line charts display information as a series of data points connected by straight-line segments. They are often used to visualize a trend in data over time. Line charts need at least one measure and one dimension.

- Dimensions for which the role is set to `category` make up the x-axis (category axis). If no dimension is specified with this role, the first dimension is used as the x-axis. We recommend using only time-based dimensions (for example, day, date, quarter, or year) for the category axis of a line chart.
- Dimensions for which the role is set to `series` make up the line segments of the chart, with different colors assigned to each dimension value. If multiple dimensions are assigned to this role, the values of all the dimensions together are considered as one dimension and a color is assigned.
- Measures make up the y-axis (value axis). If there are multiple measures, then each measure is represented by a different colored line in the chart area.

The line chart supports a color palette for semantic coloring.

Bubble Chart

Bubble charts display up to three measures and two dimensions of data. The three measures are reflected in the x- and y-axes, and in the size of the bubbles. The dimensions can be expressed in the colors and/or shapes of the bubbles. Bubble charts need to have three measures and one or two dimensions.

- The first measure for which the role is set to an axis is assigned to the `valueAxis` feed. The UID makes up the x-axis.

  > **i** Note
  >
  > The role is set to `axis1`, `axis2` (if there's no `axis1`), or `axis3` (if there's no `axis2`.)

- The first measure for which the role is set to an axis is assigned to the `valueAxis2` feed and the UID makes up the y-axis.

  > **i** Note
  >
  > The role is set to `axis2` of the first measure. If there's no `axis2`, the role is set to `axis1` of the second measure. If there's no axis1 for the second measure, the role is set to `axis3` of the first measure.

- The remaining measure is assigned to the `bubbleWidth` feed'sUID. This determines the size of the bubble.
- The dimensions for which the role is set to `series` are assigned to the feed's UID **color**. Different values for this dimension in the data set result in different colored data points in the chart. If multiple dimensions are set with the category role, only the first dimension is considered. If the role **series** is assigned to both dimensions, then each combination of the dimension member gets a unique color.
  For example, if role **series** is assigned to the dimensions "Year" and "Country" then "India/2015", "India/2016", "Germany/2015", "Germany/2016" are represented as different colored bubbles. If no role is assigned to a dimension, then the dimension members gets the same color. In the above example, if no color has been assigned to a year, then the bubbles only have two colors - one for all records for India and one for all records for Germany, irrespective of the year.

  > **i** Note
  >
  > Assigning the role of a dimension as a category leads to differently shaped data points for different values of the dimension. However, we do not recommend this for a bubble chart card.

- The dimensions for which the role is set to a **category** are assigned to the **shape** feed's UID. Different values for this dimension in the data set result in differently shaped data points in the chart. If multiples dimensions are set with the category role, only the first dimension is considered.

The code snippet shown below demonstrates how you define a bubble chart card. Note that there are three measures (under the `MeasureAttributes` property) and one dimension (under the `DimensionAttributes` property).

Example

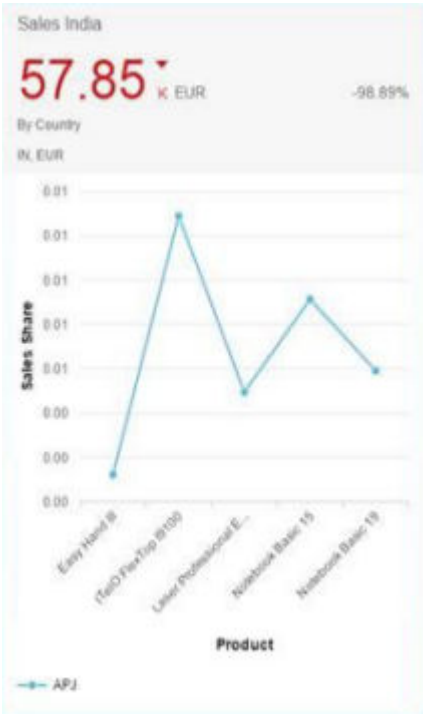
> Sample Code

```
<Annotation Term="UI.Chart" Qualifier="Qualifier_ID_1">
<Record Type="UI.ChartDefinitionType">
                                <PropertyValue Property="Title"
String="View1" />
                                <PropertyValue Property="ChartType"
EnumMember="UI.ChartType/Bubble"/>
```

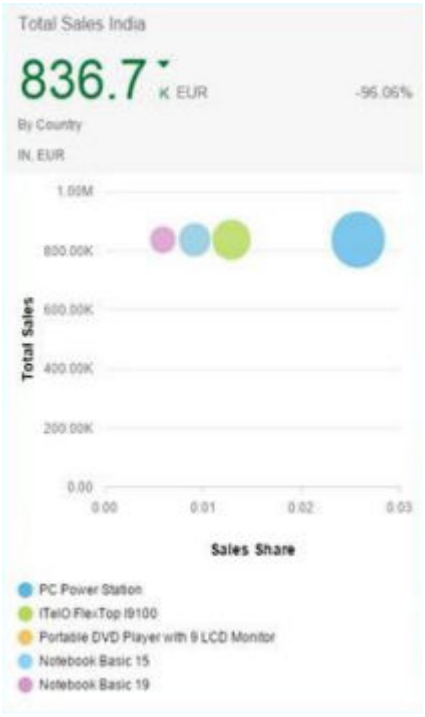
```
                                <PropertyValue Property="MeasureAttributes">
                                                <Collection>
                                                                <Record
Type="UI.ChartMeasureAttributeType">

  <PropertyValue Property="Measure" PropertyPath="SalesShare" />

  <PropertyValue Property="Role" EnumMember="UI.ChartMeasureRoleType/Axis1" />
                                                                </Record>
                                                                <Record
Type="UI.ChartMeasureAttributeType">

  <PropertyValue Property="Measure" PropertyPath="TotalSales" />

  <PropertyValue Property="Role" EnumMember="UI.ChartMeasureRoleType/Axis2" />
                                                                </Record>
                                                                <Record
Type="UI.ChartMeasureAttributeType">

  <PropertyValue Property="Measure" PropertyPath="Sales" />

  <PropertyValue Property="Role" EnumMember="UI.ChartMeasureRoleType/Axis3" />
                                                                </Record>
                                                </Collection>
                                </PropertyValue>
                                <PropertyValue Property="DimensionAttributes">
                                                <Collection>
                                                                <Record
Type="UI.ChartDimensionAttributeType">

  <PropertyValue Property="Dimension" PropertyPath="Product" />

  <PropertyValue Property="Role" EnumMember="UI.ChartDimensionRoleType/
Series" />
                                                                </Record>
                                                 </Collection>
                                </PropertyValue>
                </Record>
</Annotation>
```

The bubble chart supports a color palette for semantic coloring.

Example of Line and Bubble Chart Cards



Line Chart Card



Bubble Chart Card

Donut Chart

A donut chart displays data as the differently colored sections of a donut. The value of the measure determines the size of each section. Donut charts help the viewer to quickly determine the key area that needs attention. For example, you can view numbers and percentages. You can also disable navigation in the graph (optional).

Donut charts require exactly one measure. You can provide more than one dimension. If this is the case, the dimensions are stacked so that the sections of the chart represent the combination of all dimensions. For example, if you define **Sales** as your measure, and provide two dimensions: **Year** and **Country**, the chart displays the sales data of each combination of year and country as a separate colored section.

Stable Coloring

You can now assign specific colors to the sections in the donut chart. Each color can be assigned to a particular dimension value. To enable this feature:

- Configure a color map object that maps the key-value pairs between dimension and color values in the `colorPalette` property of the descriptor configuration.
- Enable stable coloring by setting the `bEnableStableColoring` property to **true** in card settings.
- The chart dimension property (`Role`) in the chart annotation has to be a `Category`.
- Define the `com.sap.vocabularies.Common.v1.Text` annotation for a dimension property within the entity type. This is considered to be a label for any individual dimensions value and also for rendering appropriate texts in the chart's legend.

Others Sector

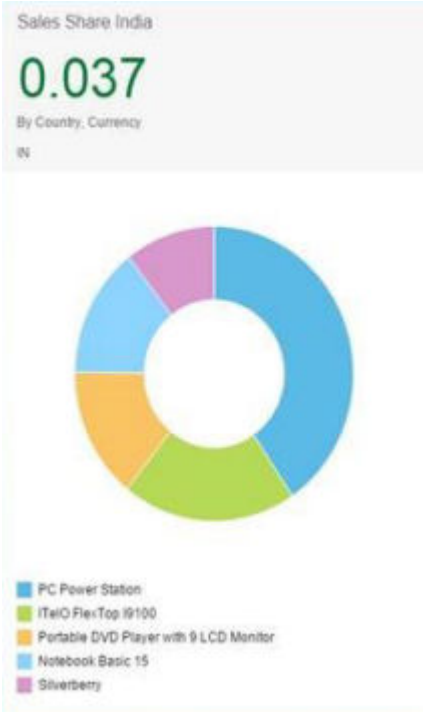
You can pass filter conditions to target applications other than the dimensions shown on the donut chart. For example, in a donut chart with the sections A, B, C, and Others, navigation from Others section leads to a filter condition that excludes A, B, and C.

Column Chart

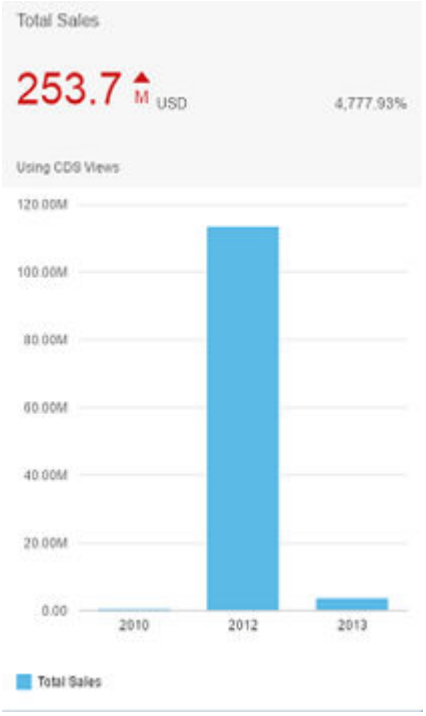
Use this chart type to display data, such as total product sales over a period of years in columns. The number of columns is equal to the number of measures in the annotation file.

Column charts need to have at least one measure and one dimension. Irrespective of the role defined for the measure in the annotation file, every measure is represented as a separate column. Similarly, regardless of the role defined in the annotation file, every dimension is added to the **axis** category (x-axis).

Example of Donut and Column Chart Cards



Donut Chart Card



Column Chart Card

Stacked Column Chart

A stacked column chart is similar to a column chart; however, all measures, irrespective of role, are stacked on top of each other. There should be at least one dimension with the assigned **category** role and all dimensions

with this role are added to the **axis** category (x-axis). All dimensions with the **series** role are also stacked. We recommend either stacking based on dimensions or measures, but not mixing both in a single chart card.

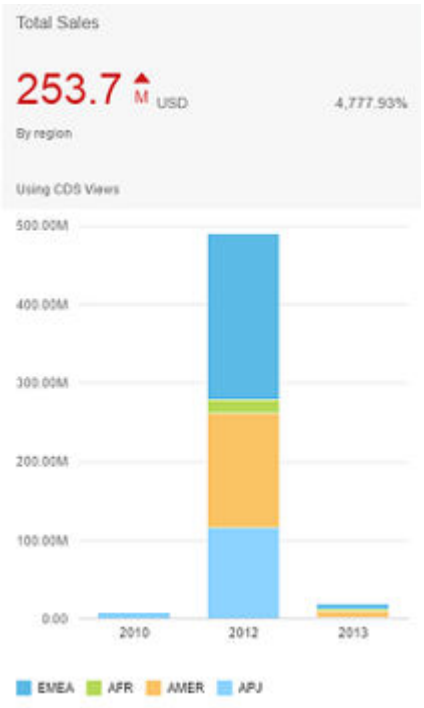
> **i** Note
>
> The column stack card can have an optional dimension with role series. Assign a dimension with the **series** role for the property containing the semantic values.

The stacked column chart supports a color palette for semantic coloring.

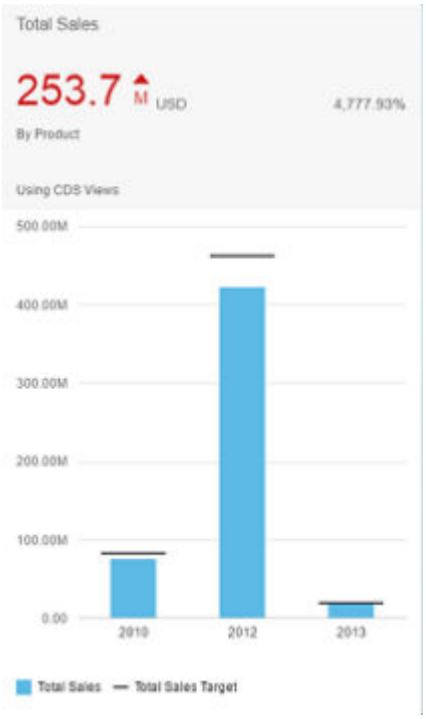
Vertical Bullet Chart

Vertical bullet charts accept at least one measure and one dimension. All dimensions, regardless of their role, are assigned to the **axis** category. All measures with the **axis1** role are represented as solid-colored columns, which represent actual values. All measures with **axis2** role are shown as a solid black line, which represents the target value.

Example of Stacked Column and Vertical Bullet Chart Cards



Stacked Column Chart Card



Vertical Bullet Chart Card

Combination Chart

A combination chart lets you combine and view two or more chart types in a single chart. For example, combining a column and line chart in the same visualization lets you compare values of different categories. This provides a clear view and helps you compare categories.

For combination charts:

- The first measure is used for the column format and the subsequent measure is displayed as a line within the chart.
- We recommend only using one time-based dimension for the **category** axis.
- All measures, irrespective of their roles, are assigned to the feed's UID value axis. You need to have at least two measures for combination chart cards.
- For all dimensions with a role:
  - A **Category** is assigned to the category axis with the default role. You need to have at least one role assigned to the category axis.
  - A **Series** is assigned to the feed UID's color and is displayed within the chart area with a differently colored column and line combinations for each of its members.

Example

Sample Code

```
<Annotation Term="UI.Chart" Qualifier="Eval_by_Currency_Combination">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Sales and Total Sales" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Combination"/>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                        <PropertyValue Property="Measure"
PropertyPath="sales" />
                        <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                </Record>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="totalsales" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="DimensionAttributes">
            <Collection>
                <Record Type="UI.ChartDimensionAttributeType">
                    <PropertyValue Property="Dimension"
PropertyPath="quarter_1" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartDimensionRoleType/Category" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

The combination chart supports a color palette for semantic coloring.

Scatter Chart

With a scatter chart card, you can visualize the distribution of data points over two measures; the first measure for which the role is set to an axis is assigned to the `valueAxis` feed UID makes up the x-axis.

> **i** Note
>
> The role is set to `axis1`, `axis2` (if there's no `axis1`), or `axis3` (if there's no `axis2`).

The other measure is plotted on the y-axis.

A scatter chart card supports a maximum of two dimensions. If the dimension is not marked with a role, then all members of the dimension are plotted as equal-sized bubbles of the same color in the chart. You can assign only one dimension to the `Series` role and all members of this dimension get a different color. A maximum of only one dimension can be assigned to the `Category` role and all members of such a dimension get a different shape.

Example

> Sample Code

```
<Annotation Term="UI.Chart" Qualifier="Eval_by_Currency_Scatter">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Scatter Chart no role" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Scatter" />
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="salesshare" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                </Record>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="totalsales" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis2" />
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="DimensionAttributes">
            <Collection>
                <Record Type="UI.ChartDimensionAttributeType">
                    <PropertyValue Property="Dimension"
PropertyPath="suppliercompany" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Example Combination and Scatter Chart Cards

Combination Chart Card

Scatter Chart Card

> **i** Note
>
> Analytic cards don't have a fixed height. The height is adjusted automatically to accommodate the data points and legends. Legends are created automatically based on the defined measures and dimensions.

Waterfall Chart

You use waterfall charts to analyze a cumulative value. Waterfall charts allow you to see the change in cumulative values from the initial state to the final state by representing the accumulation of successive values. These are the available waterfall charts:

- Waterfall charts without a time dimension
- Waterfall charts with a time dimension represent the change of a cumulative value over time
- Semantic waterfall charts (semantic coloring based on `com.sap.vocabularies.UI.v1.CriticalityCalculation` or `com.sap.vocabularies.UI.v1.Criticality` in the datapoint annotation)

> **i** Note
>
> By default the legend shows the name of the measure mapped to the chart and two groups `<0` and `>0`. If there is more than one measure, all measures are displayed instead of the measure names.

→ Remember

- Waterfall charts need at least one measure and one dimension
- Dimensions for which a role is set (for example, category) make up the x-axis (category axis). If no dimension is specified with a role, the first dimension is used as the x-axis.
- Dimensions for which a role is set (for example, series) make up the cumulative data points in the chart. A waterfall chart can have only one dimension per role.
- Dimensions with the role mapped to the `waterfallType` UID. You use this to show the intermediate totals and subtotals in the waterfall chart. Valid values:
  - `null`
  - `subtotal:2` (combines the previous two data points and shows a new column in the chart as a subtotal)
  - `total` (combines all the data points and shows a new column as the total)
- Measures make up the y-axis (value axis)

‹≡› Sample Code

Vertical Waterfall Chart Annotation Sample

```
<Annotation Term="UI.Chart" Qualifier="Waterfall_Eval_by_Country">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="Revenue Waterfall" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Waterfall"/>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="Finances" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis2" />
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="DimensionAttributes">
            <Collection>
                <Record Type="UI.ChartDimensionAttributeType">
                    <PropertyValue Property="Dimension"
PropertyPath="SpendType" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartDimensionRoleType/Category" />
                </Record>
                <Record Type="UI.ChartDimensionAttributeType">
                    <PropertyValue Property="Dimension" PropertyPath="Type" />
                    <PropertyValue Property="Role"
EnumMember="UI.ChartDimensionRoleType/Series" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Time Series Chart

A chart type with a time axis instead of a categorical axis. A time series chart represents a time-based dimension that is more responsive to a change in card size. Analytic cards use the time axis automatically if these conditions apply:

- The chart type must be either vertical bullet, stacked column, scatter, line, bubble, column, waterfall, combination, or dual combination.
- The chart is configured with only one dimension for bubble, column, waterfall, and combination charts. However, you can use two dimensions for line charts. The second dimension can be the color dimension. The dimension with the `uid` color has to have the **Series** role assigned to it.
- The data type of the dimension is either `edm.datetime` or `edm.string`. If the data type is `edm.string`, then it needs to have the additional annotation in the OData metadata annotation: `sap:semantics:"yearweek (YYYYWW) or yearmonth (YYYYMM) or yearquarter (YYYYQ)"`.
- If it is a bubble chart, it needs to have two measures. If the chart is a combination chart, it needs to have at least two measures.
- Vertical bullet, stacked column, and scatter charts need at least one measure and one dimension. Extra color and shape dimensions are suppored only in scatter charts.

Sample Code

Metadata Sample

```
<Property Name="Date" Type="Edm.DateTime" sap:display-format="Date"
sap:label="Date" sap:aggregation-role="dimension"/>
```

Sample Code

Annotation Sample

```
<Annotation Term="UI.Chart" Qualifier="Line-Time-Currency">
    <Record Type="UI.ChartDefinitionType">
        <PropertyValue Property="Title" String="View1" />
        <PropertyValue Property="ChartType" EnumMember="UI.ChartType/Line"/>
        <PropertyValue Property="MeasureAttributes">
            <Collection>
                <Record Type="UI.ChartMeasureAttributeType">
                    <PropertyValue Property="Measure"
PropertyPath="SalesShare" />
                    <PropertyValue Property="Role"
                                   EnumMember="UI.ChartMeasureRoleType/
Axis1" />
                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Eval_by_CtryCurr-SalesShare"/>
                </Record>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="DimensionAttributes">
            <Collection>
                <Record Type="UI.ChartDimensionAttributeType">
                    <PropertyValue Property="Dimension" PropertyPath="Date" />
                    <PropertyValue Property="Role"
                                   EnumMember="UI.ChartDimensionRoleType/
Category" />
                </Record>
                <Record Type="UI.ChartDimensionAttributeType">
                    <PropertyValue Property="Dimension"
PropertyPath="Sales_CURRENCY"/>
```

```
                    <PropertyValue Property="Role"
                                   EnumMember="UI.ChartDimensionRoleType/
Series"/>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Dual Combination Chart

Overview pages support a dual combination chart type that lets you view individual data points for a particular dimension. The chart contains two axis values with a line chart representing the multiple measures.

Chart area configuration:

- The first measure is displayed as a column chart type and subsequent measures display as a line within the chart.
- We recommend using only one time-based dimension for the category axis.
- Configure at least two measures. Assign the measures to the feed's UID value axis, irrespective of the roles.
- Assign at least one role to the category axis. All dimensions with the **Category** role are assigned to the category axis. The **Category** role is the default role.

Configuring Charts

Developers configure the measures and dimensions displayed in charts by setting the role property to the desired value per chart type. Additional definitions apply to all chart types.

You can use the same annotation file with different qualifiers to present charts with different chart views. You do this by specifying different qualifiers in the annotation file for each card. The following sections of the annotation file apply to all chart types:

Annotation	What it Does
`UI.Identification`	Specify the navigation targets activated when the user clicks the card and list the parameters to pass to the target application. This definition is mandatory. For more information, see Configuring Card Navigation [page 1998].
`UI.SelectionVariant.SelectOptions`	Specify the filter values that are applied to the card, which are applied when retrieving the card data.
`UI.PresentationVariant.SortOrder`	Specify the sort order to be used.

Annotation	What it Does
`UI.PresentationVariant.MaxItems`	Limit the maximum number of records to be fetched from the backend. If this variant isn't used, then all records from the backend will be displayed in the chart. **→ Tip** Don't use this for charts that rely on complete data sets, for example, the donut chart card, otherwise the results won't be meaningful.
`UI.Chart`	Specify the dimensions and measures that make up the chart, the chart type, and the way that the measures/dimensions are used for the chart. This definition is mandatory.
`UI.Chart.MeasureAttributes.Measure`	Defines the measures used in the chart.
`UI.Chart.MeasureAttributes.Role`	The manner in which a measure is used within the chart. This is configured differently for each chart type, as described below.
`UI.Chart.DimensionAttributes.Dimension`	These are the dimensions used in the chart.
`UI.Chart.DimensionAttributes.Role`	The manner in which a dimension is used within the chart. This is configured differently for each chart type, as described below.

Formatting Numeric Values in Charts

Within overview pages, analytical chart cards can have format measure values based on the `NumberOfFractionalDigits` and `ScaleFactor` properties of the `DataPoint` term in the annotation file.

Here is an example annotation that shows how it's used:

Sample Code

```
<Annotation Term="UI.DataPoint" Qualifier=" Eval_by_Currency-TotalSales ">
<Record Type="UI.DataPointType">
          <PropertyValue Property="ValueFormat">
                  <Record>
                           <PropertyValue Property="ScaleFactor"
Int="1000" />
                           <PropertyValue
Property="NumberOfFractionalDigits" Int="3" />
                  </Record>
          </PropertyValue>
  </Record>
</Annotation>
<Annotation Term="UI.Chart" Qualifier="Eval_by_Currency">
  <Record Type="UI.ChartDefinitionType">
          <PropertyValue Property="Title" String="View1" />
          <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Bubble"/>
          <PropertyValue Property="MeasureAttributes">
                  <Collection>
                           …
```

```
                              <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="TotalSales" />
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis2" />
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#Eval_by_Currency-TotalSales"/>
                              </Record>
                              …
                     </Collection>
            </PropertyValue>
            …
  </Record>
</Annotation>
```

Semantic Pattern

With the semantic pattern feature, overview page analytical cards can enable users to compare between actual and forecast values. The line, column, and vertical bullet chart cards support this feature. The forecast value comes from the datapoint annotation that is associated with the measure used in the selected analytical card. To enable the semantic pattern feature:

- Datapoint annotation should contain the `ForecastValue` property with value as a measure.
- Chart annotation should consist of:
  - 1 dimension and 1 measure for line and column chart cards
  - 1 dimension and 1-2 measures for vertical bullet chart cards

If the above conditions are not met, the chart will not inherit the semantic pattern feature. The actual measure will appear in a solid color and the forecast measure will appear as a dashed pattern for column and vertical bullet chart cards or as a dotted pattern for line charts. The following graphic is an example of what it could look like:

Here's an example annotation that shows how it's used:

Sample Code

```
<Annotation Term="UI.DataPoint" Qualifier="Column_Forecast">
<Record Type="UI.DataPointType">
            <PropertyValue Property="Title" String="Sales Performance"/>
            <PropertyValue Property="Value" Path="Sales"/>
            <PropertyValue Property="NumberFormat">
                <Record>
                    <PropertyValue Property="ScaleFactor" Int="0"/>
                    <PropertyValue Property="NumberOfFractionalDigits"
Int="3"/>
                </Record>
            </PropertyValue>
        <PropertyValue Property="ForecastValue" PropertyPath="SalesShare"/>
</Record>
</Annotation>

<Annotation Term="UI.Chart" Qualifier="Eval_by_Currency_Column">
<Record Type="UI.ChartDefinitionType">
            <PropertyValue Property="Title" String="Column chart for shape" />
            <PropertyValue Property="ChartType" EnumMember="UI.ChartType/
Column" />
            <PropertyValue Property="MeasureAttributes">
                    <Collection>
                        <Record Type="UI.ChartMeasureAttributeType">
                            <PropertyValue Property="Measure"
PropertyPath="Sales" />
                            <PropertyValue Property="DataPoint">
<AnnotationPath>@UI.DataPoint#Column_Forecast</AnnotationPath>
                </PropertyValue>
            <PropertyValue Property="Role"
            EnumMember="UI.ChartMeasureRoleType/Axis1" />
    </Record>
                    </Collection>
                </PropertyValue>
        <PropertyValue Property="DimensionAttributes">
                <Collection>
<Record Type="UI.ChartDimensionAttributeType">
                        <PropertyValue Property="Dimension"
PropertyPath="SupplierCompany" />
                        <PropertyValue Property="Role"
                            EnumMember="UI.ChartDimensionRoleType/Category" />
                    </Record>
                </Collection>
        </PropertyValue>
</Record>
</Annotation>
```

Chart Types

Overview pages can use line, donut, bubble, column, stacked column, vertical bullet, combination, and scatter analytic chart cards.

The value assigned to the role property for dimensions and measures in the annotation file determines the visualization of the chart. For dimensions, you can set the role to **category** or **series**. If no value is specified, the default is **category**.

For measures, you can set the role to the values: **axis1**, **axis2** , or **axis3**. If no value is specified, the default is **axis1**. The actual interpretation of the role value specified in the annotation file varies according to the chart type used.

Time Series Charts

Time series chart cards are cards with regular charts, but use time as the category axis instead of the categorical axis. The advantage in using a time-series axis is that the representation of the time-based dimension is much cleaner and more responsive to the change in card size. The display level and format in the time axis would be offered in the default format by the visual chart (for example day/month/year displayed as 10/Jan/2016).

Analytic cards will automatically use the time axis only if the following conditions are met:

- The chart type is either line, bubble, column, or combination
- The chart is configured with only one dimension
- The data type of the dimension is either `edm.datetime` or `edm.string`. If the data type is `edm.string`, then it must have the additional OData metadata annotation `sap:semantics` of `yearmonthday`. If it's a bubble chart, there must be exactly two measures
- If it's a combination chart card, then there must be at least two measures.

> **i** Note
>
> Only line, bubble, column, and combination chart cards support the time axis.

Color Palette

A few chart types (line, bubble, combination, and stacked column) support color palette for semantic coloring. To enable this feature, configure the required chart type and define the `colorPalette` property in the app descriptor. The `colorPalette` property is an array of four objects. Each object indicates the semantic representations:

- First object: criticality state 0
- Second object: criticality state 1
- Third object: criticality state 2
- Fourth object: criticality state 3

Every object in the `colorPalette` array has two properties `color` (a color value for a particular state) and `legendText` (the corresponding legend text).

> **i** Note
>
> - Use only the colors listed in the semantic palette that are defined by SAP Fiori guidelines for customizing the column stack card.
> - All four objects in the color `colorPalette` array are mandatory.

More Information

For more information about the type of charts used in overview pages, see Chart Cards Used in Overview Pages [page 1979].

Configuring Card Properties

This section provides the configuration items relevant for all overview page cards.

All cards inherit the generic capabilities that are shared across all cards, such as a card header, card footer, navigation support, and more. The following are the card properties mandatory for all cards:

- `model`: Name of the model to provide to the card instance
- `template`: Card type (card component package) to instantiate
- `settings`: Iternal card configuration passed to the card instance
- `entitySet`: Entity set to use in the card

Configuring an EntitySet with Input Parameters

Some entity sets requires input parameters. You can configure these parameters using the `com.sap.vocabularies.UI.v1.SelectionVariant` annotation term by setting the `Parameters` section.

In the following examples there are two entity sets - `SalesShare` and `SalesShareParameters` - with associations defined between them. The `SalesShare` entity set contains the OData information and the `SalesShareParameters` defines the input parameters to retrieve from the `SalesShare` entity set the data which will be displayed in cards. The `com.sap.vocabularies.UI.v1.SelectionVariant` annotation term contains the input parameters to be applied, so as to trigger the query and display the actual data during runtime.

OData metadata definition

Sample Code

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/
edmx" xmlns:sap="http://www.sap.com/Protocols/SAPData">
    <edmx:DataServices xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata" m:DataServiceVersion="2.0">
        <Schema Namespace="sap.smartbusinessdemo.services" xmlns:d="http://
schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://
schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://
schemas.microsoft.com/ado/2008/09/edm">
            <EntityType Name="SalesShareType" sap:semantics="aggregate">
                <Key>
                    <PropertyRef Name="ID" />
                </Key>
                <Property Name="ID" Type="Edm.String" Nullable="false"
MaxLength="2147483647" sap:filterable="false" />
                <Property Name="Country" Type="Edm.String" MaxLength="3"
sap:label="Country" sap:aggregation-role="dimension" />
                <Property Name="Region" Type="Edm.String" MaxLength="4"
sap:label="Region" sap:aggregation-role="dimension" />
                <Property Name="ProductID" Type="Edm.String" MaxLength="10"
sap:label="Product ID" sap:aggregation-role="dimension" />
                <Property Name="Currency" Type="Edm.String" MaxLength="5"
sap:label="Currency" sap:aggregation-role="dimension" />
                <Property Name="Product" Type="Edm.String" MaxLength="1024"
sap:label="Product" sap:aggregation-role="dimension" />
                <Property Name="SupplierCompany" Type="Edm.String"
MaxLength="80" sap:label="Supplier Company" sap:aggregation-
role="dimension" />
```

```
                <Property Name="BuyerCompany" Type="Edm.String"
MaxLength="80" sap:label="Buyer Company" sap:aggregation-role="dimension" />
                <Property Name="Year" Type="Edm.String" MaxLength="4"
sap:label="Year" sap:aggregation-role="dimension" />
                <Property Name="Quarter" Type="Edm.String" MaxLength="2"
sap:label="Quarter" sap:aggregation-role="dimension" />
                <Property Name="Month" Type="Edm.String" MaxLength="2"
sap:label="Month" sap:aggregation-role="dimension" />
                <Property Name="TotalSales_CURRENCY" Type="Edm.String"
MaxLength="5" sap:semantics="currency-code" />
                <Property Name="Sales_CURRENCY" Type="Edm.String"
MaxLength="5" sap:semantics="currency-code" />
                <Property Name="TotalSales" Type="Edm.Decimal" Precision="15"
Scale="2" sap:filterable="false" sap:label="Total Sales" sap:aggregation-
role="measure" sap:unit="TotalSales_CURRENCY" />
                <Property Name="Sales" Type="Edm.Decimal" Precision="15"
Scale="2" sap:filterable="false" sap:label="Sales" sap:aggregation-
role="measure" sap:unit="Sales_CURRENCY" />
                <Property Name="SalesShare" Type="Edm.Decimal" Precision="12"
Scale="5" sap:filterable="false" sap:label="Sales Share" sap:aggregation-
role="measure" />
            </EntityType>
            <EntityType Name="SalesShareParametersType"
sap:semantics="parameters">
                <Key>
                    <PropertyRef Name="P_Currency"/>
                    <PropertyRef Name="P_Country"/>
                </Key>
                <Property Name="P_Currency" Type="Edm.String"
Nullable="false" MaxLength="5" sap:label="Currency"
sap:parameter="mandatory"/>
                <Property Name="P_Country" Type="Edm.String" Nullable="false"
DefaultValue="3" MaxLength="20" sap:label="CountryCode"
sap:parameter="mandatory"/>
                <NavigationProperty Name="Results"
Relationship="sap.smartbusinessdemo.services.SalesShareParameters_SalesShareTy
pe" FromRole="SalesShareParametersPrincipal" ToRole="SalesShareDependent"/>
            </EntityType>

            <Association Name="SalesShareParameters_SalesShareType">
                <End
Type="sap.smartbusinessdemo.services.SalesShareParametersType"
Role="SalesShareParametersPrincipal" Multiplicity="*"/>
                <End Type="sap.smartbusinessdemo.services.SalesShareType"
Role="SalesShareDependent" Multiplicity="*"/>
            </Association>


            <EntityContainer Name="SalesShare"
m:IsDefaultEntityContainer="true">

                <EntitySet Name="SalesShare"
EntityType="sap.smartbusinessdemo.services.SalesShareType"/>
                <EntitySet Name="SalesShareParameters"
EntityType="sap.smartbusinessdemo.services.SalesShareParametersType"
sap:addressable="false"/>
                <AssociationSet Name="SalesShareParameters_SalesShare"
Association="sap.smartbusinessdemo.services.SalesShareParameters_SalesShareTyp
e">
                    <End Role="SalesShareParametersPrincipal"
EntitySet="SalesShareParameters"/>
                    <End Role="SalesShareDependent" EntitySet="SalesShare"/>
                </AssociationSet>

            </EntityContainer>
        </Schema>
    </edmx:DataServices>
```

```
</edmx:Edmx>
```

Annotation document containing the filters to be applied

Sample Code

```
<Annotations Target="sap.smartbusinessdemo.services.SalesShareType">
    <Annotation Term="com.sap.vocabularies.UI.v1.SelectionVariant"
Qualifier="params">
        <Record>
            <PropertyValue Property="SelectOptions">
                <Collection>
                </Collection>
            </PropertyValue>
            <PropertyValue Property="Parameters">
                <Collection>
                    <Record Type="com.sap.vocabularies.UI.v1.Parameter">
                        <PropertyValue Property="PropertyName"
PropertyPath="P_Currency" />
                        <PropertyValue Property="PropertyValue"
String="EUR" />
                    </Record>
                    <Record Type="com.sap.vocabularies.UI.v1.Parameter">
                        <PropertyValue Property="PropertyName"
PropertyPath="P_Country" />
                        <PropertyValue Property="PropertyValue" String="IN" />
                    </Record>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
    ...
</Annotations>
```

Card configuration in the manifest document

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
        ...
        "card04": {
            "model": "salesShare",
            "template": "sap.ovp.cards.list",
            "settings": {
                ...
                "selectionAnnotationPath" :
"com.sap.vocabularies.UI.v1.SelectionVariant#params",
                "entitySet": "SalesShare"
            }
        },
        ...
    }
}
```

Configuring Card Navigation

All cards support navigation, both to a different SAP Fiori application using intent based navigation, and to external applications and websites via a direct URL which opens in a new browser tab. To trigger the navigation, users click or tap on a card header and in some cases, on an item within the card.

Navigation information is taken from the `com.sap.vocabularies.UI.v1.Identification` and

- `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation` should be used to define intent based navigation to SAP Fiori application.
- `com.sap.vocabularies.UI.v1.DataFieldWithUrl term` should be used to configure navigation to external apps and websites.

The recommended way to configure intent-based navigation is to use `DataFieldForIntentBasedNavigation`. However, for navigation to a specific application route that is not configured as target mapping, you can also use `DataFieldWithUrl` to construct the specific application route. The overview page will identify that this is an intent-based navigation and open the application in the relevant context, in the same tab.

Note that information about the single record selected can only be passed on to the navigation destination from list or table cards. To support this option, provide navigation configuration in the `com.sap.vocabularies.UI.v1.LineItem` term used by that specific card.

If more than one navigation record is provided in the `com.sap.vocabularies.UI.v1.Identification` or `com.sap.vocabularies.UI.v1.LineItem` terms, the first one will be used for each term. The navigation records would be sorted according to importance, set in the `com.sap.vocabularies.UI.v1.ImportanceType` annotation, and their order of entry. The `com.sap.vocabularies.UI.v1.Identification` term can be configured in the application manifest file by setting the `identificationAnnotationPath` property with a qualifier, as shown in the example below. If the `identificationAnnotationPath` property is not configured, the `com.sap.vocabularies.UI.v1.Identification` term, without a qualifier, is used.

Sample Code

```
"sap.ovp": {
    ...
    "cards": {
        ...
        "card04": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.list",
            "settings": {
                "sortBy": "Price",
                "sortOrder": "descending",
                "listFlavor": "bar",
                "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#bar",
                "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#bar",
                "category": "{{card04_category}}",
                "entitySet": "Products"
            }
        },
        ...
    }
}
```

Custom Navigation

Overview pages support navigation breakouts (extension points) that let you configure multiple navigation targets from different areas of a card (different targets from different line items).

To use navigation breakouts:

1. Configure your app descriptor for controller extension.

   Sample Code

   ```
   "extends": {
               "extensions": {
                   "sap.ui.controllerExtensions": {
                       "sap.ovp.app.Main": {
                           "controllerName": "<custom controller path, for
   example sap.ovp.demo.ext.customController>"
                       }
                   }
               }
           }
   ```

2. In the custom controller, define the `doCustomNavigation` function with following input parameters:
   - `Card ID`: Enter a string as defined in the app descriptor
   - `Context`: Enter the object that defines the context on click of a card
   - `Navigation Entry`: Enter the object that has standard navigation defined by annotations
3. Ensure that the `doCustomNavigation` method returns an object that is similar to input `Navigation Entry` and can contain following attributes (all of type String):
   - `type`: (Mandatory) Possible values are `com.sap.vocabularies.UI.v1.DataFieldWithUrl` and `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`.
   - `semanticObject`: Required when type is `DataFieldForIntentBasedNavigation`
   - `action`: Required when type is `DataFieldForIntentBasedNavigation`
   - `url`: Required when type is `DataFieldWithUrl`
   - `label`: Optional
4. If custom targets are required for a particular set of input parameters, return an object from the `doCustomNavigation` method.

   Sample Code

   ```
   doCustomNavigation: function (sCardId, oContext, oNavigationEntry) {
            var oCustomNavigationEntry;
            var oEntity = oContext && oContext.getProperty(oContext.sPath);
            if (sCardId === "card001" && oEntity.PurchaseOrder ===
   "4500003575") {
                 oCustomNavigationEntry = {};
                 oCustomNavigationEntry.type =
   "com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation";
                 oCustomNavigationEntry.semanticObject = "Action";
                 oCustomNavigationEntry.action = "toappnavsample2";
                 oCustomNavigationEntry.url = "";
                 oCustomNavigationEntry.label = "";
            }
            return oCustomNavigationEntry;
    }
   ```

Adding Static Parameters

Static parameters are objects containing key value pairs. They provide navigation parameters during `IntentBasedNavigation` from overview page to an application. To add navigation parameters, define the card settings `staticParameters` in the descriptor file.

Sample Code

```
"staticParameters": {
              "parameter1": "parameterValue1",
              "parameter2": "parameterValue2",
            },
```

Configuring Card Filters

You can add filters to all card types, by using the `com.sap.vocabularies.UI.v1.SelectionVariant` annotation term or by passing filter parameter in the URL.

The following example shows filter configuration in the annotation document. The record set is filtered by the `TotalSum` property and returns values between 0 and 8000. You can provide multiple filters in the `SelectOptions` collection.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.SelectionVariant"
Qualifier="bubble">
    <Record>
        <PropertyValue Property="SelectOptions">
            <Collection>
            <Record>
               <PropertyValue Property="PropertyName"
PropertyPath="TotalSum" />
               <PropertyValue Property="Ranges">
                  <Collection>
                     <Record>
                        <PropertyValue Property="Sign"

EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeSignType/I" />
                        <PropertyValue Property="Option"

EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeOptionType/BT" />
                        <PropertyValue Property="Low" String="0" />
                        <PropertyValue Property="High" String="8000" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

The preference for displaying the currency type is taken from the SAP Fiori Launchpad user settings in the `DisplayCurrency` field of an application. You can also set filters for SAP Fiori overview page by passing a filter

paramter in the URL. Applicable when you launch SAP Fiori overview pages from SAP Fiori Launchpad or from any other application with a filter parameter.

> **i Note**
>
> You can only pass strings or integers as filter parameters. The filter applies automatically if the filter property exists in the entityset.

For example: http://abc#Equipment-overviewPage?EquipmentNumber=123456

Configuring Sort Properties

All cards support sorting using the `com.sap.vocabularies.UI.v1.PresentationVariant` annotation term.

List, table, and stack cards support sorting by using the `sortBy` and `sortOrder` properties in the application manifest file.

The following example shows sort configuration in the application manifest file. The records are sorted by the `Price` property in descending order.

> **Sample Code**

```
"sap.ovp": {
    ...
    "cards": {
        ...
        "card04": {
            "model": "ZCD204_EPM_DEMO_SRV",
            "template": "sap.ovp.cards.list",
            "settings": {
                "sortBy": "Price",
                "sortOrder": "descending",
                "listFlavor": "bar",
                "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#bar",
                "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#bar",
                "category": "{{card04_category}}",
                "entitySet": "Products"
            }
        },
        ...
    }
}
```

The following example shows sort configuration in the annotation file. The records are sorted by the `Sales` property in descending order. You can configure multiple sorts by defining multiple properties in the `SortOrder` collection.

> **Sample Code**

```
<Annotation Term="com.sap.vocabularies.UI.v1.PresentationVariant"
Qualifier="Eval_by_Country">
   <Record>
      <PropertyValue Property="GroupBy">
         <Collection>
```

```
            <PropertyPath>Country</PropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="SortOrder">
         <Collection>
            <Record>
               <PropertyValue Property="Property" PropertyPath="Sales" />
               <PropertyValue Property="Descending" Boolean="true" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Visualizations">
         <Collection>
            <AnnotationPath>@UI.Chart#Eval_by_Country</AnnotationPath>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

i Note

Overview pages allow you to share sort parameters defined in the presentation variant to any target SAP Fiori Elements application. The target applications can use these parameters to sort data accordingly.

Adding the OData Select Parameter

OData supports the `select` parameter, where you can specify lists of properties that are needed by the application, so that unnecessary properties are not returned by the OData request.

The `select` parameter is not added automatically to the OData request in list and table cards as this can affect navigation behavior. However, in some cases where Smart Business OData services are used, the `select` parameter might be required to show aggregated values to the user. In these cases, the `addODataSelect` property must be set in the application manifest file for the relevant card.

‹≡› Sample Code

The card configuration in the application manifest file looks like this:

```
"sap.ovp": {
    "globalFilterModel": "salesOrder",
    "globalFilterEntityType": "GlobalFilters",
    "cards": {
      "card00": {
        "model": "salesOrder",
        "template": "sap.ovp.cards.list",
        "settings": {
          "entitySet": "SalesOrderSet",
          "category": "Sales Orders with filters",
          "listType": "extended",
          "addODataSelect": true
        }
      }
    }
}
```

Configuring View Switch

Configuring this property lets you define a dropdown list to filter/view data at the card level.

You can define view switch with a single entity set or with multiple entity sets.

Sample Code

View Switch Defenition with Single Entity Set

```
"settings": {
    "entitySet": "SalesOrderSet"
    "tabs": [
        {
            ...
        },
        {
            ...
        }
    ]
}
```

Sample Code

View Switch Definition with Multiple Entity Sets

```
 "settings": {
     "tabs": [
         {
             "entitySet": "SalesOrderSet"
           ...
         },
         {
             "entitySet": " ProductSet"
           ...
         }
     ]
 }
```

Sample Code

View Switch Sample

```
"card009": {
  "model": "salesOrder",
  "template": "sap.ovp.cards.list",
  "settings": {
    "title": "Contract Monitoring",
    "subTitle": "Per Supplier",
    "valueSelectionInfo": "Total contract volume",
    "listFlavor": "bar",
    "listType": "extended",
    "showLineItemDetail":true,
    "tabs": [
      {
        "entitySet": "SalesOrderSet",
        "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
        "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View1",
        "selectionAnnotationPath":
"com.sap.vocabularies.UI.v1.SelectionVariant#line1",
```

```
        "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#line",
        "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification",
        "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
        "value": "{{dropdown_value2}}"
       },
      {
        "entitySet": "SalesOrderSet",
        "dynamicSubtitleAnnotationPath":
"com.sap.vocabularies.UI.v1.HeaderInfo#dynamicSubtitle",
        "annotationPath": "com.sap.vocabularies.UI.v1.LineItem#View3",
        "presentationAnnotationPath":
"com.sap.vocabularies.UI.v1.PresentationVariant#SP3",
        "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification",
        "dataPointAnnotationPath":
"com.sap.vocabularies.UI.v1.DataPoint#line",
        "value": "{{dropdown_value3}}"
       },
      {
        "entitySet": "ProductSet",
        "annotationPath": "com.sap.vocabularies.UI.v1.LineItem",
        "identificationAnnotationPath":
"com.sap.vocabularies.UI.v1.Identification#identify1",
        "value": "{{dropdown_value1}}"
       }
    ]
  }
}
```

Setting Units of Measure

You can display the unit of measure next to numeric values by providing the `sap:unit` attribute in the OData metadata file or by annotating the unit in the annotation document.

In the following example of the `Product` entity type definition in the OData metadata file, the `Price` property has the `CurrencyCode` property as its unit of measure; `Width`, `Depth`, and `Height` have the `DimUnit` property as their unit of measure; and the `WeightMeasure` property has the `WeightUnit` property as its unit of measure.

Sample Code

```
<EntityType Name="Product" sap:content-version="1">
    <Key>
        <PropertyRef Name="ProductID"/>
    </Key>
    <Property Name="ProductID" Type="Edm.String" Nullable="false"
MaxLength="10" sap:label="Product ID" sap:updatable="false"/>
    <Property Name="TypeCode" Type="Edm.String" MaxLength="2"
sap:label="Prod. Type Code"/>
    <Property Name="Category" Type="Edm.String" MaxLength="40"
sap:label="Prod. Cat."/>
    <Property Name="Name" Type="Edm.String" MaxLength="255"
sap:label="Product Name" sap:sortable="false" sap:filterable="false"/>
    <Property Name="NameLanguage" Type="Edm.String" MaxLength="1"
sap:label="Language" sap:creatable="false" sap:updatable="false"
sap:sortable="false" sap:filterable="false"/>
```

```
    <Property Name="Description" Type="Edm.String" MaxLength="255"
sap:label="Prod.Descrip." sap:sortable="false" sap:filterable="false"/>
    <Property Name="DescriptionLanguage" Type="Edm.String" MaxLength="1"
sap:label="Language" sap:creatable="false" sap:updatable="false"
sap:sortable="false" sap:filterable="false"/>
    <Property Name="SupplierID" Type="Edm.String" MaxLength="10"
sap:label="Bus. Part. ID"/>
    <Property Name="SupplierName" Type="Edm.String" MaxLength="80"
sap:label="Company Name" sap:creatable="false" sap:updatable="false"/>
    <Property Name="TaxTarifCode" Type="Edm.Byte" sap:label="Prod. Tax Code"/>
    <Property Name="MeasureUnit" Type="Edm.String" MaxLength="3"
sap:label="Qty. Unit" sap:semantics="unit-of-measure"/>
    <Property Name="WeightMeasure" Type="Edm.Decimal" Precision="13"
Scale="3" sap:unit="WeightUnit" sap:label="Wt. Measure"/>
    <Property Name="WeightUnit" Type="Edm.String" MaxLength="3"
sap:label="Qty. Unit" sap:semantics="unit-of-measure"/>
    <Property Name="CurrencyCode" Type="Edm.String" MaxLength="5"
sap:label="Currency" sap:semantics="currency-code"/>
    <Property Name="Price" Type="Edm.Decimal" Precision="16" Scale="3"
sap:unit="CurrencyCode" sap:label="Unit Price"/>
    <Property Name="Width" Type="Edm.Decimal" Precision="13" Scale="3"
sap:unit="DimUnit" sap:label="Dimensions"/>
    <Property Name="Depth" Type="Edm.Decimal" Precision="13" Scale="3"
sap:unit="DimUnit" sap:label="Dimensions"/>
    <Property Name="Height" Type="Edm.Decimal" Precision="13" Scale="3"
sap:unit="DimUnit" sap:label="Dimensions"/>
    <Property Name="DimUnit" Type="Edm.String" MaxLength="3" sap:label="Dim.
Unit" sap:semantics="unit-of-measure"/>
    <Property Name="CreatedAt" Type="Edm.DateTime" Precision="7"
sap:label="Time Stamp" sap:creatable="false" sap:updatable="false"/>
    <Property Name="ChangedAt" Type="Edm.DateTime" Precision="7"
ConcurrencyMode="Fixed" sap:label="Time Stamp" sap:creatable="false"
sap:updatable="false"/>
    <NavigationProperty Name="ToSalesOrderLineItems"
Relationship="GWSAMPLE_BASIC.Assoc_Product_SalesOrderLineItems"
FromRole="FromRole_Assoc_Product_SalesOrderLineItems"
ToRole="ToRole_Assoc_Product_SalesOrderLineItems"/>
    <NavigationProperty Name="ToSupplier"
Relationship="GWSAMPLE_BASIC.Assoc_BusinessPartner_Products"
FromRole="ToRole_Assoc_BusinessPartner_Products"
ToRole="FromRole_Assoc_BusinessPartner_Products"/>
</EntityType>
```

In the following example, the `Price` property is annotated in the annotation document with `Org.OData.Measures.V1.ISOCurrency` to indicate that the currency is displayed using the `CurrencyCode` property; the `Width`, `Depth`, and `Height` properties are annotated with `Org.OData.Measures.V1.Unit` to indicate that the unit is displayed using the `DimUnit` property; and `WeightMeasure` is annotated with `Org.OData.Measures.V1.Unit` to indicate that the unit is displayed using the `WeightUnit` property.

Sample Code

```
<Annotations Target="GWSAMPLE_BASIC.Product/WeightMeasure">
    <Annotation Term="Org.OData.Measures.V1.Unit" Path="WeightUnit"/>
</Annotations>
<Annotations Target="GWSAMPLE_BASIC.Product/Width">
    <Annotation Term="Org.OData.Measures.V1.Unit" Path="DimUnit"/>
</Annotations>
<Annotations Target="GWSAMPLE_BASIC.Product/Depth">
    <Annotation Term="Org.OData.Measures.V1.Unit" Path="DimUnit"/>
</Annotations>
<Annotations Target="GWSAMPLE_BASIC.Product/Height">
    <Annotation Term="Org.OData.Measures.V1.Unit" Path="DimUnit"/>
</Annotations>
<Annotations Target="GWSAMPLE_BASIC.Product/Price">
```

```
    <Annotation Term="Org.OData.Measures.V1.ISOCurrency" Path="CurrencyCode"/>
</Annotations>
```

Formatting Numeric Values

Numeric values in overview pages appear in their short format, using the SAPUI5 `sap.ui.core.format.NumberFormat` utility. You can configure the number of decimal points to display by using information provided in the OData metadata file, or by using annotations.

In the following example, the `scale` attribute in the OData metadata is set to 3, indicating that the properties `Price`, `Width`, `Depth`, and `Height` will be displayed with three decimal points.

Sample Code

```
<EntityType Name="Product" sap:content-version="1">
    ...
    <Property Name="Price" Type="Edm.Decimal" Precision="16" Scale="3"
sap:unit="CurrencyCode" sap:label="Unit Price"/>
    <Property Name="Width" Type="Edm.Decimal" Precision="13" Scale="3"
sap:unit="DimUnit" sap:label="Dimensions"/>
    <Property Name="Depth" Type="Edm.Decimal" Precision="13" Scale="3"
sap:unit="DimUnit" sap:label="Dimensions"/>
    <Property Name="Height" Type="Edm.Decimal" Precision="13" Scale="3"
sap:unit="DimUnit" sap:label="Dimensions"/>
    ...
</EntityType>
```

You can also provide number formatting information in the annotation document in the `com.sap.vocabularies.UI.v1.DataPoint` term, by using the `ValueFormat` property. The `NumberOfFractionalDigits` property can be used to determine the number of decimal points.

In the following example, using the `com.sap.vocabularies.UI.v1.DataPoint ValueFormat` property, the number of decimal points displayed for the `Price` property is 1, as defined in the `NumberOfFractionalDigits` property.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.DataPoint" Qualifier="Price">
    <Record Type="com.sap.vocabularies.UI.v1.DataPointType">
        <PropertyValue Property="Title" String="Unit Price"/>
        <PropertyValue Property="Description" Path="Name"/>
        <PropertyValue Property="Value" Path="Price"/>
        <PropertyValue Property="ValueFormat">
            <Record Type="com.sap.vocabularies.UI.v1.NumberFormat">
                <PropertyValue Property="ScaleFactor" Decimal="1000"/>
                <PropertyValue Property="NumberOfFractionalDigits" int="1"/>
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

Highlighting Numeric Values

You can highlight a numeric value (for example, with a color) by providing `Criticality` or `CriticalityCalculation` information in the `com.sap.vocabularies.UI.v1.DataPoint` annotation.

For more information, see `com.sap.vocabularies.UI.v1.DataPoint` in Annotations Used in Overview Pages [page 2010].

Coloring Cards Based on Threshold Values

With overview pages, column chart cards can be semantically colored based on threshold values.

The threshold values used to semantically color column charts come from the data point annotation that is associated with the measure used in the analytic card. We recommend to only use one measure in the chart if you intend to use this feature.

When you color the chart card, the threshold value that influences the semantic color would also be displayed in the legend; however, in instances where you have more than one measure, then the chart legend would only show "good", "bad" or "neutral" values. For example, a measure can be linked with a data point as follows, if the following two conditions are met:

- This data point should have a value that provides the improvement direction. This typically depicts whether we are dealing with:
  - a maximizing measure where a higher value of a measure is better, or
  - a minimizing measure where a lesser value of a measure is better, or
  - a target measure where the measure value is preferred within a certain range
- It is also required to have all the threshold values based on the examples below.

> **i Note**
>
> We recommend configuring only one measure in the chart if you want to use semantic coloring. If more than one measure is used and all the measures satisfy the above two conditions, then the chart would still be colored semantically with each measure colored based on its own threshold values. The legends, however, won't be very meaningful.
>
> If more than one measure is used and some of the measures don't satisfy the required conditions, those measures will be colored black and the measures that do satisfy the conditions will be semantically colored based on the threshold values.

Examples of Measures

Target Measure (for example, temperature)

From	To	Measure Value
Negative	Critical	>= ThresholdValues.DeviationRange-LowValue
Critical	Positive	>= ThresholdValues.ToleranceRange-LowValue
Positive	Critical	> ThresholdValues.ToleranceRange-HighValue
Critical	Negative	> ThresholdValues.DeviationRange-HighValue

Maximizing Measure (for example, sales)

From	To	Measure Value
Negative	Critical	>= ThresholdValues.DeviationRange-LowValue
Critical	Positive	>= ThresholdValues.ToleranceRange-LowValue

Example of Target and Maximizing Measures

Semantic coloring with target KPIs	Semantic coloring with maximizing KPIs

Minimizing Measure (for example, cost)

Positive	Critical	Negative

From	To	Measure Value
Positive	Critical	> ThresholdValues.ToleranceRange-HighValue
Critical	Negative	> ThresholdValues.DeviationRange-HighValue

Setting Authorizations for Cards

By setting authorization on cards, you can ensure that a user only sees cards or a preview of content from an application for which they have the proper authorization.

An overview page provides a wide overview to the end user so that they can look at and interact with business data in a specific domain. A user can perform key actions such as clicking on a particular card and navigating to

another application. But what if the user does an intent-based navigation to another application and gets an error because they're not authorized to access that application?

To improve this user experience, during card configuration, you can add the property `requireAppAuthorization` in the `manifest.json` file. Once a user has entered an app, this property is checked against `isIntentSupported` to see if the user has the required authorizations to access the app. If so, then the card will be displayed. The advantage here is that the intent can be set explicitly as needed.

Also, `isIntentSupported` is standard functionality in the Cross Application Navigation Service.

Sample Code

```
"sap.ovp":{
    ...
        "cards":{
            ...
            "card01":{
              "model": "salesOrder",
              "template": "sap.ovp.cards.stack",
              "settings": {
                "title": "Stack Card Title",
                "subTitle": "Stack Card",
                "requireAppAuthorization" : "#Action-toappnavsample",
                 "entitySet": "SalesOrderSet"
                    }
                 },
    ...
}
```

Annotations Used in Overview Pages

This topic provides a list of the annotations used in overview pages. They are as follows:

- com.sap.vocabularies.UI.v1.HeaderInfo
- com.sap.vocabularies.UI.v1.DataField
- com.sap.vocabularies.UI.v1.DataFieldForAnnotation
- com.sap.vocabularies.UI.v1.DataFieldForAction
- com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation
- com.sap.vocabularies.UI.v1.DataFieldWithUrl
- com.sap.vocabularies.UI.v1.DataPoint
- com.sap.vocabularies.UI.v1.Identification
- com.sap.vocabularies.UI.v1.LineItem
- com.sap.vocabularies.UI.v1.Facets
- com.sap.vocabularies.UI.v1.FieldGroup
- com.sap.vocabularies.UI.v1.SelectionVariant
- com.sap.vocabularies.UI.v1.PresentationVariant
- com.sap.vocabularies.UI.v1.SelectionPresentationVariant
- com.sap.vocabularies.UI.v1.Chart
- com.sap.vocabularies.UI.v1.KPI
- com.sap.vocabularies.Common.v1.Text

- com.sap.vocabularies.PersonalData.v1

`com.sap.vocabularies.UI.v1.HeaderInfo`

The following properties are supported:

- `TypeName`
- `TypeNamePlural`
- `ImageUrl`
- `Title`
- `Description`

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.HeaderInfo">
    <Record>
        <PropertyValue Property="TypeName" String="Product"/>
        <PropertyValue Property="TypeNamePlural" String="Products"/>
        <PropertyValue Property="Title">
            <Record Type="com.sap.vocabularies.UI.v1.DataField">
                <PropertyValue Property="Value" Path="Name"></PropertyValue>
            </Record>
        </PropertyValue>
        <PropertyValue Property="Description">
            <Record Type="com.sap.vocabularies.UI.v1.DataField">
                <PropertyValue Path="Description" Property="Value"/>
            </Record>
        </PropertyValue>
        <PropertyValue Property="ImageUrl" Path="ProductPicUrl"/>
    </Record>
</Annotation>
```

`com.sap.vocabularies.UI.v1.DataField`

`DataField` is used to display simple text fields. The following properties are supported:

- `Label`: the label of the field
- `Value`: the value of the field, usually pointing to a path in the metadata

Sample Code

```
<Record Type="com.sap.vocabularies.UI.v1.DataField">
    <PropertyValue Property="Label" String="Total Sum"/>
    <PropertyValue Property="Value" Path="TotalSum"/>
</Record>
```

`com.sap.vocabularies.UI.v1.DataFieldForAnnotation`

`DataFieldForAnnotation` can be used to reference a different annotation term, using the `Target` property. The following properties are supported:

- `Label`: the label of the field
- `Target`: reference to a different term in the annotation document

Sample Code

```
<Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation"
Qualifier="WeightMeasure">
    <PropertyValue Property="Label" String="Weight Measure"/>
    <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#WeightMeasure"/>
</Record>
```

`com.sap.vocabularies.UI.v1.DataFieldForAction`

`DataFieldForAction` is used for OData actions that can be preformed on an entity, and refer to a `FunctionImport` action definition in the OData metadata. The following properties are supported:

- `Label`: the navigation label displayed in the footer of the quick view card.
- `Action`: name of the `FunctionImport` action definition to use

Sample Code

```
<Record Type="com.sap.vocabularies.UI.v1.DataFieldForAction">
    <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/Medium" />
    <PropertyValue Property="Label" String="Confirm" />
    <PropertyValue Property="Action"
String="GWSAMPLE_BASIC.GWSAMPLE_BASIC_Entities/SalesOrder_Confirm" />
</Record>
```

`com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`

The `DataFieldForIntentBasedNavigation` record type supports the following properties:

- `SemanticObject`: intent semantic object
- `Action`: action of the intent
- `Label`: the navigation label displayed in the footer of the quick view card

Sample Code

```
<Record Type="com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation">
    <PropertyValue Property="SemanticObject" String="SemanticObject1"/>
    <PropertyValue Property="Action" String="Action1"/>
```

```
      can also contain an intent based navigation with
                                route (static or dynamic with
arguments).<PropertyValue Property="Label" String="App1"/>
    <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/Medium"/>
</Record><Record
Type="com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation"
```

com.sap.vocabularies.UI.v1.DataFieldWithUrl

The `com.sap.vocabularies.UI.v1.DataFieldWithUrl` record type supports the following properties:

- `Url`: Use this property to configure the URL details.

  > **i Note**
  >
  > `Url`

- `Label`>: Use this property to specify the navigation label. The label appears in the quick view card's action footer area for the stack card type.

```
<Record Type="com.sap.vocabularies.UI.v1.DataFieldWithUrl">
    <PropertyValue Property="Label" String="Link to"/>
    <PropertyValue Property="Value" String="Google Maps"/>
    <PropertyValue Property="Url">
        <Apply Function="odata.fillUriTemplate">
            <String>https://www.google.de/maps/place/{street},{city}</String>
            <LabeledElement Name="street">
                <Apply Function="odata.uriEncode">
                    <Path>Address/Street</Path>
                </Apply>
            </LabeledElement>
            <LabeledElement Name="city">
                <Apply Function="odata.uriEncode">
                    <Path>Address/City</Path>
                </Apply>
            </LabeledElement>
        </Apply>
    </PropertyValue>
</Record>
```

`com.sap.vocabularies.UI.v1.DataPoint`

The `DataPoint` term is used to display fields with special formatting. The following properties are supported:

- `Title`: Used in table cards for the column name
- `Value`: Field to display
- `ValueFormat`: Used with the `NumberFormat` annotation to format a decimal value
  - `com.sap.vocabularies.UI.v1.NumberFormat`:
    - `ScaleFactor`: Scale factor for large numbers.
    - `NumberOfFractionalDigits`: Number of decimal points to display.

- `com.sap.vocabularies.UI.v1.Criticality`: An enumeration value that can be used to highlight the value with a certain color. The following values are supported:
  - `com.sap.vocabularies.UI.v1.CriticalityType/Neutral`: Value is displayed using a neutral color (default).
  - `com.sap.vocabularies.UI.v1.CriticalityType/Negative`: Value is displayed using a color for errors (red).
  - `com.sap.vocabularies.UI.v1.CriticalityType/Critical`: Value is displayed using a color for critical values (orange).
  - `com.sap.vocabularies.UI.v1.CriticalityType/Positive`: Value is displayed using a color for positive values (green).
- `com.sap.vocabularies.UI.v1.CriticalityCalculation`: Used to color the value dynamically according to the value in the `com.sap.vocabularies.UI.v1.CriticalityCalculationType` record type. It supports the following values:
  - `ImprovementDirection`: Defines what is considered a positive value. Possible values are:
    - `Minimize`: If `Minimize` is the improvement direction, the `DeviationRangeHighValue` and `ToleranceRangeHighValue` properties are required. and
      - If the value is less than the `ToleranceRangeHighValue`, the value is considered positive and is displayed in green.
      - If the value is between the `ToleranceRangeHighValue` and the `DeviationRangeHighValue`, the value is considered critical and is displayed in orange.
      - If the value is greater than the `ToleranceRangeHighValue` the value is considered negative and is displayed in red.
    - `Target`: If `Target` is the improvement direction, the `DeviationRangeLowValue`, `DeviationRangeHighValue`, `ToleranceRangeLowValue`, and `ToleranceRangeHighValue` properties are required.
      - If the value is between the `ToleranceRangeHighValue` and `ToleranceRangeLowValue`, the value is considered positive and is displayed in green.
      - If the value is between `DeviationRangeLowValue` and `DeviationRangeLowValue`, or between `DeviationRangeHighValue` and `ToleranceRangeHighValue`, the value is considered critical and is displayed in orange.
      - If the value is lower than the `DeviationRangeLowValue` or higher than `DeviationRangeHighValue`, the value is considered negative and is displayed in red.
    - `Maximize`: If `Maximize` is the improvement direction, the `DeviationRangeLowValue` and `ToleranceRangeLowValue` properties are required.
      - If the value is greater than the `ToleranceRangeLowValue`, the value is considered positive and is displayed in green.
      - If the value is between `ToleranceRangeLowValue` and the `DeviationRangeLowValue`, the value is considered critical and is displayed in orange.
      - If the value is lower than `ToleranceRangeLowValue`, the value is considered negative and is displayed in red.
- `com.sap.vocabularies.UI.v1.TargetValue`: A reference value for trend calculation. Use this value to calculate the deviation percent value. For example, `Deviation = (DataPoint Value – ReferenceValue) / ReferenceValue.`

Sample Code

```
<Annotation Term="UI.DataPoint" Qualifier="Eval_by_Country">
```

```
    <Record Type="UI.DataPointType">
        <PropertyValue Property="Title" String="Sales India - Line Card" />
        <PropertyValue Property="Value" Path="Sales" />
        <PropertyValue Property="NumberFormat">
            <Record>
                <PropertyValue Property="ScaleFactor" Int="0" />
                <PropertyValue Property="NumberOfFractionalDigits"
                    Int="3" />
            </Record>
        </PropertyValue>
        <PropertyValue Property="CriticalityCalculation">
            <Record>
                <PropertyValue Property="ImprovementDirection"
                    EnumMember="UI.ImprovementDirectionType/Minimizing" />
                <PropertyValue Property="DeviationRangeHighValue"
                    String="7300" />
                <PropertyValue Property="ToleranceRangeHighValue"
                    String="7200" />
            </Record>
        </PropertyValue>
        <PropertyValue Property="TargetValue" String="2.000 " />
        <PropertyValue Property="TrendCalculation">
            <Record>
                <PropertyValue Property="ReferenceValue" String="5201680" />
                <PropertyValue Property="DownDifference" Int="10000000.0" />
            </Record>
        </PropertyValue>
    </Record>
</Annotation>
```

`com.sap.vocabularies.UI.v1.Identification`

This term is used as a container for card actions. The following record types are supported:

- `com.sap.vocabularies.UI.v1.DataFieldForAction`
- `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`
- `com.sap.vocabularies.UI.v1.DataFieldWithUrl`

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.Identification">
    <Collection>
        <Record
Type="com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation">
            <PropertyValue Property="SemanticObject" String="Action"/>
            <PropertyValue Property="Action" String="toappnavsample"/>
            <PropertyValue Property="Label" String="SO Navigation (M)"/>
            <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/Medium"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Sales Order ID"/>
            <PropertyValue Property="Value" Path="SalesOrderID"/>
        </Record>
    </Collection>
</Annotation>
```

`com.sap.vocabularies.UI.v1.LineItem`

This term is used to display lists of fields in a list or a table. The following record types are supported:

- `com.sap.vocabularies.UI.v1.DataField`
- `com.sap.vocabularies.UI.v1.DataFieldForAnnotation`
- `com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation`
- `com.sap.vocabularies.UI.v1.DataFieldWithUrl`

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.LineItem">
    <Collection>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Sales Order ID"/>
            <PropertyValue Property="Value" Path="SalesOrderID"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Customer Name"/>
            <PropertyValue Property="Value" Path="CustomerName"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Status"/>
            <PropertyValue Property="Value" Path="Status"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataField">
            <PropertyValue Property="Label" String="Note"/>
            <PropertyValue Property="Value" Path="Note"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation"
Qualifier="TotalSum">
            <PropertyValue Property="Label" String="Total Sum"/>
            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#TotalSum"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation"
Qualifier="NetSum">
            <PropertyValue Property="Label" String="Net Sum"/>
            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#NetSum"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.DataFieldForAnnotation"
Qualifier="Tax">
            <PropertyValue Property="Label" String="Tax"/>
            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.DataPoint#Tax"/>
        </Record>
        <Record
Type="com.sap.vocabularies.UI.v1.DataFieldForIntentBasedNavigation">
            <PropertyValue Property="SemanticObject" String="Action"/>
            <PropertyValue Property="Action" String="toappnavsample2"/>
            <PropertyValue Property="Label" String="SO Navigation (M)"/>
            <Annotation Term="com.sap.vocabularies.UI.v1.Importance"
EnumMember="com.sap.vocabularies.UI.v1.ImportanceType/Medium"/>
        </Record>
    </Collection>
</Annotation>
```

`com.sap.vocabularies.UI.v1.Facets`

The `Facets` term contains a collection of facets.

Note

Overview pages only support the `com.sap.vocabularies.UI.v1.ReferenceFacet` record type.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.Facets">
    <Collection>
        <Record Type="com.sap.vocabularies.UI.v1.ReferenceFacet">
            <Annotation Term="com.sap.vocabularies.UI.v1.IsSummary"/>
            <PropertyValue Property="Label" String="Amounts"/>
            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.FieldGroup#Amounts"/>
        </Record>
        <Record Type="com.sap.vocabularies.UI.v1.ReferenceFacet">
            <Annotation Term="com.sap.vocabularies.UI.v1.IsSummary"/>
            <PropertyValue Property="Label" String="Note1"/>
            <PropertyValue Property="Target"
AnnotationPath="@com.sap.vocabularies.UI.v1.FieldGroup#Status"/>
        </Record>
    </Collection>
</Annotation>
```

`com.sap.vocabularies.UI.v1.FieldGroup`

This term consists of a label and a collection of `com.sap.vocabularies.UI.v1.DataFieldAbstract` fields.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.FieldGroup" Qualifier="Amounts">
    <Record Type="com.sap.vocabularies.UI.v1.FieldGroupType">
        <PropertyValue Property="Label" String="Amounts"/>
        <PropertyValue Property="Data">
            <Collection>
                <Record Type="com.sap.vocabularies.UI.v1.DataField">
                    <PropertyValue Property="Label" String="Total Sum"/>
                    <PropertyValue Property="Value" Path="TotalSum"/>
                </Record>
                <Record Type="com.sap.vocabularies.UI.v1.DataField">
                    <PropertyValue Property="Label" String="Net Sum"/>
                    <PropertyValue Property="Value" Path="NetSum"/>
                </Record>
                <Record Type="com.sap.vocabularies.UI.v1.DataField">
                    <PropertyValue Property="Label" String="Tax"/>
                    <PropertyValue Property="Value" Path="Tax"/>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

`com.sap.vocabularies.UI.v1.SelectionVariant`

This term consists of a combination of parameters and filters that query the annotated entity set. You can use `SelectionVariant` separately, or together with the `SelectionPresentationVariant` annotation. The following properties are supported:

- `ID`: Can contain an identifier to reference this instance from external context
- `Text`: Name of the selection variant
- `Parameters`: Collection of `com.sap.vocabularies.UI.v1.ParameterAbstract` terms
- `SelectOptions`: Collection of `com.sap.vocabularies.UI.v1.SelectOptionType` terms used to define filters on this entity set

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.SelectionVariant">
   <Record>
      <PropertyValue Property="SelectOptions">
         <Collection>
            <Record>
               <PropertyValue Property="PropertyName"
PropertyPath="TotalSum" />
               <PropertyValue Property="Ranges">
                  <Collection>
                     <Record>
                        <PropertyValue Property="Sign"

EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeSignType/I" />
                        <PropertyValue Property="Option"

EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeOptionType/BT" />
                        <PropertyValue Property="Low" String="0" />
                        <PropertyValue Property="High" String="8000" />
                     </Record>
                  </Collection>
               </PropertyValue>
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

`com.sap.vocabularies.UI.v1.PresentationVariant`

Defines the way in which the result of a queried collection of entities is displayed in the KPI header. You can use `PresentationVariant` separately, or together with `SelectionPresentationVariant` annotation. The following properties are supported:

- `ID`: Can contain an identifier to reference this instance from external context
- `Text`: Name of the presentation variant
- `SortOrder`: Collection of `com.sap.vocabularies.Common.v1.SortOrderType` records
- `RequestAtLeast`: Collection of fileds that must be part of selection fields. The `requestAtLeast` lets you receive additional dimensions from backend. The additional dimensions are added to the result of a queried collection for further navigation.

> **Note**
>
> In an aggregated service, additional dimensions may cause issues while rendering the chart..

- `Visualizations`: Supports the `LineItem` or `Chart` annotation path

> **Note**
>
> You can only define this property when it is used together with the `SelectionPresentationVariant` annotation.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.PresentationVariant">
   <Record>
      <PropertyValue Property="GroupBy">
         <Collection>
            <PropertyPath>Status</PropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="SortOrder">
         <Collection>
            <Record>
               <PropertyValue Property="Property" PropertyPath="TotalSum" />
               <PropertyValue Property="Descending" Bool="true" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="RequestAtLeast">
          <Collection>
             <Record>
                <PropertyPath>CustomerID</PropertyPath>
             </Record>
          </Collection>
       </PropertyValue>
   </Record>
</Annotation>
```

`com.sap.vocabularies.UI.v1.SelectionPresentationVariant`

Provides a combination of capabilities from `SelectionVariant` and `PresentationVariant`. The `SelectionPresentationVariant` supports the following properties:

- `SelectionVariant`: specify the `SelectionVariant` annotation path.
- `PresentationVariant`: specify the `PresentationVariant` annotation path.

Sample Code

`SelectionPresentationVariant`

```
<Annotation Term="com.sap.vocabularies.UI.v1.SelectionPresentationVariant"
Qualifier="BothSelectionAndPresentation">
    <Record>
        <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#SP2"/>
        <PropertyValue Property="PresentationVariant"
Path="@UI.PresentationVariant#customer"/>
```

```
    </Record>
</Annotation>
```

Sample Code

SelectionVariant

```
<Annotation Term="com.sap.vocabularies.UI.v1.SelectionVariant"
Qualifier="SP2">
    <Record>
        <PropertyValue Property="SelectOptions">
            <Collection>
                <Record>
                    <PropertyValue Property="PropertyName"
PropertyPath="CustomerName"/>
                    <PropertyValue Property="Ranges">
                        <Collection>
                            <Record>
                                <PropertyValue Property="Sign"

EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeSignType/I"/>
                                <PropertyValue Property="Option"

EnumMember="com.sap.vocabularies.UI.v1.SelectionRangeOptionType/EQ"/>
                                <PropertyValue Property="Low" String="Asia
High tech"/>
                            </Record>
                        </Collection>
                    </PropertyValue>
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
</Annotation>
```

Sample Code

PresentationVariant

```
<Annotation Term="com.sap.vocabularies.UI.v1.PresentationVariant"
Qualifier="customer">
    <Record>
        <PropertyValue Property="GroupBy">
            <Collection>
                <PropertyPath>BillingStatusDescription</PropertyPath>
                <PropertyPath>DeliveryStatusDescription</PropertyPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="Visualizations">
            <Collection>
                <AnnotationPath>@UI.LineItem#View2</AnnotationPath>
            </Collection>
        </PropertyValue>
        <PropertyValue Property="SortOrder">
            <Collection>
                <Record>
                    <PropertyValue Property="Property"
PropertyPath="CustomerName" />
                    <PropertyValue Property="Descending" Boolean="true" />
                </Record>
            </Collection>
        </PropertyValue>
    </Record>
```

```
</Annotation>
```

`com.sap.vocabularies.UI.v1.Chart`

Defines the dimensions and measures used in charts. The following properties are supported:

- `Title`
- `Description`
- `ChartType`
- `Measures`: Collection of `PropertyPath` properties.
- `MeasureAttributes`: A collection of `com.sap.vocabularies.UI.v1.ChartMeasureAttributeType` records describing attributes for measures. All measures used in this collection must also be part of the `Measures` property.
- `Dimensions`: Collection of `PropertyPath` properties
- `DimensionAttributes`: Collection of `com.sap.vocabularies.UI.v1.ChartDimensionAttributeType` records describing attributes for dimensions. All dimensions used in this collection must also be part of the `Dimensions` property.

Sample Code

```
<Annotation Term="com.sap.vocabularies.UI.v1.Chart">
   <Record Type="com.sap.vocabularies.UI.v1.ChartDefinitionType">
      <PropertyValue Property="Title" String="View1" />
      <PropertyValue Property="MeasureAttributes">
         <Collection>
            <Record
Type="com.sap.vocabularies.UI.v1.ChartMeasureAttributeType">
               <PropertyValue Property="Measure" PropertyPath="TotalSum" />
               <PropertyValue Property="Role"
EnumMember="com.sap.vocabularies.UI.v1.ChartMeasureRoleType/Axis1" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="DimensionAttributes">
         <Collection>
            <Record
Type="com.sap.vocabularies.UI.v1.ChartDimensionAttributeType">
               <PropertyValue Property="Dimension" PropertyPath="Status" />
               <PropertyValue Property="Role"
EnumMember="com.sap.vocabularies.UI.v1.ChartDimensionRoleType/Series" />
            </Record>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

KPI Annotation

Use this annotation to create KPI tags for your overview page cards. The KPI information appears on the header area of the card and reacts to the filter conditions you set. This annotation provides the capabilities of

`SelectionVariant`, `PresentationVarint`, and `DataPoint` annotations. Additionally, the KPI annotation has the semantic object property and action to configure the navigation parameters.

```
<Annotation Term="UI.KPI" Qualifier="AllActualCosts">
   <Record Type="UI.KPIType">
      <PropertyValue Property="Detail">
         <Record Type="UI.KPIDetailType">
            <PropertyValue Property="DefaultPresentationVariant"
Path="@UI.PresentationVariant#Eval_by_Currency1" />
            <PropertyValue Property="AlternativePresentationVariants">
               <Collection>
                  <Path>@UI.PresentationVariant#Eval_by_Currency_Column</Path>
               </Collection>
            </PropertyValue>
            <PropertyValue Property="SemanticObject" String="Action" />
            <PropertyValue Property="Action" String="toappnavsample" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="SelectionVariant"
Path="@UI.SelectionVariant#Eval_by_Currency_1" />
      <PropertyValue Property="DataPoint" Path="@UI.DataPoint#Eval_by_Country-
Generic" />
      <PropertyValue Property="ID" String="String for KPI Annotation" />
   </Record>
</Annotation>
```

Sample Code

`PresentationVariant` annotation

```
<Annotation Term="UI.PresentationVariant" Qualifier="Eval_by_Currency1">
   <Record>
      <PropertyValue Property="MaxItems" Int="5" />
      <PropertyValue Property="GroupBy">
         <Collection>
            <PropertyPath>Country</PropertyPath>
            <PropertyPath>Currency</PropertyPath>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="SortOrder">
         <Collection>
            <Record>
               <PropertyValue Property="Property" PropertyPath="TotalSales" />
               <PropertyValue Property="Descending" Boolean="true" />
            </Record>
         </Collection>
      </PropertyValue>
      <PropertyValue Property="Visualizations">
         <Collection>
            <AnnotationPath>@UI.Chart#Eval_by_Currency_Donut</AnnotationPath>
         </Collection>
      </PropertyValue>
   </Record>
</Annotation>
```

Sample Code

`SelectionVariant` annotation

```
<Annotation Term="UI.SelectionVariant" Qualifier="Eval_by_Currency_1">
   <Record>
      <PropertyValue Property="SelectOptions">
         <Collection>
            <Record>
```

```
                <PropertyValue Property="PropertyName"
PropertyPath="Country" />
                <PropertyValue Property="Ranges">
                   <Collection>
                      <Record>
                         <PropertyValue Property="Sign"
EnumMember="UI.SelectionRangeSignType/I" />
                         <PropertyValue Property="Option"
EnumMember="UI.SelectionRangeOptionType/EQ" />
                         <PropertyValue Property="Low" String="IN" />
                      </Record>
                   </Collection>
                </PropertyValue>
             </Record>
          </Collection>
       </PropertyValue>
       <PropertyValue Property="Parameters">
          <Collection>
             <Record Type="UI.Parameter">
                <PropertyValue Property="PropertyName"
PropertyPath="Currency_Target" />
                <PropertyValue Property="PropertyValue" String="EUR" />
             </Record>
             <Record Type="UI.Parameter">
                <PropertyValue Property="PropertyName"
PropertyPath="UoM_Target" />
                <PropertyValue Property="PropertyValue" String="KGM" />
             </Record>
          </Collection>
       </PropertyValue>
    </Record>
</Annotation>
```

Sample Code

`DataPoint` annotation

```
<Annotation Term="UI.DataPoint" Qualifier="Eval_by_Country-Generic">
   <Record Type="UI.DataPointType">
      <PropertyValue Property="Title" String="Sales India - Generic Card" />
      <PropertyValue Property="Value" Path="Sales" />
      <PropertyValue Property="ValueFormat">
         <Record>
            <PropertyValue Property="ScaleFactor" Int="2" />
            <PropertyValue Property="NumberOfFractionalDigits" Int="1" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="CriticalityCalculation">
         <Record>
            <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Minimizing" />
            <PropertyValue Property="DeviationRangeHighValue" String="7300" />
            <PropertyValue Property="ToleranceRangeHighValue" String="7200" />
         </Record>
      </PropertyValue>
      <PropertyValue Property="TargetValue" String="2.000 " />
      <PropertyValue Property="TrendCalculation">
         <Record>
            <PropertyValue Property="ReferenceValue" String="5201680" />
            <PropertyValue Property="DownDifference" Int="10000000.0" />
         </Record>
      </PropertyValue>
   </Record>
</Annotation>
```

Text Arrangement

The text arrangement annotation lets you format text.

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="GWSAMPLE_BASIC.SalesOrder/CustomerID">
    <Annotation Term="com.sap.vocabularies.Common.v1.Text" Path="Supplier_Name"/>
    <Annotation Term="com.sap.vocabularies.UI.v1.TextArrangement"
EnumMember="com.sap.vocabularies.UI.v1.TextArrangementType/TextLast" />
</Annotations>
```

In the preceding example, the text `Customer` is bound to the `ContactID` property and appears as shown in the table:

Text Arrangement Type	Result
`TextLast`	ContractID (Customer)
`TextFirst`	Customer (ContractID)
`TextOnly`	Customer

Potentially Sensitive Personal Data

To define a property as sensitive data, configure the target property and set it as personally sensitive.

```
<Annotations Target="SEPMRA_OVW.SEPMRA_C_OVW_SalesOrderType/CompanyCode">
          <Annotation
Term="com.sap.vocabularies.PersonalData.v1.IsPotentiallySensitive"/>
</Annotations>
```

Configuring Overview Page App Extensions

Lets you customize the card and filter area to design a robust application.

Custom Actions

You can assign custom actions to quick view cards that open in the object stream of the stack card. These custom actions are displayed as buttons in the card footer.

Procedure

To define custom action in quick view cards:

1. Create a JavaScript file in your project folder for defining custom controls. For example, `customConfiguration.controller.js`.
2. Open the JavaScript file you created and define the custom actions.

```
sap.ui.define([], function () {
    "use strict";
    return sap.ui.controller("sap.ovp.demo.ext.customConfiguration", {
        /*
    The following Hook function “onCustomActionPress” accepts only one
argument name of the press handler as a string and returns the press handler
function defined in the custom controller.
         */
        onCustomActionPress: function(sCustomAction) {
            if (sCustomAction === "press1") {
                return this.press1;
            } else if (sCustomAction === "press2") {
                return this.press2;
            }
        },
        /*
    The following Press Handler contains the custom actions to be performed
on the click of the button in quickview action footer.
         */
        press1: function(oEvent) {
            window.open("https://www.google.co.in");
        },
        press2: function(oEvent) {
            window.open("http://www.sap.com/index.html");
        },
    });
});
```

3. Configure the application descriptor file.
    1. Add your JavaScript filepath in the `extends` section of the descriptor.

```
"extends": {
    "extensions": {
        "sap.ui.controllerExtensions": {
            "sap.ovp.app.Main": {
                    "controllerName":
"sap.ovp.demo.ext.customConfiguration"
            }
        }
    }
},
```

    2. Add the `customActions` property under the stack card settings `objectStreamCardsSettings` property and define the following mandatory properties:
        - `text`: Enter the string to view in the quick view action footer

- `press`: Enter the name of a press handler defined in the JavaScript file
- `position`: Enter a numeric value to position the order of actions in the quick view action footer

```
"card007_ProductsOutOfStock": {
  "model": "salesOrder",
  "template": "sap.ovp.cards.stack",
  "settings": {
    "itemText": "{{stackCard_itemText}}",
    "title": "Products Out of Stock",
    "subTitle": "SalesOrderSet Stack Card",
    "requireAppAuthorization": "#Action-toappnavsample",
    "entitySet": "SalesOrderSet",
    "identificationAnnotationPath": "com.sap.vocabularies.UI.v1.Identificat
ion,com.sap.vocabularies.UI.v1.Identification#item2",
    "objectStreamCardsSettings": {
      "showFirstActionInFooter": false,
      "customActions": [
        {
          "text": "text1",
          "press": "press1",
          "position": 1
        },
        {
          "text": "text2",
          "press": "press2",
          "position": 10
        },
        {
          "text": "text3",
          "press": "press1",
          "position": 3
        },
        {
          "text": "text4",
          "press": "press2",
          "position": 8
        },
        {
          "text": "text5",
          "press": "press1",
          "position": 5
        },
        {
          "text": "text6",
          "press": "press2",
          "position": 6
        }
      ]
    }
  }
},
```

Global Actions on the Filter Bar

To add custom global actions on the smart filter bar, you need to configure the descriptor file, define a fragment in the view extension, and create `controller.js` file to handle the events. For example:

Sample Code

Descriptor Setting

```
"sap.ui5": {
   "_version": "1.1.0",
      "dependencies": {
        "libs": {
          "sap.ovp": {}
        }
      },
   "models": {
      "i18n": {
        "type": "sap.ui.model.resource.ResourceModel",
        "uri": "i18n/i18n.properties"
      },
      "salesOrder": {
        "dataSource": "salesOrder",
        "settings": {}
      }
       },
       "extends": {
         "extensions": {
           "sap.ui.controllerExtensions": {
             "sap.ovp.app.Main": {
               "controllerName": "sap.ovp.demo.ext.customFilter"
             }
           },
           "sap.ui.viewExtensions": {
             "sap.ovp.app.Main": {
               "SmartFilterBarControlConfigurationExtension|GlobalFilters": {
                 "className": "sap.ui.core.Fragment",
                 "fragmentName": "sap.ovp.demo.ext.customFilter",
                 "type": "XML"
               },
               "SmartFilterBarGlobalActionExtension": {
                 "className": "sap.ui.core.Fragment",
                 "fragmentName": "sap.ovp.demo.ext.customAction",
                 "type": "XML"
               }
             }

           }
         }
       }
   },
```

Add controller and view extensions with a path to the custom controller name and fragment name.

Sample Code

Sample fragment for view extension: `customAction.fragment.xml`

```
<core:FragmentDefinition xmlns="sap.m"
xmlns:smartfilterbar="sap.ui.comp.smartfilterbar" xmlns:core="sap.ui.core">

    <Button text="Action" press="handleCustomAction" type="Transparent"></
Button>
```

```
</core:FragmentDefinition>
```

Sample Code

Sample `customFilter.controller.xml` file

```
sap.ui.define([
    "sap/ui/model/Filter",
    "sap/m/MessageToast"
], function (Filter, MessageToast) {
    "use strict";

    //Extension controller for ovp demo app
    return sap.ui.controller("sap.ovp.demo.ext.customFilter", {
         *  /*
            This is for Custom Global Action
         */
        handleCustomAction : function(){
             var msg = 'Custom Global Action clicked';
             MessageToast.show(msg);
        }
    });
});
```

Custom Cards

Create custom cards to view custom information relevant to your overview page.

Procedure

To create a custom card:

1. Create the following files in your project folder:
   - `Component.js` file
   - `Cardtype.controller.js` file. For example: `List.controller.js`
   - `Cardtype.fragment.xml` file. For example: `List.fragment.xml`
   - `CardtypeHeader.fragment.xml` file. For example: `ListHeader.fragment.xml`
2. Extend the custom overview page component in your component file.

```
sap.ui.define(["sap/ovp/cards/generic/Component", "jquery.sap.global"],

    function (CardComponent, jQuery) {
        "use strict";

        return CardComponent.extend("sap.ovp.cards.list.Component", {
            // use inline declaration instead of component.json to save 1
round trip
            metadata: {
                properties: {
                    "contentFragment": {
                        "type": "string",
```

```
                        "defaultValue": "sap.ovp.cards.list.List"
                    },
                    "controllerName": {
                        "type": "string",
                        "defaultValue": "sap.ovp.cards.list.List"
                    },
                    "annotationPath": {
                        "type": "string",
                        "defaultValue": "com.sap.vocabularies.UI.v1.LineItem"
                    },
                    "countHeaderFragment": {
                        "type": "string",
                        "defaultValue": "sap.ovp.cards.generic.CountHeader"
                    },
                    "headerExtensionFragment": {
                        "type": "string",
                        "defaultValue": "sap.ovp.cards.generic.KPIHeader"
                    }
                },

                version: "${version}",

                library: "sap.ovp",

                includes: [],

                dependencies: {
                    libs: ["sap.suite.ui.microchart"],
                    components: []
                },
                config: {}
            }
        });
    }
);
```

3. Define the fragment file content.

Sample Code

List.fragment.xml

```
<core:FragmentDefinition
        xmlns="sap.m"
        xmlns:core="sap.ui.core"
        xmlns:ovp="sap.ovp.ui"
        xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
    <!-- Here you can put your Card's Content Area -->
</core:FragmentDefinition>
```

Sample Code

ExtendedBarList.fragment.xml

```
<core:FragmentDefinition
        xmlns="sap.m"
        xmlns:core="sap.ui.core"
        xmlns:ovp="sap.ovp.ui"
        xmlns:template="http://schemas.sap.com/sapui5/extension/
sap.ui.core.template/1">
    <!-- Here you can put your Card's Header Area -->
</core:FragmentDefinition>
```

4. Define the content of the controller file.

Sample Code

```
sap.ui.define(["sap/ovp/cards/generic/Card.controller"],
function (Controller) {
    "use strict";
    return Controller.extend("sap.ovp.demo.ext.list.List", {

        onInit: function () {

        },

        onAfterRendering: function () {


        },
    });
});
```

5. Configure the descriptor file.

Sample Code

```
"card008_UrgentPurchaseOrdersk": {
  "model": "purchaseOrder",
  "template": "sap.ovp.demo.ext.list",
  "settings": {
    "title": "Custom Card for purchased items",
    "subTitle": "PurchaseOrderSet custom card",
    "listType": "extended",
    "entitySet": "PurchaseOrderSet",
    ...
    ...
```

Custom Filters

Add custom filters to your overview page application. It provides the end users an option to filter the data displayed in one or more cards.

Steps

1. Create a view extension fragment.

Property	Description
`groupId`	Enter a group ID to associate the custom controller to a group. **i Note** If the group ID does not exist, the filter is added to the default group.
`key`	Enter a property of an entity type to define the filter criteria.
`visibleInAdvancedArea`	Enable this property to view custom filters on the filter bar.
`Input id`	Enter a property of an entity type to define the input criteria.

Sample Code

For example, create a `customFilter.fragment.xml` file and provide the required information.

```
<core:FragmentDefinition xmlns="sap.m"
xmlns:smartfilterbar="sap.ui.comp.smartfilterbar" xmlns:core="sap.ui.core">
    <!-- Product ID Filter -->
    <smartfilterbar:ControlConfiguration groupId="_BASIC" key="ProductID"
                                         label="Product ID (Extension)"
                                         visibleInAdvancedArea="true">
        <smartfilterbar:customControl>
            <Input id="ProductID" type="Text"/>
        </smartfilterbar:customControl>
    </smartfilterbar:ControlConfiguration>
    <smartfilterbar:ControlConfiguration groupId="GlobalFilters"
key="SalesOrderID"
                                         label="Sales Order ID (Extension)"
                                         visibleInAdvancedArea="false">
        <smartfilterbar:customControl>
            <Input id="SalesOrderID" type="Text"/>
        </smartfilterbar:customControl>
    </smartfilterbar:ControlConfiguration>
</core:FragmentDefinition>
```

2. Create a controller extension. For example, create a `customFilter.controller.js` file and define the following functions:
   - Define `getCustomFilters()` to return a filter object.

```
        getCustomFilters: function () {
            var oValue1 = this.oView.byId("ProductID").getValue();
            var oValue2 = this.oView.byId("SalesOrderID").getValue();
            var aFilters = [], oFilter1, oFilter2;
            if (oValue1) {
                oFilter1 = new Filter({
                    path: "ProductID",
                    operator: "EQ",
                    value1: oValue1
                });
                aFilters.push(oFilter1);
```

```
            }
            if (oValue2) {
                oFilter2 = new Filter({
                    path: "SalesOrderID",
                    operator: "EQ",
                    value1: oValue2
                });
                aFilters.push(oFilter2);
            }
            if (aFilters && aFilters.length > 0) {
                return (new Filter(aFilters, true));
            }

        },
```

- Define `getCustomAppStateDataExtension(oCustomData)` to store the application state.

```
        getCustomAppStateDataExtension: function (oCustomData) {
            //the content of the custom field will be stored in the app
state, so that it can be restored later, for example after a back
navigation.
            //The developer has to ensure that the content of the field is
stored in the object that is returned by this method.
            if (oCustomData) {
                var oCustomField1 = this.oView.byId("ProductID");
                var oCustomField2 = this.oView.byId("SalesOrderID");
                if (oCustomField1) {
                    oCustomData.ProductID = oCustomField1.getValue();
                }
                if (oCustomField2) {
                    oCustomData.SalesOrderID = oCustomField2.getValue();
                }
            }
        },
```

- Define `restoreCustomAppStateDataExtension(oCustomData)` to restore the application state.

```
        restoreCustomAppStateDataExtension: function (oCustomData) {
            //in order to restore the content of the custom field in the
filter bar, for example after a back navigation,
            //an object with the content is handed over to this method and
the developer has to ensure that the content of the custom field is set
accordingly
            //also, empty properties have to be set
            if (oCustomData) {
                if (oCustomData.ProductID) {
                    var oCustomField1 = this.oView.byId("ProductID");
                    oCustomField1.setValue(oCustomData.ProductID);
                }
                if (oCustomData.SalesOrderID) {
                    var oCustomField2 = this.oView.byId("SalesOrderID");
                    oCustomField2.setValue(oCustomData.SalesOrderID);
                }
            }
        },
```

3. Add the controller and view extension settings to the manifest.

> **i Note**
>
> Ensure that you use the same entity type in both `viewExtensions` and `globalFilterEntityType` settings. For example, see
>
> ```
>         "extends": {
>             "extensions": {
>                 "sap.ui.controllerExtensions": {
> ```

```
                    "sap.ovp.app.Main": {
                        "controllerName": "sap.ovp.demo.ext.customFilter"
                    }
                },
                "sap.ui.viewExtensions": {
                    "sap.ovp.app.Main": {
                        "SmartFilterBarControlConfigurationExtension|
GlobalFilters": {
                            "className": "sap.ui.core.Fragment",
                            "fragmentName":
"sap.ovp.demo.ext.customFilter",
                            "type": "XML"
                        }
                    }
                }
            }
        },
        "sap.ovp": {
            "globalFilterModel": "salesOrder",
            "globalFilterEntityType": "GlobalFilters",
            ...
            ...
            }
```

Custom Navigation Parameters

Add custom parameters for intent-based navigation to the target application.

Procedure

1. Define the `onCustomParams` function in the controller file extension.

   Sample Code

   ```
            onCustomParams: function(sCustomParams) {
                if (sCustomParams === "getParameters") {
                    return this.getParameters;
                } else if (sCustomParams === "param2") {
                    return this.param2;
                }
            },
   ```

   The custom parameter function inserts URL parameters while navigating to the target application. Configure the following properties:

   - `path`: Property name
   - `operator`: Operator to apply. Possible operations are EQ,NE,LE,GE,LT,GT,BT,CP.
   - `value1`: First operator value applied
   - `value2`: Second operator value. Use only for a range of operators, such as `BT`. If empty, set the value to null.

- `sign`: Specify the current selection to be included or excluded from the filter. Use `I` to include and `E` to exclude.

```
        getParameters: function(oNavigateParams) {
            var aCustomSelectionVariant = [];
            var oCustomSelectionVariant = {
                path: "TaxTarifCode",
                operator: "EQ",
                value1: 5,
                value2: null,
                sign: "I"
            };
            aCustomSelectionVariant.push(oCustomSelectionVariant);
            return aCustomSelectionVariant;
        },
        param2: function(oNavigateParams) {
            oNavigateParams.TaxTarifCode = '3';
            return oNavigateParams;
        }
```

Sample Code

Adding parameters during navigation

```
getParameters: function(oNavigateParams,oSelectionVariantParams) {


            // to get the select option property names, make use of this
to check what values are available to modify
            var aSelectOptionNames =
oSelectionVariantParams.getSelectOptionsPropertyNames();

            var oFilter1 =
oSelectionVariantParams.getSelectOption("Filter1");
            var oFilter2 =
oSelectionVariantParams.getSelectOption("Filter2");


            ///
                Your logic to extract values from oFilter1 and oFilter2
            ///



            /// logic to remove Filter1 and Filter2
            /// assigning empty values to Filter1 and Filter2, with
ignoreEmptyString as true, this will be removed from the Selection Variant


                var Filter1 = {
                    path: "Filter1",
                    operator: "EQ",
                    value1: "",
                    value2: null,
                    sign: "I"
                };


                var Filter2 = {
                    path: "Filter2",
                    operator: "EQ",
                    value1: "",
                    value2: null,
                    sign: "I"
                };
```

```
            /// logic to remove Filter1 and Filter2


            var aCustomSelectionVariant = [];
            var oFilter3 = {
                path: "Filter3PropertyName",
                operator: "EQ",
                value1: "< Value you want to include >",
                value2: null,
                sign: "I"
            };
            aCustomSelectionVariant.push(oFilter3);
            aCustomSelectionVariant.push(oFilter2);
            aCustomSelectionVariant.push(oFilter1);
            return {
                selectionVariant: aCustomSelectionVariant,
                ignoreEmptyString: true
            };
        },
```

2. Configure the descriptor file.
    1. Add a controller extension and specify the path to the custom controller.

```
"extends": {
    "extensions": {
        "sap.ui.controller.Extensions": {
            "sap.ovp.app.Main": {
                    "controllerName":
"sap.ovp.demo.ext.customConfiguration"
            }
        }
    }
},
```

> **i Note**
>
> If a controller file already exists, add the new extension code in the same file.

3. Configure the `customParams` card setting type to return custom parameters. Enter the name of the custom parameter function defined in your custom controller file.

```
"card002_ReorderSoon": {
  "model": "purchaseOrder",
  "template": "sap.ovp.cards.list",
  "settings": {
    "title": "reorder Soon",
    "subTitle": "Less than 10 in stock",
    "listType": "condensed",
    "entitySet": "PurchaseSet",
    "customParams": "<function-name>"    // Depending on the logic you define
in step 1, input the function name.
    ...
    ...
```

Custom Messages

You can customize messages for success, success with no data, and error scenarios. Also, you can add an icon for success scenarios.For error scenarios, default icon is displayed.

To configure custom messages, define `getCustomMessage` in your extension controller file.

```
getCustomMessage: function (oResponse, sCardId) {
 if (sCardId == "card001") {
  if (oResponse && oResponse.getParameters() &&
oResponse.getParameters().success) {
    return {
       sMessage: "My Custom Message for No Data", //message in case of success
and no data
       sIcon: "sap-icon://message-information"    //icon in case of success and
no data
     }
  } else {
    return {
       sMessage: "My Custom Message for Error" //message in case of error
           }
         }
  }
},
```

Custom View Switch

Extend view switch so it reacts based on filter conditions or custom configuration.

Procedure

To define the custom view switch:

1. Create a controller extension (example, `customViewswitch.controller.js` file) and define the `onBeforeRebindPageExtension` function with these input parameters:
   - `aCards` [Type: Array]: List of all visible cards
   - `oSelectionvariant` [Type: Object]: Object containing filter values
2. Define `setTabIndex()` method to pass `<Cardid>` and `<TabIndex>` as parameters. For example, var `oTabIndexList = {"card1" : 2, "card2": 1};`

   > **i Note**
   >
   > The `<TabIndex>` starts with the value one and should not be greater than the length of tabs.

   Configure the key value according to your filter values and pass the `oTabIndexList` object to `this.setTabIndex(oTabIndexList)` as shown here:

Sample Code

Breakout function

```
/*
* Breakout function for dynamic view switch
* */
onBeforeRebindPageExtension: function (aCards, oSelectionVariant) {
    var oTabIndexList = {};
    var oFilterList = this._getFilterList(oSelectionVariant); //Sample
logic
    var oTabIndexList = {};
    if (aCards && aCards.length > 0) {
        for (var i = 0; i < aCards.length; i++) {
            if (aCards[i].id == "card012") {
                if (oFilterList &&
oFilterList.hasOwnProperty("SupplierName")) {
                    if (oFilterList.SupplierName == "SAP") {
                        oTabIndexList["card012"] = 1;
                    } else if (oFilterList.SupplierName == "Talpa") {
                        oTabIndexList["card012"] = 2;
                    }
                }
            }
        }
    }//End of sample logic
    this.setTabIndex(oTabIndexList); //Pass updated oTabIndexList object
here
}
```

Sample Code

Supporting function

```
//get all filters with values
_getFilterList: function (oSelectionVariant) {
    var oFilterList = {};
    if (oSelectionVariant && oSelectionVariant.Parameters &&
oSelectionVariant.Parameters.length > 0) {
        for (var i = 0; i < oSelectionVariant.Parameters.length; i++) {
            oFilterList[oSelectionVariant.Parameters[i].PropertyName] =
oSelectionVariant.Parameters[i].PropertyValue;
        }
    }
    if (oSelectionVariant && oSelectionVariant.SelectOptions &&
oSelectionVariant.SelectOptions.length > 0) {
        for (var j = 0; j < oSelectionVariant.SelectOptions.length; j++) {
            var aRanges = oSelectionVariant.SelectOptions[j].Ranges;
            for (var k = 0; k < aRanges.length; k++) {
                if (aRanges[k].Option == "EQ" && aRanges[k].Low !== "") {

oFilterList[oSelectionVariant.SelectOptions[j].PropertyName] =
aRanges[k].Low;
                }
            }
        }
    }
    return oFilterList;
},
```

3. Configure the controller extension in the descriptor file.

```
"extends": {
            "extensions": {
```

```
                "sap.ui.controllerExtensions": {
                    "sap.ovp.app.Main": {
                        "controllerName": "sap.ovp.demo.ext.customController"
                    }
                }
            }
        }
```

Sharing Overview Pages

Share your application as a tile on the SAP Fiori launchpad or by sending an email link.

SAP Fiori Launchpad

The current state of the filter bar in your application is stored as a tile on the SAP Fiori launchpad.

To save the application, on the filter bar, click *Share* > *Save As Tile* > *Fill required fields* > *Ok*.

Email

You can also share the current state of a overview page by email, which contains a link to your application.

To send an email link, on the filter bar, click *Share* > *Send Email*.

Customizing Overview Pages Using Runtime Capabilities

End users can customize their overview pages by rearranging cards and by hiding or showing cards. They can also apply a filter to the displayed information, which will affect all relevant cards.

End users can customize their overview pages in the following ways:

- Drag and drop cards to rearrange them. You can do this using the mouse, or with the keyboard by pressing `Ctrl` and the arrow buttons.
- Hide cards by doing the following:
  1. In the header bar, click or tap the *Options* icon and select *Manage Cards*.
  2. Use the switch control to hide or show the relevant card.
  3. Click *OK*.
  4. To reset the view to the default settings, in the *Manage Cards* window, click *Reset*.
- Resize a card in horizontal and vertical directions. Or, you can load cards with specific default size. For example, to set default size for a card, configure the card settings in the descriptor file.
  Prerequisite: Set the property `containerLayout: "resizable"` in descriptor configuration file.

```
"defaultSpan": {
```

```
  "rows": 7,
  "cols": 2
}
```

> **i Note**
>
> Resizing of stack card is not supported.

- Filter the information displayed in cards by defining the values or range of values to be displayed for a specific field. For example, display only orders for which the supply date has passed.

Key User Capabilities

A key user is a special role that lets you perform user interface adaptation in overview page application.

Prerequisites

Ensure that you have the required user role (For more information, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152]):

- For SAP Cloud Platform applications, you must have *FlexKeyUser* role assigned.
- For OnPremise applications, you must have *SAP_UI_FLEX_KEY_USER* role assigned.

To enable the user interface adaptation mode, click *Me Area* and select *Adapt UI*.

Table 96: User Interface Adaptation Features

Feature	Description
Hide Card	Lets you hide cards during initial load. The hidden cards are available in the *Manage Cards* section.
Add Card	Lets you add cards from a list of available cards that are hidden.
Edit Card	Lets you modify title, subtitle, KPI header, value, and description, and type of chart. Also, you can modify sorting and navigation properties.
Add/Delete Views	Lets you add view switch field to a card
Modify Card Size	Lets you resize cards, change the number of rows and columns.
Create Card	Lets you create new cards. Currently you can create static link list card and KPI card only.
Others	Similarly you can perform basic card operations like cut, copy, paste, save, cancel, reset and publish.

Best Practices

Maintain property labels for these annotations to enhance readability in the user interface adaptation mode:

Annotation	Property
`PresentationVariant`	text
`SelectionVariant`	text
`DataPoint`	title
`IdentificationAnnotation`	The label is displayed based on the priority set in the `DataFieldForIntentBasedNavigation/ DataFieldWithURL` property
`chartAnnotation`	description

Developing Apps with Analysis Path Framework (APF)

Analysis Path Framework (APF) provides reuse components that allow you to build and enhance interactive analytical Web applications.

You can use APF-based applications to explore KPIs and their influencing factors by drilling down into multidimensional representations of data, such as charts or tables.

APF is available with:

- SAP Business Suite powered by SAP HANA
- SAP S/4HANA

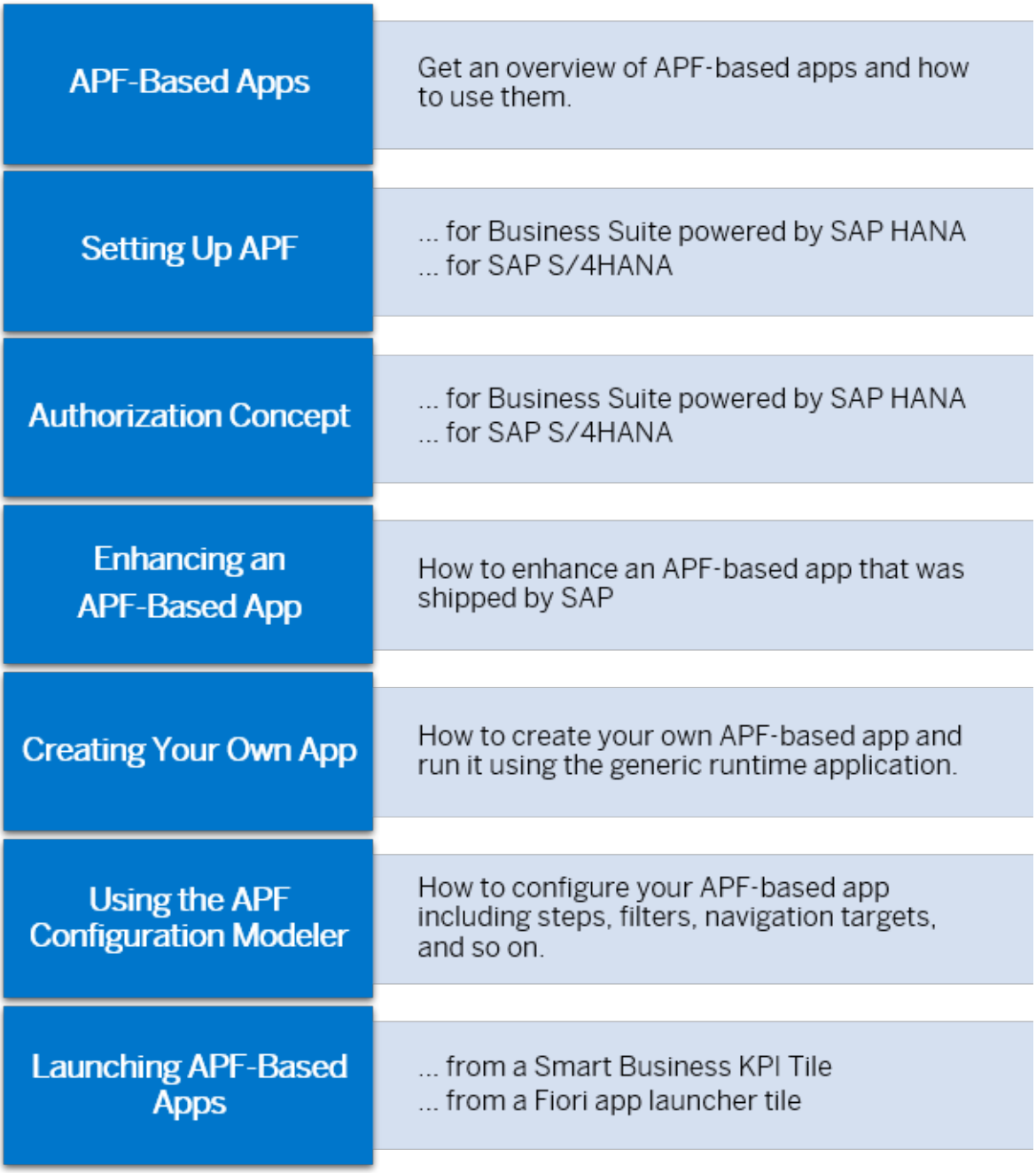
Key Features

The key features of APF include the following:

- APF-based applications:
  - SAP Fiori applications for a data driven, chart-based drill-down analysis targeted at business users
  - Step-by-step analysis of data by building analysis paths using a set of preconfigured analysis steps
  - Select data to filter the information provided in subsequent steps
  - Save, retrieve, and print analysis paths
  - Use of OData services to expose the data on the UI. Each step can have its own OData service so that you can define cross-scenarios that use data from different application areas or systems.
  - Supported data sources for the OData services:
    - Calculation views
    - ABAP CDS views
    - BW OData queries

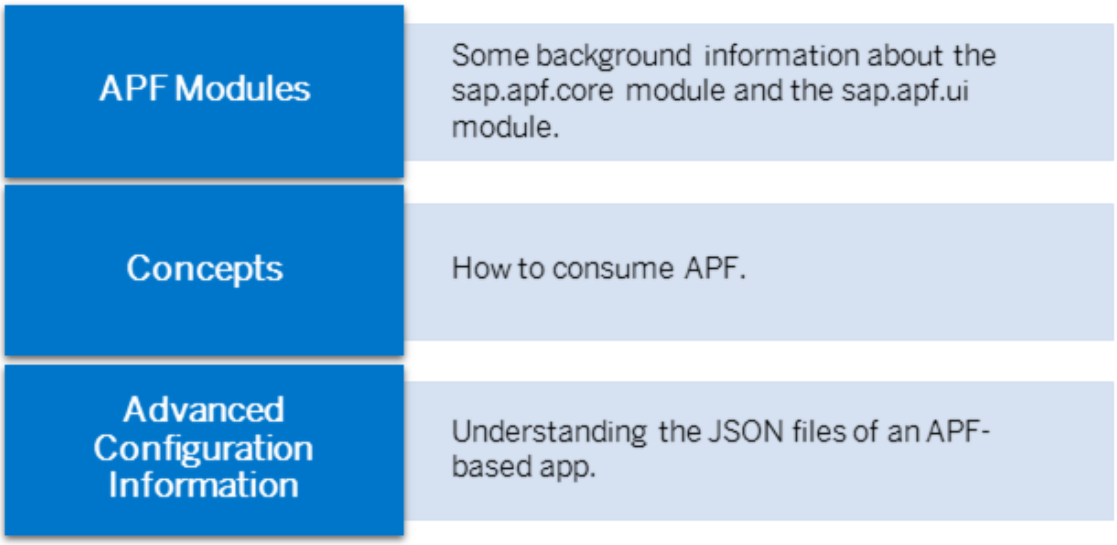
- APF Configuration Modeler:
  - Enables you to create your own APF-based applications in a very quick and easy way and to enhance shipped applications.
  - Configure an APF-based app without having to code:
    - Choose from predefined UI elements, such as chart types, tables, and filters. APF takes care of the interaction between these UI elements.
    - Assign OData services to analysis steps or filters, for example. APF then takes care of the OData request handling, such as when a request is triggered and how it is parameterized.
  - Execute APF-based applications directly from the APF Configuration Modeler using the generic runtime application provided by APF.
  - In SAP S/4HANA, the APF Configuration Modeler provides in-app help to get context-sensitive user assistance for individual entry fields.
- Integration
  - Integration with SAP Fiori and SAP Smart Business, which allows you to launch APF-based applications from different tiles or other apps, handing over filters and parameters to the APF-based application.
  - Navigation from APF-based applications to other apps, for example, to view additional information or to take action in a transactional app, handing over filters and parameters of the APF-based application to the target application.

To learn more about APF-based apps and how to configure them, see the following sections:



- Analytical Applications Based on APF [page 2043]
- Setting Up APF and the APF Configuration Modeler [page 2045]
- Authorization Concept [page 2049]
- Enhancing an APF-Based Application [page 2050]
- Creating Your Own Application [page 2052]
- APF Configuration Modeler [page 2054]
- Launching APF-Based Applications [page 2094]

If you need more technical background information, see the following sections:



- APF Modules [page 2101]
- Concepts [page 2108]
- Configuration Files and Their Structure [page 2121]

Related Information

https://wiki.scn.sap.com/wiki/x/lCe7Gg

Analytical Applications Based on APF

APF supports a sophisticated user interaction paradigm that you can apply to your own APF-based analytical Web application. Users can perform a step-by-step analysis of KPIs by looking at them from different perspectives. For example, they can compare a KPI across different countries or customers, and they can examine tendencies over time. To cover these aspects, you can define different **analysis steps** that the users can then choose from in an analysis step gallery. Analysis steps consist of data that is depicted on the UI in various types of representations, such as charts or tables.

In each analysis step, the users can select data to filter the information provided in subsequent steps. By combining different analysis steps and applying filters, they interactively create their own flexible **analysis paths**.

To get an overview of how an APF-based app looks and how to use it, you can view the APF Demo App in the Demo Kit and watch the following videos:

Title	Video
UI Overview	
Creating an Analysis Path	
Further Options for Analysis Paths	
Filtering Data in Analysis Paths	
Insight to Action	

Configured APF-Based Applications

A number of APF-based applications are available that are already configured and ready to be used.

APF-based applications that are shipped by SAP consist of a Business Server Page (BSP) application along with an app descriptor, the `manifest.json` file. This `manifest.json` file refers to the analytical configuration file, also in JSON format. If you want to run an APF-based application without making any changes to it, the configuration can be read directly from the JSON files. You don't need the APF Configuration Modeler to use this scenario.

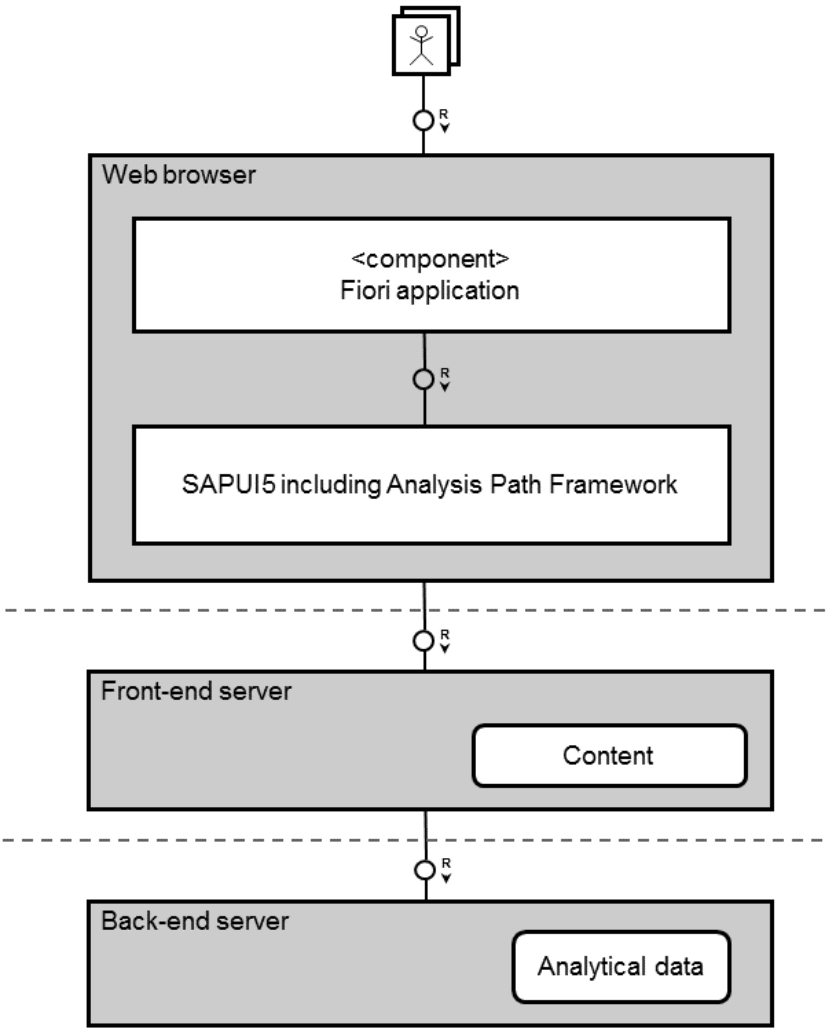
When you import the JSON file of a shipped application into the APF Configuration Modeler to enhance the application, the configuration is written into a repository. For more information, see Import [page 2090] and Enhancing an APF-Based Application [page 2050].

For information about shipped APF-based apps, see the documentation of the individual apps.

Architecture

APF-based Web applications run in a Web browser and communicate with a server using OData service requests.

The architecture is depicted in the following figure:



Setting Up APF and the APF Configuration Modeler

The following sections give information about the different data sources you can use and how APF is implemented depending on the platform you use:

- SAP Business Suite powered by SAP HANA
- SAP Business Warehouse powered by SAP HANA
- SAP S/4HANA

Data Sources

APF consumes OData services. These OData services can be based on different data sources:

- Calculation views
- ABAP CDS views
- BW OData queries

When you use **calculation views**, APF and the APF Configuration Modeler only support the consumption of OData Version 2.0 services provided on SAP HANA extended application services (SAP HANA XS).

All required OData services must exist and must fulfill the following requirements:

- The service definition must contain:

```
annotations {
    enable OData4SAP;
}
```

- For each entity set you must specify the key words “aggregates always”.
  In addition, for each entity set that corresponds to an analytical query view, the statement “keys generate local” must be included.
  Example:

```
service  {
    "sap.hba.ecc/YearMonthQuery.calculationview" as "YearMonthQueryResults"
        keys generate local "GenID"
        aggregates always;
}
```

When you use **ABAP CDS views**, SAP HANA XS is not required. ABAP CDS views are exposed using the SADL framework and SAP NetWeaver Gateway.

For more information about CDS views, go to SAP Help Portal at http://help.sap.com/ and search for "ABAP CDS Entitiy".

BW OData queries are required when you use APF in SAP BW on SAP HANA. For more information about BW OData queries, search for "Creating Easy Queries and Creating OData Queries: A Comparison" in the documentation for your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

Implementation Information for SAP Business Suite powered by SAP HANA and SAP BW on SAP HANA

The following products are required:

- SAP Smart Business foundation component 1.0 (SAP ANALYTICS FOUNDATION 1.0)
  The following software component versions that are relevant for APF are included in this product and installed automatically:
  - HANA CONTENT HBA APF CORE 100 (technical name: HCO_HBA_R_APF_CORE)
    This component contains the SAP HANA server part of APF.

- UISAFND1 100 SP01 (technical name: UISAFND1)
  This component contains the Fiori content for the APF Configuration Modeler.

> **i Note**
>
> This product is also included in all SAP Smart Business products.

- SAP HANA appliance software SPS08 or higher.

Installation of APF on the Front-End Server

As of SAP NetWeaver 7.4, APF is part of software component User Interface Technology (`SAP_UI`) in SAP NetWeaver. For SAP NetWeaver 7.31 and lower releases, you install User Interface Add-On for SAP NetWeaver to use APF.

Implementation of Fiori Content for APF

The following catalog, group, and role are relevant for the APF Configuration Modeler:

- Catalog `SAP_APF_DT_TC_A`
- Group `SAP_APF_DT_TCG_A`
- Role `SAP_APF_DT_TCR_A`

The following catalog and role are relevant for the generic runtime application:

- Catalog `SAP_APF_RT_TC_A`
- Role `SAP_APF_RT_TCR_A`

Implementation of SAP HANA Content for APF

The following role is relevant for the APF Configuration Modeler:

- `sap.hba.r.apf.core.roles:AnalyticalConfiguration`

The following role is relevant for all APF-based applications :

- `sap.hba.r.apf.core.roles::AnalysisPath`

Implementation Information for SAP S/4HANA

The following product is required: `SAP Fiori 1.0 for SAP S/4HANA SPS 01`

The Fiori content for the APF Configuration Modeler is contained in the following software component: `UIS4HOP1 100 SP01`

Installation of APF on the Front-End Server

As of SAP NetWeaver 7.4, APF is part of software component User Interface Technology (`SAP_UI`) in SAP NetWeaver. For SAP NetWeaver 7.31 and lower releases, you install User Interface Add-On for SAP NetWeaver to use APF.

Implementation of Fiori Content for APF

The following technical catalog contains all tiles and target mappings required for APF-related apps: `SAP_TC_CA_APF_COMMON` (SAP: APF Technical Catalog)

The following role, group, and catalogs are relevant for the APF Configuration Modeler:

- Group `SAP_CA_BCG_APF_MODELING` (Analysis Path Framework)
- Business Role `SAP_BR_ANALYTICS_SPECIALIST` (Analytics Specialist)
  This role has the following business catalogs assigned:
  - `SAP_CA_BC_APF_MODELING` (APF Modeling) for the APF Configuration Modeler
  - `SAP_CA_BC_APF_EXECUTION` (APF Execution) for the generic runtime application

The following catalog is relevant for the generic runtime application: `SAP_CA_BC_APF_EXECUTION` (APF Execution)

Assign this catalog to a role. Users that are assigned this role can then use the generic runtime application.

Path Persistence on the Back-End Server

To enable the path persistence on the back-end server, you must ensure that the OData service `BSANLY_APF_RUNTIME_SRV` is activated. If you also use a front-end server, you must also activate the service there.

You can activate the service using transaction `/IWFND/MAINT_SERVICE`.

Administration Information for SAP S/4HANA Cloud

When you, as an administrator, create business users, ensure that they have the business catalog `SAP_CA_BC_APF_MODELING_PC` assigned.

You can also create users based on the business role template `SAP_BC_ANALYTICS_SPECIALIST`. This template includes the business catalog mentioned above, but also other business catalogs, which you may or may not need. You can use the template or assign the business catalog directly to the users as required.

> **i Note**
>
> To authorize users for the OData services required to create a configuration for an APF-based app, you must assign them additional business catalogs as the start authorizations for OData services are not included in the business catalog `SAP_CA_BC_APF_MODELING_PC`.

Authorization Concept

The following sections provide authorization information depending on the platform you use:

- SAP Business Suite powered by SAP HANA
- SAP S/4HANA

Authorization Information for SAP Business Suite powered by SAP HANA

The general authorization concept applies as described in the SAP HANA Security Guide on SAP Help Portal at https://help.sap.com/viewer/p/SAP_HANA_PLATFORM.

In particular, for APF runtime and design time, analytic privileges are needed for the following calculation views:

Runtime	`sap.hba.r.apf.core.v/AnalysisPathQuery`
	`sap.hba.r.apf.core.v/AnalysisPathCountQuery`
	`sap.hba.r.apf.core.v/AnalyticalConfigurationQuery`
	`sap.hba.r.apf.core.v/TextElementQuery`
Design Time	`sap.hba.r.apf.core.v/ApplicationQuery`
	`sap.hba.r.apf.core.v/AnalyticalConfigurationQuery`
	`sap.hba.r.apf.core.v/TextElementQuery`

Security Considerations

You can define various analytic privileges based on the `sap.hba.r.apf.core.v/AnalyticalConfigurationQuery` calculation view and assign them to different users. This can be useful if you use the generic runtime application to run several configurations with various configuration IDs and you want to ensure that only authorized users can see the content of an app. You can specify whether you want to restrict the access to certain applications or even configurations.

Authorization Information for SAP S/4HANA

Configuration Persistence on the Front-End Server

Authorization object /UIF/LREP must be added to any role. Depending on what you want to do, select one of the following values for the field /UIF/ROLE of this authorization object:

- To create configurations using the APF Configuration Modeler and to execute them, select APFADMIN.
- To run a configuration, select APFUSER

Path Persistence on the Back-End Server

To enable the path persistence on the back-end server, you must ensure that the user has the start authorization `S_SERVICE` for the OData service `BSANLY_APF_RUNTIME` both on the front-end server (`IWSG`) and on the back-end server (`IWSV`).

Enhancing an APF-Based Application

You have installed an APF-based application that was shipped by SAP. If you want to make changes to the application, you can do so using the APF Configuration Modeler. Proceed as follows:

> **i Note**
>
> Steps 1 through 8 are relevant for SAP Business Suite powered by SAP HANA only. If you use SAP S/4HANA, proceed with step 9.

1. In your SAP Fiori frontend server system, run transaction `SE80`.
2. In the Repository Browser, select **`BSP Application`** and enter the name of your Business Server Page (BSP) application.
3. In the folder structure, open *Page Fragments* > *config* and double-click the analytical configuration file to open it.
4. Copy the entire content of the file and paste it into a text editor.
5. Save the text file using the file extension `.json`.
6. Go to *Page Fragments* > *i18n* and double-click the .properties file for the required language.
7. Copy the entire content of the file and paste it into a text editor.
8. Save the text file using the file extension .properties.
9. Import the JSON file and the `.properties` text file into the APF Configuration Modeler app in one of the following ways:
   - If you use SAP Business Suite powered by SAP HANA, choose *Import* and specify the files you want to import.

> **i Note**
>
> You can import the `.properties` file in any language. This allows you to switch the development language.

- If you use SAP S/4HANA, choose *Import Delivered Content* and select the desired configuration from the value help.

> **i Note**
>
> You can import the `.properties` file in the development language only.

10. Make the required changes in the APF Configuration Modeler as explained under APF Configuration Modeler [page 2054] and save the configuration.
    For example:
    - Change requests, for example, to replace the shipped data provisioning with your own one.
    - Create new categories for the analysis step gallery.
    - Reassign steps to other categories.
    - Create new analysis steps.
    - Add representation types to a step.
    - Change filter configurations.
    - Add or change navigation targets.
11. Ensure that the correct configuration has been specified in the tile definition. To do so, proceed as follows:
    - For SAP Business Suite powered by SAP HANA:
      If you launch your application from a Smart Business KPI tile, ensure that the correct configuration is selected in the *Configure KPI Tiles* app.
      If you launch your application from a Fiori app launcher tile, maintain the configuration ID as the value for parameter `sap-apf-configuration-id`.
    - For SAP S/4HANA:
      If you launch your application from a Smart Business KPI tile, ensure that the correct configuration is selected in the *Create Tile* app.
      If you launch your application from a Fiori app launcher tile, maintain the application ID and the configuration ID as the value for parameter `sap-apf-configuration-id` in the format `<application ID>.<configuration ID>`.
12. If you changed or created texts that must be translated into languages other than your development language, export the text pool into a .properties file and perform the file-based translation process.

Related Information

APF Configuration Modeler [page 2054]
Configuring the SAP Smart Business KPI Tile [page 2094]
Import [page 2090]
Translation [page 2093]

Creating Your Own Application

If you want to create your own APF-based application, we recommend using the generic APF runtime application, which is shipped with APF. However, you can also create your own runtime application.

Using the Generic Runtime Application

This section describes how to create an APF-based application using the generic APF runtime application.

The generic application already contains important elements that are required for an APF-based application. For example, a BSP application along with the `manifest.json` file is already there so that you only need to create a configuration using the APF Configuration Modeler. In addition, the semantic object and the action, which you need to integrate the application with SAP Smart Business, are already defined for the generic application so that you can easily configure a Smart Business KPI tile to launch the app.

To create a configuration and run it using the generic runtime application, proceed as follows:

1. Open the APF Configuration Modeler
2. Click the + icon to create a new application and enter a description. The *Semantic Object* field is already filled with **`FioriApplication`**. Save your application.

   > **i Note**
   >
   > This step is not required if you use an already existing APF application and just create an additional configuration.

3. Open the application. If the application does not yet have any configurations, you are immediately directed to the screen where you can define one. If you want to create an additional configuration, click *Add* > *New Configuration*. Add categories, steps, representations, filters, and navigation targets as required and save your configuration.
4. You can now execute your application immediately from the APF Configuration Modeler using the *Execute* button or you can launch it from a Smart Business KPI tile or a Fiori app launcher tile.

   > **i Note**
   >
   > A few restrictions apply with regards to the execute feature. For more information, see Executing a Configuration [page 2089].

Generic APF Runtime Application

The generic APF runtime application can be used to execute APF configurations that have been created with the APF Configuration Modeler.

Some restrictions exist for the generic APF runtime application:

- You can run it in one language only. This is because the repository where the APF Configuration Modeler saves the configuration and the texts supports only a single language. The generic application reads texts from this repository and not from the text properties files.
- It cannot have coded elements, such as additional footer elements.
- Using functions for setting default values in the filters is not possible. You can, however, select any of the other default value modes when configuring the filters.

If you don't need more than one language, a footer toolbar, nor a function for defaults in the filters, you can use the generic application to set up an APF-based application in a quick and easy way.

Settings for the Generic APF Runtime Application

The following settings are relevant when you use the generic APF runtime application:

- You must activate the service `af_apf_launch` using transaction `SICF` in the Fiori frontend server system.
- If you use BW OData queries, set the URL parameter `sap-apf-filter-reduction` to **`true`**.

Related Information

APF Configuration Modeler [page 2054]
Configuring the SAP Smart Business KPI Tile [page 2094]

Using Your Own Runtime Application

This section describes how to create an APF-based SAPUI5 application using your own runtime application.

You need your own runtime application if you can't use the generic runtime application shipped with APF, for example, because you need more than one language or a footer toolbar, or because you want to implement extensions, such as your own start filters.

The following steps are required:

1. Build content using the APF Configuration Modeler
   For more information, see APF Configuration Modeler [page 2054].
2. Develop the Component.js file
   For more information, see Consuming APF [page 2111].
3. Maintain the manifest.json file
   For more information, see Descriptor (manifest.json) [page 2124].

In particular, you must maintain the following properties:

Property	Description
`"sap.app.i18n"`	The location of the text resource files. A text resource file contains the texts that the text keys used in the configuration refer to. When you export a configuration using the APF Configuration Modeler, a text file is also exported. More text files containing translations can be added in the same location. Enter the path of the text resource files relative to the web root of the component.
`"sap.app.title"`	Enter "{{`AnalyticalConfigurationName`}}". This refers to the text key included in the exported text properties file and defines the app name.
`"sap.app.dataSources.AnalyticalConfigurationLocation"`	The location of the analytical configuration file, which is created when you export a configuration. Enter the path of the configuration file relative to the web root of the component.
`"sap.app.dataSources.PathPersistenceServiceRoot"`	Enter the service root for the path persistence service. See Descriptor (manifest.json) [page 2124] for the correct values depending on the platform you use.
`"sap.apf.activateFilterReduction"`	This property is relevant if you use CDS views that are executed on the Analytic Engine or BW OData queries. If this is the case, set it to true. If not, set it to false.
`"sap.apf.activateLrep"`	If you use SAP S/4HANA, this property is mandatory and must be set to **`true`**. Otherwise set it to **`false`**.
`"sap.ui5.dependencies"`	Enter your current SAPUI5 version.

APF Configuration Modeler

You use the APF Configuration Modeler to design or enhance your APF-based applications.

To get an overview of the APF Configuration Modeler UI, you can view the APF Configuration Modeler Demo App in the Demo Kit and watch the following video:

To configure a **new** application, the following steps are relevant and build on one another:

1. Create an **application**
   In this step, you create a node under which you can define several different configurations. This step only serves to organize configurations in a structured way. At runtime, the information you enter here is not reflected on the UI.

All configurations that you create under one application node share a common text pool, that is, one text file that contains the texts for all configurations belonging to an application node. This is useful if you have configurations that have very similar content and therefore share large parts of their texts. Having all these texts in one text file can also save translation costs.

2. Create one or more **configurations**
   You can define several configurations under one application. At runtime, only one configuration is used at a time.
3. Create **categories** for a configuration
   The categories you define here are displayed on the first level of the analysis step gallery. They help to clearly arrange the analysis steps, for example, by drill-down dimensions or by KPIs.
   Note that the assignment of steps to categories is done manually. The steps cannot be assigned automatically based on their content.
4. Create **analysis steps** for each category
   Here, you define the steps that a user can choose from in the analysis step gallery and that make up the actual analysis in an APF-based application. Analysis steps have a request assigned that retrieves the data to be analyzed, and one or more representations that determine the chart types.
5. Create **representations** for each analysis step
   Representations define how the data of an analysis step is visualized on the UI, that is, which chart type is used. You can define several representations for each step so that the user can then select the required one in the analysis step gallery. At runtime, the user can switch between the different representations of a step.
6. Create **filters** for a configuration
   Using filters, the user can set global filter values that apply to an entire analysis path.
7. Create **navigation targets** for a configuration
   Navigation targets are displayed in the *Open In...* menu of an APF-based application. When you navigate to another application, the context of the current analysis path is handed over to the navigation target.

> **i Note**
>
> In SAP S/4HANA, the APF Configuration Modeler provides in-app help to get context-sensitive user assistance for individual entry fields. To use the in-app help, you must configure your SAP Web dispatcher accordingly. For more information, see the UI Technology Guide for S/4HANA, section 4.3 "User Assistance Settings".

To **change** an application that was shipped by SAP, you must first import the relevant files into the APF Configuration Modeler. You can then edit the elements listed above. For more information about importing files, see Import [page 2090].

A configuration that was created or enhanced using the APF Configuration Modeler is stored in a table. Each configuration has its own table entry with its own ID. At runtime, the configuration is read from the table, that is, JSON files are not used in this scenario.When you import the JSON file of a shipped application into the APF Configuration Modeler to enhance the application, the configuration is also written into a table.

Adding an Application

For an APF-based application to run, you need to add an application node and create at least one configuration under this node in the APF Configuration Modeler.

To add an application, choose *New* and enter the following:

Field	Explanation
Description	Text used to distinguish the different applications at design time. This description is visible in the APF Configuration Modeler only and not at runtime. It is not translated.
Semantic Object	For SAP Business Suite powered by SAP HANA, this field is prefilled with `FioriApplication`. Keep this default entry if you use the generic APF runtime application. You can change the semantic object any time, but the entry here must be the same as the one you make when configuring the Smart Business KPI tile. In the KPI tile configuration, the semantic object filters the list of configurations so that only those are listed that have been created for the specified semantic object. Otherwise you have to type in your configuration manually. For SAP S/4HANA, this field is not relevant and can be left empty.

To edit the description or the semantic object, choose the *Edit Application* icon for the application you want to change.

Related Information

Configuring the SAP Smart Business KPI Tile [page 2094]

Creating a Configuration

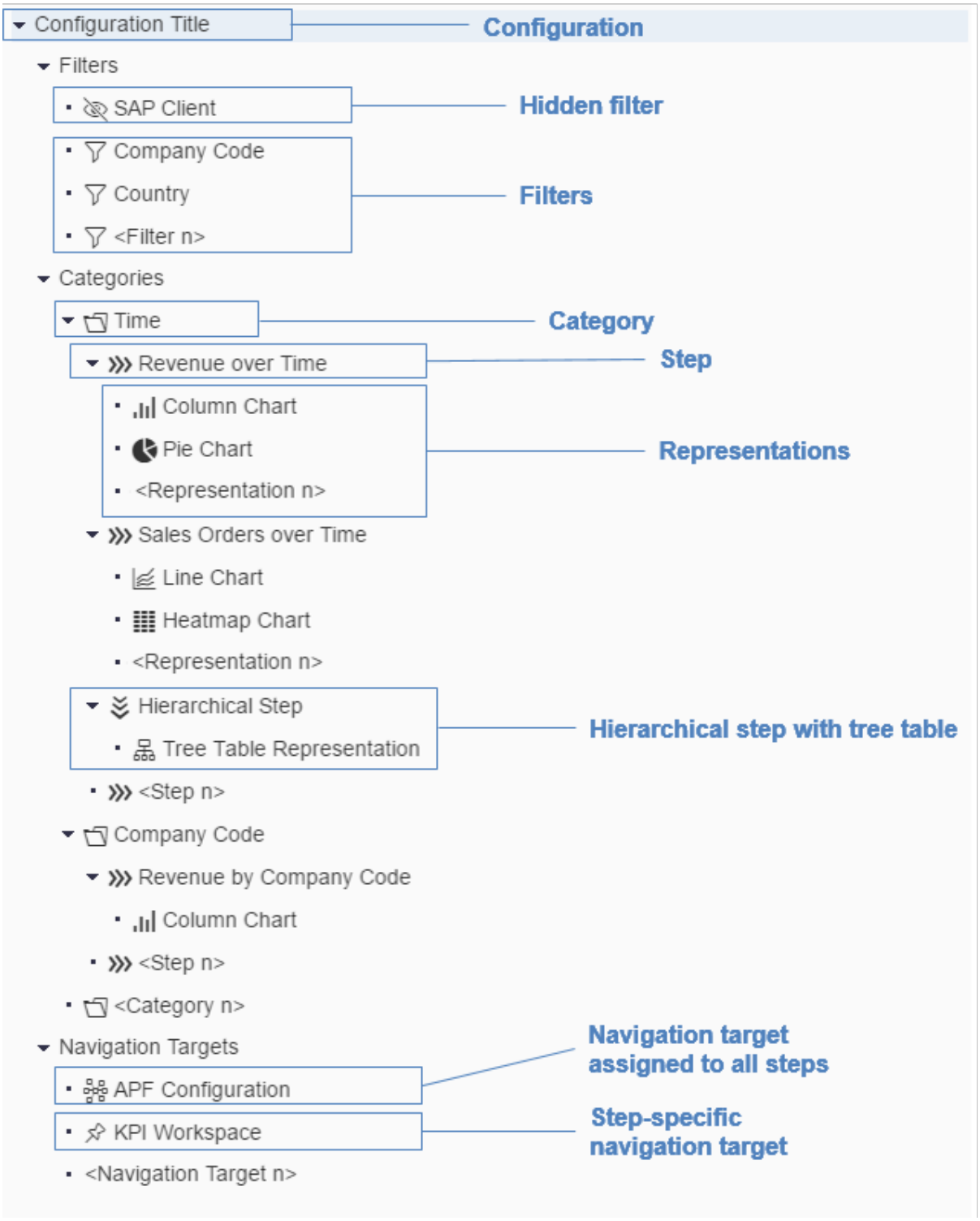
Once you have created an application, you can add a configuration to it. First, navigate to the application by clicking the corresponding row in the application overview. If the application does not yet have any configurations, you are immediately directed to the screen where you can define one. If you want to create an additional configuration, click *Add* > *New Configuration*. In both cases, enter the following:

Field	Explanation
Configuration Title	Text used as title at runtime. This title is translated.
Configuration ID	Generated GUID that is required for configuring a Fiori app launcher tile. This ID cannot be changed.
Semantic Object	Inherited from the application. It is displayed for information only and cannot be changed here.

Field	Explanation
Filter Type	Choose which filter type you want to use for the entire configuration: • Smart filter bar, rendered using the `SmartFilterBar` control, which uses the OData metadata of an entity type • Individually configured filters, rendered using the `FacetFilter` control • No filters

Adding Objects to a Configuration

Within a configuration, you can now start adding objects according to the following structure:



The design time layout of these objects is organized as a tree structure to reflect the dependencies between them. For example, steps can only be added under a category, and representations can only be added under a step.

The order in which the objects are arranged in the APF Configuration Modeler also determines the order in which they are displayed on the UI at runtime. In the analysis step gallery, the categories, for instance, are listed in the same order as they were designed in the APF Configuration Modeler.

When you add a new object, for example, an additional step in a category, it is always added at the end of the list. You can use the *Up* and *Down* arrows to move an object to another position in the list.

Copying Objects

The following objects can be copied:

- Configuration
- Filters
- Categories
- Steps
- Representations
- Navigation targets

In all cases, the object is always copied including all subordinate objects. For example, a category is copied including all steps and representations assigned to it.

A copy is always added in the same place as the original object. For example, when you copy a configuration, the application GUID is retained so that the new configuration is created in the same application. The same applies to, for example, a step: The copy of a step is added in the same category as the original step. You can move the copy to a different category by changing the category assignment in the step itself.

Creating Categories

Categories help to group analysis steps and organize the analysis step gallery. To create a category, click *Add* > *New Category*. A category only needs a title before you can continue to assign steps to it.

At runtime, the categories are displayed in the same order as they appear here.

Creating Steps

The following video shows how to configure an analysis step:

To create an analysis step, select the category you want to create the step in and click *Add* > *Step* > *New Step*.

Analysis steps can be assigned to multiple categories. To assign a previously created step to another category, proceed in one of the following ways:

- Select the category you want to assign the step to and click *Add* > *Step* > *Existing Step*.
- Select the step and add the category in the *Category Title* field.

> **i Note**
>
> When you edit an analysis step that is assigned to multiple categories, the changes take effect in all categories in which this step is used.

At runtime, the steps are displayed in the analysis step gallery in the same order as they appear in the tree structure. You can change the order of the steps by moving them up or down the tree structure using the arrow icons. The order of the steps can be defined for each category individually, that is, steps can have a different order in different categories.

To create or edit an analysis step, enter the following:

Title and Assignment to Categories

Field	Explanation
Step Title	The title displayed in the analysis step gallery and with the thumbnails in the analysis path display.
Step Long Title	The title displayed above the representation in the analysis step display. If no long title is defined, the title entered in the *Step Title* field is used instead.
Category Assignments	The categories in which the analysis step is displayed in the analysis step gallery. At least one category must be entered here. When first creating a step, this field is already filled with the category in which you create the step. You can enter additional categories so that the step is displayed in all of these categories. You can also remove category assignments here and you can use this field to change the category assignment of a step by selecting a different category from the value help. This is particularly useful if you copy a step and then want to assign it to a different category.

Request

Here, you enter information for the request that defines the data provisioning for the analysis step.

Field	Explanation
Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
Entity Set	Entity set that corresponds to the data source, for example, the SAP HANA view.
Properties	The drill-down dimensions, measures, and additional information, for example, the currency, used in the step. Ensure you select only those properties that shall be displayed in the representation.
Selectable Property	The property that is selectable in the chart. If nothing is entered here, nothing can be selected in any representation of this step. If you have selected both a key property and the corresponding text property in the *Properties* field, you can define for the selectable property what is displayed at runtime: the key only, the text only, or both key and text. This takes effect in the selection information popup, showing which elements are selected in the current analysis step as well as in the filter information popup, showing the filters that affect the current analysis step.
Label	The label of the selectable property, which is displayed in the selection information popup as well as in the filter information popup at runtime. The default label text for each property is derived from the `sap:label` annotation of the property. You can see that the default text is used when you see *Label (Default)* in front of the entry field. You can overwrite the default text with your own label text as required. When you delete the label text, the default label is displayed again.

Data Reduction

In this section, you can reduce the number of data records that are sent to the frontend for this analysis step. The top n approach is useful if you know that for this analysis step it is sufficient to look at the most relevant data records only instead of sending all data points to the UI.

> → Tip
>
> Add "top <n>" to the title of the analysis step, for example, "Revenue by Customer (Top 10)". This is helpful because, in the chart, you cannot see that data reduction has been applied.

> **i Note**
>
> The top n values you see in the analysis step are not necessarily the highest numbers in the data set. Depending on the sorting direction and the measure itself, it is also possible that you see the lowest values. Examples:
>
> - If you sort by revenue in ascending direction, you get the bottom n instead of the top n.
> - If you choose a measure for which low values are positive, such as costs or reject rates, the lowest values are interpreted as top n.

> **i Note**
>
> You can use the top n function if you want to restrict your data to up to 10,000 records. If you need more than 10,000 records, don't use data reduction.

Field	Explanation
Data Reduction Type	Choose whether you want to use top n or no data reduction.
Number of Records	Enter a number between 1 and 10,000.
Sorting Field, Direction	You can define which properties are applied to the data request and subsequently to the chart as sorting criteria. You can also specify the sorting direction (ascending or descending) for each property. The sorting criteria you enter here are copied to each representation of the step and cannot be changed in the representation configuration afterwards.

Filter Mapping

This section becomes visible when you choose a selectable property in the *Request* section above.

Filter mapping is useful when a selection that can be made in one analysis step cannot be handled by a subsequent step. In this case, the system determines the source filter property based on the selections made in a chart and maps them to other filters that can be used in the requests for subsequent steps in the path (target properties). Filter mapping is optional, that is, if the fields are not filled, no filter mapping is executed for this step.

Define the lookup request that maps the source property to the target properties as follows:

Field	Explanation
Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
Entity Set	Entity set that corresponds to the SAP HANA view.

Field	Explanation
Target Property	Filter properties that the selectable property shall be mapped to. These are filter properties that can be used in the request for subsequent steps. If both a key property and the corresponding text property exist, you can define what is displayed at runtime: the key only, the text only, or both key and text.
Label	The label of the target property, which is displayed in the filter information popup at runtime. The default label text for each property is derived from the `sap:label` annotation of the property. You can see that the default text is used when you see *Label (Default)* in front of the entry field. You can overwrite the default text with your own label text as required. When you delete the label text, the default label is displayed again.
Keep Selected Property as Filter	Determines whether the source filter property is kept in the cumulative filter in addition to the mapped filter properties.

Assignment of Navigation Targets

Here, you can assign step-specific navigation targets to your analysis step. At runtime, these navigation targets are displayed in the *Open In...* menu when this analysis step is the active one. Navigation targets that are relevant for all steps are also displayed, but cannot be changed here.

Field	Explanation
Step-Specific	Shows the assigned step-specific navigation targets for this step. You can assign additional ones by selecting them from the list of previously created step-specific navigation targets. You can also delete assignments.
Assigned to All Steps	Navigation targets that are relevant for all analysis steps and are always displayed in the *Open In...* menu at runtime. This field is read-only.

Corner Texts for Thumbnails

The texts entered here are displayed in the four corners of the step thumbnail in the analysis path display. All of them are optional.

Corner texts entered at step level are used as default for the corner texts at runtime as long as no text has been maintained at representation level for the specific corner. They are also used to prefill the corner text fields at representation level.

When you change a specific corner text at step level, this change is reflected in the representation and at runtime only if the corresponding field has not yet been edited on representation level.

Related Information

Creating Navigation Targets [page 2085]

Creating Hierarchical Steps

Hierarchical steps are analysis steps in which the data is depicted in a hierarchical table. The data for hierarchical steps is provided in a hierarchical form, that is, it has a parent-child-relationship. The property in the first column of the hierarchical table has the parent information and provides the hierarchical structure. You can expand it to show the child items.

To create an analysis step, select the category you want to create the step in and click *Add* > *Step* > *New Hierarchical Step*.

To create or edit a hierarchical step, enter the following information:

Title and Assignment to Categories

Field	Explanation
Step Title	The title displayed in the analysis step gallery and with the thumbnails in the analysis path display.
Step Long Title	The title displayed above the representation in the analysis step display. If no long title is defined, the title entered in the *Step Title* field is used instead.
Category Assignments	The categories in which the analysis step is displayed in the analysis step gallery. At least one category must be entered here. When first creating a step, this field is already filled with the category in which you create the step. You can enter additional categories so that the step is displayed in all of these categories. You can also remove category assignments here and you can use this field to change the category assignment of a step by selecting a different category from the value help. This is particularly useful if you copy a step and then want to assign it to a different category.

Request

Here, you enter information for the request that defines the data provisioning for the analysis step.

Field	Explanation
Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
Hierarchical Entity Set	Entity set that corresponds to the data source, for example, the SAP HANA view. As soon as a service has been entered, this field is filled automatically.
Hierarchical Property	The property that is provided in a hierarchical form. This property is always displayed as the first column of the hierarchical table. As soon as a service has been entered, this field is filled automatically.
Non-Hierarchical Properties	The properties that are displayed in the following columns of the table. Ensure you only enter the following: • Measures • The text of the hierarchical property • Properties that have the same value across the entire hierarchy, for example, the reporting currency **i Note** Do not use other drill-down properties apart from the hierarchical property because the tree table cannot handle additional drill-down criteria.
Selectable Property	The property that is selectable in the chart. **i Note** At runtime, you cannot add more than one hierarchical step with the same hierarchical selectable property.

Filter Mapping

This section becomes visible when you choose a selectable property in the *Request* section above.

Filter mapping is useful when a selection that can be made in one analysis step cannot be handled by a subsequent step. In this case, the system determines the source filter property based on the selections made in a chart and maps them to other filters that can be used in the requests for subsequent steps in the path (target properties). Filter mapping is optional, that is, if the fields are not filled, no filter mapping is executed for this step.

Define the lookup request that maps the source property to the target properties as follows:

Field	Explanation
Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
Entity Set	Entity set that corresponds to the SAP HANA view.
Target Property	Filter properties that the selectable property shall be mapped to. These are filter properties that can be used in the request for subsequent steps.
Keep Selected Property as Filter	Determines whether the source filter property is kept in the cumulative filter in addition to the mapped filter properties.

Assignment of Navigation Targets

Here, you can assign step-specific navigation targets to your analysis step. At runtime, these navigation targets are displayed in the *Open In...* menu when this analysis step is the active one. Navigation targets that are relevant for all steps are also displayed, but cannot be changed here.

Field	Explanation
Step-Specific	Shows the assigned step-specific navigation targets for this step. You can assign additional ones by selecting them from the list of previously created step-specific navigation targets. You can also delete assignments.
Assigned to All Steps	Navigation targets that are relevant for all analysis steps and are always displayed in the *Open In...* menu at runtime. This field is read-only.

Corner Texts for Thumbnails

The texts entered here are displayed in the four corners of the step thumbnail in the analysis path display. All of them are optional.

Corner texts entered at step level are used as default for the corner texts at runtime as long as no text has been maintained at representation level for the specific corner. They are also used to prefill the corner text fields at representation level.

When you change a specific corner text at step level, this change is reflected in the representation and at runtime only if the corresponding field has not yet been edited on representation level.

Tree Table

For hierarchical steps, there is only one representation available to vizualize the data: the tree table. To create a representation for a hierarchical step, select the step, click *Add* > *Representation* and enter the following:

Field	Explanation
Visualization	Prefilled with *Tree Table*
Basic Data	Enter the hierarchical property for the first column of the tree table and properties for further columns. If you have selected both a key property and the corresponding text property for the hierarchical property at step level, you can define what is displayed in the first column of the tree table: the key or the text. If you want both key and text to be displayed, use *Key* for the hierarchical column and define an additional column for the corresponding text property. The default label text for each property is derived from the `sap:label` annotation of the property. You can see that the default text is used when you see *Label (Default)* in front of the entry field. You can overwrite the default text with your own label text as required. When you delete the label text, the default label is displayed again.
Sorting	You can define whether certain measures are applied to the data request and subsequently to the chart as sorting criteria. You can also specify the sorting direction (ascending or descending) for each measure. If you don't specify a sorting field, the data is displayed in the order provided by the OData service.
Corner Texts for Thumbnail	The texts that are displayed in the four corners of the thumbnail in the analysis path display. All of them are optional. The fields are prefilled with the texts entered at step level, if applicable. When a text is entered or changed on representation level, this text takes precedence over the corresponding text entered at step level and is displayed at runtime. When you now again change the same text at step level, this change is not copied to the representation and also not reflected at runtime. This only applies to those representations where a change has been made on representation level.

> **i Note**
>
> A preview is not available for tree tables.

Creating Representations

The following video shows how to configure a representation:

To create a representation, select the step you want to create the representation for, click *Add* > *Representation* and enter the following:

Field	Explanation
Visualization	Select a chart type or table representation from the value help. **i Note** If there is an asterisk in front of the chart type, this indicates that the properties selected on step level are not sufficient to configure this chart type. To be able to use it, you must go back to the analysis step configuration and add more dimensions or measures as required. In the analysis step gallery, the chart type is indicated by an icon and a label.
Basic Data	Depending on the chart type you chose, enter data such as the dimension for the horizontal axis and the legend, and the measure. Some chart types offer more fields, for example, the stacked column chart, or fewer fields, for example, the pie chart. Mandatory fields are marked with an asterisk. If you have selected both a key property and the corresponding text property for a dimension at step level, you can define for each representation what is displayed at runtime: the key only, the text only, or both key and text. The default label text for each property is derived from the `sap:label` annotation of the property. You can see that the default text is used when you see *Label (Default)* in front of the entry field. You can overwrite the default text with your own label text as required. When you delete the label text, the default label is displayed again. Ensure that all properties of drill-down dimensions that are selected for the step are also used in the representation for this step. Otherwise the data may not be displayed correctly.

Field	Explanation
Sorting	You can define whether certain properties are applied to the data request and subsequently to the chart as sorting criteria. You can also specify the sorting direction (ascending or descending) for each property. If you don't specify a sorting field, the data is displayed in the order provided by the OData service. If sorting criteria have been defined in the Data Reduction section of the step configuration, they are copied to the representation and cannot be changed here. **i Note** Not all representations support sorting by more than one property to the full extent.
Corner Texts for Thumbnail	The texts that are displayed in the four corners of the thumbnail in the analysis path display. All of them are optional. The fields are prefilled with the texts entered at step level, if applicable. When a text is entered or changed on representation level, this text takes precedence over the corresponding text entered at step level and is displayed at runtime. When you now again change the same text at step level, this change is not copied to the representation and also not reflected at runtime. This only applies to those representations where a change has been made on representation level.

You can click *Preview* to see what the representation and the thumbnail will look like at runtime.

> **i Note**
>
> The preview is based on dummy data and does not show the real data configured for the step.

At runtime, for each representation there is also an alternative list view available. The user can toggle between the chart representation and the list view. Exception: The alternative list view is not available if the representation is a table. The alternative list view is created implicitly for each representation and you do not have to define it in the APF Configuration Modeler.

Configuring Filters

APF-based applications can use filters that narrow down the data of an entire analysis path. You can either configure each filter individually or you can use a smart filter bar:

- Individually configured filters
  You can configure separate filters that are rendered using the `FacetFilter` control. This option is useful if you don't have a service available that you can use to configure a smart filter bar and if you don't require

features such as range selections, date picker, or paging, which are supported by the `SmartFilterBar` control only.

- Smart filter bar
  You can configure an entire filter bar in just one step using the `SmartFilterBar` control. This option is useful if you have an entity type that provides all of the properties that you want to offer as filters as well as the required annotations to configure the smart filter bar. The `SmartFilterBar` control uses the OData metadata of the entity type to create the filter bar. The possible filter criteria by which you can filter are automatically derived based on the annotations of the used service.
  The advanced options of the smart filter bar include the following:
  - Rule-based filtering using operators such as "contains", "between", or "greater than"
  - Use of the `DatePicker` control
  - Paging for value lists, which can avoid performance issues in the case of very large lists.

For each configuration, you define whether you want to use individually configured filters, or the smart filter bar, or no filters at all. Once you have chosen one of the filter types, the tree structure of your configuration is adapted accordingly.

> **i Note**
>
> When you switch between the two filter types or from one of the filter types to no filter, you lose the current settings. When you switch the filter type and then want to switch back again, you have to define your settings once more.

Individually Configured Filters

Use individually configured filters if you want to use more than one OData service to configure your filters.

The following video shows how to configure filters:

For each filter, you can configure the following:

- The property for which a filter is displayed
- The values that are listed in the value help for the property
  You may have to configure a value help request to generate the list of values that the user can choose from to filter the data.
  You can also maintain the values manually if you don't have a service available that produces the desired values.
- The values that are preselected in the value help.

When you launch your APF-based application from a Smart Business KPI tile, you must also consider whether an additional filter is handed over with the Smart Business context and how that affects the filter you are configuring.

> **i Note**
>
> Do not define a filter for URL parameters. For example, if you configure SAPClient as URL parameter, you cannot define an additional filter for SAPClient because this may result in an empty data response.

You can use the fields listed in the following table to configure a filter.

Not all of the fields are relevant for each use case. For example, you can configure a value help request to determine the entries in the value help list. A context resolution request is necessary if the Smart Business context contains not only single values, but also other operators. The context resolution request then resolves the context into a list of single values. This is necessary because a filter can only display single values. For details about possible use cases, see Use Cases for Configuring Filters [page 2077].

Field		Explanation
Basic Data	Filter Title	The label displayed for the filter at runtime.
	Property	The property for which you want to configure a filter. The value help is populated with the properties of all entity sets that are used in the configuration. **i Note** The value help is empty if you have not yet entered any request data in the configuration. To avoid this, ensure you enter the required request data before configuring a filter. Request data can be entered, for example, in the *Value Help* section below or in an analysis step. The filter property is also used as the property displayed in the filter at runtime if an alias is not defined. Values that the user selects in the filter are handed over to the entire analysis path as filter.

Field		Explanation
	Do Not Show Filter at Runtime	Select this checkbox if you do not want to expose this filter at runtime. This is useful when a request requires a mandatory filter or parameter that is not coming in from outside APF, for example, with the Smart Business context, and if it is not necessary that users see or change the filter. **i Note** When you select this checkbox, any information you may have entered for value help request and context resolution request is deleted. When you decide later on to show the filter at runtime, you must fill in this information again as required.
	Selection Mode	Choose whether you want to allow single selection or multiple selections.

Field		Explanation
Default Values	Default Value Mode	Select *None* if you do not want any values to be selected by default. This option is useful for filters that are optional, have a lot of values, and support multiple selections. Select *Automatic Values* if you want the system to determine the default values automatically. In case of single selection, the first entry in the value help of the filter is selected by default. In case of multiple selection, all values are selected by default. Select *Fixed Values* if you want to list specific values that are preselected in the value help. **i Note** Do not enter sensitive data as default values for filters. Other APF users can read the configuration and thus have access to this data. Select *Function* if you want to specify a JavaScript function that calculates the default values. This function must be included in the Business Server Pages (BSP) application. **i Note** Using a function to calculate the default values is not possible if you use the generic APF runtime application.

Field		Explanation
	Default Values/Function	Depending on the default value mode, either specify the fixed default values or a function. If you want to enter a date as a fixed default value, you can do so in one of the following formats: • <dd.mm.yyyy>, for example, 28.04.2018 • <mm/dd/yyyy>, for example, 04/28/2018 • <yyyy-mm-dd>, for example, 2018-04-28 **i Note** If a context is handed over from a Smart Business KPI tile for this filter property, this context replaces the default values. The default values are used as a fallback only. This is necessary because it is not possible to have an empty filter.
Value Help	Value Help Mode	Select *Value Help Request* if you want to specify a request to generate the list of values in the value help. Select *Configured List of Values* if you want to enter the values for the value help manually. Select *None* if none of the above options applies. The values in the value help can then result only from default values or from a context that is handed over from outside, for example, from a Smart Business KPI tile.
	Values (displayed only when you select *Configured List of Values*)	Manual list of values for the value help.
	The following fields are displayed only when you select *Value Help Request*.	

Field		Explanation
	Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
	Entity Set	Entity set that corresponds to the data source, for example, the SAP HANA view. This field is mandatory if a service has been entered.
	Properties	The properties that determine the values in the value help. Select at least the filter property or the alias property. You can also select further properties. For example, you can select CompanyCode and Revenue so that the value help lists those company codes for which revenue exists. If you select both a filter property and the corresponding text property, for example, CompanyCode and CompanyCodeName, a concatenation of key and text is displayed in the filter at runtime and you can search for both of them.
	Alias	The property displayed in the filter. It is used if the field name of the filter's property is different in the value help request. If the property and the alias are the same, the alias can be omitted.

Field		Explanation
Context Resolution	Use Value Help Request	You can configure the same request for both value help and context resolution by selecting the *Use Value Help Request* checkbox in the context resolution request. When you select the checkbox, all context resolution fields are filled with the same entries as the corresponding value help fields and cannot be edited. When you deselect the checkbox, the value help entries remain but can now be edited. If you change the value help entries, the context resolution entries are updated accordingly.
	Service	The request used for resolving a context into single values for the value help. Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually. You should specify a context resolution request if there is a possibility that a context is handed over to the APF-based app that does not contain single values. If you did not specify a context resolution request, but a context resolution is necessary to be able to display the values in the value help, the filter for the corresponding property is not displayed on the UI. Instead, the context is applied to the analysis path in the background.
	Entity Set	Entity set that corresponds to the data source, for example, the SAP HANA view.
	Properties	The properties that determine which values are preselected in the filter. Select at least the filter property or the alias property. You can also select further properties.

Filter Dependencies

At runtime, selections you make in a filter are applied to all subsequent filters, just as selections in an analysis step filter the data of subsequent steps. Selections in a filter are exposed as filter criteria to the value help request or context resolution request of any subsequent filter. Subsequent filters are positioned to the right of the filter where the selection is made (in left-to-right languages). This dependency only exists if a property of a filter is also used as filterable property or parameter in the value help request or context resolution request of a subsequent filter.

At design time, the order in which the filters are arranged in the APF Configuration Modeler app determines the order in which the filters appear at runtime. The filter that is at the top of the list at design time appears on the left at runtime. You can determine the way in which filters influence each other by arranging them in the required sequence. A filter that you position at the top of the list in the APF Configuration Modeler influences any filter that is further down the list and that uses the same filter property. In other words, a filter can depend on the selections in any filter that is higher up in the list.

> **i** Note
>
> If filter A has a property that filter B requires as parameter or mandatory filter, filter A must be positioned in front of filter B so that the dependency takes effect. Otherwise, filter B is not provided with this property and therefore does not work.

Use Cases for Configuring Filters

A filter for an APF-based application can be based on:

- A configured filter
- A filter that is handed over with the Smart Business context of a KPI tile
- Both

The following table compares the typical use cases derived from these options and the resulting configuration:

	Use Case 1: Filter Independent of Smart Business	Use Case 2: Filter Determined by Smart Business Context	Use Case 3: Filter with Default Values Determined by Smart Business Context	Use Case 4: No Filter
Filter configured?	Yes	Yes	Yes	No
Value help request configured?	Yes	No	Yes	No
Filter passed with Smart Business context?	No	Yes	Yes	Yes

For a detailed explanation, see the following sections.

Use Case 1: Filter Independent of Smart Business

For this use case, all of the following apply:

- You configure a filter for a property in your application.
- You configure a value help request to determine the entries in the value help list.
- The Smart Business KPI tile does not pass a corresponding filter in the context.

As a result, the filter is displayed in the application and its value help is populated with all values retrieved by the configured value help request.

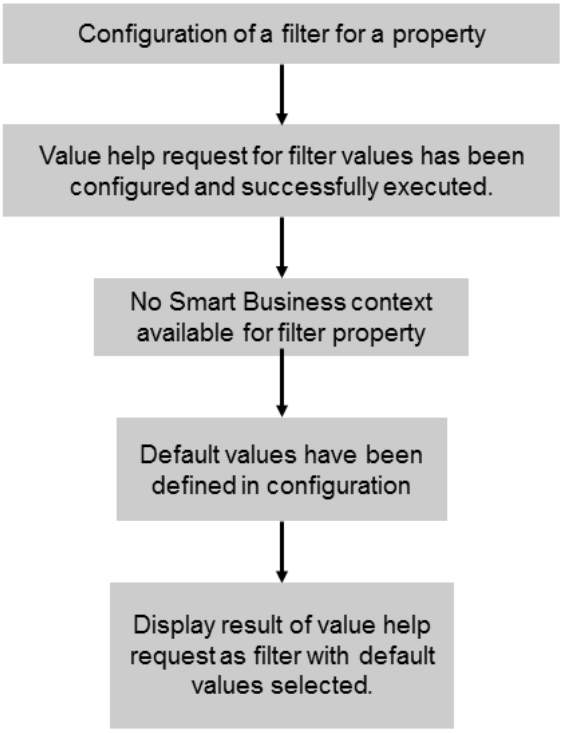
To configure the filter for this use case, you must enter basic data for the filter and information for the value help. No entries for the context resolution are required.

You can configure whether multiple selections are possible in the value help or not. If multiple selections are possible you can specify one or several values that are used as default values or you can enter a function that determines the default values. If you select *Automatic Values*, all values are selected in the filter by default.

If only single selection is allowed, you can specify one value that is used as default value or you can enter a function that determines the default value. If you select *Automatic Values*, the first entry in the value help of the filter is automatically selected by default.

> **i** Note
>
> If you specify default values that do not exist in the value help, these values will occur in the filter in addition to the values provided by the value help request.

The option described above is depicted in the following figure:



Use Case 2: Filter Determined by Smart Business Context Only

For this use case, all of the following apply:

- You configure a filter for a property in your application.
- No value help request has been specified.
- The Smart Business KPI tile passes a filter for the same property in the context.

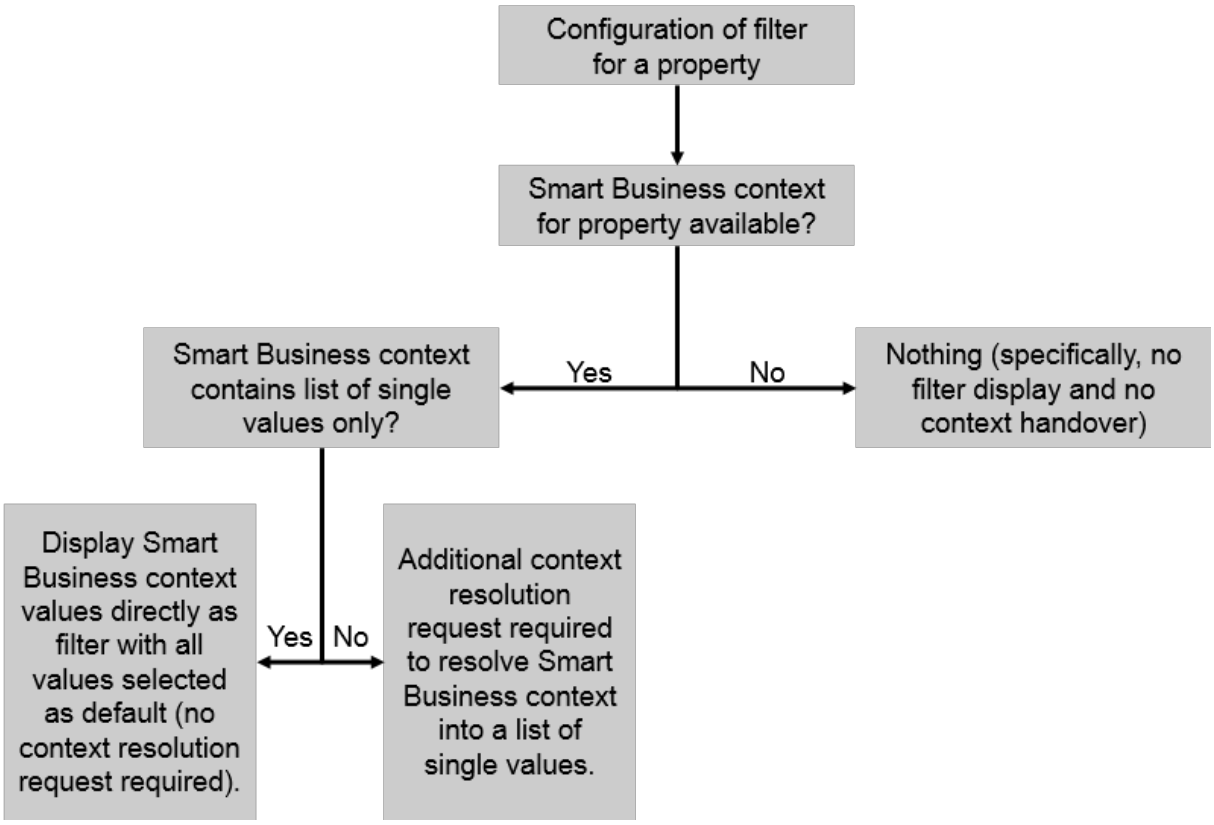
In this case, the filter is displayed in the application and its content is determined by the information passed in the Smart Business context. If the context contains a list of single values only, the value help lists all these values and by default, all values are selected. If the context contains other operators, an additional context resolution request is required. This request resolves the context into a list of single values. The value help then lists all these single values and by default, all values are selected.

To configure a filter for this use case, only the basic data for the filter is required. You don't have to enter information for the value help request.

To configure the context resolution request, enter the required information in the *Context Resolution* section.

> **i Note**
>
> If you did not specify a context resolution request, but a context resolution is necessary to be able to display the values in the value help, the filter for the corresponding property is not displayed on the UI. Instead, the context is applied to the analysis path in the background.

The options described above are depicted in the following figure:



Use Case 3: Filter with Default Values Determined by Smart Business Context

For this use case, all of the following apply:

- You configure a filter for a property in your application.
- A value help request determines the entries in the value help list.
- The Smart Business KPI tile passes a filter for the same property in the context.

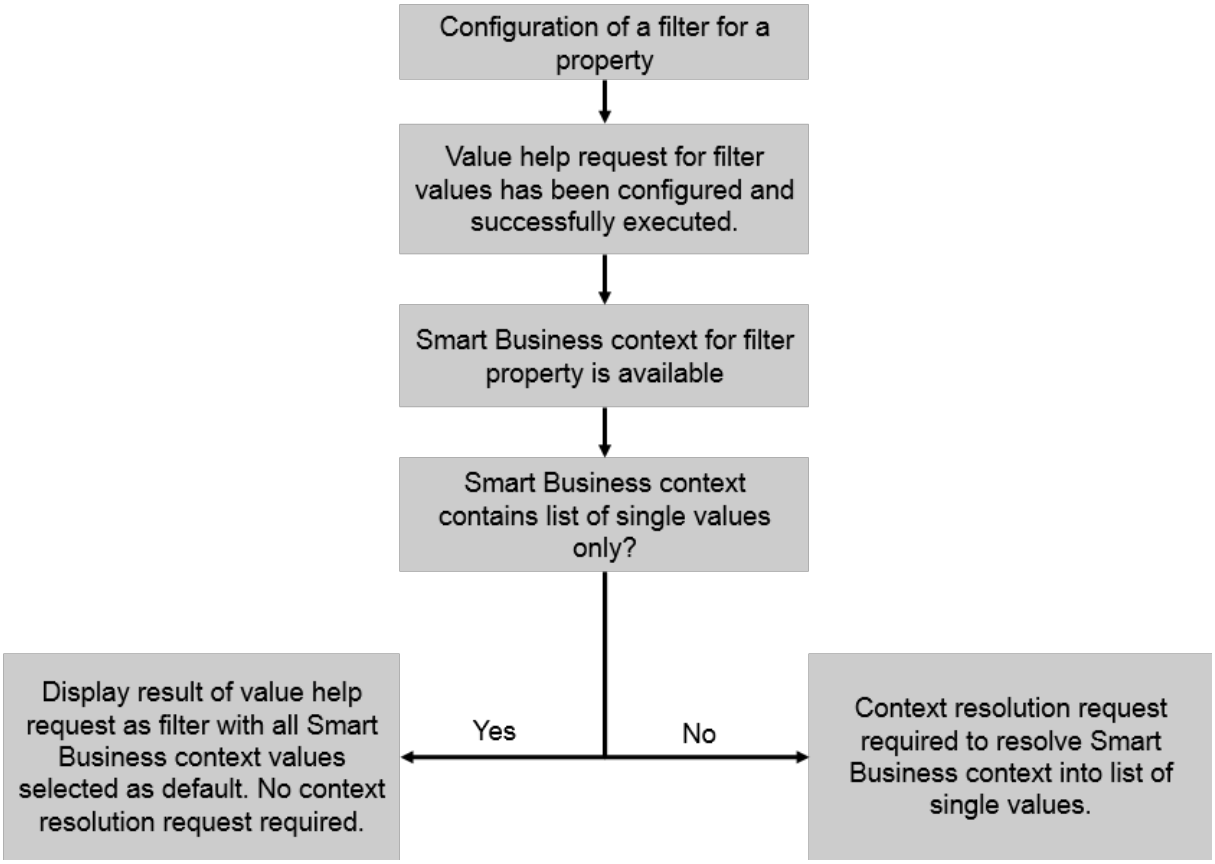
As a result, the value help for the filter lists all values retrieved by the value help request. By default, those values are preselected that were passed as filters with the Smart Business context.

As in use case 2, if the context contains operators other than just single values, an additional context resolution request is required. This request resolves the context into a list of single values. In the value help, these single values are then selected by default.

To configure a filter for this use case, you must enter basic data for the filter definition and information for the value help.

To configure the context resolution request, enter the required information in the *Context Resolution* section.

The option described above is depicted in the following figure:



Example

You configure a filter for the property `CompanyCode`. You define a value help request that retrieves all company codes that are available. These are listed in the value help for the filter.

The Smart Business context passes company codes 1000 and 2000 as filters. These company codes are then preselected in the filter. The user can, for example, add more company codes to the analysis as required.

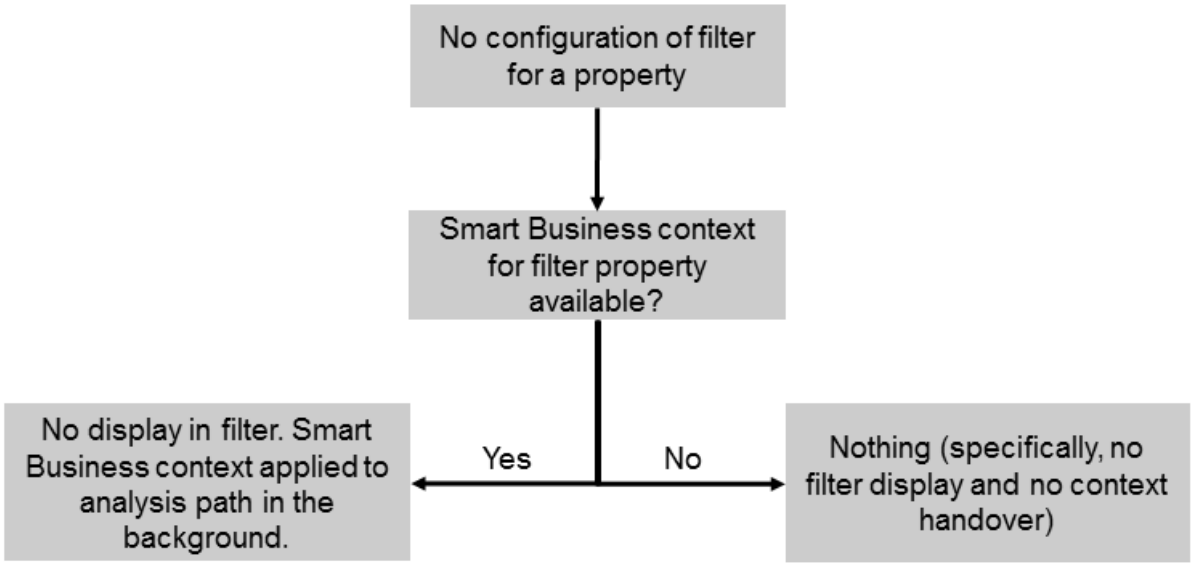
Use Case 4: No Filter

For this use case, the following applies:

- The Smart Business KPI tile passes a context for a property.
- It is not required to display the corresponding filter on the UI and it shall not be changed by the user. Therefore, you don't configure a filter for this property.

In this case, the context is applied to the analysis path in the background. This is useful, for example, for technical properties or for properties that the user shall not be able to change, such as, `SAPClient`.

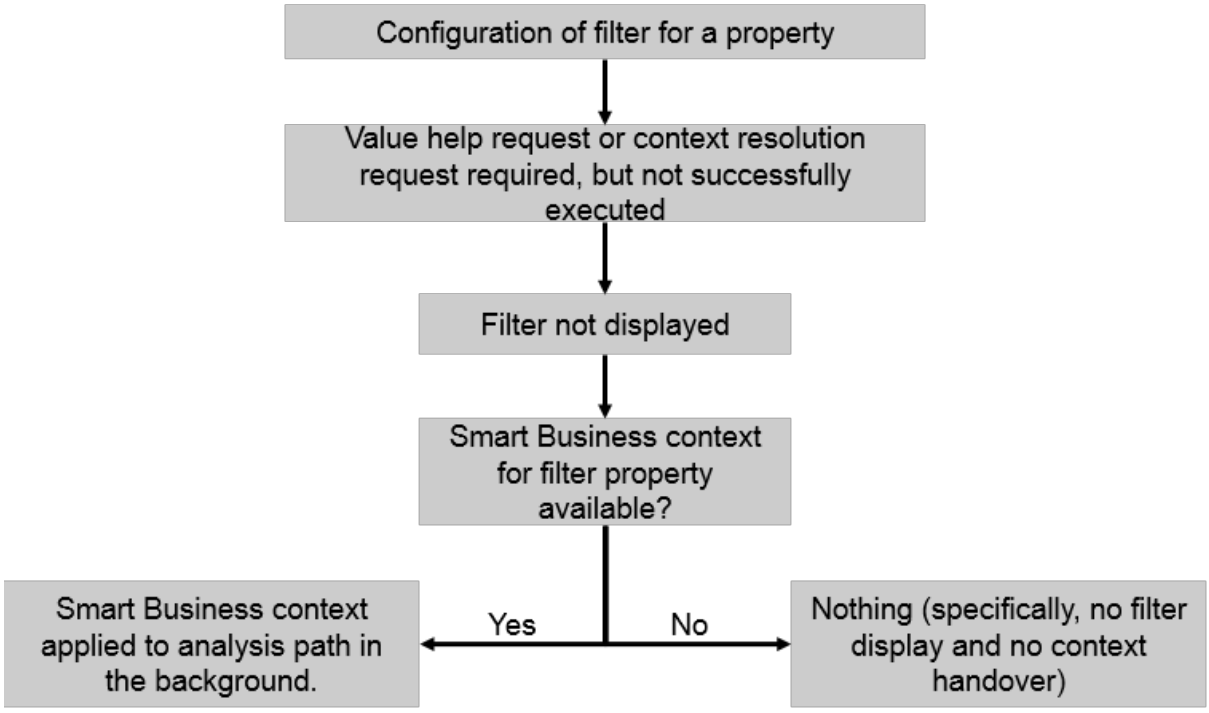
The option described above is depicted in the following figure:



Erroneous Filter Configuration

If a filter has been configured for a property, situations may occur where the filter cannot be displayed. This can be the case, for example, if a context resolution is required, but a context resolution request has not been configured or the request fails. As in use case 4, if a filter is passed with the Smart Business context, it is directly applied to the analysis path.

The option described above is depicted in the following figure:



Smart Filter Bar

The `SmartFilterBar` control uses the OData metadata of an entity set to create a filter bar. The metadata define, for example, the possible filter criteria by which you can filter, whether a field supports the type-ahead feature and whether it has a value help.

To configure a smart filter bar, enter the following:

Field	Explanation
Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
Entity Set	Entity set that corresponds to the data source, for example, the SAP HANA view.

> **!** Restriction
>
> If you use an entity set that has an associated entity set, you cannot use the properties from the associated entity set as filters in APF, because APF cannot process these properties. The properties from the associated entity set can be selected as filters in the filter bar settings and are shown in the smart filter bar, but they do not have any effect on the analysis path.

Runtime

At runtime, the smart filter bar is rendered based on the annotations of the specified service. Whether certain filters are already visible or not depends on the metadata. You can add more filters and select data as required. The settings you make are applied to the analysis path as soon as you do one of the following:

- Enter a value in one of the input fields
- Use the value help to select values and choose *OK*

The annotations also define whether the smart filter bar has mandatory fields. If there are mandatory fields, you can start your analysis only if they are filled. Otherwise the *Add Analysis Step* button is inactive.

When an external context is handed over to the APF-based app, for example, from a Smart Business KPI tile, all properties from the context that are part of the smart filter bar are preselected in the filter bar. All other properties are applied to the analysis path in the background, like hidden filters.

> **! Restriction**
>
> When you define conditions for filters, you cannot use the 'Exclude' option because APF cannot handle this.

As of SAPUI5 1.46, the property `useDateRangeType` is set to true for APF-based applications. This enables you to make dynamic time selections if a date field has the filter restriction interval. For example, you can select *Last X days* and define the number of days. When you save an analysis path with a dynamic time selection, the time selection is always updated in relation to the current date when you open the analysis path.

> **i Note**
>
> For analysis paths and variants that have been saved prior to upgrading to SAPUI5 1.46, date intervals are lost and must be set again.

Variant Management

When you have adjusted the smart filter bar, for example, you have added further filters or selected data, you can save these settings as a variant. Saving a variant is independent from saving a path. Therefore, if you use a certain variant for an analysis path, change the settings in the filter and then save the path, the path is saved including all filters, but the variant is not overwritten.

Related Information

API Reference: `sap.ui.comp.smartfilterbar.SmartFilterBar`

Sample: `sap.ui.comp.smartfilterbar.SmartFilterBar`

API Reference: `sap.ui.comp.variants.VariantManagement`

Creating Navigation Targets

Users can use the *Open In...* button in the footer of an APF-based runtime application to navigate to other applications. The context of the current analysis path, including start filters and other filters, such as selections made in the charts up to the active analysis step, is handed over to the other application.

When the user navigates back to the APF-based application, the analysis path is reloaded in exactly the same status as it was when navigating to the other application; that is, all analysis steps and all selections made are retained.

The following video shows an introduction to navigation targets and how to configure them:

The following types of navigation targets exist:

- Navigation targets that are relevant for all analysis steps
  Choose this type if the navigation target is always available no matter which analysis steps are added to the analysis path.
- Step-specific navigation targets
  Choose this type if the navigation target is relevant only in the context of a specific dimension or selection. For example, navigating to a customer fact sheet is useful only when a customer has been selected. A step-specific navigation target is displayed in the *Open In...* dialog only if the step to which it is assigned is the active one.

Step-specific navigation targets can be assigned to multiple analysis steps. To assign a previously created navigation target to an analysis step, proceed in one of the following ways:

- Select the navigation target and add the analysis step in the *Step Assignments* field.
- Select the analysis step and add the navigation target in the *Navigation Target Assignment* section in the *Step-Specific* field.

You can define multiple navigation targets for each configuration. To create a navigation target, click *Add* > *New Navigation Target* and enter the following:

Basic Data

Field	Explanation
Semantic Object	Semantic object as defined in the corresponding target mapping configured in the Fiori launchpad designer. Semantic object as defined by the Fiori launchpad configuration. **i Note** Currently, you can only navigate to other APF applications. Therefore, the semantic object must be **`FioriApplication`**.
Action	As soon as you enter a semantic object, the value help for the action is filled with all actions that are available for this target mapping and that you are authorized for. The *Action* field is prefilled with the first action in this list. **i Note** Currently, you can only navigate to other APF applications. Therefore, the action must be **`executeAPFConfiguration`**.
Navigation Target Title	Taken over from the description in the target mapping.

Navigation Parameters

Field	Explanation
Use Dynamic Parameters	By default, this checkbox is not selected. In this case, the application context is handed over to the navigation target using an app state. Select this checkbox to expose all single value filters and parameters from the context as URL parameters as well. In this case, each property of the context that has a single value only is exposed as key-value pair in the URL. This includes global filters set in the filter bar, selections in analysis steps, and filters and parameters handed over to APF from another application. This is useful, for example, if the navigation target is unable to consume a context from the app state or if the intent of the navigation target has mandatory parameters that must be provided as URL parameters. **i Note** Dynamic parameters occur in the URL of the called application. Therefore, data protection and privacy aspects have to be considered.
Static Parameter/Value	At runtime, static parameters are added to the URL of the navigation target. You can, for example, add a specific APF configuration ID to the navigation target to execute the generic APF runtime.

Assignment to Steps

Field	Explanation
Assignment Type	Choose one of the following: • Assign to All Steps The navigation target is available in all analysis steps at runtime. • Assign to Specific Steps The navigation target is available only in the specified analysis steps.

Field	Explanation
Step Assignments	If you are creating a step-specific navigation target, assign one or more analysis steps to it. If you leave this field empty, you can still save the navigation target and assign it to steps at a later point in time.

Context Mapping

If a certain property is required for launching the navigation target, you can define a request for it. This ensures that the property is available as a context when navigating to this navigation target at runtime. The result of the context mapping request is a list of values for the specified property. If you have selected more than one property, the result is a list of value tuples. The results are added to the existing context that is handed over to the navigation target.

Field	Explanation
Service	Path to the OData service root. If you use ABAP CDS views or BW OData queries, select a service from the value help, which lists all services available on SAP Gateway. If you use calculation views, you must enter the service manually.
Entity Set	Entity set that corresponds to the data source, for example, the SAP HANA view.
Mapped Properties	Select one or more properties from the list of properties that are available for the selected entity set.

Runtime

At runtime, the navigation targets are displayed in the *Open In...* menu. Step-specific navigation targets appear at the top of the list. Navigation targets that are relevant for all analysis steps appear at the bottom of the list so that this part of the *Open In...* menu is stable for all steps.

Navigation targets are displayed in the same order as they appear in the tree structure. You can change the order of the navigation targets by moving them up or down the tree structure using the arrow icons.

The list of navigation targets in the *Open In...* menu can be different from user to user. Whether a navigation target is visible for a user depends on the following:

- The user is authorized for the target mapping of the navigation target. This is the case if the user has a role to which the catalog is assigned which contains the target mapping.
- The navigation target supports the form factor the user is currently using: desktop, tablet, or smartphone. This is also configured in the target mapping.

Related Information

Outbound Navigation and Inbound Navigation [page 2116]

Executing a Configuration

To get a preview of how your configuration looks at runtime, you can execute it using the *Execute* button on the footer of the APF Configuration Modeler. The configuration is launched in a separate browser window so that you can see the configuration and the corresponding runtime application side by side.

The configuration preview is generated using the generic APF runtime application. Since the generic APF runtime application does not support filters that use a function to determine the values, there is a restriction in previewing those filters: They are shown in the preview with all values selected by default, but it is not possible to select other values.

If a request in your configuration requires parameters or mandatory filters that will be handed over by another application in a productive system, for example, by a Smart Business KPI tile, the preview will not work properly. To simulate that the required parameters are handed over to the app, you can configure hidden filters and put them in front of all other filters. This ensures that the preview shows the correct data and you can test your application. The hidden filters do not show up on the runtime user interface. If a context value is handed over from outside at a later stage, this context value will take precedence over the default value in the configuration of the hidden filter, so you do not even have to remove the hidden filter.

Deleting Objects

To delete an application, select it in the *Application Overview* and click *Edit*. You can then delete it from the list. Note, however, that an application is always deleted along with all configurations assigned to it.

To delete an object of an application, select it in the tree structure and click *Delete*. Note the following:

- Deleting a configuration always includes all subordinate objects.
- Deleting a category also deletes those steps that are not assigned to any other category. Steps that are also assigned to other categories are not deleted.
- Deleting a step removes it from any category it is assigned to. To remove a step from a particular category without deleting it altogether, select the step and remove the category assignment from the *Category Title* field.

Text Pool Cleanup

All configurations that you create under one application node share a common text pool, that is, one set of texts for all configurations belonging to an application node. This is useful if you have configurations that share large parts of their texts, for example, because they are very similar as far as their content is concerned.

When you export a configuration, the corresponding text pool can be exported too, for example, for translation purposes. However, in certain cases, texts that are no longer used may exist in the text pool. For example, if you change a text, both the old text and the new one are retained in the text pool, even if you only correct a typo. When workin in the APF Configuration Modeler, you can only add texts to the text pool, but not change or remove them. Therefore, to keep your text pool as small as possible and save translation costs, we recommend performing a text pool cleanup whenever you want to export a text pool. Otherwise, obsolete texts may end up in the translation process.

On the *Application Overview* screen, choose the *Text Pool Cleanup* icon for the application for which you want to perform the text pool cleanup.

Related Information

Export [page 2092]
Translation [page 2093]

Import

To change the configuration of an APF-based application that was shipped by SAP, you must first import the corresponding JSON file as well as the text properties file into the APF Configuration Modeler. The options that are available for importing files depend on whether you use SAP Business Suite powered by SAP HANA, SAP S/4HANA, or SAP Cloud Platform.

Importing Files on an SAP Business Suite powered by SAP HANA Platform

The following video shows how to import and export configurations:

Open this video in a new window

The files to be imported can either result from a configuration export or you can copy them from a shipped BSP application.

> **i Note**
>
> Configuration files of APF-based applications that were not created using the APF Configuration Modeler cannot be imported. You can easily recognize these configuration files because they do not contain a configuration GUID.

To import the files, proceed as follows:

1. In the toolbar of the *Application Overview* screen, click *Import*.
2. Choose a configuration file for upload. The file must be in JSON format.

3. Choose a text properties file for upload. The file must be in .properties format. It is also possible to upload a text properties file only, for example, after translation or to switch the development language.
4. Click *Upload*.

During import, the content of the configuration file is written into a configuration table. The content of the text properties file is written into a text table.

If the application with the GUID specified in the JSON file does not yet exist in the APF Configuration Modeler, it is created and the description and semantic object are filled based on the information in the JSON file. Otherwise, the configuration of the JSON file is added to an existing application.

If a configuration with the same GUID already exists, you can decide whether you want to overwrite it or create a new one. You can now edit the configuration and save the changes.

Importing Files on an SAP S/4HANA Platform

The following video shows how to import files on an SAP S/4HANA platform:

If you use SAP S/4HANA, you have the following options:

- *Import Delivered Content*
- *Import Files*

You can use *Import Files* to import files that result from a configuration export. The procedure is the same as described above for SAP Business Suite powered by SAP HANA.

If you want to import content that was delivered by SAP, you don't have to copy the files from a BSP application. You can import them from the layered repository of SAPUI5 flexibility using the *Import Delivered Content* function.

The configuration file of a shipped application and the text properties file in the development language reside in the `VENDOR` layer of the layered repository. When you import the content, it is copied to the `CUSTOMER` layer. Changes you make to the configuration or the texts are also written into the `CUSTOMER` layer, so the original files in the `VENDOR` layer remain unchanged.

However, when a new version of a configuration is delivered with a support package, this update overwrites the version in the `VENDOR` layer, so in this case the version in the `CUSTOMER` layer remains unchanged. To see the updated version in the APF Configuration Modeler, you must first import the configuration again from the `VENDOR` layer into the `CUSTOMER` layer using the *Import Delivered Content* function.

> ⚠ Caution
>
> When you have imported and edited a configuration and then later on import an updated version from the `VENDOR` layer, this new import overwrites the changes you have already made in the `CUSTOMER` layer. To prevent this, you can copy the configuration before editing it and then work on the copy.

At runtime, the system first checks whether there is a configuration file and a text properties file in the `CUSTOMER` layer. Only if there are no files in the `CUSTOMER` layer, is the application run from the files in the `VENDOR` layer.

Related Information

Enhancing an APF-Based Application [page 2050]

Export

The following video shows how to import and export configurations:

Open this video in a new window

You can export the table entries for configurations and text pools. An export results in the following files:

- By exporting a configuration, you create a configuration file in JSON format. This format is required, for example, to be able to transport a configuration between different systems.
- By exporting a text pool, you create a text file in .properties format. This format is necessary to be able to translate the texts. Exporting is also useful to create a backup copy of your configurations and texts.

> **i Note**
>
> In SAP S/4HANA Cloud, you must use the apps *Export Software Collection* and *Import Collection* to transport a configuration between different systems.

To use the export function, proceed as follows:

1. Navigate into an application.
2. Select a configuration.
3. Click *Export* in the footer toolbar.
4. Download the configuration file.
5. Download the text properties file.

Related Information

Translation [page 2093]
Transporting Configurations in SAP S/4HANA Cloud [page 2092]

Transporting Configurations in SAP S/4HANA Cloud

In SAP S/4HANA Cloud, you must use the apps *Export Software Collection* and *Import Collection* to transport configurations between systems. Whenever you create, change, or delete a configuration, a notification is created in the system. You can then use these apps to transport the files.

> **i Note**
>
> If you just want to create a backup copy of your files, you can use the export function of the APF Configuration Modeler and don't need the apps *Export Software Collection* and *Import Collection*.

To find the configuration you want to export in the *Import Collection* app, you can search, for example, for "Analysis Path Framework", the configuration name, or the configuration GUID.

For more information, see the documentation for your Cloud version at https://help.sap.com/viewer/p/SAP_S4HANA_CLOUD under *Product Assistance* > `<language>` > *Generic Information* > *General Functions for the Key User* > *Extensibility* >, chapters *Export Software Collection* and *Import Collection*.

Related Information

Export [page 2092]

Translation

Texts in the development language are stored in a text table. If you need more languages than just the development language, you must export the text pool from the table into a text properties file. Based on this file, translation can take place, resulting in one text properties file for each language.

> **i Note**
>
> We recommend performing a text pool cleanup before the texts are exported for translation. Otherwise, obsolete texts may end up in the translation process.

Texts must be translated directly in the text properties file, that is, you must create one file for each language and maintain the translated texts manually. The following examples illustrate the naming convention for the files:

- Original: `<texts>`.properties
- German: `<texts>`_de.properties
- English: `<texts>`_en.properties

New texts are always added at the end of the file so that they can easily be identified after a new export.

Translated texts are always read from the text properties files and usually not reimported into the APF Configuration Modeler unless you want to change the development language.

Related Information

Export [page 2092]
Text Pool Cleanup [page 2089]

Launching APF-Based Applications

You can launch an APF-based application in the following ways:

- From a Smart Business KPI tile
  You can configure a Smart Business KPI tile so that an APF-based application is used for drill-down.
- From a Fiori app launcher tile
  You can use a Fiori app launcher tile to launch your APF-based application. However, this is not supported if you use the generic APF runtime application.

Configuring the SAP Smart Business KPI Tile

The following tasks are performed using the Smart Business modeler apps.

Prerequisites:

- You have used the SAP Smart Business modeler apps to create the following:
  - A KPI
  - An evaluation
  - A KPI tile
- You have deployed a shipped APF-based application or created your own APF configuration that you want to launch using an SAP Smart Business KPI tile.

You can now configure the KPI tile using the SAP Smart Business modeler apps. The data you must enter depends on, among other things, whether you use shipped content or content you created yourself.

You can create a KPI tile for the following use cases:

Use Case	Description
Shipped APF-based application – unchanged	A shipped Business Server Pages (BSP) application that you use as is without making any changes.
Shipped APF-based application – enhanced	A shipped BSP application that you have enhanced, that is, you have imported the APF configuration and changed it using the APF Configuration Modeler.
Generic runtime application	The generic APF runtime application is used to execute an APF configuration that you created using the APF Configuration Modeler.
New application	An application you have created using the APF Configuration Modeler without using the generic runtime application. This requires defining your own BSP application and target mapping.

The data you must enter also depends on the SAP Smart Business KPI Modeler version you are using. There are two versions of the KPI Modeler available:

- A KPI Modeler for apps that use calculation views, which is used for SAP Business Suite powered by SAP HANA.
- A KPI Modeler for apps that use ABAP CDS views, which is available both for SAP Business Suite powered by SAP HANA and for SAP S/4HANA.

The following sections differentiate the data you must enter depending on the use cases and the KPI Modeler version mentioned above:

Using the SAP Smart Business Modeler Apps for SAP Business Suite powered by SAP HANA

To configure a KPI tile for an APF-based application, open the *Configure KPI Tiles* app, go to the *Navigation* area and select the *Other Drill-Down* radio button. Select **`Analysis Path Framework`** as drill-down.

Depending on your use case, enter the following data:

	Shipped BSP Application - Unchanged	Shipped BSP Application – Enhanced	Generic Runtime Application	New Application
Semantic Object	As defined in target mapping of the application	As defined in target mapping of the application	Prefilled: FioriApplication	As defined in your own target mapping
Action	As defined in target mapping of the application	As defined in target mapping of the application	Prefilled: executeAPFConfiguration	As defined in your own target mapping
Configuration	Not applicable	Select the configuration you want to launch	Select the configuration you want to launch	Select the configuration you want to launch

If you use a shipped BSP application, a KPI tile may have been shipped along with it. In this case, semantic object and action are already filled.

> **i** Note
>
> We recommend copying the shipped KPI, evaluation, and tile. You can then adapt them to your needs.

Shipped BSP-Application - Unchanged

If you want to launch a shipped APF-based application without making any changes to it, entering a configuration title is not required. The configuration can be read directly from the analytical configuration file of the BSP application. The location of this file is specified in the `manifest.json` file, which is part of the BSP application.

Shipped BSP-Application - Enhanced

If you have imported the JSON file of a shipped application into the APF Configuration Modeler, ensure that the semantic object entered in the APF Configuration Modeler app is the same as the one you enter here. The semantic object and the action are used to determine the KPIs navigation target.

You must select a configuration to be able to launch the application. The corresponding configuration ID takes precedence over the location of the analytical configuration file of the BSP application. To select the configuration title, use the value help. The semantic object filters the list of configurations so that only those are listed that have been created for the specified semantic object. A configuration title has a corresponding configuration ID, which is unique.

Generic Runtime Application

Selecting a configuration is also mandatory if you use the generic runtime application. The generic runtime application does not contain a reference to the location of the analytical configuration file and therefore must be parameterized using a configuration ID. To select the configuration title, use the value help. The semantic object filters the list of configurations so that only those are listed that have been created for the specified semantic object. A configuration title has a corresponding configuration ID, which is unique.

New Application

If you have used the APF Configuration Modeler to create a new application, ensure that the semantic object entered in the APF Configuration Modeler app is the same as the one you enter here. The semantic object and the action are used to determine the KPIs navigation target.

To select the configuration title, use the value help. The semantic object filters the list of configurations so that only those are listed that have been created for the specified semantic object. A configuration title has a corresponding configuration ID, which is unique.

Navigating to the APF Configuration Modeler

By clicking *Save and Configure Drill-Down*, you can navigate to the APF Configuration Modeler to view or edit the configuration. This is useful because from here you can navigate directly to the relevant configuration. You cannot search for a configuration in the APF Configuration Modeler itself.

> **i** Note
>
> When you click *Save and Configure Drill-Down*, your changes are saved as draft, but not yet activated.

Using the SAP Smart Business Modeler Apps for SAP S/4HANA

> **i** Note
>
> The KPI Modeler for SAP S/4HANA may also be used on SAP Business Suite powered by SAP HANA.

To configure a KPI tile for an APF-based application in the SAP Smart Business modeler apps for SAP S/4HANA, open the *Create Tile* app and select your tile or create a new one. In the *Navigation* area, go to the *Select Drill-Down* field and select `Analysis Path Framework`.

Depending on your use case, enter the following data:

	Shipped BSP Application – Enhanced	Generic Runtime Application	New Application
Semantic Object	As defined in target mapping of the application	`FioriApplication` (automatically filled based on selected configuration)	As defined in your own target mapping
Action	As defined in target mapping of the application	`executeAPFConfigurationS4HANA` or `executeAPFConfiguration`(automatically filled based on selected configuration.)	As defined in your own target mapping
Configuration	Select the configuration you want to launch.	Select the configuration you want to launch.	Select the configuration you want to launch.

If you use a shipped BSP application, a KPI tile may have been shipped along with it. In this case, semantic object and action are already filled.

> **i Note**
>
> We recommend copying the shipped KPI, evaluation, and tile. You can then adapt them to your needs.

By clicking *Save and Configure Drill-Down*, you can navigate to the APF Configuration Modeler to view or edit the configuration. This is useful because from here you can navigate directly to the relevant configuration. You cannot search for a configuration in the APF Configuration Modeler itself.

Launching an APF-Based Application from an SAP Smart Business Generic Drill-Down

Instead of launching an APF-based app from an SAP Smart Business KPI tile, you can first launch an SAP Smart Business generic drill-down app and then navigate to an APF-based app from there using the *Open In...* menu. In this case, filters set in the generic drill-down app are also handed over as context to the APF-based app.

To configure the navigation from a generic drill-down app to the APF-based app, proceed as follows:

1. Open the *KPI Workspace* app.
2. Choose your evaluation and switch to edit mode.
3. Enter the semantic object and, optionally, an action.

> **i Note**
>
> If you don't have your own BSP application and target mapping, but use the APF generic runtime, you must create a target mapping that launches the generic runtime and specifies the configuration as follows:
>
> - Name: `sap-apfconfigurationid`
> - Value: configuration ID in format `<application ID>.<configuration ID>`

You can find the values for the configuration ID in the following places:

- application ID: in the app URL behind the parameter "app"
- configuration ID: in the app URL behind the parameter “config” or in the configuration details in the APF Configuration Modeler app

Application Parameters

You can configure your KPI tile in a way that it launches an APF-based application and immediately opens a specific analysis step and representation. You can do so by entering the following application parameters:

- Analysis step:
  - Name: `sap-apf-step-id`
  - Value: step ID, for example, “Step-23”
- Representation:
  - Name: `sap-apf-representation-id`
  - Value: representation ID, for example, “ Step-23-Representation-1”

You can look up the step ID and representation ID in the JSON file of the configuration. You may have to export your configuration first to get an up-to-date JSON file. You can also find the IDs in the URL of the APF Configuration Modeler while you are editing a step or representation. You can find the step ID behind the parameter "step" and the representation ID behind the parameter "repn".

Entering a step ID is sufficient; you don’t have to enter a representation ID. If you only enter a step ID, the default representation is used.

Configuring the Fiori App Launcher Tile

You can use a Fiori app launcher tile to launch your APF-based application.

Prerequisites

Mandatory HANA view parameters must be filled to be able to launch the application. This can be implemented in the BSP application by extracting the parameter values from the URL created by the Fiori app launcher tile. Another option is to configure a filter for a parameter. In this case, the user can set the values at runtime.

Procedure

Configure the Fiori app launcher tile using the Fiori launchpad designer. For more information, search for "Setting Up Content With the Launchpad Designer" and "About Navigation" in the documentation for your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

If required, enter fixed values for the HANA view parameters in the Fiori tile. This is relevant for parameters that shall not be displayed in the application and for which you therefore do not configure filters.

If you want to specify a configuration ID, you can also enter it as a parameter. The parameter name is sap-apf-configuration-id.

For SAP S/4HANA, you must maintain the application ID and the configuration ID as value for parameter `sap-apf-configuration-id` in the format `<application ID>.<configuration ID>`.

If you have imported the JSON file of a shipped application into the APF Configuration Modeler, the configuration ID takes precedence over the location of the analytical configuration specified in the `manifest.json` file.

You can also use parameters to define that a specific analysis step is immediately opened when launching the APF-based application. To do so, enter the following parameter name and value pairs:

- sap-apf-step-id=`<step ID>`
- sap-apf-representation-ID=`<representation ID>`

Example:

- sap-apf-step-id=Step-23
- sap-apf-representation-ID=Step-23-Representation-1

You can look up the step ID and representation ID in the JSON file of the configuration. You may have to export your configuration first to get an up-to-date JSON file.

Entering a step ID is sufficient; you don't have to enter a representation ID. If you only enter a step ID, the default representation is used, that is, the representation that comes first in the tree structure of the APF Configuration Modeler.

> **i Note**
>
> You can define the parameters either in the tile configuration or in the corresponding target mapping.

Data Protection and Privacy

To ensure data protection and privacy, it is required to be able to delete personal data from the system.

The following sections give information about how to delete personal data depending on the platform you use.

Deletion of Personal Data in SAP Business Suite powered by SAP HANA

This section explains how an administrator or a user can delete personal data in SAP Business Suite powered by SAP HANA.

Deleting Analysis Paths

When you create an analysis path using an APF-based app, the path as well as the user who created the path are stored in a SAP HANA database table.

Deletion by an Administrator

When a user does not have access to the system any longer, for example, because he or she left the company, an administrator can delete the user's analysis paths. The analysis paths are stored in the following HANA database table:

Table name: `"sap.hba.r.apf.core.db::ANALYSIS_PATH"`

Schema name: `"SAP_HBA". The user is stored in the table field "CREATED_BY_USER"`

The administrator can delete the analysis paths with an SQL statement such as the following:

```
delete from "SAP_HBA"."sap.hba.r.apf.core.db::ANALYSIS_PATH" where
"CREATED_BY_USER" = '<user name>'
```

Deletion by a User

Users who still have system access can delete their own saved analysis paths directly in the analysis path gallery of the runtime application. To do so, open the analysis path menu and choose *Delete*. This opens a list of all saved analysis paths, which can be deleted individually. Repeat this for each APF-based application since you can see only those analysis paths that were created in the application you're currently using.

Personal Data in Analysis Paths

In an APF-based application, personal data can also appear in an analysis path, for example, as a measure in an analysis step such as ‘revenue by consultant’. APF does not persist these measures, but only filter criteria that are relevant for selections made in a chart. When a user is deleted from the system, the personal data of this user is not displayed any longer in an analysis path. The filter criteria is also removed from the analysis path so that no personal data of the deleted user shows up.

Deletion of Personal Data in SAP S/4HANA

This section explains how an administrator or a user can delete personal data in SAP S/4HANA.

Deleting Analysis Paths

When you create an analysis path using an APF-based app, the path as well as the user who created the path are stored in a database table.

Deletion by an Administrator

When a user left the company, the administrator deletes the user master record from the system. The user master record includes the personalization object `BSANLY_APF` (Analysis Path Framework personalization object). This ensures that deleting the user master record automatically also deletes analysis paths created by the user.

Deletion by a User

Users who still have system access can delete their own saved analysis paths directly in the analysis path gallery of the runtime application. To do so, open the analysis path menu and choose *Delete*. This opens a list of all saved analysis paths, which can be deleted individually. Repeat this for each APF-based application since you can see only those analysis paths that were created in the application you're currently using.

Personal Data in Analysis Paths

In an APF-based application, personal data can also appear in an analysis path, for example, as a measure in an analysis step such as 'revenue by consultant'. APF does not persist these measures, but only filter criteria that are relevant for selections made in a chart. When a user is deleted from the system, the personal data of this user is not displayed any longer in an analysis path. The filter criteria is also removed from the analysis path so that no personal data of the deleted user shows up.

APF Modules

Analysis Path Framework (APF) is an SAPUI5 component that can be used as a foundation for creating analytical Web applications. APF has a public API that supports the following:

- Building a Web application by using predefined UI elements
- Configuring the Web application
- Creating and processing analysis paths
- Displaying data, for example, in interactive charts
- Interaction between UI and path processing
- Error and message handling
- Interaction with the server using OData services
- Saving analysis paths on the server

Internally, APF consists of the following submodules:

- `sap.apf.core`
- `sap.apf.ui`

The `sap.apf.core` module defines the foundation for analysis path processing. The `sap.apf.ui` module defines the UI and the rendering of the analysis path on the UI. It depends on the `sap.apf.core` module.

The Core Module (sap.apf.core)

The core module (`sap.apf.core`) has a public API that supports the following:

- Loading an analytical configuration and other static resources
- Creating an analysis step from the given configuration
- Creating OData requests
- Adding a step to an analysis path
- Manipulating, processing, and updating the analysis path and its steps
- Creating filters and setting a context
- Interaction between APF and an external Web application (UI), for example, by passing callback functions
- Error handling and sending error messages to a registered handler
- Saving and retrieving a named analysis path on the server
- Extending possible configurations by new representation entities

The core component contains the entire logic for path processing, but no UI logic. For more information, see Analysis Path Processing [page 2108].

The UI Module

The UI module provides reusable UI components to support the interaction paradigm of an APF-based Web application.

The Analysis Step Container

The analysis step container is the UI area where the visualization of a step result, additional information, and actions related to the analysis step are displayed. It defines the rendering and styling of an analysis step. Analysis steps are configured in the `step` object of the analytical configuration file.

The analysis step container contains the following entities:

- The title of the analysis step as defined in the `manifest.json` file.
- The Step Toolbar [page 2102]

On the UI, the contents of the analysis step container are rendered in the analysis step display, where the visualization of the step result, additional information and possible actions related to the step are displayed.

The Step Toolbar

The step toolbar is part of the analysis step container and provides the following actions that the user can perform for the active analysis step:

- Selection Count

Displays the number of selected items. The selection count is displayed only when a selection is made in the active representation.

- Filter Information
  Displays a list of the filters that were set in previous analysis steps and that are also applied to the active analysis step. If filter mapping has been applied, you are informed about the mapped filter property. In addition, if a previous step has filters that do not affect the current analysis step, you are also informed about this.
- Toggle Legend
  Shows or hides the chart legend. This option is displayed only when the user has chosen a chart representation.
- Zoom in and zoom out (if the chart type supports zooming)
- Toggle Fullscreen
  Displays the chart in full screen mode or returns to the original size.
- Representation Type
  Depicts the current representation type. If the analysis step has multiple representations, clicking the icon opens a dialog with the available representation types for this step.
- Table Representation
  Allows the user to display the data in a table. The related icon is displayed only if an alternative table representation has been configured for the active chart representation.
- View Settings
  Allows the user to sort the table. This option is displayed only when the user has chosen the table representation.
- Export to Excel
  You can download the data that is available on the front end to a Microsoft Excel file. This feature is available in the following cases:
  - When the active analysis step is a table representation
  - When you have switched to the alternative list view of any chart

  > **i** Note
  >
  > If you use this feature on an iPad, you must first add the file extension `.xlsx` to the download file before you can open it in Microsoft Excel.
- Load All
  Loads all data records to the front end so that you don't have to page down to the end of the table to ensure that all data records are loaded. This is useful, for example, when you want to print or export the entire table. A counter shows the number of data records that are already on the front end and the total number of data records.

The Analysis Path Display

The analysis path display is the UI area where thumbnails of all analysis steps of the current analysis path are displayed. In addition, actions such as editing analysis paths or saving them are performed in the analysis path display. It consists of the following subcomponents:

- Analysis path title
  Contains the name of the analysis path and the icon to open the actions menu.
  Once an analysis path has been saved, the name of the saved path is displayed. Unsaved changes are indicated by an asterisk in front of the path name.

The actions menu offers the following options:
    - *New*
    - *Open*
    - *Save*
    - *Save As*
    - *Delete*
    - *Print*
- Analysis path carousel
  Contains the thumbnails and titles of the analysis steps. More precisely, the carousel renders previews of the analysis step instances that are displayed in the representation container. It also provides the capabilities for adding, rearranging, and deleting analysis steps.

The analysis path controller manages the analysis path with its analysis steps. It provides the APIs to add, delete, and rearrange analysis steps, propagate filters, invalidate path, and recalculation of filters.

The Analysis Step Gallery

When the user clicks *Add Analysis Step*, the analysis step gallery is instantiated. The analysis step gallery displays all available analysis step templates in a hierarchical select dialog. This dialog first lists all available categories. The titles of the categories as well as the order in which they are displayed are defined in the APF Configuration Modeler app.

When the user selects a category, all analysis steps available for this category are listed on the next level of the select dialog. The titles of the analysis steps are defined in the step configuration of the APF Configuration Modeler app. The step configuration also contains one or more category assignments to define the categories in which a particular step is displayed in the analysis step gallery.

On the final level, the representation types for the selected step are displayed.

When the user selects an analysis step to add it to the analysis path, the analysis step gallery calls the `createStepAndSetActive()` API and adds the step to the analysis path as the active analysis step. The analysis step gallery is then closed and the `getSteps` API is called to get all analysis steps in the correct sequence.

The Analysis Path Gallery

The analysis path gallery displays the list of saved analysis paths in a hierarchical select dialog. You can select a saved path to display the sequence of steps contained in the path.

You can click one of the analysis steps to load the analysis path. The step you clicked is displayed as the active analysis step in the analysis step container.

The saved analysis paths are displayed in the order of the most recent modification date.

Predefined Representation Types

Use

The UI component of APF provides predefined representation types that can be used to display data.

These representation types are predefined in the file `sap/apf/core/representationTypes.js`. They can be referenced in the analytical configuration using their IDs.

Each representation type has a constructor assigned. The constructor is used as a parameter in the representation type object of the configuration file and points to the implementation of the representation type. The charts shipped with APF are implemented using the VizFrame charting library (`sap.viz.ui5.controls.VizFrame`) that is available with SAP UI5.

The following table lists the order of available representation types:

Chart Name	ID	Constructor	Chart Type in VizFrame Charting Library
Column chart	ColumnChart	`sap.apf.ui.representations.columnChart`	`sap.viz.ui5.controls.VizFrame({vizType : column});`
Bar chart	BarChart	`sap.apf.ui.representations.barChart`	`sap.viz.ui5.controls.VizFrame({vizType : bar});`
Line chart	LineChart	`sap.apf.ui.representations.lineChart`	`sap.viz.ui5.controls.VizFrame({vizType : line});`
Line chart with two vertical axes	LineChartWithTwoVerticalAxes	`sap.apf.ui.representations.lineChartWithTwoVerticalAxes`	`sap.viz.ui5.controls.VizFrame({vizType : dual_line});`
Line chart with time axis	LineChartWithTimeAxis	`sap.apf.ui.representations.lineChartWithTimeAxis`	`sap.viz.ui5.controls.VizFrame({vizType : timeseries_line});`
Pie chart	PieChart	`sap.apf.ui.representations.pieChart`	`sap.viz.ui5.controls.VizFrame({vizType : pie});`
Donut chart	DonutChart	`sap.apf.ui.representations.donutChart`	`sap.viz.ui5.controls.VizFrame({vizType : donut});`

Chart Name	ID	Constructor	Chart Type in VizFrame Charting Library
Scatter plot chart	ScatterPlotChart	`sap.apf.ui.representations.scatterPlotChart`	`sap.viz.ui5.controls.VizFrame({vizType : scatter});`
Bubble chart	BubbleChart	`sap.apf.ui.representations.bubbleChart`	`sap.viz.ui5.controls.VizFrame({vizType : bubble});`
Stacked column chart	StackedColumnChart	`sap.apf.ui.representations.stackedColumnChart`	`sap.viz.ui5.controls.VizFrame({vizType : 100_stacked_column});`
Stacked bar chart	StackedBarChart	`sap.apf.ui.representations.stackedBarChart`	`sap.viz.ui5.controls.VizFrame({vizType : 100_stacked_bar});`
100% stacked column chart	PercentageStackedColumn-Chart	`sap.apf.ui.representations.percentageStackedColumnChart`	`sap.viz.ui5.controls.VizFrame({vizType : 100_stacked_column});)`
100% stacked bar chart	PercentageStackedBarChart	`sap.apf.ui.representations.percentageStackedBarChart`	`sap.viz.ui5.controls.VizFrame({vizType : 100_stacked_bar});)`
Combined column line chart	CombinationChart	`sap.apf.ui.representations.combinationChart`	`sap.viz.ui5.controls .VizFrame({vizType : combination});)`
Combined stacked column line chart	StackedCombinationChart	`sap.apf.ui.representations.stackedCombinationChart`	`sap.viz.ui5.controls .VizFrame({vizType : stacked combination});)`

Chart Name	ID	Constructor	Chart Type in VizFrame Charting Library
Combined column line chart with two vertical axes	DualCombinationChart	`sap.apf.ui.represent ations.dualCombinationChart`	`sap.viz.ui5.controls .VizFrame({vizType : dual combination});)`
Combined stacked column line chart with two vertical axes	DualStackedCombination-Chart	`sap.apf.ui.represent ations.dualStackedCombinationChart`	`sap.viz.ui5.controls .VizFrame({vizType : dual stacked combination});)`
Heatmap chart	HeatmapChart	`Sap.apf.ui.representations.heatmapChart`	`sap.viz.ui5.controls.VizFrame({vizType :heatmap});)`
Table	TableRepresentation	`sap.apf.ui.representations.tableRepresentation`	`SAPUI5 table (sap.ui.table.Table)`
Tree table	TreeTableRepresentation	`sap.apf.ui.representations.treeTable`	`SAPUI5 tree table (sap.ui.table.TreeTable)`

You can use the predefined representation types to define representations in your application. To do so, depending on the chart type you choose, you must define parameters such as the fields used for the horizontal axis and the vertical axis, or the field by which a chart is sorted.

Related Information

Analytical Configuration [page 2128]
The Representation Type Object [page 2138]

Rendering of Charts

The rendering of charts includes the following elements that are defined in the APF Configuration Modeler:

- Chart title
  The chart title is taken from the *Step Long Title* field of the step configuration. It is displayed above any representation of a particular analysis step. If no long title is defined, the step title is used instead.

- Titles of the horizontal axis and the vertical axis
  The general format of the title is <title (unit)>. It can be determined in one of the following ways:
  - The title is first read from the *Label* field of the representation configuration in the APF Configuration Modeler, if this property has been configured.
  - If the `fieldDesc` property has not been configured, but the field has a corresponding `sap:text` annotation in the OData metadata, the system looks up the field name specified in the annotation and retrieves the `sap:label` value of that field. This applies to those fields for which you want to display the name rather than an ID, for example, the customer ID.
  - If the *Label* field has not been configured and a `sap:text` annotation does not exist, the system looks up the `sap:label` annotation of the OData metadata, which retrieves the label specified for this field in the SAP HANA model.

  > **i Note**
  >
  > The unit is concatenated to the title only if the axis shows a measure that has a unit and if all data records have the same unit.
- Value formatting:
  All fields are formatted according to the supported OData 2.0 annotations. If no annotations exist, the value is displayed without formatting.
- Legend
  A legend is displayed for all charts that have multiple dimensions or measures. The same formatting applies as described above.
- Labels of the horizontal axis and the vertical axis
  The rendering of the labels of the axes is defined by the behavior of the core chart. The same formatting applies as described above.

Concepts

The following sections explain the basic concepts of Analysis Path Framework (APF).

Analysis Path Processing

The main purpose of the core module (`sap.apf.core`) is to handle and process analysis paths. A path and its steps are created, accessed, and processed using the APF API. During path processing, the following tasks are executed going through all steps in the order of their positions in the analysis path:

1. Execute the OData requests of the steps
2. Supply the representations of the steps with the response data
3. Notify the application UI using a callback function

In addition, the path logic accumulates filters that are derived from selections made in representations, and applies those filters in subsequent OData requests.

An analysis path contains an ordered sequence of one or more analysis steps.

A step is created using the APF API method `createStep(<id>,<callback>)`. It is inserted at the end of the path. The first method parameter is a unique identifier, which refers to the configuration entity that defines the

step. The second parameter supplies a callback function, which is called once after the step has been created and processed.

Path processing is triggered using the APF API method `updatePath(<callback>)`. The callback function is called each time after a step has been processed.

Runtime Objects

The following runtime objects exist:

- A `step` object consists of a `request` object and a `binding` object.
- A `request` object defines an OData server request. It creates and processes the OData request, processes the response, and sends the response data to the corresponding step.
- The `binding` object sends the response data to the selected `representation` object. A `binding` object associates a step with one or more `representation` objects and identifies and handles the `representation` object that is currently selected. The `binding` object also defines how the selected representation translates its selection into a filter object.
- A `representation` object wraps a chart, a table, or any other representation of data. When the user switches the representation, a different `representation` object is selected and supplied with data.
- A `filter` object represents the selections made on the charts.

The relation between the objects described above is depicted in the following figure:

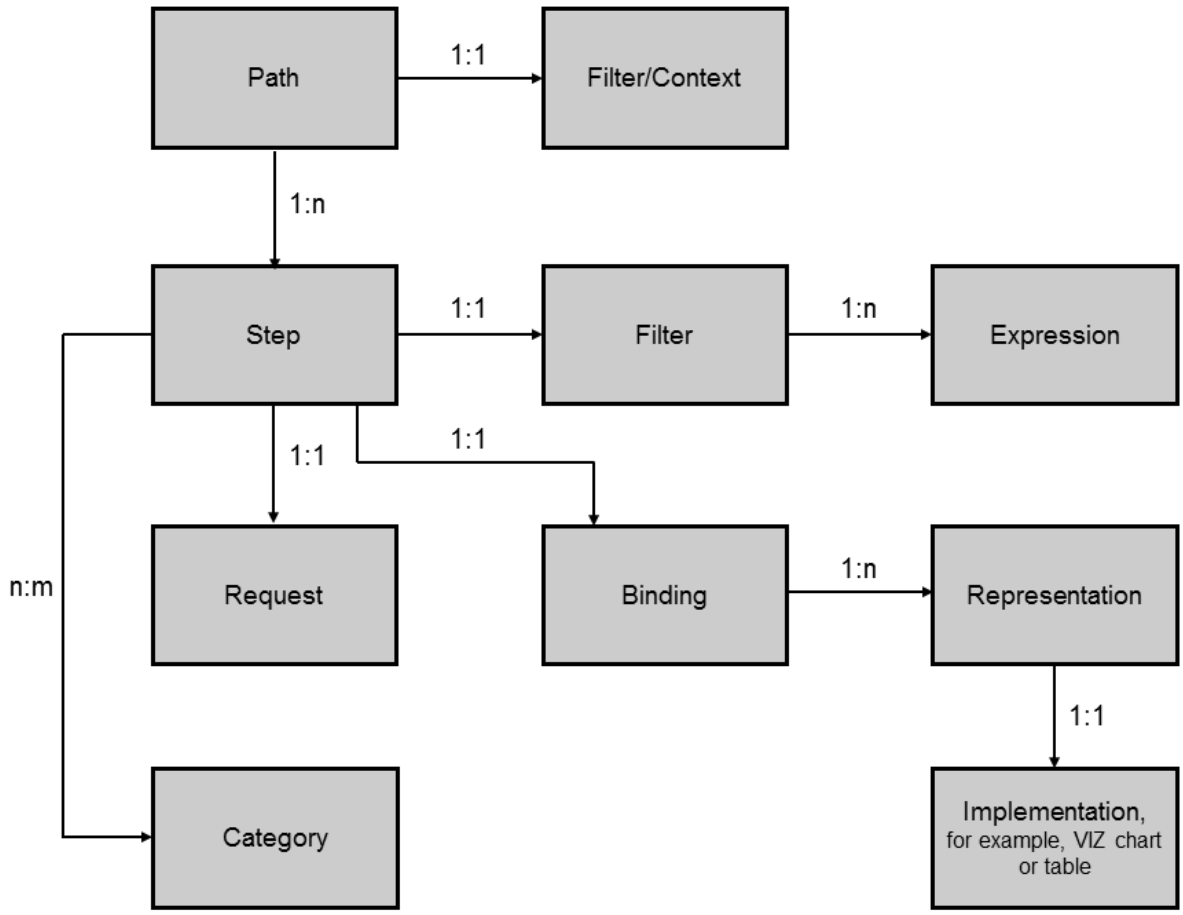


Figure 337: Runtime Objects of a Path

Processing of Runtime Objects

The steps are processed in the order of their positions in the path. The first step is processed first. The filter resulting from the first step is used to process the second step, and so on.

To describe the processing of an analysis path in more details, the following variables are relevant:

Variable	Description
`S_1,..,S_n`	Path of length n
`S_i`	Step
`R_i`	Request for step `S_i`
`B_i`	Binding for step `S_i`
`RO_i_s`	Selected representation object for step `S_i`
`F_i`	Filter object associated to step `S_i`

Variable	Description
`F_0`	Filter for the entire path. This filter is either empty or it is the filter set by the APF API method `addPathFilter()`. This filter defines the OData filter for request `R_1` of step `S_1`.

The processing of step `S_i` begins with requesting the filter `F_(i-1)` of the previous step. Note that for step `S_1`, this is filter `F_0`.

Step `S_i` is further processed by sending an OData request `R_i` to the server. The filter expression of request `R_i` is defined by the previous filter `F_(i-1)`.

When request `R_i` returns successfully, it sends its response data to the selected representation object `RO_i_s`. This representation object then creates its own filter based on this data and its own UI selections. This filter can be requested by the subsequent step.

The filter `F_i` is defined as a conjunctive accumulation of the previous filter `F_(i-1)` and the filter defined by the selected representation object `(RO_i_s)`. The filter `F_i` is used for processing step `S_(i+1)`.

Filter F_0 is defined by URL parameters passed to the application, such as SAPCLIENT, and by filters. In the figure above, F_0 is represented as the “Filter/Context” object directly associated to the path object.

Consuming APF

The following code snippet is an example of how to consume APF in an SAPUI5-based application:

```
sap.ui.define("myApp.Component", [
    "sap/apf/base/Component"
], function(ApfComponent){
    'use strict';
    return ApfComponent.extend("myApp.Component",
    {
        metadata : {
            "name" : <name>,
            "manifest" : "json",
            "version" : <version>
        },
        /**
        * Initialize the application
        *
        * @returns
        */
        init : function() {
            // Initialize application here. No APF specific operation done here,
since APF API is not yet available.
            // Call APF Component init
            ApfComponent.prototype.init.apply(this, arguments);
        },
        /**
        * Creates the application layout and returns the outer layout of APF
        *
        * @returns {sap.ui.core.Control} the content
        */
        createContent : function() {
            // Attach APF start-up callbacks
            this.getApi().setCallbackBeforeApfStartup(this.onBeforeApfStartup);
```

```
            this.getApi().setCallbackAfterApfStartup(this.onAfterApfStartup);

            // Return whatever is returned by parent (APF Component)
createContent method
            return ApfComponent.prototype.createContent.apply(this, arguments);
        },
        onBeforeApfStartup: function() {       //optional
            // Code executed before APF startup
        },

        onAfterApfStartup: function() {        //optional
            // Code executed after APF startup
        },
        destroy : function() {
            // Destroy application instances

            // Call destroy on APF Component
            ApfComponent.prototype.destroy.apply(this, arguments);
        }
    });
});
```

Replace *myApp* with the application-specific namespace.

Method `this.getApi()` provides a reference to the APF instance.

The function registered through `onBeforeApfStartup` is executed after the execution of method `init()`, at the beginning of method `createContent()` of `sap.apf.base.Component`. This registered function is useful, for example, for defining application-specific filters.

The function registered through `onAfterApfStartup` is executed after all asynchronous startup operations have been terminated, that is, at the end of method `createContent()` of `sap.apf.base.Component`. This registered function is useful, for example, for adding footer content to the APF UI.

Footer Content

You can add footer content to your APF-based application, for example, to allow users to make settings such as defining a reporting currency or adjusting the exchange rate settings.

To add footer content, attach the following APF start-up callbacks at APF API level in the `Component.js` file inside the `createContent()` method:

1. `setCallbackBeforeApfStartup`, where you can build footer controls.
2. `setCallbackAfterApfStartup`, where you can insert the footer content into the UI layout using the `addMasterFooterContent` API.

Footer controls need to register a listener to the `contextChanged` event of APF to listen to context changes at startup or when a saved path is opened. The event listener for the `contextChanged` event is defined as follows:
`oApi.setEventCallback(oApi.constants.eventTypes.contextChanged,fnCallbackForContextChange);`

Apart from the `contextChanged` event, you can also register the following events for `setEventCallback`:

- `printTrigerred` (to implement a specific formatting for a print page)

- `format` (to implement a specific formatting for an entire application)

Creating Filters for the Footer Content

To ensure that the filters you create as footer content are recognized by APF, create them in the following format:

```
var oFilter = this.oApi.createFilter();
var orExpression = oFilter.getTopAnd().addOr();
orExpression.addExpression({
    name : "<name>",
    operator : "<operator>",
    value : "<value>",
  });
```

For example, if you want to configure a filter for the exchange rate type, the filter expression looks as follows:

```
orExpression.addExpression({
    name : "P_ExchangeRateType",
    operator : "EQ",
    value : "USD",
  });
```

Use the API `addPathFilter()` to ensure that the created filter is applied to the analysis path. When the API `addPathFilter()` is called with the filter created above as a parameter, it returns an ID. The filter can be updated using the API `updatePathFilter()` by passing the filter ID and the filter as parameters:

```
var sFilterId = this.oApi.addPathFilter(oFilter);
this.oApi.updatePathFilter(sFilterId, oFilter);
```

To retrieve the filter that was applied to the analysis path, use the API getPathFilter() and pass the filter ID as a parameter:

```
var oFilter = this.oApi.getPathFilter(sFilterId);
var sValue = oFilter.getInternalFilter().getFilterTerms()[0].getValue();
```

Security Considerations

For any application extension, ensure that both the extension and its libraries are trustworthy.

Related Information

Components [page 720]
Descriptor (manifest.json) [page 2124]

Consuming APF in SAPUI5 1.28 and Prior Releases

The following code snippet is an example of how to consume APF in a UI5 based application up until SAPUI5 1.28:

```
jQuery.sap.declare("myApp.Component");
sap.ui.getCore().loadLibrary("sap.apf");
jQuery.sap.require("sap.apf.Component");

sap.apf.Component.extend("myApp.Component",
{
        metadata : {
            name : <name>,
            version : <version>
        },
        /**
        * Initialize the application
        *
        * @returns
        */
        init : function() {
            // Initialize application here. No APF specific operation done here,
since APF API is not yet available.
            // Call APF Component init
            sap.apf.Component.prototype.init.apply(this, arguments);
        },
        /**
        * Creates the application layout and returns the outer layout of APF
        *
        * @returns {sap.ui.core.Control} the content
        */
        createContent : function() {
            // Attach APF start-up callbacks
            this.getApi().setCallbackBeforeApfStartup(this.onBeforeApfStartup);
            this.getApi().setCallbackAfterApfStartup(this.onAfterApfStartup);

            // Prepare path to application configuration file
            var modPath = jQuery.sap.getModulePath('myApp');
            var configFilePath = modPath + "/config/
myApplicationConfiguration.json";
            this.getApi().loadApplicationConfig(configFilePath);

            // Return whatever is returned by parent (APF Component)
createContent method
            return sap.apf.Component.prototype.createContent.apply(this,
arguments);
        },

        onBeforeApfStartup: function() {       //optional
            // Code executed before APF startup
        },

        onAfterApfStartup: function() {        //optional
            // Code executed after APF startup
        },
        destroy : function() {
            // Destroy application instances

            // Call destroy on APF Component
            sap.apf.Component.prototype.destroy.apply(this, arguments);
        }
    });
```

Replace *myApp* with the application-specific namespace.

Method `this.getApi()` provides a reference to the APF instance.

When you use the start parameter `sap-apf-app-config-path`, APF executes method `loadApplicationConfig()` in the `init()` method of `sap.apf.Component`, that is, before `createContent()` of the application component is executed. APF ensures that method `loadApplicationConfig()` is executed not more than once.

The function registered through `onBeforeApfStartup` is executed after the execution of method `init()`, at the beginning of method `createContent()` of `sap.apf.Component`. This registered function is useful, for example, for defining application-specific filters.

The function registered through `onAfterApfStartup` is executed after all asynchronous startup operations have been terminated, that is, at the end of method `createContent()` of `sap.apf.Component`. This registered function is useful, for example, for adding footer content to the APF UI.

Footer Content

You can add footer content to your APF-based application, for example, to allow users to make settings such as defining a reporting currency or adjusting the exchange rate settings.

To add footer content, attach the following APF start-up callbacks at APF API level in the `Component.js` file inside the `createContent()` method:

1. `setCallbackBeforeApfStartup`, where you can build footer controls.
2. `setCallbackAfterApfStartup`, where you can insert the footer content into the UI layout using the `addMasterFooterContent` API.

Footer controls need to register a listener to the `contextChanged` event of APF to listen to context changes at startup or when a saved path is opened. The event listener for the `contextChanged` event is defined as follows:
`oApi.setEventCallback(oApi.constants.eventTypes.contextChanged,fnCallbackForContextChange);`

Creating Filters for the Footer Content

To ensure that the filters you create as footer content are recognized by APF, create them in the following format:

```
var oFilter = this.oApi.createFilter();
var orExpression = oFilter.getTopAnd().addOr();
orExpression.addExpression({
    name : "<name>",
    operator : "<operator>",
    value : "<value>",
  });
```

For example, if you want to configure a filter for the exchange rate type, the filter expression looks as follows:

```
orExpression.addExpression({
    name : "P_ExchangeRateType",
    operator : "EQ",
    value : "USD",
  });
```

Use the API `addPathFilter()` to ensure that the created filter is applied to the analysis path. When the API `addPathFilter()` is called with the filter created above as a parameter, it returns an ID. The filter can be updated using the API `updatePathFilter()` by passing the filter ID and the filter as parameters:

Source Code

```
var sFilterId = this.oApi.addPathFilter(oFilter);
this.oApi.updatePathFilter(sFilterId, oFilter);
```

To retrieve the filter that was applied to the analysis path, use the API getPathFilter() and pass the filter ID as a parameter:

```
var oFilter = this.oApi.getPathFilter(sFilterId);
var sValue = oFilter.getInternalFilter().getFilterTerms()[0].getValue();
```

Security Considerations

For any application extension, ensure that both the extension and its libraries are trustworthy.

Related Information

Components [page 720]

Outbound Navigation and Inbound Navigation

You can navigate from an APF-based application to another SAP Fiori application (outbound navigation) and you can also navigate from an SAP Fiori application to an APF-based application (inbound navigation).

In both cases, the source application sends a context object to the target application using the app state. The context object contains a filter object. This filter object represents selections made in the source application and hands over these selections to the target application in the form of a filter. The target application applies the filter or parts of it. For example, the target application can apply the filter to its data requests or visualize the filter as selections in the charts on the UI.

Outbound Navigation

When navigating from an APF-based application to another application, APF puts the cumulative filter of the current analysis path into the context object. The cumulative filter includes start filters and other filters such as selections made in the charts up to the active analysis step. The context object stores the cumulative filter in the formats described in the section Filter Formats [page 2117].

APF tries to reduce the cumulative filter to the select options format. If this is successful, the resulting filter is stored in the context object in the property `selectionVariant`.

If it is not possible to reduce the cumulative filter, the value of the property `selectionVariant` contains an error text instead of a select option. This is because select options can only express a subset of all possible filters whereas the format `sap.ui.model.Filter` can express all filters created by APF.

In addition, APF always creates a filter in the `sap.ui.model.Filter` format and stores it in the property `sapApfCumulativeFilter`.

The consuming application can read the app state as follows:

```
sap.ushell.Container
    .getService("CrossApplicationNavigation")
    .getAppState(oInject.instances.component, crossAppStateKey)
    .done(function(appState) {
        contextObject = appState.getData();
        if (contextObject && contextObject.sapApfCumulativeFilter) {
            // your code that processes the filter
        }
    });
```

Inbound Navigation

When you navigate from an SAP Fiori application to an APF-based application, the SAP Fiori application can hand over a filter in the context object. The APF-based application automatically applies this filter to the analysis path. The context object contains the filter in one of the formats described in the section Filter Formats [page 2117].

The source application can set the app state as follows:

```
var oCrossAppNavigator =
sap.ushell.Container.getService("CrossApplicationNavigation");
var contextObject = {};
contextObject.sapApfCumulativeFilter = //add filter here
contextObject.selectionVariant = //add selectionVariant here
appState = oCrossAppNavigator.createEmptyAppState(oInject.instances.component);
appState.setData(contextObject);
appState.save();
oCrossAppNavigator.toExternal({
    target : {
        semanticObject : oNavigationTarget.semanticObject,
        action : oNavigationTarget.action
    },
    appStateKey : appState.getKey()
});
```

Filter Formats

APF hands over the cumulative filter of the current analysis path including all analysis steps up to the current step using the `CrossApplicationNavigation` service of the unified shell. For more information, see the API Reference in the Demo Kit.

The context object contains two different properties that hand over the filter in two different formats:

- Property `sapApfCumulativeFilter`:
  Filter object in the format of `sap.ui.model.Filter`
  For more information, see the API Reference in the Demo Kit.
- Property `selectionVariant`:
  Filter object in a select options format
  For more information, search for "Selection Variants" in the documentation of your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

> **i Note**
>
> The type information of a property is not included in any of the filters. If the consuming application requires the type information, it can be derived from the metadata.

The Select Options Format

When the property `selectionVariant` of a context object is not undefined, it contains an object of the following form:

```
{
    "SelectionVariantID": <String>,
    "ParameterContextUrl": <String>,
    "FilterContextUrl": <String>,
    "Text": <String>,
    "Parameters": [],
    "ODataFilterExpression": <String>,
    "SelectOptions": […]
}
```

The property `SelectOptions` contains the filter object. The filter object is an array that expresses a conjunction. The elements of the array are range expressions.

```
"SelectOptions": [
    {
        "PropertyName": <String>,
        "Ranges": […]
    }
]
```

A range expression is a filter that represents ranges and disjunctions of values.

```
{
    "Sign": "I" | "E",
    "Option": <Char(2)>,
    "Low": <String>,
    "High": <String> | null
}
```

`Sign` expresses inclusion or exclusion. `Option` expresses the operator, for instance, "EQ" for equal or "BT" for between. The other two properties express a low value and a high value. The high value is optional. If it is not used, it is set to null.

Example: The following filter expresses that values are either equal to 1 or between 3 and 5:

```
"Ranges": [
    {
        "Sign": "I",
        "Option": "EQ",
```

```
        "Low": "0001",
        "High": null
    },
    {
        "Sign": "I",
        "Option": "BT",
        "Low": "0003",
        "High": "0005"
    }
]
```

> **i** Note
>
> APF does not support exclusions (property `Sign` equals "E").

Example

```
{
    "SelectionVariantID" : "20141023134315",
    "ParameterContextUrl" : "/(..)/AccountBalance/
$metadata#AccountBalanceQueryParameters",
    "FilterContextUrl" : "/(..)/AccountBalance/
$metadata#AccountBalanceQueryResult",
    "Text" : "Temporary Selection Variant, Account Balance, 24.10.2014 13:43:15",
    "Parameters" : [
        {
            "PropertyName" : "DisplayCurrency",
            "PropertyValue" : "EUR"
        },
        {
            "PropertyName" : "ExchangeRateType",
            "PropertyValue" : "M"
        }
    ],
    "ODataFilterExpression" : "",
    "SelectOptions" : [
        {
            "PropertyName" : "CompanyCode",
            "Ranges" : [
                {
                "Sign" : "I",
                "Option" : "EQ",
                "Low" : "0001",
                "High" : null
                },
                {
                "Sign" : "I",
                "Option" : "EQ",
                "Low" : "0002",
                "High" : null
                }
            ]
        },
        {
            "PropertyName" : "FiscalYear",
            "Ranges" : [
                {
                "Sign" : "I",
                "Option" : "EQ",
                "Low" : "2014",
                "High" : null
                }
```

```
            ]
        },
        {
            "PropertyName" : "GLAccount",
            "Ranges" : [
                {
                "Sign" : "I",
                "Option" : "BT",
                "Low" : "10000",
                "High" : "20000"
                },
                {
                "Sign" : "I",
                "Option" : "EQ",
                "Low" : "30000",
                "High" : null
                }
            ]
        }
] }
```

Related Information

Creating Navigation Targets [page 2085]
Analysis Path Processing [page 2108]

Working with Multiple Back-End Systems

If you work with multiple back-end systems because, for example, you have a system landscape with regional back-end systems, you can use the `sap-system` parameter to ensure that SAP Gateway directs the OData service requests to the correct back-end system. For each back-end system, you can configure a SAP Fiori tile with the corresponding `sap-system` parameter, so you can have several tiles for the same APF-based app accessing data from different systems.

APF uses the general concept for SAP Fiori applications on multiple back-end systems. For more information, search for "Configuring Multiple Back-End Systems Using the sap-system Parameter" in the documentation of your SAP NetWeaver version on the SAP Help Portal at https://help.sap.com/viewer/p/SAP_NETWEAVER.

In particular, you must do the following:

- Assign all required back-end systems to your analytical OData services.
- If you want to save your analysis paths on multiple back-end systems, you must also assign those back-end systems to the service `BSANLY_APF_RUNTIME_SRV`, which is used for path persistence.
- If you use static app launcher tiles to launch an APF-based app, go to the tile configuration and add the parameter `sap-system=<SYSTEM_ALIAS>`. Create one tile for each back-end system you want to connect the app to. For each tile, enter the system alias of one back-end system as the `sap-system` parameter.
- If you navigate to an APF-based app from somewhere else, for example, from a SAP Smart Business generic drill-down app that also uses the `sap-system` parameter, the `sap-system` parameter is handed over to APF via intent-based navigation.

As a result, APF interprets the `sap-system` parameter and reacts as follows:

- All OData requests that are used when running an APF-based app are sent with an origin segment parameter `o` that corresponds to the specified `sap-system` parameter.
- When navigating away from the APF-based app, the `sap-system` parameter is added to the URL of the navigation target.
- When executing a configuration from the APF Configuration Modeler, the `sap-system` parameter is added to the URL of the generic runtime application.

Configuration Files and Their Structure

To configure or enhance an APF-based application, you can use the APF Configuration Modeler app. When you export a configuration, a JSON file is created that contains all configuration objects, such as categories, steps, and representations. The following chapters explain the configuration on a technical level and show how the JSON files are structured and what the properties used in the configuration objects mean.

The following JSON files are relevant:

- The `manifest.json` file, which defines static information about the application.

  > **i Note**
  >
  > In SAPUI5 1.28 and prior releases, the static information was defined in the application configuration file.

- The analytical configuration file, which defines the content of the application and how it is represented on the user interface.

  > **i Note**
  >
  > We recommend not to edit the analytical configuration file manually. However, there are exceptions such as a mass change of OData service paths.

The configuration options include the following tasks:

- Changing the analytical configuration file

  > **i Note**
  >
  > When you want to change the analytical configuration file of a delivered application, you must first make a copy and save it to a new location.

  Some possible changes to the analytical configuration file are:
  - You can change a request, which defines the access to a server resource.
  - You can change the relation between the requested data and the representations by adapting the binding.
  - You can add a category for the analysis step gallery.
  - You can add new analysis steps or adapt existing ones, for example, by replacing the request or the binding for a step.
- Changing text resource files

> **Note**
>
> When you want to change a text resource file of a delivered application, you must first make a copy and save it to a new location.

- Changing the `manifest.json` file
  Whenever you have changed the analytical configuration file or text resource files and have copied the files to a new location, you must ensure that the locations are adapted accordingly in the `manifest.json` file.

The configuration options are explained in detail in the following sections.

Application Configuration in SAPUI5 1.28 and Prior Releases

> **Note**
>
> As of SAPUI5 1.30, the application configuration file has been replaced with the `manifest.json` file. If you build your app based on SAPUI5 1.30 or higher, you can skip this section. For information about the `manifest.json` file, see Descriptor (manifest.json) [page 2124]

The application configuration file defines static information about the application, such as the name of the application or the location of various files. It is written in JavaScript Object Notation (JSON) format.

In addition to the properties described below, you can define further properties as required using method `getApplicationConfigurationProperties`. For example, you can define default values for fields on the UI.

> **Note**
>
> If you create an APF-based application using the generic APF runtime application, you can omit this step, because the application configuration is already contained in it. However, if you create your own BSP application, you must also create an application configuration file.

> **Note**
>
> Customer modifications may conflict with the SAP namespace and can be overwritten when updates are imported.

The application configuration file has the following format:

```
{
    "applicationConfiguration" : {
        "type" : "applicationConfiguration",
        "appName" : "<key>",
        "appTitle" : "<key> | <text>",
        "analyticalConfigurationLocation" : "<applicationPath>/config/
configuration.json",
        "applicationMessageDefinitionLocation" : "<applicationPath>/config/
applicationMessageDefinition.json",
        "textResourceLocations" : {
            "applicationMessageTextBundle" : "<applicationPath>/i18n/
applicationMessages.properties",
            "applicationUiTextBundle" : "<applicationPath>/i18n/
applicationUi.properties"
```

```
        },
        "persistence" : {
            "path" : {
                "service" : "<service root>",
                "entitySet" : "<entity set name>"
            },
            "logicalSystem" : {
                "service" : null
            }
        }
        "smartBusiness" : {
            "runtime" : {
                "service" : "/sap/hba/r/sb/core/odata/runtime/
SMART_BUSINESS.xsodata",
            }
        }
    }
}
```

> **i Note**
>
> `<applicationPath>` denotes the location of the Web application on the Web server.

The properties used in the application configuration file denote the following:

Property	Description
`appName`	The name of the application is displayed in the header of the application. It is retrieved from a translatable text property using a text key. Therefore, the value of the property `appName` must be a text key.
`appTitle`	The title is displayed in the title bar of the Web browser. It is optional and can be used to overwrite the `<title>` tag of the index.html file. The property `appTitle` can be a text key, but it can also be text that is not translated.
`analyticalConfigurationLocation`	The location of analytical configuration file for APF; for more information, see Analytical Configuration [page 2128].
`applicationMessageDefinitionLocation`	The location of the message definition file for the application. If the application uses the `MessageHandling` component of APF, this definition is used to retrieve further information for message handling, such as a text.
`textResourceLocations`	The location of the text resource files. Text resource files contain the texts that the text keys used in the code and the message definition files refer to. Text resource files can be of type `*.hdbtextbundle` or `*.properties` and contain UI texts and message texts that can be translated. At runtime, a text key is resolved by checking all specified resource files in turn.

Property	Description
`persistence`	Contains two properties: • `path` Specifies the service to create, read, update, and delete analysis paths on the persistence layer. The property `service` defines the service document. • `logicalSystem` Specifies the service to determine the logical system. In most cases, you can set the service to **`null`**.
`smartBusiness`	The Smart Business runtime service, which is used to fetch information related to Smart Business filters using the evaluation ID.

Descriptor (manifest.json)

As of SAPUI5 version 1.30, APF uses the `manifest.json` file as descriptor. It replaces the application configuration file. The `manifest.json` file defines static information about the application, such as the name of the application or the location of various files. It is written in JavaScript Object Notation (JSON) format.

> **i Note**
>
> If your application still uses the application configuration file, you can skip this section.
>
> You can also omit this step if you create an APF-based application using the generic APF runtime application because the `manifest.json` file is already contained in it. However, if you create your own BSP application with a component that extends `sap.apf.base.Component`, you must also create a `manifest.json` file.

> **i Note**
>
> Customer modifications may conflict with the SAP namespace and can be overwritten when updates are imported.

For information about the structure and content of the `manifest.json` file, see Descriptor for Applications, Components, and Libraries [page 734].

APF expects certain entries in the manifest of a component that extends `sap.apf.core.Component`. Entries for four different data sources have to be defined in the `sap.app` namespace:

- For the data source `AnalyticalConfigurationLocation`, you must specify the location of the analytical configuration file. Enter the relative path from the Component.js.
- The following three data sources are predefined and must not be changed:
  - `PathPersistenceServiceRoot`
  - `SmartBusiness`
  - `LogicalSystem`

The data sources differ depending on whether you use SAP Business Suite powered by SAP HANA, or SAP S/4HANA. The following table gives an overview about the relevant entries that you must use in your own `manifest.json` file:

Data Source	...for SAP Business Suite powered by SAP HANA and SAP BW on SAP HANA	...for SAP S/4HANA
`PathPersistenceServiceRoot`	`/sap/hba/r/apf/core/odata/apf.xsodata`	`/sap/opu/odata/sap/BSANLY_APF_RUNTIME_SRV`
`SmartBusiness`	`/sap/hba/r/sb/core/odata/runtime/SMART_BUSINESS.xsodata`	Not required
`LogicalSystem`	`/sap/hba/apps/wca/dso/s/odata/wca.xsodata`	Not required

The `sap.app` namespace also contains the title of the application. This “title” entry references the text key `AnalyticalConfigurationName`. When you export the text pool from APF Configuration Modeler, the up-to-date configuration title is written into the .properties file with the text key `AnalyticalConfigurationName`. Ensure you also keep the location of the .properties file up to date in the `manifest.json` file (entry “i18n”). At runtime, this title is displayed as the browser tab title.

Example

```
{
        "_version": "1.1.0",
        "sap.app": {
                "_version": "1.1.0",
                "id": "<component ID>",
                "type": "application",
                "i18n": "i18n/texts.properties",
                "title": "{{AnalyticalConfigurationName}}",
                "description": "{{<key in .properties file>}}",
                "applicationVersion": {
                        "version": "${project.version}"
                },
                "ach": "<ach>",
                "dataSources": {
                        "<data source for analytical request>": {
                                "uri": "<service root of analytical request>",
                                "type": "OData",
                                "settings": {
                                        "annotations": [
                                                "<annotation data source>"
                                        ],
                                        "odataVersion": "2.0"
                                }
                        },
                        "<annotation data source>": {
                                "uri": "<location of annotation file>",
                                "type": "ODataAnnotation",
                                "settings": {
                                        "localUri": "<location of local
annotation file>"
                                }
```

```
                        },
                        "PathPersistenceServiceRoot": {
                                "uri": "<URI of path persistence>",
                                "type": "OData",
                                "settings": {
                                        "odataVersion": "2.0"
                                }
                        },
                        "SmartBusiness": {
                                "uri": "/sap/hba/r/sb/core/odata/runtime/
SMART_BUSINESS.xsodata",
                                "type": "OData",
                                "settings": {
                                        "odataVersion": "2.0"
                                }
                        },
                        "LogicalSystem": {
                                "uri": "/sap/hba/apps/wca/dso/s/odata/
wca.xsodata",
                                "type": "OData",
                                "settings": {
                                        "odataVersion": "2.0"
                                }
                        }
                },
                "AnalyticalConfigurationLocation": {
                        "uri": "./config/analyticalConfiguration.json",
                        "type": "JSON"
                },
                "resources": "resources.json",
                "offline": false
        },
        "sap.fiori": {
                "_version": "1.1.0",
                "registrationIds": [
                        "<FioriId>"
                ],
                "archeType": "analytical"
        },
        "sap.ui": {
                "_version": "1.1.0",
                "technology": "UI5",
                "deviceTypes": {
                        "desktop": true,
                        "tablet": true,
                        "phone": false
                },
                "supportedThemes": [
                        "sap_hcb",
                        "sap_belize"
                ]
        },
        "sap.ui5": {
                "_version": "1.1.0",
                "dependencies": {
                        "minUI5Version": "1.38.1",
                        "libs": {
                                "sap.ui.core": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
                                },
                                "sap.ca.ui": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
                                },
                                "sap.m": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
```

```
                                },
                                "sap.ui.layout": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
                                },
                                "sap.ushell": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
                                },
                                "sap.apf": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
                                },
                                "sap.viz": {
                                        "minVersion": "1.38.1"
                                        "lazy" : false
                                }
                        }
                },
                "extends": {
                        "component": "sap.apf.base"
                },
                "contentDensities": {
                        "compact": true,
                        "cozy": true
                },
                "models": {
                        "i18n": {
                                "type": "sap.ui.model.resource.ResourceModel",
                                "uri": "i18n/texts.properties"
                        }
                }
                "config": {
                        "sapFiori2Adaptation": true
                },
        }
}
```

APF-Specific Settings

The following settings reside in the `sap.apf` name space:

```
"sap.apf": {
    "activateFilterReduction" : true,
    "activateLrep" : true,
    },
```

- `activateFilterReduction`:
  This property is relevant if you use CDS views that are executed on the Analytic Engine or BW OData queries. It defines whether filters that are generated during a path update are reduced so that the Analytic Engine can handle them.
  Alternatively, you can use the equivalent boolean URL parameter `sap-apf-filter-reduction`, which you can set to true. If a URL parameter exists, it overrules the entry in the `manifest.json`. If you use the generic APF runtime application, you must use the URL parameter.
  If you use filter reduction, make sure you don't use more than one property in the filter mapping configuration.
- `activateLrep`

> **i Note**
>
> If you use SAP S/4HANA, this property is mandatory and must be set to `true`.

This property defines that the layered repository of SAPUI5 flexibility is used for persistence of the analytical configuration.

Analytical Configuration

The analytical configuration of APF specifies, for example, the available analysis steps, the representations that are used for each step, and the categories in which the analysis steps are displayed. It is written in JavaScript Object Notation (JSON) format.

Some of the objects used to specify the analytical content must be registered under a unique ID and can be accessed using this ID. The IDs must be unique within the configuration of an APF-based Web application.

The following table lists the objects used in an analytical configuration:

Object Type	Unique ID Required
`step`	Yes
`hierarchicalStep`	Yes
`request`	Yes
`binding`	Yes
`representation`	Yes
`representationType` (optional)	Yes
`facetFilter`	Yes
`smartFilterBarConfiguration`	Yes
`navigationTarget`	Yes
`category`	Yes
`label`	No
`thumbnail`	No

General Rules

The following rules apply when working with configuration objects in Analysis Path Framework:

- Value representations in configuration objects must comply with the JSON format.
- String values must be in double quotes.
- The *type* property is optional, but it is useful for debugging and comprehensibility.
- The *id* property is mandatory for those objects that require a unique ID. It must be a unique ID of type string.
- Depending on the object type, configuration objects can contain various other properties. Changes to a set of properties must be compatible with the original configuration object. You can, for example, add new properties.

> **i Note**
>
> Customer modifications may conflict with the SAP namespace and can be overwritten when updates are imported.

The Configuration Root Object

The configuration root object encapsulates all configuration objects that represent a function and that exist as runtime objects. The format is as follows:

```
{
  steps : [ <step> | <hierarchicalStep>* ],
  requests : [ <request>* ],
  bindings : [ <binding>* ],
  categories : [ <category>* ],
  representationTypes : [ <representationType>* ],
  facetFilters : [ <facetFilter>* ],    // optional
  smartFilterBar : <smartFilterBarConfiguration>,    // optional
  navigationTargets : [ <navigationTarget>* ],    // optional
  configHeader : <configurationInformation>
 }
```

> **i Note**
>
> The notation `[ <type>* ]` indicates an array of one or more elements.
>
> The configuration can have either facet filters or a smart filter bar, not both.

The Step Object

The `step` object in the analytical configuration file defines static data such as labels. It is referenced in the `category` objects in which the step is displayed in the analysis step gallery.

The `step` object consists of a `request` object and a `binding` object. The `request` defines the access to the server resource, that is, it retrieves the data that is displayed on the UI. The `binding` maps the required

representation types to the analysis step, which defines the way in which the data is displayed on the UI, for example, in a chart or a list.

On the UI, analysis steps are shown in the analysis step gallery as template objects. The step template object is derived from the `step` object in the configuration file, but contains only the information that is required on the UI, for example, the texts and the representations. When a user selects an analysis step from the gallery, a new step instance is created. An instance of a `step` object provides the representation type that is currently used both in the analysis step display as a chart or a list, and in the analysis path display as a thumbnail.

Optionally, the step object can contain the filter mapping configuration. Filter mapping can be useful when a selection that can be made in one analysis step cannot be handled by a subsequent step. In this case, the system determines the source filter property based on the selections made in a chart and maps them to another filter that can be used in the requests for subsequent steps in the path (mapped filter property).

The step object has the following format:

```
{
  "type" : "step",    // optional
  "id" : "<step ID>",
  "title" : <label>
  "longTitle" : <label>    // optional
  "request" : "<request ID>",
  "binding" : "<binding ID>",
  "topNSettings" : {    //optional
      "top" : "<number>",
      "orderby" : [{
          "property" : "<property name>",
          "ascending" : <boolean>
      }*]
  },
  "navigationTargets" : [ {"type" : "navigationTarget" , "id" : "<navigation
target ID>"}* ]
  "thumbnail" : <thumbnail>,
   "filterMapping" : {    // optional
      "requestForMappedFilter" : "<request ID>",
      "target" : [ <mapped filter properties>*],
      "keepSource" : <boolean>
  }
}
```

The properties used in the step object denote the following:

Property	Description
`id`	Unique ID of type string
`title`	Title displayed in the analysis step gallery and in the analysis path display.
`longTitle`	Title displayed above the representation in the analysis step display. If no long title is defined, the title is used instead.
`request`	ID of the request used for this analysis step.
`binding`	ID of the binding used for this analysis step.
`thumbnail`	Displayed in the analysis path display.

Property	Description
`filterMapping`	Maps a filter derived from the selection in a chart (source filter property) to another filter that can be used in the requests for subsequent steps in the path (mapped filter property). If the `filterMapping` property does not exist, no filter mapping is required for this step.
`requestForMappedFilter`	Contains the ID of the `request` object used to do the lookup request. The lookup request maps the source filter properties to the mapped filter properties. The source filter properties are derived from the selections made in the current step. They correspond to property `requiredFilter` of the `binding` object. Therefore, the properties used in the source filter must be contained as filterable properties in `requestForMappedFilter`.
`target`	Array of properties that are used to determine the mapped filter properties based on the response from `requestForMappedFilter`.
`keepSource`	Determines whether the source filter property is kept in the cumulative filter in addition to the mapped filter property. true = Both filter properties are kept in the cumulative filter for subsequent steps. false = The mapped filter property only is kept in the cumulative filter.

Related Information

The Request Object [page 2133]
The Binding Object [page 2134]

The Hierarchical Step Object

The hierarchical step uses the tree table to visualize hierarchical data.

The hierarchical step object has the following format:

```
{
  "type" : "hierarchical step",    // optional
  "id" : "<step ID>",
  "title" : <label>
  "longTitle" : <label>    // optional
  "request" : "<request ID>",
```

```
  "binding" : "<binding ID>",
  "navigationTargets" : [ {"type" : "navigationTarget" , "id" : "<navigation
target ID>"}* ]
  "hierarchyProperty": "<property name>"
  "thumbnail" : <thumbnail>,
  "filterMapping" : {    // optional
      "requestForMappedFilter" : "<request ID>",
      "target" : [ <mapped filter properties>*],
      "keepSource" : <boolean>
  }
}
```

The properties used in the step object denote the following:

Property	Description
`id`	Unique ID of type string
`title`	Title displayed in the analysis step gallery and in the analysis path display.
`longTitle`	Title displayed above the representation in the analysis step display. If no long title is defined, the title is used instead.
`request`	ID of the request used for this analysis step.
`binding`	ID of the binding used for this analysis step.
navigationTargets	IDs of the navigation targets used for this analysis step.
hierarchyProperty	The property that is provided in a hierarchical form.
`thumbnail`	Displayed in the analysis path display.
`filterMapping`	Maps a filter derived from the selection in a chart (source filter property) to another filter that can be used in the requests for subsequent steps in the path (mapped filter property). If the `filterMapping` property does not exist, no filter mapping is required for this step.
`requestForMappedFilter`	Contains the ID of the `request` object used to do the lookup request. The lookup request maps the source filter properties to the mapped filter properties. The source filter properties are derived from the selections made in the current step. They correspond to property `requiredFilter` of the `binding` object. Therefore, the properties used in the source filter must be contained as filterable properties in `requestForMappedFilter`.
`target`	Array of properties that are used to determine the mapped filter properties based on the response from `requestForMappedFilter`.

Property	Description
`keepSource`	Determines whether the source filter property is kept in the cumulative filter in addition to the mapped filter property. true = Both filter properties are kept in the cumulative filter for subsequent steps. false = The mapped filter property only is kept in the cumulative filter.

The Request Object

The `request` object defines the access to a server resource by specifying an OData service and an entity set.

A `request` object has the following format:

```
<id>"
  "service" : "
{
  "type" : "request",    // optional
  "id" : "<service root>",
  "entitySet" : "<entity set name>",
  "selectProperties" : [ "<property name>"* ],
}
```

The properties used in the `request` object denote the following:

Property	Description
`id`	Unique ID of type string.
`service`	Path to OData service root. The service root is defined by an absolute path on the OData server according to the following pattern: `<service root>: ".../odata/<odata-service-document>"` The `request` { and its entity set name are absolute to the service root, which starts after the server and port information: `[protocol]://[server]:[port]/<service root>`
`entitySet`	Entity set that corresponds to the data source, for example, the SAP HANA view.

Property	Description
`selectProperties`	Set of property names. The selected properties determine which properties of the entity are contained in the server response. Therefore, they also determine the analytical processing of the request on the server.

The Binding Object

The `binding` object defines the relation between the requested data and one or more representations. It also defines the mapping of selections made on the UI to OData filter expressions. The `binding` object has the following format:

```
{
  "type" : "binding",    //optional
  "id" : "<id>"
  "requiredFilters" : [ "<property name>"* ]
  "representations" : [ <representation>* ],
}
```

The properties used in the `binding` object denote the following:

Property	Description
`id`	Unique ID of type string
`requiredFilters`	Contract between binding and request; specify the target properties for converting selected data into filter expressions
`representations`	An array of `representation` objects.

Related Information

The Representation Object [page 2134]

The Representation Object

The `representation` object defines the relation between the requested data and the representation type, for example, a particular chart type or a list. The format is as follows:

```
{
```

```
  "type" : "representation",    // optional
  "id" : "<id>"
  "representationTypeId" : "<representation type ID>",
  "parameter" : {
    "type" : "parameter"
    "dimensions" : [ <dimensions>* ]    //optional
    "measures" : [ <measures>* ]    // optional
    "properties" : [ <properties>* ]    //optional
    "hierarchicalProperty" : [ <hierarchical property> ]    //optional
    "width" : "<width>"
    "alternateRepresentationType" : "TableRepresentation"       //optional
    "top" : "<number>"       //optional
    "orderby" : [ <orderby>* ]             //optional
            }
}
```

The properties used in the `representation` object denote the following:

Property	Description
`id`	Unique ID of type string.
`representationTypeId`	ID of the `representation type`
`parameter`	Defines specific information for the representation, for example, dimensions, measures, properties, the optional alternate table representation, and the column width for table representations. Dimensions, measures, properties and hierarchical property are all optional, but at least one of them must exist, depending on the representation type: Charts have dimensions and measures, tables have properties, and tree tables have a hierarchical property and optionally more properties. At runtime, the constructor defined in the object, which contains a label, an image, and a reference to the constructor function of the representation type `representation type` object is called with the parameters defined by this property.

Properties

The `properties` property has the following format:

```
"properties": [
    {
        "fieldName": "Customer",
        "kind": "column",
        "fieldDesc": {
            "type": "label",
            "kind": "text",
            "key": "<key>"
        }
    }
],
```

Hierarchical Properties

The `hierarchicalProperty` property has the following format:

```
"hierarchicalProperty": [
    {
        "fieldName": "Customer",
        "kind": "hierarchicalColumn",
        "fieldDesc": {
            "type": "label",
            "kind": "text",
            "key": "<key>"
        },
        "labelDisplayOption": "text"
    }
],
```

Dimensions and Measures

The `parameter` property can contain dimensions and measures for the representation.

The `dimensions` property has the following format:

```
*  object, which contains a"dimensions" : [
    {
        "fieldName" : "<field name>",
        "kind" : "<value>",
        "fieldDesc" : {
            "type" : "label",
            "kind" : "text",
            "key" : "<key>"
        },
        "labelDisplayOption": "text"
    }
]
```

The kind attribute provides the option to maintain multiple dimensions. It is mapped to `feedItemId`. The following table lists the `feedItemId` for the available charts:

Chart Type	Dimension	feedItemId
Line chart, line chart with two vertical axes, column chart, bar chart, stacked column chart, stacked bar chart, 100% stacked column chart, 100% stacked bar chart, combined column line chart, combined stacked column line chart, combined column line chart with two vertical axes, combined stacked column line chart with two vertical axes	First dimension	categoryAxis
	Second dimension	color
Line chart with time axis	First dimension	timeAxis

Chart Type	Dimension	feedItemId
	Second dimension	color
Heatmap chart	First dimension	categoryAxis
	Second dimension	valueAxis
Pie chart, donut chart	Dimension	color
Scatter chart, bubble chart	First dimension	color
	Second dimension	shape

The `measures` property has the following format:

```
"measures" : [
    {
        "fieldName" : "<field name>",
        "kind" : "<value>",
        "fieldDesc" : {
            "type" : "label",
            "kind" : "text",
            "key" : "<key>"
        },
        "measureDisplayOption" : "bar"
    }
]
```

The kind attribute provides the option to maintain multiple measures. It is mapped to `feedItemId`. The following table lists the `feedItemId` for the available charts:

Chart Type	Measure	feedItemId
Line chart, line chart with time axis, column chart, bar chart, stacked column chart, stacked bar chart, 100% stacked column chart, 100% stacked bar chart, combined column line chart, combined stacked column line chart	Measure	valueAxis
Heatmap chart	Measure	Color
Pie chart, donut chart	Measure	size
Scatter chart, line chart with two vertical axes, combined column line chart with two vertical axes, combined stacked column line chart with two vertical axes	First measure	valueAxis
	Second measure	valueAxis2
Bubble chart	First measure	valueAxis

Chart Type	Measure	feedItemId
	Second measure	valueAxis2
	Third measure	bubbleWidth

The optional `fieldDesc` property of the `dimensions` and `measures` properties can be used for rendering the axis titles in the chart. For more information, see Rendering of Charts [page 2107].

Alternate Representation

For each representation, you can define an alternate representation. A user can switch to the alternate representation directly from any representation without having to access the gallery of available representation types.

To define an alternate representation, the configuration file must contain the following parameter:

```
"alternateRepresentationType" : "tableRepresentation"
```

The property `alternateRepresentationType` contains a reference to the `representation type ID` that is configured with details such as the constructor. The constructor of an alternate representation type is handed over to the representation instance using the parameter object.

Ordering

Using the `orderby` property, you can define the properties by which the data depicted in the representation is ordered. You can also specify the ordering direction (true = ascending, false = descending).

The `orderby` property has the following format:

```
"orderby" : [{              //optional
        "property" : "<property name>",
        "ascending" : <boolean>
        }*]
```

The Representation Type Object

> **i Note**
>
> This step is required only if you want to define your own representation types in addition to the predefined ones shipped with APF. For more information, see Predefined Representation Types [page 2105].

The `representation type` object defines how data is depicted on the UI. It has the following format:

```
{
```

```
  "type" : "representationType",    // optional
  "id" : "<id>",
  "constructor" : <constructor>,
  "picture" : "<path/file>",    // relative to Web application root
  "label" : <label>
}
```

The properties used in the `representation type` object denote the following:

Properties	Description
`id`	Unique ID of type string.
`constructor`	Defines the implementing function object of the representation.
`picture`	Icon indicating the representation type.
`label`	Defines the label text.

The Facet Filter Object

The analytical configuration can optionally contain one or more facet filter objects, which define the configuration of the facet filters used in an APF-based application. Facet filters provide global filters that are applied to all analysis steps of an analysis path.

The facet filter object has the following format:

```
{
  "type" : "facetFilter",    //optional
  "description" : <same as label text>
  "id" : "<id>",
  "property" : "<property>",
  "alias" : "<property>",   //optional
  "invisible" : <boolean>    //optional
  "valueHelpRequest" : "<request ID>",    //optional
  "filterResolutionRequest" : "<request ID>",    //optional
  "multiSelection" : <boolean>,    //optional
  "preselectionDefaults" : [ "<value>"* ],     //optional
  "preselectionFunction" : <function path>,    //optional
  "label" : <label>    //optional
}
```

The properties used in the facet filter object denote the following:

Property	Description
`description`	Contains the string entered as label so that a specific facet filter object can easily be found in the file.
`id`	Unique ID of type string

Property	Description
`property`	Filter property for the facet filter. If `alias` is not defined, `property` is also used as select property for `valueHelpRequest`.
`alias`	Select property for `valueHelpRequest`. Used if the field name of the property of the facet filter is different in the `valueHelpRequest`. If `property` and `alias` are the same, `alias` can be omitted.
`valueHelpRequest`	ID of the request used for providing a value help to select values in the facet filter.
`filterResolutionRequest`	ID of the request used for resolving a filter into single values for the value help.
`multiSelection`	Defines whether selection of multiple values is possible.
`preselectionDefaults`	Array of values that are preselected in the facet filter.
`preselectionFunction`	Path to the function used if the preselected values need to be calculated.
`label`	Defines the label text.

If a `preselectionFunction` is defined as string in the facet filter object, the system internally converts it into a callable function before the configuration object is returned. If the defined function string cannot be resolved or does not point to a function, the property `preselectFunction` is set to *undefined* .

The Smart Filter Bar Object

The analytical configuration can optionally contain a smart filter bar object, which defines the configuration of the smart filter bar used in an APF-based application. The smart filter bar can be used as an alternative for facet filters and provides global filters that are applied to all analysis steps of an analysis path. A configuration can have either facet filters or a smart filter bar, not both.

The annotation file of the service root defines which properties and parameters are displayed in the smart filter bar. Therefore, the manifest.json file of the component must contain the data source definition of the service root as well as the data source definition of the corresponding annotation file.

The smart filter bar object has the following format:

```
{
    "type" : "smartFilterBar",
    "id" : "SmartFilterBar ID",
    "service" : "<service root>",
```

```
    "entitySet" : "<entity set name>",
}
```

The properties used in the step object denote the following:

Property	Description
`id`	Unique ID of type string
`service`	Path to OData service root
`entitySet`	Entity set that corresponds to the data source, for example, the SAP HANA view. The entity set provides the properties and parameters that can be displayed in the smart filter bar.

The Navigation Target Object

The navigation target object defines applications that a user can navigate to from an APF-based application. It has the following format:

Source Code

```
{
  "type" : "navigationTarget",    // optional
  "id" : "<id>",
  "semanticObject" : "<semantic object>"
  "action" : "<action>"
  "isStepSpecific" : <boolean>
}
```

The properties used in the navigation target object denote the following:

Table 97:

Property	Description
`id`	Unique ID of type string
`semanticObject`	Semantic object as defined in the corresponding target mapping configured in the Fiori launchpad designer.
`action`	Action as defined in the corresponding target mapping configured in the Fiori launchpad designer.
`isStepSpecific`	Determines whether the navigation is assigned to all steps or to specific steps. true = assigned to specific steps false = assigned to all steps

The Configuration Header Object

The configuration header holds administrative data of the analytical configuration. It is an optional part of the configuration, but it is automatically created when you export a configuration using the APF Configuration Modeler. If you manually create your own analytical configuration file, you don't have to add a configuration header.

The configuration header has the following format:

```
"configHeader": {
    "Application": "<32-digit GUID>",
    "ApplicationName": "<application description>",
    "SemanticObject": "<semantic object name>",
    "AnalyticalConfiguration": "<32-digit GUID>",
    "AnalyticalConfigurationName": "<configuration title>",
    "UI5Version": "<your SAPUI5 version, for example, 1.38.1-SNAPSHOT>",
    "CreationUTCDateTime": "/Date(1415784024931)/",
    "LastChangeUTCDateTime": "/Date(1415815299519)/"
}
```

The Category Object

The `category` object defines the categories that are displayed in the analysis step gallery. It has the following format:

```
{
  "type" : "category",    // optional
  "id" : "<id>",
  "label" : <label>,
  "steps" : [{"type" : "step", "id" : "<step ID>"}*]
}
```

The order in which the steps are listed in the steps array determines the order in which the steps are displayed in the analysis step gallery at runtime.

The Label Object

The `label` object is a subobject that defines a label text. It can be used, for example, in the `thumbnail` object or the `representation type` object to define the texts for the UI.

The `label` object contains a key that refers to a text resource file, where all texts are maintained and identified by the registered key. The path to this resource file is configured in the `manifest.json` file.

The `label` object has the following format:

```
{
  "type" : "label",    // optional
  "kind" : "text",
  "key" : "<key>"    // text key related to resource file
}
```

The Thumbnail Object

The `thumbnail` object defines the visualization of a selected analysis step in the analysis path display of the UI. It has the following format:

```
{
  "type" : "thumbnail",    // optional
  "leftUpper" : <label>,
  "leftLower" : <label>,
  "rightUpper" : <label>,
  "rightLower" : <label>
}
```

The labels define the text that is displayed in the four corners of the thumbnail. All properties of the `thumbnail` object are optional.

API Reference

You can find the API Reference for Analysis Path Framework in the Demo Kit under `sap.apf`.

Extending Apps

You can adapt an SAPUI5 app to your specific requirements. For example, you can adapt or replace views, extend or replace controllers, or change language-specific texts.

With SAPUI5 you have the option to extend your applications. You can do so by using SAPUI5 flexibility or component configuration.

SAPUI5 flexibility [page 2144] is the preferred way to extend SAP Fiori elements-based apps for versions 1.56 or higher. It uses a better interface, supports layering as well as lifecycle hooks.

Using Component Configuration [page 2145] is intended for versions below 1.56 and for freestyle applications. It is based on merge, supports only override of methods and requires additional component configurations. The extension information is stored in a specific area of the component configuration. It can be performed on a custom app that extends a delivered standard app. A replacement or extension of views and custom controllers can also be part of a custom app, but may not always be required. If no replacement and no custom controller exists, the custom app project only contains the component definition with the extension configuration. The standard app itself is not changed. The customized app becomes the start-up project and launches the standard app with the additional extension configuration.

For more information on how to create layered controller extensions in SAP Web IDE, see Related Information.

By using key user adaptation, users with the key user authorization role can also make extensive UI changes for apps without having to change the code. For more information, see SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450].

Related Information

Layering Concept [page 1156]

Using SAPUI5 Flexibility

You can use SAPUI5 flexibility to extend your SAP Fiori apps.

Overview

SAPUI5 flexibility (see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152]) allows you to extend SAP Fiori elements-based applications without modifications of base artefacts. To use this type of extensibility, you need an adaptation project in SAP Web IDE. It allows you to make changes that are also available in key user adaptation (see SAPUI5 Flexibility: Enable Your App for UI Adaptation [page 1450]) as well as to extend views with XML fragments (see XML Fragments [page 1005]) and extend controllers with controller extensions.

Using SAPUI5 Flexibility for Controller Extensions

Controller extensions add or override functionality of existing applications. For more information, see Using Controller Extension [page 810].

By using SAPUI5 flexibility and adaptation projects, controller extensions are added to the reserved `extension` namespace of the controller. This is done in order to avoid name clashes with already existing functionality in the controller, for example `oMainControllerInstance.extension.my.sample.ControllerExtension.publicMethod()`.

Usage in a View

The controller extension functionality can be used in views in the same way as a regular controller function. Deep namespaces are already supported in this scenario and it should work without additional effort.

```
<Button press=".extension.my.sample.ControllerExtension.publicMethod"
text="Execute"></Button>
```

Override Extension Functionality

It is possible to override public or extend lifecycle methods of controller extension in another controller extension.

```
override: {
    //override other extensions method by the namespace extension name
    extension: {
        "my.sample.ControllerExtension": {
            publicMethod: function() {
                //...
            }
```

```
        }
    }
}
```

You can also override an extension directly in a controller.

Sample Code

```
sap.ui.define(['sap/ui/core/mvc/Controller', 'my/extension/SampleExtension'],
    function(Controller, SampleExtension) {
    "use strict";
    return MainController = Controller.extend("sample.Main", {
        //inline override of an extension. E.g. to provide a hook
implementation
        sample: SampleExtension.override({
            someHook: function() {},
            someOtherMethod: function() {}
        }),
        onLifecycleHook: function() {
        }
    });
});
```

Using Component Configuration

SAPUI5 supports the extension of a base controller by merging the delivered standard controller with a custom controller on JavaScript object level.

The SAPUI5 controller extension concept does **not** use inheritance. Instead, methods of the custom controller override standard methods with the same name. The following controller lifecycle methods are, however, an exception to this rule: `onInit`, `onExit`, `onBeforeRendering`, `onAfterRendering`. For these methods, the controller methods of your custom application are called either after (for `onInit` and `onAfterRendering`), or before (for `onExit` and `onBeforeRendering`) the standard lifecycle methods.

The following examples show how controller extension concept in SAPUI5 works. The following code snippet shows the standard controller `Main.controller.js` of the delivered standard application:

```
sap.ui.define(["sap/ui/core/mvc/Controller"], function(Controller) {
    "use strict";
    return Controller.extend("samples.components.ext.sap.Main", {
        onInit : function () {
            console.log("samples.components.ext.sap.Main - onInit");
        },
            doSomething: function() {
            alert("this is an original standard action");
        },
            doSomeStandardAction: function() {
            alert("this is another original standard action");
        }
    });
});
```

The following code snippet represents the custom controller `CustomMain.controller.js`:

```
sap.ui.define(["sap/ui/core/mvc/Controller"], function(Controller) {
    "use strict";
    return Controller.extend("samples.components.ext.customer.CustomMain", {
```

```
        onInit : function () {
             console.log("samples.components.ext.customer.CustomMain - onInit");
         },
             doSomething: function() {
             alert("this is a customer action");
         },
             doSomeCustomAction: function() {
             alert("this is another customer action");
         }
    });
});
```

The following extension in component configuration merges the two controllers:

```
extensions: {
    "sap.ui.controllerExtensions": {
        "samples.components.ext.sap.Main": {
            controllerName: "samples.components.ext.customer.CustomMain"
        }
        }
    // .....some more content
}
```

As a result, the `samples.components.ext.customer.CustomMain` controller functions are merged when the controller `samples.components.ext.sap.Main` is called. After initialization, the log contains the following messages:

```
samples.components.ext.sap.Main - onInit
samples.components.ext.customer.CustomMain - onInit
```

The `doSomething` method of the new controller overwrites the `doSomething` method of the standard controller. Thus, if the method is invoked, an alert popup with the following text appears: *this is a customer action*.

The `doSomeStandardAction` method remains available without changes, as no method with the same name exists in the new controller.

The `doSomeCustomAction` method is additionally available and you can use it, for example, in a view extension.

The controller extensions are applied to all controllers with the specified name within the customized component, regardless of whether the controller is instantiated explicitly or belongs to a view.

Example: Component Configuration

The component configuration contains the information about the extension metadata and the objects that are replaced or extended in the custom view or control.

The configuration is stored in the component.js file of the custom application. The component of the custom application needs to inherit from the main component of the original application. To make the location of the original application or component known to SAPUI5, it may be necessary to use `registerModulePath(...)`. The configuration in the extension section contains the extension metadata and describes the objects that are replaced or extended.

The following code snippet shows an example of a configuration structure.

```
some.sap.Component.extend("some.customer.Component", {

    metadata : {
        .....some configuration
        config: {
            .....some configuration
        },
        extensions: {

            "sap.ui.viewExtensions": {
                "samples.components.ext.sap.Sub2": {
                    "extension2": {
                        className: "sap.ui.core.Fragment",
                        fragmentName:
"samples.components.ext.customer.CustomFrag1",
                        type: "XML"
                    },
                    "extension3": {
                        className: "sap.ui.core.mvc.View",
                        viewName:
"samples.components.ext.customer.CustomSubSubView1",
                        type: "XML"
                    }
                },
                "samples.components.ext.sap.Sub4": {
                    "extension4": {
                        className: "sap.ui.core.Fragment",
                        fragmentName:
"samples.components.ext.customer.CustomFrag2",
                        type: "JS"
                    }
                }
            },

            "sap.ui.viewModifications": {
                "samples.components.ext.sap.Sub3": {
                    "someCustomizableTextControl": {
                        "visible": false
                    }
                }
            },
            "sap.ui.viewReplacements": {
                "samples.components.ext.sap.Sub1": {
                    viewName: "samples.components.ext.customer.CustomSub1",
                    type: "XML"
                }
            },

            "sap.ui.controllerExtensions": {
                "samples.components.ext.sap.Main": {
                    "controllerName":
"samples.components.ext.customer.MainExtension"
                }
            },

            "sap.ui.controllerReplacements": {
                "samples.components.ext.sap.Main":
"samples.components.ext.customer.MainReplacement"
            }
        }
    }
});
```

`"sap.ui.viewExtensions"`: Provides custom view content in a specified extension point in the delivered standard application

`"sap.ui.viewModifications"`: Used for overriding control properties of the delivered standard application

`"sap.ui.viewReplacements"`: Used for replacing a standard view with a custom view

`"sap.ui.controllerExtensions"`: Used for extending a controller in a delivered standard application with a custom controller

`sap.ui.controllerReplacements`: Used for replacing a controller in a delivered standard application with a custom controller

Providing Hooks in the Standard Controller

Hooks are extension points in the controller code that are used to make controller extensions more stable.

The controller extension concept enables you to override any method. This is a powerful but also fragile feature. Extension points, so-called hooks, can be provided in the controller code. These hooks can be documented and kept stable, thus providing more robust hooks across application updates for controller extensions.

The process for this is as follows:

1. In the application, identify a strategic location within the controller code where customers may want to plug in and execute their customized code.
2. In the application, define a new function name which is reserved for the extension, document the function and any arguments the function may receive or return.
3. Add code lines in the application (see code snippet below) to check whether the function has been implemented, and, if so, to call the function. We also recommend to implement sanity checks for return values.
4. The customer can then configure a controller extension, implementing exactly this one function.
5. SAPUI5 runtime merges the new controller extension into the standard controller. If customizing is enabled, the new function can be executed.

Example

By receiving the data object `oSomeData` from the server, the application enables you to access and modify the data object. The extension function name is `onDataReceived` and gets a reference to the data object as argument.

Code of the standard controller:

```
// ...data object oSomeData has been received, possibly from an Ajax response...
   if (this.onDataReceived) {         // check whether any extension has
implemented the hook...
      this.onDataReceived(oSomeData); // ...and call it
   }
   // ...continue working with the (now possibly modified) data...
```

Code of the custom controller:

```
sap.ui.controller("customer.xy.Sub2ControllerExtension", {
   onDataReceived: function(oData){ // oSomeData will be passed in
      if (oData && oData.status === "important") {
         oData.message = oData.message + "!!!"; // modify some part of the data
object, adding exclamation marks to a message text
      }
   } // no need to return anything as in this example the original object is
modified
});
```

> **i Note**
>
> This only works for one extension layer as the most specific or last extension overrides any other hook implementations. To allow multi-layer extensions, we recommend that middle-layer extensions provide and document their own hook functions.
>
> This also requires flat, non-inherited controllers defined with the `sap.ui.controller(...)` function used as extension controller, and not with typed controllers.

View Extension

SAPUI5 uses extension points for extending standard views with custom content. The assignment of a custom view to an extension point is done in component customizing.

You can add extension points in a standard view to indicate the position within the view where you can insert custom content. In XML views, the `<ExtensionPoint>` tag is used and replaced by the controls provided by the customer. The tag should therefore be placed in the view where also a control would be placed and document the types of controls that are suitable.

In the XML view below, for example, three extension points are defined: `extension1`, `extension2`, and `extension3`. The extension name together with the view name identifies an extension point.

```
<mvc:View xmlns="sap.m"  xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc">
    <core:ExtensionPoint name="extension1" />
    <TextView text="SAP View 'Sub2' - this one is extended by the customer and
there should be a button after this text"></TextView>
    <core:ExtensionPoint name="extension2" />
    <core:ExtensionPoint name="extension3" />
</mvc:View>
```

In JS views, extension points can be created within the `createContent` method by using the `sap.ui.extensionpoint` function. The following example shows the simplest way to initiate an extension point in a JS view.

```
[...]
      createContent(oController){
           return sap.ui.extensionpoint(this, "extension4");
      }
[...]
```

> **i Note**
>
> This example creates a view with one extension point, which can be customized to hold controls, but do not show any default content.

You can add an extension point to an aggregation of another control by specifying the target control and, optionally, a target aggregation. The target aggregation is only required when you do not want to add the extension point to your target control's default aggregation. In the following example, an extension point is added to `VerticalLayout`:

```
[...]
      createContent(oController){
         var oLayout = new sap.ui.layout.VerticalLayout("Layout1");
         sap.ui.extensionpoint(this, "extension4", null, oLayout, "content" /
*not mandatory, as content is the default aggregation*/);
      }
[...]
```

You can also use the JSON notation to create the surrounding control and add the extension point to an aggregation as follows:

```
[...]
    var oLayout = new sap.ui.layout.VerticalLayout({
        content: [
                  new sap.m.Button({text: "I am preceding the extension point"}),
                  sap.ui.extensionpoint(this, "extension4"),
                  new sap.m.Button({text: "I am following the extension point"})
                 ]
    });
[...]
```

For table-like controls with aggregations that span two dimensions, for example, rows and columns, extension points must be provided for both dimensions. In the `sap.m.Table`, for example, one extension point needs to be provided in the `columns` aggregation, another one in the provided `cells` aggregation of the template`ColumnListItem`.

Extension Points with Default Content

Applications can also use extension points to provide default content, which is used as long as no custom content is defined and ignored when custom content is defined. This feature is particularly interesting for aggregations, which are filled by data binding: In XML views, one item is given which is used as a template. In applications that enable exchanging these items by custom `ListItems`, the default list items can be enclosed in an `<ExtensionPoint>` tag.

```
<mvc:View xmlns="sap.m"  xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc">
    <ListBox items="{/names}">
        <core:ExtensionPoint name="extension1" />
            <ListItem text="{firstName} />
        </core:ExtensionPoint>
    </ListBox>
</mvc:View>
```

This can be used for all multiple aggregations, not only for lists.

To define default content for extension points in JS views or fragments, specify the value of another parameter:

```
sap.ui.extensionpoint(this, "extension4", fnCreateDefaultContent); // this
extension point has a callback function creating default content
```

The function provided as a callback needs to return a control or an array of controls and is only executed when no customizing for the extension is configured, or when customizing is disabled.

Extension Point Implementation

The extension content, which will then be inserted at the position of an extension point, is defined in the custom application. For example, for the extension points that have been defined in the standard application described in the section above, custom content can be defined for each extension. This is shown in the example below together with a description of the customizing that connects the extension point in the delivered standard application and the extension content in the customer application.

```
extensions: {

    "sap.ui.viewExtensions": {
        "samples.components.ext.sap.Sub2": {
            "extension2": {
                className: "sap.ui.core.Fragment",
                fragmentName: "samples.components.ext.customer.CustomFrag1",
                type: "XML"
                id: "stableid"
            },
            "extension3": {
                className: "sap.ui.core.mvc.View",
                viewName: "samples.components.ext.customer.CustomSubSubView1",
                type: "XML"
            }
        },
    .....some more content
```

If you add an ID to the view extension, this ID overrules the original view ID. For view fragments like in the example, also IDs of nested controls are then prefixed with this ID.

Extension content in the custom application in `CustomFrag1.fragment.xml` file:

```
<Button xmlns="sap.m" text="This Button is in an Extension Fragment" />
```

> **i Note**
>
> You can also add multiple root-level controls using one fragment.

Extension content in the custom application in `CustomSubSubView1.view.xml` file.

```
<mvc:View xmlns="sap.m"  xmlns:core="sap.ui.core" xmlns:mvc="sap.ui.core.mvc">
    <core:ExtensionPoint name="extension1" />
    <TextView text="Customer View 'SubSubView1' - this one extends the original
SAP View 'Sub2' - and even custom Views can be extended:"></TextView>
    <core:ExtensionPoint name="extension2" />
</mvc:View>
```

View Modification

For modifying views, control properties of standard views can be changed.

The view modification is currently restricted to the `visible` property of controls, meaning that controls can be hidden in the custom application. The controls must have the `visible` property and the control's ID must be defined in the view. The view name together with the control ID uniquely determines the control in the standard application.

View modification is available for XML views, JS views and HTML views. Below is an example that explains how a view modification works. The first code snippet describes the `Sub3.view.xml` view in the delivered standard application.

```
<mvc:View xmlns="sap." xmlns:mvc="sap.ui.core.mvc">
    <TextView text="SAP View 'Sub3' - the text after this one is hidden by
customizing: "></TextView>
    <TextView id="someCustomizableTextControl" text="This text is made invisible
by customization"></TextView>
</mvc:View>
```

The following code snippet describes the extensions for the control `someCustomizableTextControl` with the `visible` property set to `false`.

```
extensions: {
    "sap.ui.viewModifications": {
        "samples.components.ext.sap.Sub3": {
            "someCustomizableTextControl": {
                "visible": false
            }
        }
    }
}
```

View Replacement

Views of a delivered standard application can be replaced to adapt the application to the customer needs.

If the extension points provided for view extension are not sufficient to meet the requirements of the custom application, you can replace the standard view with a custom view.

The following view is delivered in the standard application:

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <TextView text="SAP View 'Sub1' - this one should have been replaced by the
customer View"></TextView>
</mvc:View>
```

This is the custom view to replace the standard view:

```
<mvc:View xmlns="sap.m" xmlns:mvc="sap.ui.core.mvc">
    <TextView text="Custom View 'Sub1' - this one replaces the original SAP View
'Sub1'"></TextView>
</mvc:View>
```

The following extension replaces the custom view with the view in the standard application

```
extensions: {
    .....some more content
    "sap.ui.viewReplacements": {
        "samples.components.ext.sap.Sub1": {
            viewName: "samples.components.ext.customer.CustomSub1",
            type: "XML"
        }
    },
    .....some more content
```

Controller Replacement

Standard controller can be replaced by specifying a new controller name in a replacement View and implementing this Controller.

For a view replacement, you can either use the standard controller of the replaced view by setting its name as `controllerName`, or use and extend the standard controller, or you can replace the controller by specifying a new controller name in the new view and implementing the new controller.

An extension option is available that allows to replace an original controller without replacing its view. This is especially useful for typed controllers, that is, controllers that have been defined with the `extend` syntax:

```
sap.ui.define([
    "sap/ui/core/mvc/Controller"
], function(Controller) {
    "use strict";

    return Controller.extend("samples.components.ext.customer.CustomMain", {
        onInit: function() { /* do something */ },
        onBeforeRendering: function() { /* do something */ },
        onAfterRendering: function() { /* do something */ },
        onExit: function() { /* do something */ },
        myEventHandler: function(oEvent) { /* do something */ }
    });

});
```

To replace the controller of the standard application with the custom controller, use the following extension configuration:

```
extensions: {
    "sap.ui.controllerReplacements": {
        "samples.components.ext.sap.Main":
"samples.components.ext.customer.CustomMain"
    },
    .....some more content
```

> ⚠ Caution
>
> Typed controllers cannot be extended by using the controller extension configuration (`sap.ui.controllerExtensions`). Instead, you use the controller replacement configuration (`sap.ui.controllerReplacements`) to extend a typed controller with the `extend` syntax and call the original methods in the custom implementation:
>
> ```
> sap.ui.define([
> ```

```
    "samples/components/ext/customer/CustomMain"
], function(CustomController) {
    "use strict";

    return
CustomController.extend("samples.components.ext.partner.PartnerMain", {
        onInit: function() {
            CustomController.prototype.onInit.apply(this, arguments);
            /* do something */
        },
        onBeforeRendering: function() {
            CustomController.prototype.onBeforeRendering.apply(this,
arguments);
            /* do something */
        },
        onAfterRendering: function() {
            /* do something */
            CustomController.prototype.onAfterRendering.apply(this,
arguments);
        },
        onExit: function() {
            /* do something */
            CustomController.prototype.onExit.apply(this, arguments);
        },
        myEventHandler: function(oEvent) {
            CustomController.prototype.myEventHandler.apply(this, arguments);
            /* do something */
        }
    });

});
```

The chaining of the lifecycle methods is **not** done automatically. You can control on your own, if or when to call the parent lifecycle methods. In addition, you can always access the methods defined in the original controller.

Localized Texts for Extended Apps

You can add custom localized text files that contain additional texts or texts that overwrite the original texts to the `sap.ui.model.resource.ResourceModel`

The enhanced resource model tries to resolve the localized texts from the custom resource bundle first. If a text does not exist there, it tries to look up the text in the resource bundle of the original app. Custom resource bundles cannot be added by standard extension configuration, but must be added as part of a controller extension as shown in the following example:

```
var oModel = new sap.ui.model.resource.ResourceModel({bundleUrl:"./testdata/
messages.properties"});
oModel.enhance({bundleUrl:"./testdata/messages_custom.properties"});
```

Specifics for Extended Apps in the ABAP Repository

For applications located in the ABAP back end, the resource model of the original application can be accessed by its relative path. If the code is not located at the root level of the application, adjust the path with an additional `../` at the beginning:

```
var oModel = new sap.ui.model.resource.ResourceModel({bundleUrl:"../<original
bsp application name>/i18n/i18n.properties"});
```

You can enhance this with an additional custom text properties file, which resides in the copied application:

```
oModel.enhance({bundleUrl:"./other18n.properties"});
```

If an SAPUI5 application is extended or copied, the GUID in the SAPUI5 translation key in the copied properties file must be exchanged with a new one. Each properties file must contain a unique GUID. You can then upload the application with the new translation key in the text properties file to the ABAP back end by means of the team provider or the `/UI5/UI5_REPOSITORY_LOAD` upload report. To localize the custom texts in the copied application you can use transaction `se63`.

> ⚠ Caution
>
> Although it is generally possible to copy SAPUI5 applications, copying and editing of applications in the SAPUI5 text repository must be avoided and we strongly recommend to only use the extension concepts.
>
> If an application is copied, the texts stored in the SAPUI5 text repository of the original application are **not** copied and are, thus, not available in the copied SAPUI5 application. This also applies to applications using the extension concepts of SAPUI5.

Limitations

Known limitations for extending SAPUI5 applications

The control property modification is only supported for the `visible` property and only for controls with a given ID specified in the XML view.

Caveats Regarding Stability Across Application Upgrades

There are a few limitations in the compatibility of custom applications that have to be considered when extending SAPUI5 standard applications.

Offering modification-free extensions and customizing allows to ship new versions of the application without overwriting customer-specific modifications. Thus, these modifications can survive an application upgrade. However, the degree of compatibility which can be guaranteed is limited. If a view is replaced by a custom view but the original view is no longer used, the custom view will, of course, also no longer be used. Or, if a custom controller extension accesses a field in the original view, it has to be able to cope with the possibility that the field may no longer exist or have a different type in an upgraded version of the application, otherwise it will break.

Many possible reasons exist that modifications can not be applied any longer or even break the upgraded version of an application and it is, therefore, not possible to compile an exhaustive list of all possible reasons. The following list gives some examples:

- View modifications (hiding controls)
  - If the original control is no longer used or has a different or not a given ID, a view modification will no longer be applied.
  - If the original view name is changed or the view is no longer used, an invalid view modification will no longer be applied.
  - However, a view modification is simply ignored and will never lead to a crash.
- View extensions (added content in extension points)
  - If the extension point is removed or renamed or in an area or container which is invisible under certain conditions, the view extension will no longer be applied.
  - If the view name is changed or the extension point is moved to a different view, the view extension will no longer be applied.
  - If the controls around the extension point have changed or the extension point has been moved to a different environment inside the same view, view extensions may look weird, have a broken layout or display or do not really fit the new environment.
  - If the updated application requires the extension to be of a certain control type, the view extension may break the application.
  - If custom code relies on the presence of the extensions, the view extension may break the application.
- View replacements
  - If the original view name is changed or the view is no longer used, view replacements will no longer be applied.
  - As long as no other custom code relies on the view to be present, view replacements should not cause a crash.
- Controller extensions
  - If the extension code accesses parts of the original application which are changed, for example, removed, have a different type or a different ID, controller extensions can cause a crash.
  - If the extension code makes assumptions about the application which are no longer valid after an update, controller extensions can lead to a crash.
  - If original code is overwritten which is required for the application to run properly, controller extensions can lead to a crash.
  - If the controller name is changed, controller extensions are no longer applied.
- Other extension types have similar caveats.

In addition to this, all content changes such as additions, removals, and structure changes may affect CSS and JavaScript code which relies of the position of certain elements in the DOM. Partly these caveats are compatibility requirements for applications, partly they are suggestions for customizing development how to create more robust extensions.

Supportability

In case of problems or errors in the custom application, several options exist that support you in resolving the issues.

Disabling Extensions/Customizing

If a customized application does not run properly, you can disable the customizing. In a support case, for example, you can set a breakpoint early in the `sap-ui-core.js` and then execute the following code in the browser developer tool console:

```
window["sap-ui-config"] = window["sap-ui-config"] ||{};
window["sap-ui-config"]["xx-disableCustomizing"] = true;
```

> **i** Note
>
> For security reasons, it is not possible to use a URL parameter.

Using the Log

The console log contains information about the processing of customizing or extensibility information. Depending on the importance of the respective information, different log levels are used. To enable a certain log level, execute the following code in the browser console:

```
// "Log" required from module "sap/base/Log"
Log.setLevel(Log.Level.INFO)
```

As an alternative, you can set the log level to `INFO` in the support popup if you want to see all messages of level "INFO" or more important.

The following information is provided per log level:

- `WARNING/ERROR`: Any critical or error situation; such messages must be checked because something may have gone wrong
- `INFO`: Information about successful customizing activities, such as applying a view extension or modifying a control property
- `DEBUG`: Information about "non-activities", for example, if no extension configured for an extension point was found; this provides comprehensive information for each situation where an extension might be configured. Exception: Non-existing control property modifications are **not** logged at this log level (see below how you can enable this). In addition, a complete dump of the extensibility configuration is logged when it is activated.

Certain information is only logged when an additional URL parameter is used, because otherwise there would be too much information in the log for other debugging scenarios: To enable this extra logging, add `sap-ui-xx-debugCustomizing` to the query part of the URL. This extra logging enables an explicit log statement for each control, for which **no** control modification has been found.

Reading the Current Customizing Data

To dump the complete extensibility data, use the following command in the browser console:

```
sap.ui.require(["sap/ui/core/CustomizingConfiguration"],
function(CustomizingConfiguration) {
  CustomizingConfiguration.log()
});
```

As a result, an object is returned in the console which contains the customizing configuration, structured by type of customization and view name.

> **i** Note
>
> If Customizing is not enabled, this command causes an error because `sap.ui.core.CustomizingConfiguration` is not defined.

Developing Controls

You can create own content for SAPUI5. To develop controls in JavaScript, you can either extend existing controls or create new ones.

> **i** Note
>
> If you want to contribute to SAPUI5, you have to consider our guidelines and recommendations with regard to, for example, product standards, file names and encoding.

As a control developer, you create or modify UI libraries and their pieces, i.e. controls and types. You define the set of properties your control provides as well as events or aggregations. A major task is the implementation of a control-specific renderer, which knows how to create suitable HTML markup for a given control instance, taking its current state into account. A renderer is written in JavaScript and produces HTML output which is styled by means of CSS. Such style sheets are another important part of a UI library.

Controls can be defined on the fly without a library definition or running generation steps. These controls are also called notepad controls.

When you want to develop several controls for reuse in different applications, we recommend creating a control library instead of using these notepad controls. Control libraries have additional features such as automatic support for theming and right-to-left languages, but the implementation of the controls is the same as for notepad controls.

> **i** Note
>
> This functionality is not restricted to controls. It can also be used to create or extend arbitrary objects, such as components, that are derived from `sap.ui.base.ManagedObject`. For more information, see API Reference: `sap.ui.base.ManagedObject`.

Development Conventions and Guidelines

To keep the SAPUI5 code readable and maintainable, development conventions and guidelines are introduced. We strongly recommend that you follow these guidelines even if you find them violated somewhere. For files that are consistently **not** following these rules and for which adhering to the rules would make the code worse, follow the local style. If you want to contribute your content to SAPUI5, you **have to** follow these conventions and guidelines.

> **i** Note
>
> This list is not complete.

General Guidelines

The following list gives some general guidelines to be adhered to when developing content for SAPUI5:

- Always consider the developers who **use** your control or code! Do not surprise them, but give them what they expect. And make it simple.
- Use tabs, not spaces, for indentation; adhere to local standards in the file.
- Use Unix line endings (LF-only).
- Text files must be UTF-8 encoded (HANA); only `*.properties` and `*.hdbtextbundle` files must be ISO8859-1 encoded as defined in the corresponding standard.
  This is the status quo. As this causes issues, it may be subject to change..
- An 80-character line length guideline does **not** exist.
- Use comments; do **not** rephrase the code, but tell the reader what is **not** in the code. Describe why your code does what it does. Prefer line comments.

JavaScript Coding Guidelines

Provides an overview of the guidelines for JavaScript coding for SAPUI5 with regard to code formatting, naming conventions, and creating classes.

For JavaScript, the following **general** guidelines apply:

- Do **not** use global JavaScript variables; organize all global objects in an `sap.*` namespace structure. The module `sap/base/util/ObjectPath` assists in doing so. For more information, see JavaScript Namespaces [page 2162] and API Reference: `jQuery.sap.getObject`.
  This also means: Do **not** use undeclared variables. When using global variables introduced by other libraries, declare the use in a special global comment: `/*global JSZip, OpenAjax */`.
- Do **not** access internal (private) members of other objects.
- Do **not** use `console.log()`
- Use `window.document.getElementById("<someId>")` instead of `jQuery("#<someId>")` when `<someId>` is not a known string; certain characters in IDs need to be escaped for jQuery to work correctly.

- Keep modifications of jQuery and other embedded Open Source to a minimum and document them clearly with the term *SAP modification*. Such modifications may **not** alter the standard behavior of the used library in a way that breaks other libraries

Code Formatting

For any code becoming part of SAPUI5, an ESLint check needs to run successfully, see Tools [page 2179]. The following list contains the most important formatting rules:

- Add a semicolon after each statement, even if optional
- No spaces before and after round braces (function calls, function parameters), but...
- ...use spaces after `if/else/for/while/do/switch/try/catch/finally`, around curly braces, around operators and after commas
- Opening curly brace (functions, for, if-else, switch) is on the same line
- Use "===" and "!==" instead of "==" and "!="; see the ESLint docu for special cases where "==" is allowed
- The code should therefore look like this:

```
function outer(c, d) {
        var e = c * d;
        if (e === 0) {
            e++;
        }
        for (var i = 0; i < e; i++) {
            // do nothing
        }
        function inner(a, b) {
            return (e * a) + b;
        }
        return inner(0, 1);
    }
```

Naming Conventions

We strongly recommend to use the Hungarian notation where name prefixes indicate the type for variables and object field names. But do **not** use the Hungarian notation for API method parameters: The documentation specifies the type in this case.

When using the Hungarian notation, use the prefixes highlighted below and continue with an uppercase letter (camelCase):

Sample	Type
sId	string
oDomRef	object
$DomRef	jQuery object
iCount	int
mParameters	map / assoc. array

Sample	Type
aEntries	array
dToday	date
fDecimal	float
bEnabled	boolean
rPattern	RegExp
fnFunction	function
vVariant	variant types

Use CamelCase for class names, starting with an uppercase letter. HTML element IDs starting with `sap-ui-` are reserved for SAPUI5. DOM attribute names starting with `data-sap-ui-` as well as URL parameter names starting with `sap-` and `sap-ui-` are reserved for SAPUI5.

The following IDs are currently used:

ID	Description
`sap-ui-bootstrap`	ID of the bootstrap script tag
`sap-ui-library-*`	Prefix for UI libraries script tags
`sap-ui-theme-*`	Prefix for theme stylesheets link tags
`sap-ui-highlightrect`	ID of the highlight rect for controls in TestSuite
`sap-ui-blindlayer-*`	ID for `BlockLayer`
`sap-ui-static`	ID of the static popup area of UI5
`sap-ui-TraceWindowRoot`	ID of the `TraceWindowRoot`
`sap-ui-xmldata`	ID of the `XML Data Island`

Creating Classes

For the creation of classes, the following rules and guidelines apply:

- Initialize and describe instance fields in the constructor function: `this._bReady = false; // ready to handle requests`
- Define instance methods as members of the prototype of the constructor function: `MyClass.prototype.doSomething = function(){...`
- Define static members (fields and functions) as members of the constructor function object itself: `MyClass.doSomething = function(){...`
- Start the name of private members with an underscore: `this._bFinalized`
- Combine constructor + methods + statics in a single JS source file named and located after the qualified name of the class; this is a precondition for class loading
- Static classes do not have a constructor but an object literal; there is no pattern for inheritance of such classes. If inheritance is needed, use a normal class and create a singleton in the class.

- Do not use `SuperClass.extend(…)` for subclasses. If no base class exists, the prototype is automatically initialized by JavaScript as an empty object literal and must not be assigned manually. Consider inheriting from `sap/ui/base/Object`
- Subclasses call (or apply) the constructor of their base class: `SuperClass.apply(this, arguments);`

For more information, see Example for Defining a Class [page 2162].

JavaScript Namespaces

SAPUI5 modules such as classes, components, and controls, should use a consistent qualified naming scheme. Each module should reside in a unique namespace.

Naming Conventions

A namespace should be lowercase and each word should be separated by a dot (`.`), like the Java package notation. The class name should be camelcase, starting with a capital letter.

In the following example, `my.app` is the general namespace and `my.app.MyControl` is the fully qualified class name.

```
sap.ui.define(["sap/ui/core/Control"], function(Control) {
    return Control.extend("my.app.MyControl", {});
});
```

For JavaScript global names, module names, and SAPUI5 qualified names (class names, interface names, DataType names), use the same naming prefix, only with varying separators. For example, use a slash (/) instead of a dot (.) when requiring the class from the example above.

```
sap.ui.define(["my/app/MyControl"], function(MyControl) {
    ...
});
```

> **i** Note
>
> To avoid conflicts with other frameworks or developments, the `sap` namespace is reserved for SAP. Therefore, any non-SAPUI5 content, such as application code or custom controls, must **not** use namespaces that start with the `sap` prefix.

Example for Defining a Class

Full example of a class definition, including JSDoc

```
sap.ui.define(["jQuery.sap.global", "sap/ui/base/Object", "sap/ui/model/json/
JSONModel"], function (jQuery, BaseObject, JSONModel) {

    // declare and document the constructor function
```

```
    /**
     * Some short sentence that summarizes the functionality of the class.
     *
     * A more detailed explanation of the class that might consist of multiple
sentences
     * and paragraphs. It is <i>possible</i> to use <code>XHTML</code>
<b>markup</b>
     * but this should be used only rarely, as it makes the doclet harder to read
     * in the JS editor.
     *
     * Paragraphs that are separated by empty lines will be formatted as separate
     * paragraphs in the final JSDOC documentation. This makes the addition of
     * <p> or <br/> tags unnecessary.
     *
     * It is possible to reference members of this class (like {@link
#ownMethod})
     * or even of other classes (like {@link sap.ui.Object#destroy}). But be
careful:
     * in contrast to JavaDoc, the signature (parameters) of a method must not be
     * included with the @link tag, only the name of the method (as !JavaScript
does not support
     * method overloading).
     *
     * @class (mandatory) Marks the function as a constructor (defining a class).
     * @param {string} sId Documentation of constructor parameters.
     * @param {object} [mProperties=null] For optional parameters, the name is
enclosed
     *                  in square brackets.
     *                  A default value can be appended with a '='.
     * @param {string} [mProperties.text] Even members of a configuration
parameter
     *                 can be configured.
     * @see (optional, multiple) Fully qualified HTTP links to external
documentation
     *                 are also possible.
     *
     * @public|@private - (optional) Declares the class as public or private
(default).
     * @author (optional, multiple) Author is referenced by user Id, not by name.
     *                              Multiple authors are possible, order is
     *                              significant (first named author is the
default
     *                              contact).
     * @since (optional) When the class/function has been introduced.
     * @extends sap.ui.base.Object Documents the inheritance relationship.
     * @name foo.bar.MyClass (Mandatory when defining a class with extend).
     */
    var Foo = BaseObject.extend("foo.bar.MyClass", /** @lends foo.bar.MyClass */
{

      constructor: function(sId, mProperties) {

        // init and document members here
        /**
         * The ID of a MyClass.
         *
         * @private
         */
        this.mId = sId || Utils.createGUID();
      },

      // now add further methods to that prototype
      /**
       * Again a summary in one sentence.
       *
       * More details can be documented, when the method is more complex.
       * @param {string} sMethod The same mechanism as above can be used to
```

```
         *                            document the parameters.
         * @param {object} [oListener] An optional parameter. If empty, the
         *                             <code>window</code> is used instead.
         * @experimental Since 1.24 Behavior might change.
         * @public
         */
        ownMethod: function(sMethod, oListener) {

          // ... impl
        },

        /**
         * A private method.
         *
         * Every member with a doc comment is included in the public JSDOC.
         * So we explicitly declare this as a private member:
         *
         * Additionally, using an underscore prefix prevents this method
         * from being added to the public facade.
         *
         * @private
         */
        _myVeryPrivateMethod: function() {
        }

    });

    // return the module value, in this example a class
    return Foo;
});
```

Virtual Methods

```
/**
 * A 'virtual' method, that doesn't exist in this class but should be declared
 * in subclasses.
 *
 * It is even possible to document things that aren't there. Only useful use
 * case is the documentation of abstract methods.
 *
 * @name foo.bar.MyClass.prototype.abstractMethod
 * @function
 * @protected
 */
```

SAPUI5 Control Development Guidelines

Content developers developing SAPUI5 controls should follow the guidelines outlined below with regard to APIs, behavior, and themes/CSS.

General Remarks

- Keep things simple! Keep the number of entities created for a new control minimal.
- Reuse is good, but carefully compare how many features of the reused control are needed, and how big the impact on performance would be. For example, if a control needs a clickable area, you can simply implement `onclick` and check where the click came from - this has zero impact on performance. Only if you need more features should you think about instantiating and aggregating. For example, you could use a `Button` control and use its `press` event, but this would cost performance.

API

For APIs, the following guidelines apply:

- Get the API right the first time, you will not be able to change it later (compatibility).
- Control names start with an uppercase letter and use CamelCase for concatenated words.
- Property, event, aggregation, association, method, and parameter names start with a lowercase letter and also use camelCase.
- Do **not** use Hungarian notation for API parameters, as their type is documented in JSDoc.
- Provide a reasonable default value for properties. Consider the most frequent use case.
- Let block elements autofill the available width instead of explicitly setting "100%" as the default width.
- `editable` and `enabled` are two different properties. "Not enabled" controls are **not** in the focus tab chain.
- Check similar controls for consistent naming and modeling of public APIs.
  Controls for text input have a `value` property. Container controls with one generic area for child controls have a 1..n `content` aggregation. When the child controls are not generic, but have specific semantics, arrangement, or type, the name should be chosen accordingly ("items", "buttons",...).
- When there is one most important aggregation, it should be marked as the default aggregation; this facilitates the use in XMLViews.
- Properties, associations, and aggregations should be preferred over API methods due to data binding support and easier usage in XMLViews.
- Make sure not to break use in XMLViews; for example, types like `object` and `any` may not be used for mandatory properties.
- Be careful about initial dependencies. The `Input` control, for example, should not always load the table library just because some inputs may show a value help table after certain user interaction

Behavior

For behavior-related development, the following guidelines apply:

- Do **not** use hardcoded IDs. When creating internal subcontrols, their ID should be prefixed with `this.getId() + "-"`.
- Make sure not to break data binding.
- Do **not** make assumptions about how your control is used.
- Do **not** use `oEvent.preventDefault()` or `oEvent.stopPropagation()` without a good reason and clear documentation why it is required.
- Use the SAPUI5 event handling methods when available instead of `jQuery.bind()/.on()`. When you use `jQuery.bind()` or `jQuery.on()`, always unbind them again, for example in `onBeforeRendering()` and in `exit()` and rebind after rendering.
- Use CSS3 for animations and fall back to no animation for legacy browsers; there are only a few exceptions where the animation is important.
- Keep in mind that a control can be used multiple times in a page.
- Provide immediate feedback for user interaction.
- If an action takes a longer period of time, visualize this, for example by using a `BusyIndicator`.
- When you create HTML markup for a control outside a renderer, for example, by writing to the `innerHTML` property of a DOM element, or by calling `jQuery.html()` or similar helpers, make sure to escape any unchecked data first with the function provided by `sap/base/security/encodeXML`. This is mandatory to prevent cross-site scripting issues. For more information, see Cross-Site Scripting [page 1475].

Renderer

With regard to the renderer, the following guidelines apply:

- Produce clean, semantic HTML5, as compact as reasonably possible.
- Each control instance must have exactly one root HTML element and can have any HTML element structure below that.
- Unknown strings, such as values coming from string properties, need to be escaped before writing to HTML; this avoids security risks via XSS attacks.
- Use `RenderManager.writeEscaped(…)`, or the function provided by module `sap/base/security/encodeXML`.
- Container controls such as `Panel` or `Page`, as opposed to layout controls with a generic "content" aggregation, should render the children directly next to each other with no additional HTML or layout applied.
- Use the Icon pool for images.
- Provide a sufficiently large touch area for interaction on touch devices (usually 3rem/48px).
- When internal HTML elements of the control below the root element need an ID, construct the ID as follows: `<control ID> + "-" + <someSuffix>`.
- The HTML should adhere to the basic XHTML rules; close all tags, enclose attribute values in quotes and do **not** use empty attributes without value.
- Avoid <table>-based layouts when there is no logical table. If a table is used for layout, try to use "display:table" or even "table-layout:fixed" tables.

- `RenderManager.writeControlData()` must be called in the root HTML element of the control to make events work.
- `RenderManager.writeClasses()` must be called in the root HTML element of a control; otherwise `addStyleClass` does not work. this does not need to be used in subelements.

Control Development Guidelines: Theming/CSS

For themes and CSS for control development in SAPUI5, the following guidelines apply.

General Guidelines

- Write semicolons, even where optional.
- Use `rem` for dimensions; use `px` only for dimensions that do not depend on the font size.
- The root element of a control should come without outer margins; add any required padding inside. Root margins are owned by the parent control.
- Do not hard-code any colors, use LESS parameters and color calculations instead; this is also recommended for other significant theme aspects such as fonts and background images.
- Use other LESS features moderately. The more LESS processing happens, the less clear it is where the runtime CSS originates from.
- Do **not** style HTML elements directly; all selectors must include a SAPUI5-specific CSS class to avoid affecting non-owned HTML.
- Avoid the star selector (such as: `* { color: black;}`) in CSS, in particular without a "direct child" selector (">") in front of it (for performance reasons).
- Only use inline CSS for control-instance specific style, for example the button width.
- Do **not** use `!important` as it makes custom adaptations more difficult; use more specific selectors instead.
  There are rare justified exceptions, but they need to be documented.
- Put browser-prefixed properties before the un-prefixed variant.
- When the visuals of certain controls are different depending on the context/container where they are used, use CSS cascades along with marker CSS classes in the parent control:
  - The area/container should write a certain marker CSS class to the HTML and document this CSS class in its JSDoc.
  - The documentation should mention the purpose and contract/meaning of this class, for example, that it is meant to modify the appearance of children in a way that better fits table cells, toolbars, or headers.
  - This CSS class may not have any CSS styles attached. It is a pure marker.
  - This CSS class has the suffix `-CTX` (e.g. `sapUiTable-CTX` or `sapUiBorderless-CTX`) to make it distinguishable from "normal" CSS class names.
  - Controls that want to modify their appearance in such an area use the marker class in a cascade: `.sapUiTable-CTX .sapUiInput { border: none; }`

Naming

The following naming guidelines apply:

- All CSS classes must begin with the `sapUi` prefix (or `sapM` in the sap.m library). Exception: some global CSS classes used in the core start with "sap-".
- For each control there must be one unique control-specific prefix for CSS classes.
  For example, `sapUiBtn` for a `Button` control, or `sapMITB` for an `IconTabBar` in the `sap.m` library. This class must be written to the HTML root element of the control. All CSS classes within the HTML of this control must append a suffix to this class name, for example: `sapUiBtnInner` or `sapMITBHeader`.

Images

Themes (including "base") should only refer to existing images inside that theme. Images will be loaded relative to the theme where they are referenced (see LESS option `"relativeUrls"`)

If an image URL defined in `base` stays active in another theme `'mytheme'`, derived from `base`, LESS will calculate a relative URL that points from the `mytheme/library.css` to the `base/library.css`.

Similar path calculation is necessary when the URL is defined in another library (e.g. from `sap/m/themes/mytheme/library.css` to `sap/ui/core/themes/base/image.png`).

These URL transformations assume a single repository for all sources. When resources for different themes or libraries are located in different libraries, such relative URLs might not work.

To override an image within the base theme an additional rule has to be added to the individual theme referencing the image. Otherwise the base image will be loaded.

LESS Theme Parameters

For LESS theme parameters, the following guidelines apply:

- Use the correct theme parameter - do not find by color value, but by semantics. In general, let the visual designers give the correct parameter to use.
  If finding a color for a text, do not use any border or background color parameter. Start with `@sapUiText` and try to find something more specific such as `@sapUiHeaderText`. Use parameters such as `@sapUiTextInverted` for bright-on-dark scenarios.
  If no suitable parameter exists, derive the color by calculation from a suitable parameter.
- Do **not** add parameters to the public API (using annotations) without sufficient clarification with designers and product owners.
- If you create your own local parameters, you must ensure that the names you define are unique by using name(space) prefixes.
  For **control-specific** parameters in `*.less` files, use a combination of the library name and the `*.less` file name for the prefix. Start with an underscore. Separate each part of the library namespace and the file name from each other using underscores as well.

> → Tip
>
> For example, you can define the following prefix:
>
> **Library**: `sap.ui.core`
>
> **File**: `sap/ui/core/themes/base/MyControl.less`
>
> **Prefix**: `@_sap_ui_core_MyControl_`

For **library-specific** parameters in `library.source.less` files, use the library name for the prefix. Start with an underscore. Separate each part of the library namespace from each other using underscores.

> → Tip
>
> For example, you can define the following prefix:
>
> **Library**: `sap.ui.core`
>
> **File**: `sap/ui/core/themes/base/library.source.less`
>
> **Prefix**: `@_sap_ui_core_`

> ⚠ Caution
>
> Local parameters themselves must **not** contain underscores. For example, do not write `@_sap_ui_core_MyControl_Some_Color`, but write `@_sap_ui_core_MyControl_`**`SomeColor`** instead.

- When defining URLs as parameters use the proper `url()` format: `@sapUiMyUrl: url(./path/to/img.png)`
  - Do **not** use escaped strings (~): `@sapUiMyUrl: ~"path/to/img.png"`
  - Do **not** use absolute URLs: `@sapUiMyUrl: url(/absolute/path/to/img.png)`

Product Standards and Acceptance Criteria

To be of high quality and usable in mission-critical business software, SAPUI5 needs to fulfill specific product standards and acceptance criteria. While these are not directly related to code conventions, the most important standards and criteria are mentioned here, because new code needs to fulfill these requirements.

General product standards and acceptance criteria:

- Browser support, see Browser and Platform Support [page 20]
- Security (such as output encoding to prevent XSS attacks)
- Performance needs to be in focus
- Automated tests, such as QUnit
- Proper API documentation
- Translation: All texts that are visible on the UI must be enabled for translation.
  Do **not** provide the translations, but only provide a raw english version in the `messagebundle.properties` file.
  Make sure that you annotate the strings properly for translation. .

- Adhere to the compatibility rules, see Compatibility Rules [page 17].
- Make sure that any Open Source libraries (or parts of them) that you use are officially approved before you add them to SAPUI5. Do **not** add any code you "found" somewhere.

Additional product standard and acceptance criteria for controls:

- Screenreader support (ARIA)
- Keyboard navigation support
- Required themes; what themes are required depends on the library, but always includes the High Contrast theme for accessibility
- Right-to-left support
- Example page in the *Samples* in the Demo Kit

File Names and Encoding

Some target platforms of SAPUI5 impose technical restrictions on the naming or structure of resources (files).For this, rules for file names and encoding have been introduced.

When developing content for SAPUI5, adhere to the following rules:

- Folder names must **not** contain spaces.
- To avoid issues with the SAPUI5 module loading and with URL handling in general, resource names should **not** contain spaces.
- Single folder names must **not** be longer than 40 characters.
- Two resource names must **not** only differ in case.
- Avoid non-ASCII characters in resource names.

JSDoc Guidelines

Provides an overview of guidelines for creating JSDoc documentation.

To document JavaScript coding, you can add documentation comments to the code. Based on these comments, the descriptions of the SAPUI5 entities are generated and shown in the *API Reference* of the Demo Kit. SAPUI5 uses the JSDoc3 toolkit, which resembles JavaDoc, to generate the descriptions. For an explanation of the available tags, see https://jsdoc.app .

Basics of JSDoc

Here are some general principles for writing comments:

- Document the constructor with `@class`, `@author`, `@since`, and so on.
- For subclasses, document the inheritance by using an `@extends` tag in their constructor doclet.
- Document at least public and protected methods with JSDoc, mark them as `@public` or `@protected`. If you also document private methods with JSDoc, mark them as `@private`. This is currently the default in SAPUI5, but not in JSDoc, so it is safer to explicitly specify this. `@protected` is not clearly defined for a

JavaScript environment. In SAPUI5, it denotes a method that is not meant to be used by applications. It might be used outside the relevant class or subclasses, but only in closely related classes.
To explicitly specify which modules are allowed to use a class or function, mark the latter as `@private` followed by `@ui5-restricted <modulenames>`, with a comma-separated list of the modules that have access to this class or function.

> **i Note**
>
> To ensure that external JSDoc generators can also produce proper documentation, `@private` must be used first followed by `@ui5-restricted`. `@ui5-restricted` overrules `@private`, if it can be interpreted by the generator.

- Document method parameters with type (in curly braces) and parameter name (in square brackets if optional).
- Use `@namespace` for static helper classes that only provide static methods.

For an example of how to create a class, see Example for Defining a Class [page 2162].

Descriptions

A documentation comment should provide the following content:

- Summary sentence at the beginning; the summary is reused, for example, for tooltips and in summaries in the *API Reference*
- Background information required to understand the object
- Special considerations that apply
- Detailed description with additional information that does not repeat the self-explanatory API name or summary

> **i Note**
>
> Avoid implementation details and dependencies unless they are important for usage.

Dos and Don'ts

- To avoid line wrapping, make sure that each line of the description has a similar length as the code. In the *API Reference*, the line breaks in a description are ignored, and it appears as a continuous text.
- Use a period at the end of each summary sentence. The punctuation is required for JSDoc to identify the first sentence.
- Don't use a period inside a summary sentence. For example, don't use "e.g.", but write "for example" instead. Otherwise the summary sentence will be cut off.

> **i Note**
>
> You can create links to external sources. The source should comply with standard legal requirements. The required icons are added to the link as described in the Demo Kit under *Terms of Use* > *Disclaimer*. For more information about creating links, see the explanations below (@see and {@link}).

Recommendations for Writing Descriptions

- Don't use exclamation marks.
- Make sure you spell acronyms correctly, for example, ID, JSON, URL.
- In the summary sentence, omit repetitive clauses like "This class" or "This method".
- For actions, start directly with an appropriate verb in the third person: Adds, allocates, constructs, converts, deallocates, destroys, gets, provides, reads, removes, represents, returns, sets, saves, and so on. For methods, use the following verbs:

Type	Verb
Constructor	Constructs
Boolean	Indicates (whether)
Getter	Gets
Setter	Sets
Other	Adds/Removes/Creates/Releases/Other verb that applies

- For objects, use a noun phrase.
  Example: Base class for navigation

Inline and HTML Tags

You can use inline and HTML tags in your comments.

Inline tags can be placed anywhere in the comments. Inline tags are denoted by curly brackets and have the following syntax: {@tagname comment}.

HTML tags are used to format documentation comments. HTML tags have the standard HTML syntax: <tag>...</tag>.

The table provides an overview of the most common inline and HTML tags.

Table 98: Inline and HTML Tags

Tag	Use	Example	How to Use / Details	Type of Tag
{@link}	Links within API Reference	`{@link sap.ui.generic.app.navigation.service.NavError Error}` `{@link sap.ui.comp.smarttable.SmartTable#event:beforeRebindTable}`	To replace the path with a display text, use it like this: {@link <path> space <display text>}. You can also use `#myMethod` for links within a class or control to individual methods, for example. The leading hash will then be removed automatically. For other links, use the required syntax, for example, `#event:name`.	Inline
Empty line	Creates a paragraph		Using <p> is not necessary, since empty lines are used to define paragraphs.	HTML
<code>...</code>	Technical entities (optional)	the <code>Button</code> control		
<pre>...</pre>	Code samples			
   	Unordered lists			
   	Ordered lists			
... or ...	Bold font			
...	Italics			
	Non-breaking space			

Block Tags

You can also use block tags in your comments.

Block tags can only be placed in the tag section below the comment. They are separated from the comment by an empty line (recommended, but not a technical requirement). Block tags have the following syntax: @tagname comment.

The table provides an overview of the most common block tags.

Table 99: Block Tags

Tag	Use	Example	How to Use / Details

@param	Adds parameters	`/**`
		` * ...`
		` * @param {string} statement The SQL statement to be prepared`
		` * ...`
		` */`
			Begin description with a capital letter.

@returns	Adds return values	`@returns {type1|type2|...} Description`	Begin description with a capital letter.
@throws	Adds the description of an exception if an error occurs	`@throws {type} Description`	Begin description with a capital letter.
@author	Adds the name of the developer responsible for the code	`@author Max Mustermann`	This is an optional tag that is not displayed in JSDoc.
@version	Names the version for an entity	`@version 14.1.2`	If you need to use the version tag, use ${version} so you don't have to update this manually for each new version.

Tag	Use	Example	How to Use / Details

@see	Adds information (for example, link to documentation or the SAP Fiori Design Guidelines) in the header section of the *API Reference*	`@see path`
		`@see free text`
		`@see {@link topic:bed8274140d04fc0b9bcb2db42d8bac2 Smart Table}`
		`@see {@link fiori:/flexible-column-layout/ Flexible Column Layout}`
			@see {@link topic:loio <semantic control name>} provides a link to the documentation (developer guide).
			If there are several @see tags with documentation links, only the first one is shown in the header. The other ones are displayed under *Documentation Links* in the *Overview* section.
			For more generic topics that are not directly related to a class or control, use inline links.

@since	Adds the version in which an entity was first introduced	`@since 1.30`	Be as specific as possible (without mentioning patch levels for new development), since this information is useful even for internal purposes. For example, mention 1.27, even though this is not an external release.
@deprecated	Adds the version in which an entity was deprecated	`@deprecated As of version 1.28, replaced by {@link class name}`	Be as specific as possible (without mentioning patch levels), since this information is useful even for internal purposes. For example, mention 1.27, even though this is not an external release. Provide information about what replaces the deprecated entity.
@experimental	Classifies an entity that is not ready for production use yet, but available for testing purposes	`@experimental As of version 1.56.0`	

Tag	Use	Example	How to Use / Details

@example	Inserts a code sample after the comment	`/**`
		` * ...`
		` * @example`
		` * var id = myjob.schedules.add({`
		` * description: "Added at runtime, run every 10 minutes",`
		` * xscron: "* * * * * *\/10 0",`
		` * parameter: {`
		` * a: "c"`
			The code sample is inserted automatically with <pre>. It is always inserted right after the comment.
			
			To insert an example somewhere else, for example, in the middle of a comment, use <pre>.
			
			You can add a header for the example by using <caption>.

Tips for Using Block Tags

- The order of the block tags is not mandatory from a technical perspective, but recommended to ensure consistency.
  For parameters, however, a fixed order is mandatory.
- There are more tags available, such as `@class` or `@name`.

Links to API Documentation

To refer to another entity within the *API Reference*, you can use `{@link}` in combination with the reference types shown in the table below.

Table 100: Reference Types within API Reference

Type of Reference	Description	Example	Comment
<full.path.ClassName>	Refers to a class, interface, enumeration, or namespace	`sap.ui.comp.smarttable.SmartTable`	
full.path.ClassName**#**method	Refers to an instance method of a class	`sap.ui.comp.smarttable.SmartTable#getHeader`	`.prototype.` and # are interchangeable
full.path.ClassName**.prototype.**method	Refers to an instance method of a class		
full.path.ClassName**#event:**name	Refers to an event fired by an instance of a class	`sap.ui.comp.smarttable.SmartTable#event:beforeRebindTable`	

Type of Reference	Description	Example	Comment
full.path.ClassName.method	Refers to a static method (or any other static property)		
`#method`	Refers to an instance method **within** a class	`#getHeader`	You can use this type of reference within an API that you are documenting, for example, within the `SmartTable` control documentation if you want to link to a method that belongs to the control itself.
`#.method`	Refers to a static method **within** a class		
`#event:name`	Refers to an event **within** a class		

Common Pitfalls in JSDoc

The use of the JSDoc toolkit has some pitfalls. By following the guidelines outlined below, these issues can be avoided.

Multiple Documentation Comments Before a Symbol

In case of multiple documentation comments before a JavaScript symbol, JSDoc only associates the last comment with the symbol. Therefore, multiple documentation comments before a symbol must be avoided. The comment and code sequence below results in an unwanted documentation for the `adjustFilters` symbol. To avoid this, move the TODOs, either within the function or before the documentation comment.

```
/**
 * Maps the UI filter objects to the internal Filter object.
 *
 * @param filteredColumns The current UI filters that will be mapped
 * to the internal format.
 * @returns The newly formatted format.
 * @private
 */
/** TODO: Call getOperator when custom filters are supported */
/** TODO: getValue2 to fix later when we have ranges with BETWEEN operator */
sap.ui.table.internal.BehaviorManager.prototype.adjustFilters =
function(filteredColumns) {
  // ...
}
```

Special Case: Section Separators

JSDoc interprets any multiline comment starting with a double asterisks (`/**`) as a documentation comment for the JavaScript symbol that follows the documentation comment. However, some developers use "decorative" documentation comments to separate sections in their JavaScript modules, and using a multiline comment consisting of asterisks is just one example for such decorative comments:

```
// Update aggregation
  this.insertSection(oSection, iTargetIndex, true);
  /****Update index/id mapping table*******************************/
  aSections = this.getSections();
  for (var i = 0; i < aSections.length; i++) {
    this.aIdMappings[aSections[i].getId()] = i;
  }
```

For JSDoc, however, this looks like a documentation comment for the `aSections` variable. And if this is the last documentation comment for `aSections`, it appears in the generated JSDoc for the enclosing module or class. The only way to avoid such silly mistakes in documentation, is to avoid the pairing of multiline documentation comments and symbols to be documented. So do **not** use stars/asterisks for a separating banner comment. You can use other characters, for example `/* ==== */` or `/* ----- */`, or at least avoid the double asterisks at the beginning. A very unnoticeable replacement would be to use a double quote `/*"**********/` as only the last documentation comment before a symbol is used. Another very good option to avoid misinterpretation of banner comments is to document the symbol that follows.

HTML Tags in Documentation Comments

JSDoc explicitly allows HTML tags in documentation comments. This allows, as in JavaDoc, to structure longer or more complex documentation comments with the help of some HTML markup. Typical use cases are ordered or unordered lists or semantic tags like `<code>` or `<em>`. But be aware that the support for HTML tags for formatting purposes unfortunately implies that JSDoc must not escape them. So, if you want to include an HTML literally, for example, to explain what kind of HTML is produced by a control, the HTML tag must be HTML-escaped in the source code. Otherwise, it will mark up the final JSDoc output, which will most likely break.

Example

In the example, note the escaped `<TR>` in the first and the third line of the documentation comment:

```
/**
 * This function return the rowNumber given a row&lt;TR>.
 *
 * @private
 * @param {DomRow} &lt;TR> dom object.
 * @returns {int} the row number maintained in the data object.
 */
sap.ui.table.Table.prototype.getRowNumber = function(oDomRow) {
```

Tools

For the tools used for SAPUI5 content development, guidelines and recommendations have been introduced.

The following guidelines and recommendations apply:

- To make issues with mixed tabs/spaces and windows-style linebreaks visible immediately, we recommend to configure your JavaScript editor to display whitespace and linebreak characters.
- We also recommend to configure the code formatter of your code editor accordingly.
- Do **not** use the auto formatter to format entire files. The auto formatter handles many lines and makes it more difficult to find out who actually wrote the code.

ESLint

SAPUI5 contains a mandatory ruleset for ESLint. For a complete list of rules and settings, see ESLint Code Checks [page 2179]. For the ESLint configuration file, see ESLint Configuration File [page 2184].

ESLint Code Checks

SAPUI5 uses ESLint to check JavaScript sources.

The following tables show the ESLint rules that should be enabled for the SAPUI5 projects.

Rule Sets

Table 101: Possible Errors

Rule	ESLint default	Core	Comment
`no-cond-assign`	error	error	
`no-console`	error	error	
`no-constant-condition`	error	error	
`no-comma-dangle`	error	**error/warning**	can be set to warning if lib only supports IE9
`no-control-regex`	error	error	
`no-debugger`	error	error	
`no-dupe-keys`	error	error	
`no-empty`	error	error	
`no-empty-class`	error	error	

Rule	ESLint default	Core	Comment
no-ex-assign	error	error	
no-extra-boolean-cast	error	**warning**	
no-extra-parens	off	off	
no-extra-semi	error	error	
no-func-assign	error	error	
no-inner-declarations	error	error	
no-invalid-regexp	error	error	
no-negated-in-lhs	error	error	
no-obj-calls	error	error	
no-regex-spaces	error	error	
no-sparse-arrays	error	error	
no-unreachable	error	error	
use-isnan	error	error	
valid-jsdoc	off	**warning**	requireReturn = false
valid-typeof	error	error	

Table 102: Best Practices

Rule	ESLint default	Core	Comment
block-scoped-var	off	**error**	currently only warning because of wrong behaviour in switch statement
complexity	off	off	
consistent-return	error	**warning**	
curly	error	error	
default-case	off	**warning**	
dot-notation	error	**off**	
eqeqeq	error	**warning**	smart
guard-for-in	off	**error**	
no-alert	error	error	
no-caller	error	error	
no-div-regex	off	**error**	
no-else-return	off	off	
no-empty-label	error	error	

Rule	ESLint default	Core	Comment
no-eq-null	off	off	
no-eval	error	error	
no-extend-native	error	error	
no-fallthrough	error	error	
no-floating-decimal	off	**error**	
no-implied-eval	error	error	
no-labels	error	error	
no-iterator	error	error	
no-lone-blocks	error	error	
no-loop-func	error	error	
no-multi-str	error	error	
no-native-reassign	error	error	
no-new	error	error	
no-new-func	error	error	
no-new-wrappers	error	**warning**	
no-octal	error	error	
no-octal-escape	error	error	
no-proto	error	error	
no-redeclare	error	**warning**	
no-return-assign	error	error	
no-script-url	error	error	
no-self-compare	off	**error**	
no-sequences	error	error	
no-unused-expressions	error	**warning**	
no-warning-comments	off	**warning**	
no-with	error	error	
radix	off	**error**	
wrap-life	off	**error**	any
yoda	error	error	

Table 103: Strict Mode

Rule	ESLint defaul	Core
global-strict	off	**error**

Rule	ESLint defaul	Core
no-extra-strict	error	error
strict	error	**warning**

Table 104: Variables

Rule	ESLint default	Core	Comment
no-catch-shadow	error	error	
no-delete-var	error	error	
no-label-var	error	error	
no-shadow	error	error	
no-shadow-restricted-names	error	error	
no-undef	error	error	
no-undefined	off	off	
no-undef-init	error	error	
no-unused-vars	error	error	vars = all, args = none
no-use-before-define	error	**warning**	

Table 105: Node.js

Rule	ESLint default	Core
handle-callback-err	off	off
no-mixed-requires	off	off
no-new-require	off	off
no-path-concat	off	off
no-process-exit	off	off
no-restricted-modules	off	off
no-sync	off	off

Table 106: Stylistic

Rule	ESLint default	Core	Comment
brace-style	off	**error**	singleLine = false
camelcase	error	warning	
consistent-this	off	**error**	that
eol-last	error	**off**	
func-names	off	off	
func-style	off	off	

Rule	ESLint default	Core	Comment
new-cap	error	**warning**	
new-parens	error	error	
no-nested-ternary	off	**error**	
no-array-constructor	error	error	
no-lonely-if	off	**error**	
no-new-object	error	error	
no-spaced-func	error	error	
no-space-before-semi	error	error	
no-ternary	off	off	
no-trailing-spaces	error	**off**	error, but too many places to change
no-underscore-dangle	error	**off**	
no-wrap-func	error	error	
no-mixed-spaces-and-tabs	error	error	smart
quotes	off	off	
quote-props	off	off	
semi	error	error	
sort-vars	off	off	
space-after-keywords	off	**error**	always
space-in-brackets	off	off	
space-infix-ops	error	error	
space-return-throw-case	error	error	
space-unary-word-ops	off	**error**	
max-nested-callbacks	off	**warning**	3
one-var	off	off	
wrap-regex	off	off	

Table 107: Legacy

Rule	ESLint default	SAPUI5 Core
`max-depth`	off	off
`max-len`	off	off
`max-params`	off	off
`max-statements`	off	off
`no-bitwise`	off	off
`no-plus`	off	off

For more information about the rules, see the rules documentation provided on http://eslint.org .

ESLint Configuration File

Content of the ESLint configuration file

```
{
    "env": {
        "browser": true
    },
    "globals": {
        "sap": true,
        "jQuery": true
    },
    "rules": {
        "block-scoped-var": 1,
        "brace-style": [2, "1tbs", { "allowSingleLine": true }],
        "consistent-this": 2,
        "global-strict": 2,
        "no-div-regex": 2,
        "no-floating-decimal": 2,
        "no-self-compare": 2,
        "no-mixed-spaces-and-tabs": [2, true],
        "no-nested-ternary": 2,
        "no-unused-vars": [2, {"vars":"all", "args":"none"}],
        "radix": 2,
        "space-after-keywords": [2, "always"],
        "space-unary-word-ops": 2,
        "wrap-iife": [2, "any"],
        "camelcase": 1,
        "consistent-return": 1,
        "max-nested-callbacks": [1, 3],
        "new-cap": 1,
        "no-extra-boolean-cast": 1,
        "no-lonely-if": 1,
        "no-new": 1,
        "no-new-wrappers": 1,
        "no-redeclare": 1,
        "no-unused-expressions": 1,
        "no-use-before-define": [1, "nofunc"],
        "no-warning-comments": 1,
        "strict": 1,
        "valid-jsdoc": [1, {
            "requireReturn": false
        }],
        "default-case": 1,
        "dot-notation": 0,
```

```
        "eol-last": 0,
        "eqeqeq": 0,
        "no-trailing-spaces": 0,
        "no-underscore-dangle": 0,
        "quotes": 0
    }
}
```

The library.js File

The `library.js` file is a JavaScript file that contains the JavaScript code for all enumeration types provided by the library as well as library-specific initialization code that is independent from the controls in the library.

The file calls the `sap.ui.getCore().initLibrary` method with an object that describes the content of the library (list of contained controls, elements etc.). For more informarion about the object parameter, see sap.ui.getCore().initLibrary..

The library style sheet file (`library.css`) contains all styles relevant for this library. For libraries that have been developed with the SAPUI5 application development tools, this file is also generated automatically during the build.

In a `library.js` file, the call to `sap.ui.getCore().initLibraries()` takes care of creating the namespace object of the library, exports it under its global name and returns the namespace to the caller. In the `library.js` module, you don't need to write types or helpers to the global name, but can use the returned namespace like this:

```
sap.ui.define(function() {

    "use strict";

    /**
     * @alias my.lib
     */
    var oThisLibrary = sap.ui.getCore().initLibrary({…});

    /**
    * An addition to mylib. If you used the @alias tag above, JSDoc will
recognize this as my.lib.ValueColor.
    * @ui5-metamodel The UI5 metamodel restoration logic also can handle this
kind of definition and will create an enumeration type
    *         my/lib/ValueColor.type. The name of the variable
(<code>thisLibrary</code>) is not mandatory, just an example.
     */
    oThisLibrary.ValueColor = {
        Color1: …
    };

    // don't forget to return the value
    return oThisLibrary;

});
```

Enumerations and RegEx Types

We recommend to add all simple types of a library to the `library.js` module. Other modules that need to work with such types can simply include the respective library as a module dependency:

```
// requiring a library
sap.ui.require(["sap/ui/core/library"], function(library) {
    var sAlign = library.HorizontalAlign.Begin;
});

// defining a module with a library dependency
sap.ui.define(["sap/ui/core/library"], function(library) {
    var sAlign = library.HorizontalAlign.Begin;
});
```

ManagedObject Metadata

In the metadata definition of `ManagedObject` subclasses, types for properties, aggregations, associations and event parameters have to be specified with global names as strings.

The default values, however, should be referenced via the correct type value from the `library.js` module because it avoids the usage of globals.

Define the `library.js` as static dependency and use it as a local variable for convenience:

```
sap.ui.define(["sap/ui/core/Control", "./library"], function(Control, library) {
    // shortcut on Enum
    var SizeMode = library.SizeMode;

    var MyControl = Control.extend("my.lib.MyControl", {
        metadata : {
            library : "my.lib",
            properties : {
                sizeMode: {type : "my.lib.SizeMode", group : "Appearance",
defaultValue : SizeMode.Auto}
            }
        }
    });

    MyControl.prototype.setSizeMode = function (sMode) {
        switch(sMode) {
            case SizeMode.Auto: ... break;
            case SizeMode.Full: ... break;
            ...
        }
    };

    return MyControl;
});
```

Creating Control and Class Modules

Modules do not only require and use functionality from other modules, they also expose their own functionality to the outside. In asynchronous module definition (AMD) syntax, there are several ways to expose such functionality.

However, SAPUI5 only supports the "module return value".

If you want to export the "module value" of an AMD module under a global name, you have two options:

- You rely on the SAPUI5 methods that already do the exposure as a side effect, such as:
  - Classes created by the `extend` method
  - Libraries that call `initLibrary()` in their `library.js` module
  - Renderers that are created with `sap.ui.core.Renderer.extend("….")`
- You set the fourth parameter `bExport of sap.ui.define(sModuleName, aDependencies, vFactory, bExport)` to `true`. This will expose the module value under the global name that is derived from the module name.
  The global JavaScript namespace is based on a "slash to dot replacement".

  > **i Note**
  >
  > The resulting namespace might not have the expected result if the AMD module name contains dots!

In control and class modules, you should not use global variables at all. When you derive a custom control from an existing superclass via the the `extend` method, the resulting subclass is returned.

You can store the return value of the extend function in a local variable, make changes to the prototype and then return this variable as the modules return value.

JSDoc for the class should use the `@alias` tag to make sure that the variable is known under the global name in the generated JSDoc.

The `extend` function makes sure that the respective namespace is created:

```
sap.ui.define(["sap/ui/base/Object", "sap/ui/model/json/JSONModel"], function
(BaseObject, JSONModel) {

    var Foo = BaseObject.extend("foo.bar.MyClass", /** @lends foo.bar.MyClass */
{

      constructor: function(sId, mProperties) {
        this.mId = sId;
      }
    });

    Foo.prototype.ownMethod = function (a) {
        return a * 2;
    };

    // return the module value, in this example a class
    return Foo;
});
```

Related Information

Example for Defining a Class [page 2162]

Defining the Control Metadata

Control metadata consists of properties, events, as well as aggregations and associations.

The control metadata is defined as follows:

- Properties
  A property has a name and an associated data type. It has a well-defined default value expressed as a literal of that data type. Properties are accessible to application code via the element's API as getters and setters, but are also used by a control's renderer in a read-only way. The following list gives an overview of the most important settings for a property:
  - `type`: Data type of the control property; SAPUI5 provides an automatic type validation. Valid types are, for example, `string` (default) for a string property, `int` or `float` for number properties, `int[]`, etc. for arrays and `sap.ui.core.CSSSize` for a custom-defined type.
    For more information, see Defining Control Properties [page 2190].
  - `defaultValue`: Default value that is set if the application does not set a value; if no default value is defined, the property value is `undefined`.

  These control-specific settings are only available when inheriting from a control or one of the base classes `sap.ui.core.Control`, `sap.ui.core.Element`, `sap.ui.base.ManagedObject`, see Object Metadata and Implementation [page 2190]. For a complete list of the possible settings, see API Reference: `sap.ui.base.ManagedObject.extend`.
- Events
  An event has a name as well as any number of parameters. The element's API offers support to manage event subscriptions.
  It is defined by its name only.

  ```
  events: {
     "logout": {}
  }
  ```

  For each event, methods for registering, de-registering and firing the event are created. For the `logout` event, for example, the `attachLogout`, `detachLogout`, `fireLogout` methods are created.
  A control can enable events to be interrupted by the application. A tab control, for example, can enable the application to cancel a `close` event by setting the `allowPreventDefault` property of the event to `true` and checking the return value after firing the event:

  ```
  events: {
     "close": {allowPreventDefault : true}
  }
  ```
- Aggregations and Associations
  An **aggregation** is a special relation between two UI element types. It is used to define the parent-child relationship within the above mentioned tree structure. The parent end of the aggregation has cardinality `0..1`, while the child end may have `0..1` or `0..*`. The element's API offers convenient and consistent methods to deal with aggregations (e.g. to get, set, or remove target elements). Examples are table rows and cells, or the content of a table cell.

An **association** is another type of relation between two UI element types which is independent of the parent-child relationship within the above mentioned tree structure. Directed outgoing associations to a target of cardinality `0..1` are supported. They represent a loose coupling only and are thus implemented by storing the target element instance's ID. The most prominent example is the association between a label and its field.
Aggregations and associations are defined by their name and a configuration object with the following information:

- `type`: The type should be a subclass of the element or the control; the default is `sap.ui.core.control`
- `multiple`: Defines whether it is a 0..1 aggregation or a 0..n aggregation; the default for aggregations is `true` = 0..n, and for associations the default is `false`
- `singularName`: For 0..n aggregations, the aggregation name typically is plural, but certain methods are created where the singular form is required (for example, `addWorksetItem`} for the "worksetItems" aggregation).

If only the type needs to be set, you can just give it as a string instead of the configuration object.
One example:

```
aggregations: {
   "acceptButton" : "sap.m.Button", // if only type is given, no object is
required
   "content" : {singularName: "content"},    // default type is
"sap.ui.core.Control",
                                             // which is appropriate for
generic containers
   "toolbarItems" : {type : "sap.m.Button", multiple : true, singularName :
"toolbarItem"}
                                             // a fully specified aggregation
}
```

Multiple methods are created automatically at runtime, depending on the multiplicity, for example `getWorksetItems`, `insertWorksetItem`, `addWorksetItem`, `removeWorksetItem`, `removeAllWorksetItems`, `indexOfWorksetItem`, `destroyWorksetItems`. These methods have a default implementation which does everything to handle the aggregation properly, but they can be overridden and extended by the control implementation.
If you want to mark one aggregation as default aggregation in order to be able to omit the aggregation tag in XML views, you can do this by setting the `defaultAggregation` property to the name of the aggregation as shown in the following code snippet:

```
aggregations: {
   "content": {singularName: "content"} // default type is
"sap.ui.core.Control", multiple is "true"
},
defaultAggregation: "content"
```

For a brief explanation of the differences between an aggregation and an association, see the *Control Metadata* section under Working with Controls [page 1041]. For a complete list of the possible settings, see API Reference: `sap.ui.base.ManagedObject.extend`.

Object Metadata and Implementation

SAPUI5 supports the extension of plain objects that are not elements or controls.

For these objects, only the following metadata is available:

- `interfaces`: String array that denotes the implemented interfaces (optional)
- `publicMethods`: List of methods that should be part of the public API (optional)
- `abstract`: Flag to mark the type as abstract (optional)
- `final`: Flag to mark the type as final (optional)

> **i** Note
>
> This metadata can also be used when extending controls.

Regarding the implementation, all methods given outside the metadata are attached to the new type. The method name `constructor` is reserved for the constructor function of the new class. Although it is possible from a technical point of view, we recommend **not** to define a constructor for new elements and controls. Your control may otherwise break in some scenarios, such as in combination with list bindings, or may no longer be compatible in later versions of SAPUI5 when the constructor signature is extended.

Defining Control Properties

Control properties are defined as follows:

```
properties: {
   "title" : "string",                          // a simple string property,
default value is undefined
   "buttonText" : {defaultValue: "Search"},    // when no type is given, the
type is string
   "showLogoutButton" : {type : "boolean", defaultValue : true},   // a boolean
property where a default value is given
   "width" : {type : "sap.ui.core.CSSSize", defaultValue : "50px"} // a CSS size
property where a default value is given
}
```

After the property is defined, the control automatically has the `setShowLogoutButton` and `getShowLogoutButton` methods for storing data. It is possible to assign custom definitions to these methods by overriding them and calling the generic property methods `setProperty` and `getProperty`:

```
MyControl.prototype.setShowLogoutButton = function(show) {
    // …here anything in addition to the default handling can be done…
    // then do the default handling:
    this.setProperty("showLogoutButton", show); // this validates and stores the
new value
    return this; // return "this" to allow method chaining
};
```

Allowed Property Types

Table 108: Built-in Types

Type	Description
`boolean`	Can either be `true` or `false`. Properties of that type should not be set to `undefined` or `null`. The default value is `false`.
`int`	JavaScript primitive values of type `number` and that don't have a fractional part. To keep the implementation efficient, the constraint is not enforced. Declaring a property as type `int` is rather for information reasons. The corresponding object expects any given value to be an integer value. The default value of the type is the number `0`.
`float`	JavaScript primitive values of type `number` that can have a fractional part. It is named `float` instead of `number` to differentiate it from type `int` . The default value is the number `0`.
`string`	JavaScript string literal (`typeof value === ,string'`) or a `String` object (`value instanceof String`). The default value is an empty string.
`object`	Plain JavaScript object (an object whose constructor is `Object`). Most of the time, other objects are accepted as well, but deserializers (e.g. for XML views) will try to convert the object from or to a JSON string. The default value is `null`. Don't mix this type up with the `any` type! (Sorry, we maybe should have named it „`serializable`“ or „`JSON`“ or something like that, to make this more clear...).
`any`	Any valid Javascript value (including primitives, objects, functions, regular expressions, and native objects). The support in serialized formats is quite limited. Valid JSON strings will be deserialized to an object. The default value is `null`.
`function`	Can be any JavaScript function. Note that properties of this type currently can't be used in serialized formats like XMLViews. If an XMLView needs to set a value for a control property of type function, it has to set the value in its controller code (e.g. in the `onInit` hook).

Table 109: Derived Types

Category	Description
regular expression (RegExp)	Derived from the built-in type `string`. Restricted subtypes that limit their valid values to strings that match a given regular expression. `RegExp` types can only be defined by calling the `DataType.createType()` method. Example: `var fooType = DataType.createType('foo', { isValid : function(vValue) { return /^(foo(bar)?)$/.test(vValue); } }, DataType.getType('string'));` If `mSettings` contains an implementation for `isValid`, then the validity check of the newly created type will first execute the check of the base type and then call the given `isValid` function. For more information, see the API Reference.

Category	Description

enumeration (enum)	Derived from the built-in type `string`.
	
	Restricted subtypes can be derived that limit their valid values to a fixed set of values (enumeration). An `enum` type is defined through an object literal whose keys represent the allowed values.
	
	Restrictions:
	
	• The value for each key must be a string literal, equal to the key itself.
	• Renamings or aliases are not supported and only keys and values of type `string` are supported.
	
	This was an early design decision in SAPUI5 and framework code relies on it. That code might fail for enumerations that don't obey these restrictions.
	
	To reference an `enum` type in a property definition, its global name must be used (like `sap.m.ValueColor` in the example below).
	
	.
	
	Example:
	
	`/**`
	`* Enumeration of possible value color settings.`
	`*`
	`* @enum {string}`
	`* @public`
	`*/`
	`sap.m.ValueColor = {`
	`/**`
	`* Neutral value color.`
	`* @public`
	`*/`
	`Neutral : "Neutral",`
	`[…]`
	`};`

Category	Description
array	You don't have to define array types before using an array. From each valid type above, an array type with one or more dimensions can be derived by simply appending a pair of square brackets (`[]`) for each dimension, for example: • `int[]` is a one-dimensional array of integers • `int[][]` is a two-dimensional array of integers (or more precisely an array of integer arrays) The type of an element in an array is called the component type (`int` in the first example, `int[]` in the second). The `DataType` object for an array type has a method `getComponentType` to retrieve the component `type`. For non-array types, this method returns `null`. The default value for any array type is the empty array.

Runtime Behavior and API of Property Types

At runtime, each type is represented as instance of the class `DataType` (`sap/ui/base/DataType.js`). A type can be looked up by its name by calling `DataType.getType(name)`. The framework treats all values for the above types as if they would be atomic. Changes inside a value (e.g. changing the property of a value of type `object` or adding, removing, or changing members of an array) are not detected by the framework and might lead to unexpected results. Instead, any change should be applied by setting a new (or modified) value with a call like `setText`, `setCaption`, `setColor`.

Related Information

API Reference: `sap.ui.base.DataType`

Adding Method Implementations

After defining the metadata of a control, you add the method implementation to the control.

The following restrictions apply with regard to the method names:

- Do not use names of methods that are or will be provided by a superclass. Due to inheritance, your implementation would overwrite the implementation of the superclass.
- Names starting with `set.../get.../insert.../add.../remove.../indexOf.../destroy...` shall not be used because they may collide with setters/getters for properties or aggregations that are defned explicitely or by a superclass.

- Names starting with `attach.../detach.../fire` may collide with methods created for events.

The following method names have a specific meaning and should be used accordingly:

- `on...`: Used for event handlers that are automatically bound to browser events
- `init`: Used for the initialization function that is called after control instantiation
- `renderer`: Used for the function that creates the control's HTML

> **i Note**
>
> Any method in your inheriting control overrides methods with the same name in the superclass. If, for example, your control implements the `init()` method, the `init()` of the superclass will no longer be executed. The control is then no longer properly initialized and this typically causes an error. To avoid breaking the control, call the superclass method.
>
> Consider also that the superclass might implement the method later on, or removes its own method implementation because it is not needed anymore. We recommend that you check for the existence of the superclass method before calling it:

```
sap.ui.somelib.SomeControl.extend("my.OwnControl", {
       ...
       init: function() {
             if (sap.ui.somelib.SomeControl.prototype.init) { // check
whether superclass implements the method
                   sap.ui.somelib.SomeControl.prototype.init.apply(this,
arguments); // call the method with the original arguments
             }
             //... do any further initialization of your subclass...
       }
```

> **i Note**
>
> When you modify the HTML of a control using the code in the control behavior file, make sure to escape any unchecked data you write with `jQuery.sap.encodeHTML(...)` to prevent cross-site-scripting issues. For more information, see Cross-Site Scripting [page 1475].

Normal Methods

Normal or public methods comprise all methods that do not belong to one of the special method types.

All methods are appended to the implementation object. Private methods are identified by a name starting with an underscore and must **not** be called from outside the control. The following code snippet is an example for the public method `divide` that calls the private helper method `_checkForZero` within the control:

```
  divide: function(x, y) {      // a public method of the Control
      if (this._checkForZero(y)) {
         throw new Error("Second parameter may not be zero");
      }
      return x / y;
   },
   _checkForZero: function(y) { // private helper method
      if (y === 0) {
         return true;
      }
      return false;
```

```
    }
```

init() Method

The `init()` method can be used to set up, for example, internal variables or subcontrols of a composite control.

If the `init()` method is implemented, SAPUI5 invokes the method for each control instance directly after the constructor method.

> **i** Note
>
> Values that are given in the `constructor` method are **not** yet available in the `init` method. This is to prevent that a control only works when the values are set initially, but not when the values are changed.

The `init` method is considered a private method that must only be called by the SAPUI5 core.

```
   init: function() {
      this._bSearchHasBeenTriggered = false;
      this._oSearchButton = new sap.m.Button(this.getId() + "-searchBtn", {text:
"Search"});
   }
```

> **i** Note
>
> Any method in your inheriting control overrides methods with the same name in the superclass. If, for example, your control implements the `init()` method, the `init()` of the superclass will no longer be executed. The control is then no longer properly initialized and this typically causes an error. To avoid breaking the control, call the superclass method.
>
> Consider also that the superclass might implement the method later on, or removes its own method implementation because it is not needed anymore. We recommend that you check for the existence of the superclass method before calling it:
>
> ```
> sap.ui.somelib.SomeControl.extend("my.OwnControl", {
>       ...
>       init: function() {
>             if (sap.ui.somelib.SomeControl.prototype.init) { // check
> whether superclass implements the method
>                   sap.ui.somelib.SomeControl.prototype.init.apply(this,
> arguments); // call the method with the original arguments
>             }
>             //... do any further initialization of your subclass...
>       }
> ```

exit() Method

The `exit()` method is used to clean up resources and to deregister event handlers.

If the `exit()` method is implemented, SAPUI5 core invokes the method for each control instance when it is destroyed.

> **i Note**
>
> Any method in your inheriting control overrides methods with the same name in the superclass. If, for example, your control implements the `init()` method, the `init()` of the superclass will no longer be executed. The control is then no longer properly initialized and this typically causes an error. To avoid breaking the control, call the superclass method.
>
> Consider also that the superclass might implement the method later on, or removes its own method implementation because it is not needed anymore. We recommend that you check for the existence of the superclass method before calling it:
>
> ```
> sap.ui.somelib.SomeControl.extend("my.OwnControl", {
>        ...
>        init: function() {
>              if (sap.ui.somelib.SomeControl.prototype.init) { // check
> whether superclass implements the method
>                    sap.ui.somelib.SomeControl.prototype.init.apply(this,
> arguments); // call the method with the original arguments
>              }
>              //... do any further initialization of your subclass...
>        }
> ```

Event Handler Methods

Event handler methods are invoked when an event occurs. Method names starting with `on` are reserved for event handler methods.

For common events, such as `click` or `keydown`, it is sufficient to add a handler method. These events are defined in the module `sap/ui/events/ControlEventsAs`. As SAPUI5 core automatically registers browser event handlers for these methods, they are called automatically. SAPUI5 core also fires events with a richer semantic meaning, so that control developers do not need to check various keycodes.

```
onclick: function(oEvent) {
   alert("Control " + this.getId() + " was clicked.");
}
```

Internally used events, which start with "sap", are defined in the `sap/ui/events/PseudoEvent` module. An example is the `sapnext` event, which is triggered by "arrow down" or "arrow right" (or "arrow left" in right-to-left mode). The `sapnext` event performs all checks that are required to check whether the user wants to navigate to the next item. The `event` object that is passed to the handler method contains more information. These methods are private methods and must only be called by SAPUI5 core.

```
onsapnext: function(events) {
   // navigate to next item, an arrow key was pressed
}
```

Related Information

API Reference: `ControlEvents`

API Reference: `PseudoEvents`

Browser Events

To react to browser events, a control needs to register for the event either explicitly, or by implementing the event handler.

SAPUI5 applications can have the following two event types:

- Browser events: These events are fired by the browser; examples for browser events are `click` and `blur`.
- Control events: These events are fired by SAPUI5 controls. They contain more semantic information than browser events and relate to the control functionality. An example for a control event is when a browser's a `click` event on an icon in a panel header that triggers a `maximize` or `minimize` event of the control.

To register browser events explicitly for certain DOM elements, use either `jQuery.bind()` or the respective browser methods, such as `addEventListener` and register the event in the `onAfterRendering` method of the control. This ensures that the event binding is repeated after the control is rerendered, meaning that new DOM elements are created and old DOM elements are discarded. The event binding must be removed in the `onBeforeRendering` and `exit` methods by using `jQuery.unbind()` to prevent memory leaks. The `exit` method is called before the control is destroyed.

The explicit registering for browser events enables you to handle any type of browser event and works exactly the same way as in web pages or jQuery-based web applications. On the other hand side, it requires some coding to do the binding and unbinding of the event handler and registering many event handlers can affect the performance.

Example for explicit registration of browser events:

```
MyControl.prototype.onAfterRendering = function() {
        this.$().bind("click", this.handleClick.bind(this));
}
MyControl.prototype.onBeforeRendering = function() {
        this.$().unbind("click", this.handleClick);
}
MyControl.prototype.exit = function() {
        this.$().unbind("click", this.handleClick);
}
MyControl.prototype.handleClick = function(oEvent) {
        // do something...
}
```

Instead of explicitly registering for browser events, you can implement the event handler directly for certain common event types by using a naming convention for the handler method. SAPUI5 automatically registers event handlers for a list of commonly used event types on the root element of a complete tree of SAPUI5 controls. For more information about these event types, see the `sap.ui.events.ControlEvents` in the API Reference. If the respective event occurs at any position in the tree and the respective control implements the `on<eventName>` method, this method is invoked as if it had been registered with `jQuery.bind()`.

The event handler implementation requires less code, reduces the number of event handler registrations in the DOM and also reduces the number of event handler registrations and deregistrations that are executed on every rerendering action. On the other hand, this option is only available for specific events.

Example for registering the event handler directly:

```
MyControl.prototype.onclick = function(oEvent) {
   // do something...
}
```

SAPUI5 also provides so-called pseudo events. Pseudo events are semantically enriched and can be handled by just implementing an `on<eventName>` method. They **cannot** be used with `jQuery.bind()`. By using pseudo events, you avoid additional checks for modifier keys in the event handler or for certain keycodes. For a list of Pseudo Events, see jQuery.sap.PseudoEvents.

Mobile Events

In addition to the general browser and control events you can access specific events for touch-enabled devices.

When implementing SAPUI5 controls, some browser events can be handled very easily by implementing a method named `on<eventName>`, so all the bind/unbind effort is avoided. This is equally possible on mobile.

On touch-enabled platforms additional browser and pseudo events are available:

Additional Mobile Browser Events

On touch-enabled platforms the following events are also provided within UI5 controls to be handled in `on<eventName>` methods:

- `touchstart`
- `touchend`
- `touchmove`
- `touchcancel`

Additional Mobile Pseudo Events

jQuery mobile event handling is used in SAPUI5 when running on touch-enabled devices. From the basic browser events it creates semantically richer events. Some of them are also provided automatically in SAPUI5 controls:

- `swipe`
- `tap`
- `swipeleft`
- `swiperight`
- `scrollstart`
- `scrollstop`

For more information, see jQuery mobile .

Windows 8

With Microsoft Windows 8, new devices are also introduced that allow user interaction with both mouse and touch. To be able to react to both kind of events, some new functionality was introduced.

For more information, see Windows 8 Support [page 2200]

Simulation of touch events on non-touch platforms

For testing or demonstration purposes, the events listed above can also be simulated on non-touch devices. When this simulation is enabled, the touch events will also be triggered by mouse interaction.

> ⚠ Caution
>
> Due to technical constraints the simulation cannot be perfect, so it may not be used productively.

To enable the simulation mode, set the SAPUI5 configuration parameter `xx-test-mobile` to `<true>`, for example by appending the URL parameter `sap-ui-xx-test-mobile=true`.

Related Information

Event Handler Methods [page 2197]
Browser Events [page 2198]

Windows 8 Support

Devices such as Windows 8 touch-enabled laptops now support both mouse and touch input methods together. As a control developer you have to take this into consideration.

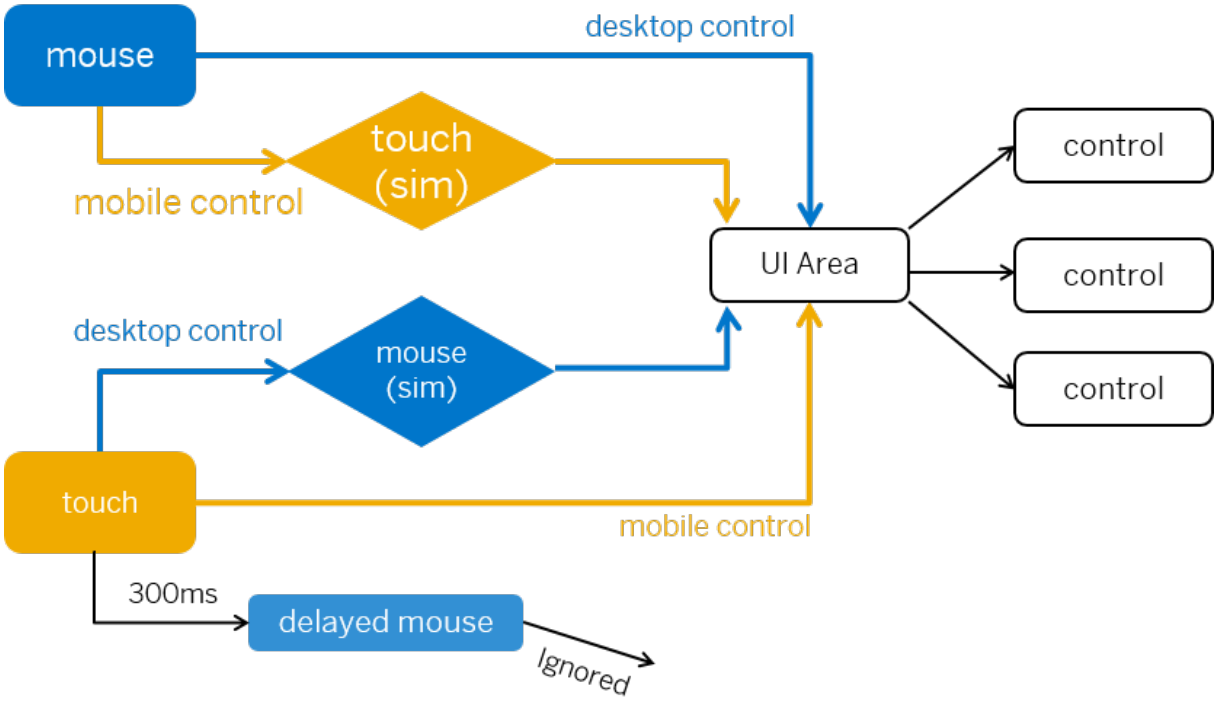
> ℹ Note
>
> SAPUI5 event delegation is adapted and the `ontouch` or `ontap` functions on the control prototype are called on both touch and mouse (exclude the emulated mouse events on touch interfaces) events. This enables the controls which only uses SAPUI5 event delegation for event handling to work seamlessly on Windows 8 touch-enabled devices without being changed. So for you as an SAPUI5 application developer there is no need to change or adapt your applications to support devices that allow input from both mouse and touch.

When you develop your own controls, then there are some things to consider:

Background: How SAPUI5 handles events

With the introduction of touch-enabled Window 8 devices, touch is becoming part of the expected desktop experience. In the past, UI5 statically detected whether the running environment supports touch events. Then the assumption was made that only touch (and not mouse) events need to be supported. This assumption became faulty with the emerging of touch-enabled Windows 8 devices. The fact that touch events are supported doesn't mean that user can't use other input device than touch. Therefore, "support touch" is not equal to "doesn't need mouse support" anymore. For all these reasons, we don't switch between touch and mouse - we now just support both of them!

The following figure shows how this is implemented:



A desktop control is defined as a control that listens to mouse events, whereas a mobile control listens to touch events. To ensure that all events can be received, for mouse events touch simulation events are created and for touch events mouse simulation events, respectively. So the UI Area, which acts as an event delegate, receives the correct events. In detail:

- When a mouse event is triggered for a desktop control, it's handed over to the UI area directly.
- When a mouse event is triggered for a mobile control, a touch event is simulated and handed over to the UI area.
- When a touch event is triggered for a mobile control, it's handed over to the UI area directly.
- When a touch event is triggered for a desktop control, a mouse event is simulated and handed over to the UI area.
- Some browsers send a delayed mouse event after a touch event, this is just ignored.

So it is ensured that all events can be handled and no event is triggered twice.

Support mouse and touch events together

Touch interfaces try to emulate mouse and click events obviously because touch interfaces need to work with applications that have only interacted with mouse events before. This means for a single tap on touch interfaces, the following events are fired in the written order:

1. `touchstart`
2. `touchend`
3. `mousedown`
4. `mouseup`
5. `click`

If we support mouse and touch together, the event handler is called twice for a single tap because there are touchstart and mousedown fired by the browser. Fortunately, we have found a way to set a flag on the emulated mouse events from touch interfaces and suppress those events when they reach the event handler.

Related Information

Adapting Event Handling to Support Windows 8 Devices [page 2203]
Device-specific Behavior of Controls [page 2206]
Windows 8 Support - Known Issues [page 2202]

Windows 8 Support - Known Issues

Information regarding known issues with regard to Windows 8 support.

Firefox doesn't simulate mouse events correctly for touch input method

Firefox currently only fires mouse events on Windows 8 for both touch and mouse input methods. When a touch input method is used to perform a single tap, all of the mouse events aren't fired until the touch input is released from the screen. This causes an issue that all UI updates between `mousedown` and `mouseup` events can't be reflected on the UI because UI Tray doesn't get the chance to update. For example, a blue color should be shown when button is tapped and set back to the normal color after tap. This blue color can't be seen on Firefox because the setting and resetting of blue color are executed in a row and UI Tray doesn't have the time to reflect the blue color.

A fix is only available in the beta version of Mozilla Firefox. Here you can start Firefox in Windows 8 metro mode in which touch events are fired instead of mouse events when touch input method is used.

In the desktop mode of Firefox, touch event support is off and can be turned on only by modifying the `dom.w3c_touch_events.enabled` parameter.

More Information

- Mozilla Firefox Beta version
- Mozilla Support: How do I launch Firefox for Windows 8 Touch?
- Mozilla Firefox - Touch events

Adapting Event Handling to Support Windows 8 Devices

SAPUI5 event delegation automatically handles both mouse and touch events simultaneously. If you are using jQuery or native browser API (`domElement.addEventListener`) to handle events, you have to adapt your coding to support both.

Context

In addition, emulated mouse events shouldn't be handled otherwise the event handler is called twice for the touch. This can be achieved by checking the flag (`_sapui_delayedMouseEvent`) we set to the emulated mouse events. We have extended the `jQuery.Event` object with `isMarked` function for checking UI5 specific flags on the real event object. The `sapui` prefix is already considered within the `isMarked` function so only the semantic part needs to be passed into the function (for example, for checking if the flag `_sapui_delayedMouseEvent` is marked, simply call `isMark(delayedMouseEvent)`). But when event handler is bound using browser API like `addEventListener`, flag needs to be checked by using the full flag name since there's no `isMarked` function on the browser event object.

In most of the cases, the event handler was bound by checking if touch is supported, like the code below:

```
// "Device" required from module "sap/ui/Device"
jQuery(document).on(Device.support.touch ? "touchmove" : "mousemove",
function(oEvent) {
    ......
});
```

Procedure

1. Since both mouse and touch should be supported, the code needs to be changed as follows.
    - When using jQuery:

```
jQuery(document).on("touchmove mousemove", function(oEvent) {
    if (oEvent.isMarked("delayedMouseEvent")) {
        // Suppress the emulated mouse event from touch interface
        return;
    }
    ......
});
```

- When using native browser event listeners:

```
var fnHandler = function (oEvent ) { if (oEvent .
_sapui_delayedMouseEvent ) { // Suppress the emulated mouse event from
touch interface
 return ; } ..... };
document . addEventListener ( "touchmove" , fnHandler );
document . addEventListener ( "mousemove" , fnHandler );
```

2. Same should be applied for the all touch and mouse event pairs when the events are needed:
   - `touchstart` and `mousedown`
   - `touchmove` and `mousemove`
   - `touchend`, `touchcancel` and `mouseup`

Example

We have also adapted some controls within `sap.m` for Windows 8 support. Let's take `sap/m/RatingIndicator` as an example. User can drag the rating indicator to change the rating value. This is implemented by registering to `touchmove` or `mousemove` event in `ontouchstart` function and `deregister` from `touchmove` or `mousemove` by listening to `touchend`, `touchcancel` or `mouseup`.

Before the adaptation, code was:

```
// "RatingIndicator" defined in module "sap/m/RatingIndicator"
// "Device" required from module "sap/ui/Device"
RatingIndicator.prototype.ontouchstart = function(oEvent) {
    if (this.getEnabled()) {
        ......
        if (!this._touchEndProxy) {
            this._touchEndProxy = jQuery.proxy(this._ontouchend, this);
        }
        if (!this._touchMoveProxy) {
            this._touchMoveProxy = jQuery.proxy(this._ontouchmove, this);
        }
        // here also bound to the mouseup mousemove event to enable it working in
        // desktop browsers
        if (Device.support.touch) {
            jQuery(window.document).on("touchend touchcancel",
this._touchEndProxy);
            jQuery(window.document).on("touchmove", this._touchMoveProxy);
        } else {
            jQuery(window.document).on("mouseup", this._touchEndProxy);
            jQuery(window.document).on("mousemove", this._touchMoveProxy);
        }
        ......
    }
};
RatingIndicator.prototype._ontouchmove = function(oEvent) {
    ......
};
RatingIndicator.prototype._ontouchend = function(oEvent) {
    if (this.getEnabled()) {
        ......
        if (Device.support.touch) {
            jQuery(window.document).off("touchend touchcancel",
this._touchEndProxy);
            jQuery(window.document).off("touchmove", this._touchMoveProxy);
        } else {
            jQuery(window.document).off("mouseup", this._touchEndProxy);
```

```
            jQuery(window.document).off("mousemove", this._touchMoveProxy);
        }
        ......
    }
};
```

After the code adaptation:

```
// "RatingIndicator" defined in module "sap/m/RatingIndicator"
// "Device" required from module "sap/ui/Device"
RatingIndicator.prototype.ontouchstart = function (oEvent) {
    if (this.getEnabled()) {
        ......
        if (!this._touchEndProxy) {
            this._touchEndProxy = jQuery.proxy(this._ontouchend, this);
        }
        if (!this._touchMoveProxy) {
            this._touchMoveProxy = jQuery.proxy(this._ontouchmove, this);
        }
        // The if (Device.support.touch) is removed and both mouse and touch
events are supported always
        jQuery(window.document).on("touchend touchcancel mouseup",
this._touchEndProxy);
        jQuery(window.document).on("touchmove mousemove", this._touchMoveProxy);
        ......
    }
};
RatingIndicator.prototype._ontouchmove = function (oEvent) {
    // Check the _sapui_delayedMouseEvent flag for the emulated mouse event from
touch interface
    if (oEvent.isMarked("delayedMouseEvent")) {
        return;
    }
    ......
};
RatingIndicator.prototype._ontouchend = function (oEvent) {
    // Check the _sapui_delayedMouseEvent flag for the emulated mouse event from
touch interface
    if (oEvent.isMarked("delayedMouseEvent")) {
        return;
    }

    if (this.getEnabled()) {
        ......
        // The if (Device.support.touch) is removed, just deregister from every
event
        jQuery(window.document).off("touchend touchcancel mouseup",
this._touchEndProxy);
        jQuery(window.document).off("touchmove mousemove", this._touchMoveProxy);
        ......
    }
};
```

Renderer Methods

The `renderer` method is responsible for creating the HTML structure for the control.

The `renderer` method is a static method, so no `this` keyword is available, but a control instance and a `RenderManager` instance are given to the method. The `RenderManager` collects and concatenates string fragments and places them in the DOM at the appropriate position.

```
renderer: function(oRM, oControl) {
```

```
   oRM.write("<div>", oControl.getText(), "</div>");
}
```

A control must have exactly one HTML element as a root node, additional elements may be added below this node. `oRM.writeControlData(oControl);` must be called while the root node is written. Thus, the root element can be marked as SAPUI5 control root and the ID of the control is written. Also, `oRM.writeClasses();` must be called in the root element of the control, in order to enable support of `addStyleClass()` for the control.

A new renderer type inherits from the renderer of the parent control. If a control extends, for example, the `InputBase` control, the function is added to a class that inherits from `sap.m.InputBaseRenderer` and can access the respective functions.

```
sap.ui.define(['sap/ui/core/Renderer', 'sap/m/InputBaseRenderer'],
               function(Renderer, InputBaseRenderer) {
               "use strict";

               var CustomInputRenderer = Renderer.extend(InputBaseRenderer);
               ...
               return CustomInputRenderer;
}, /* bExport= */ true);
```

If an existing renderer is used without modification, you can use the name of the respective renderer class:

```
renderer: "sap.m.ButtonRenderer"
```

A control renderer can also override or implement methods from the renderer superclass. And it can separate out helper functions.

This is shown in the following example. Note that the methods need to be packed together into an object to indicate that they all go into the control renderer. The main rendering method is called `render`. The `this` keyword refers to the control renderer type and is used to access the other methods:

```
renderer: {
   render: function(oRM, oControl) {
      oRM.write("<div>");
      oRM.writeEscaped(this.square(oControl.getValue()));
      oRM.write("</div>");
   },
   square: function(value) {
      return value * value;
   }
}
```

Device-specific Behavior of Controls

Some controls have different behaviors between running on different device types (mobile, desktop, tablet).

For example, the `sap.m.Carousel` control shows the left and right navigation buttons in addition to gesture only when it runs on desktop. The distinction between desktop and mobile behaviors should be done by checking `sap.ui.Device.system.desktop` but NOT `sap.ui.Device.support.touch` because desktop browser can also support touch, for example Windows 8 touch-enabled device.

A property `combi` is provided in `sap.ui.Device.support` with which you can tell that the browser supports both touch and mouse interfaces.

Not all browser on Windows 8 touch-enabled device support touch events. Some browser fires mouse events when user operates with finger. Since those mouse events are not marked with `_sapui_delayedMouseEvents`, they can still be handled with SAPUI5 event delegation and the registered handler.

Examples for Creating and Extending Controls

Examples how to create and extend controls in SAPUI5.

To create a new control type, you extend the `sap.ui.core.Control` class, and define the control API and the implementation from scratch.

Creating a Simple Control

Example of a simple control with a `name` property

The control is used to render the text "Hello <name>":

```
// "Control" required from "sap/ui/core/Control"
Control.extend("my.Hello", {                  // call the new Control type
"my.Hello"
                                              // and let it inherit from
sap.ui.core.Control
    metadata : {                              // the Control API
        properties : {
            "name" : "string"                 // setter and getter are created
behind the scenes,
                                              // including data binding and type
validation
        }
    },
    renderer : {
        apiVersion: 2,                        // see 'Renderer Methods' for an
explanation of this flag
        render: function(oRm, oControl) {     // the part creating the HTML
            oRm.openStart("span", oControl).openEnd();
            oRm.text("Hello " + oControl.getName()); // write the Control
property 'name', with automatic XSS protection
            oRm.close("span");
        }
    }
});
```

The new control is ready for use now. To instantiate and display the control, use the following code:

```
new my.Hello({name:"UI5"}).placeAt("content");
```

Creating a Simple Square Control

Example of a simple `Square` control that is rendered as a red square with text inside that pops up an alert when clicked

```
// "Control" required from "sap/ui/core/Control"
var Square = Control.extend("my.Square", { // call the new Control type
"my.Square" and let it inherit
                                          // from sap.ui.core.Control
      // the Control API:
      metadata : {
          properties : {           // setter and getter are created behind the
scenes,
                                   // incl. data binding and type validation
              "text" : "string",   // in simple cases, just define the type
              "size" : {type: "sap.ui.core.CSSSize", defaultValue: "200px"}
                                   // you can also give a default value and more
          }
      },

      // the part creating the HTML:
      renderer : {
                                               // instead of "this" in the
render function
              oRm.openStart("div", oControl);  // creates the root element incl.
the Control ID and enables event handling - important!
              oRm.style("width", oControl.getSize());  // write the Control
property size; the Control has validated it to be a CSS size
              oRm.style("height", oControl.getSize());
              oRm.class("mySquare");           // add a CSS class for styles
common to all Control instances
              oRm.openEnd();                   // this call writes the above
class plus enables support
                                               // for Square.addStyleClass(...)
              oRm.text(oControl.getText());    // write another Control
property, with protection against cross-site-scripting
              oRm.close("div");
          }
      },
      // an event handler:
      onclick : function(evt) {   // is called when the Control's area is
clicked - no event registration required
          alert("Control clicked! Text of the Control is:\n" + this.getText());
      }
  });
```

The information for the visual appearance can be written to the control HTML in the `render` method in the same way as the instance-specific width and height. We recommend, however, to define style information that is common to all control instances in a CSS file or in a <style> tag. Thus, it is only written once and can be easily modified by the application.

In general, however, when controls need their own CSS and are also supposed to participate in the theming concept, it is recommended not to use on-the-fly controls, but to create real control libraries. Those take care of loading the CSS, providing right-to-left support, and so on.

To add a grey background, a red border and some alignment information, use the following code:

```
<style>
    .mySquare {                 /* style the CSS class that has been written by
the renderer method */
        display: inline-block;  /* enable squares to appear next to each other
within one line */
```

```
        border: 1px solid red;   /* add some border, so the square can actually
be seen */
        background-color: #ddd;
        padding: 8px;
        text-align: center;
        -moz-box-sizing: border-box; /* consider padding+border part of the
width/height */
        box-sizing: border-box;
    }
</style>
```

This custom control can now be used like any SAPUI5 control:

```
// "Square" required from "my/Square"
var myControl = new Square({text:"Hello", size: "100px"});
myControl.placeAt("content");
```

Creating a Simple Container Control

Example of a container control with arbitrary child controls that are rendered in a row and a colored box around each child

```
sap.ui.core.Control.extend("ColorBoxContainer", { // call the new Control type
"ColorBoxContainer"
    // and let it inherit from sap.ui.core.Control
    // the Control API:
    metadata : {
        properties : {            // setter and getter are created behind the
scenes,
            // incl. data binding and type validation
            "boxColor" : "string"  // the color to use for the frame around each
child Control
        },
        aggregations: {
            content: {singularName: "content"} // default type is
"sap.ui.core.Control", multiple is "true"
        }
    },
    // the part creating the HTML:
    renderer : function(oRm, oControl) { // static function, so use the given
"oControl" instance
        // instead of "this" in the renderer function
        oRm.write("<div");
        oRm.writeControlData(oControl);  // writes the Control ID and enables
event handling - important!
        oRm.writeClasses();              // there is no class to write, but this
enables
                                         // support for
ColorBoxContainer.addStyleClass(...)
        oRm.write(">");
        var aChildren = oControl.getContent();
        for (var i = 0; i < aChildren.length; i++) { // loop over all child
Controls,
            // render the colored box around them
            oRm.write("<div");
            oRm.addStyle("display", "inline-block");
            oRm.addStyle("border", "3px solid " + oControl.getBoxColor()); //
specify the border around the child
            oRm.writeStyles();
            oRm.write(">");
            oRm.renderControl(aChildren[i]);   // render the child Control
```

```
                                                    // (could even be a big Control
tree, but you don't need to care)
            oRm.write("</div>"); // end of the box around the respective child
        }
        oRm.write("</div>"); // end of the complete Control
    }
});
```

As the control itself has no appearance, no additional CSS is required.

You can use this container control like any SAPUI5 container:

```
var oButton = new sap.m.Button({text:'Hello World'});
var oInput = new sap.m.Input({placeholder:'edit text here'});
var container = new ColorBoxContainer({
    boxColor: "#ff7700",
    content:[
        oButton,
        oInput
    ]});
container.placeAt('content');
```

Extending Buttons with Additional Events

Example of a button control that is extended with additional events.

To create a `HoverButton` control, that is, a button that fires a `hover` event when the mouse enters its area, use the following code:

```
sap.m.Button.extend("HoverButton", { // call the new Control type "HoverButton"
                                                    // and let it inherit from
sap.m.Button
      metadata: {
          events: {
              "hover" : {}  // this Button has also a "hover" event, in addition
to "press" of the normal Button
          }
      },

      // the hover event handler:
      onmouseover : function(evt) {   // is called when the Button is hovered -
no event registration required
          this.fireHover();
      },
      renderer: {} // add nothing, just inherit the ButtonRenderer as is;
                   // In this case (since the renderer is not changed) you could
also specify this explicitly with:  renderer:"sap.m.ButtonRenderer"
                   // (means you reuse the ButtonRenderer instead of creating a
new view
  });
```

The `HoverButton` control is used in the application in the same way as a regular button. The following code snippet shows how to attach a handler to the `hover` event:

```
var myControl = new HoverButton("myBtn", {
      text: "Hover Me",
      hover: function(evt) {
          alert("Button " + evt.getSource().getId() + " was hovered.");
      }
```

```
});
myControl.placeAt("content");
```

Extending Input Rendering

Example of an `Input` control with changed rendering.

The following code snippet creates a control type that inherits from `sap.m.Input`. The new control has all features from the `Input` control, but the rendering is changed to be highlighted with yellow background.

The control API and the `render` method can be inherited as it is and the `renderInnerAttributes` method of the `InputRenderer` is overwritten:

```
sap.m.Input.extend("HighlightInput", {// call the new Control type
"HighlightInput"
                                // and let it inherit from sap.m.Input
            renderer: {
                // note that no render() function is given here. The Input's
render() function is used.
                // But one function is overwritten:
                writeInnerAttributes : function(oRm, oInput) {
                    sap.m.InputRenderer.writeInnerAttributes.apply(this,
arguments); // the default method should be called
                                                                        // this
will make sure that all default input attributes will be there
                    oRm.addStyle('background-color', '#ffff00');  // this change
could also be done with plain CSS.
                                                            // But you get the
idea...
                    }
                }
            });
```

The `HighlightInput` control can be used in an application in the same way as `Input`:

```
var myControl = new HighlightInput({value:"Highlighted editing"});
myControl.placeAt("content");
```

Writing a Control Renderer

SAPUI5 provides three classes for control rendering: `sap.ui.core.Control`, `sap.ui.core.RenderManager`, and `sap.ui.core.Renderer`.

Control Class (`sap.ui.core.Control`)

The control class contains the control for rendering. A control consists of properties, events, aggregations, associations, and methods. They define the behavior of the control. The appearance and data of the control is

determined by properties, associations, and aggregations. The `get...` methods of the control are used to access this information during the execution of the `render()` method:

- Accessing properties:

```
// var oValue = oControl.get<Property>();
// for example for the 'text'-property
var oValue = oControl.getText();
```

- Accessing 1..1 aggregations

```
// var oAggregation = oControl.get<Aggregation>();
// for example for content-aggregation
var oAggregation = oControl.getContent();
```

- Accessing 1..n aggregrations:

```
// var aAggregations = oControl.get<Aggregation>s();
// for example for rows-aggregation
var aAggregations = oControl.getRows();
```

- Accessing associations:

```
// var sAssociatedControlId = oControl.get<Association>();
// for example labelFor-association
var sAssociatedControlId = oControl.getLabelFor();
```

RenderManager Class (`sap.ui.core.RenderManager`)

The render manager class collects pieces of HTML and injects the generated markup into the DOM. The `RenderManager` determines and loads the corresponding renderer and delegates the control rendering to the renderer. The `RenderManager` also provides, amongst others, the following helper functions for rendering:

Method	Description
`write()`	Writes string information to the HTML
`writeControlData()`	Writes the ID and the recognition data of the control to the HTML
`renderControl()`	Converts the specified control into HTML representation and adds it to the HTML; used for rendering child controls

For more information, see sap.ui.core.RenderManager.

Renderer Class (`sap.ui.core.Renderer`)

The renderer class is the base class for control renderers. The `Renderer` implements the static `render` method that is called when a control is added to the DOM. To render a control, the `RenderManager` executes the `render` method on the corresponding `Renderer` of the respective control and passes the reference to itself and to the control.

For notepad controls, the renderer class is normally not directly used, the "renderer" method is directly part of the control implementation and will be added to a renderer class behind the scenes.

Related Information

Prevention of Cross-site Scripting [page 2213]

Prevention of Cross-site Scripting

Cross-site scripting (XSS) can be prevented by ensuring that it is **not** possible to inject script code into an application page that runs in a browser.

Controls must prohibit writing scripts to the page that comes from the application or from business data saved by a different user. To ensure this, the following two measures must be combined:

- Validation of typed control properties
  SAPUI5 core validates the value of properties set by the application against the type of the property. This guarantees that an `int` is always an `int`, and a `sap/ui/core/library.CSSSize` is a string representing a CSS size and does **not** contain a script tag. This also applies to enumerations and control IDs. The control renderer can rely on this check when writing the HTML. Property values that are typed in this way can be written without escaping.
- Escaping
  Control developers must ensure that string control properties and other values coming from the application and not sufficiently typed to rule out script tags being contained are escaped when written to the HTML. For this, the `sap/ui/core/RenderManager` and SAPUI5 core provide helper methods.

Avoiding XSS for a New Renderer

To ensure maximum security for a renderer, note the following:

- For control properties, always use the most specific type that is available. For example, use `sap.ui.core.CSSSize` instead of `string` and instead of `sap/ui/core/library.string` for control properties that refer to a CSS size.
- Use helper methods from the `RenderManager` instance (used below as `oRenderManager`), which is provided to the `render` method of the respective renderer to escape the value of a string property that is written to the HTML:
  - Use `oRenderManager.writeEscaped(oControl.getSomeStringProperty()` instead of just `write(...)` for writing plainly to the HTML.
  - Use `oRenderManager.writeAttributeEscaped("someHtmlProperty", oControl.getSomeStringProperty())` instead of just `writeAttribute(...)` for writing attributes.
  - Use `sap/base/security/encodeXML` for string properties where none of the other two options is possible to escape the string and then process it further.
- Check your HTML coding whether application values can make their way into the HTML:
  - Check where the variable values come from: Can the application set a value directly or only decide which of the hardcoded values are used?
  - Escape values given in parameters in method calls of controls because they are currently not validated by SAPUI5 core.

- Keep in mind that XSS can happen anywhere and anytime in CSS classes, or in styles.

Related Information

Cross-Site Scripting [page 1475]

Implementing Animation Modes

Some UI elements can have animations like page transitions or dynamic buttons. There may be cases where the animation has to be suppressed, for example, for performance reasons or for specific users. As a control developer, you have to make sure that your control supports the animation modes that are offered to the users.

The following animation modes are available:

- `full`: all animations are shown
- `basic`: a reduced, more light-weight set of animations
- `minimal`: no animations are shown, except animations of fundamental functionality
- `none`: deactivates the animation completely

The animation mode can either be set using the configuration parameter `window['sap-ui-config']['animationMode']` or as URL parameter (see Configuration Options and URL Parameters [page 703]).

The value for the attribute `data-sap-ui-animation-mode` on the `<html>` document root element is injected automatically on loading with the return value by using the `getAnimationMode` method of the configuration object.

If the mode is changed, the value of `data-sap-ui-animation-mode` is updated correspondingly.

The attribute can be selected in CSS with `html[data-sap-ui-animation-mode="<mode>"]`.

Example

The following implementation defines the duration for animation mode `full` with 1 sec, `minimal` and `basic` with 0.1 sec, and `none` with 0.0 sec.

```
html[data-sap-ui-animation-mode="full"] .someClassName{
    transition-duration: 1.0s;
}
html[data-sap-ui-animation-mode="minimal"] .someClassName,
html[data-sap-ui-animation-mode="basic"] .someClassName {
    transition-duration: 0.1s;
}
html[data-sap-ui-animation-mode="none"] .someClassName {
    transition-duration: 0.0s;
}
```

Related Information

Configuration Options and URL Parameters [page 703]

Implementing Focus Handling

SAPUI5 provides mechanisms for observing the moving focus in an application page for controls. This information is then preserved for refocusing elements after rerendering. The focus triggers event firing. However, due to the high degree of flexibility in control rendering, a functionality tailored to the respective controls is required. For this, the framework provides helper functions for the implementation of focus handling.

Each control provided by the SAPUI5 framework has its own behavior for focus handling, depending on the functionality that is provided by the control. Complex controls and their embedded content constitute the highest level of complexity.

The base class for elements (`Element.js`) provides the following four methods to support the implementation of focus handling:

- `Element.getFocusDomRef()`
  Once a visible element is rendered, it has a Document Object Model (DOM) representation. The root DOM node can be accessed by using the method `getDomRef()` on the element. The root DOM node is the default focused DOM node. After rendering, when a control is supposed to be focused, the framework asks the control for its focus DOM node by using the `getFocusDomRef()` method. If the root DOM node does **not** represent the element that should have the focus, you have to return another DOM node by overriding the `getFocusDomRef()` method.
- `Element.focus()`
  The `focus()` method sets the focus on the element. This is done using the focus DOM node.
- `Element.getFocusInfo()`
  For some controls, it is even more difficult to apply the focus once the control has been rerendered. List controls, for example, have their own internal focus handling and set the focus on the different items. A data table moves the focus over a matrix of cells. The requirement is that a control can apply the focus to its exact previous position after rerendering. In cases where the SAPUI5 rendering mechanism fails to find the correct element after rendering (for example, because it does not have an ID or the ID changed), override the `getFocusInfo()` method and serialize the focus state into a JSON object and return it. Before rendering, the render manager calls this method for the element instance and stores this information for future use. After rendering, it calls the `applyFocusInfo()` method and passes back the serialized object. This is not only useful for focus information, but also, for example, the exact cursor position of a `TextField` control, can be stored in such an object.
- `Element.applyFocusInfo(oFocusInfo)`
  The `applyFocusInfo()` method applies the focus to the element after rerendering. You use this method if a different behavior is expected for the element. The default implementation of this method sets the focus as it is implemented in the `focus()` method (see above).

Example

In the following example, the control would usually set the focus on the second child node of its root node. In this case, simply override the `getFocusDomRef()` method:

```
sap.m.<SampleControl>.getFocusDomRef = function() {
   return this.getDomRef().firstChild.nextSibling;
}
```

Another control generally sets the focus back to the element that previously had the focus. Therefore, it overrides the methods `getFocusInfo` and `applyFocusInfo`.

```
sap.m.<SampleControl>.getFocusInfo = function() {
   return {id:this.getId(),idx:this.<myFocusElementIndex>};
}
sap.m.<SampleControl>.applyFocusInfo = function(oFocusInfo) {
   var oDomRef = this.getDomRef();
   if (oDomRef) {
      this.<myFocusElementIndex> = oFocusInfo.idx;
      this.focus();
   }
}
```

API Reference

sap.ui.core.Element.

Related Information

Convenience Functionality [page 2216]

Convenience Functionality

In addition to automatic focus handling, SAPUI5 provides further functions to support focus handling.

Automatic focus handling is provided by the interaction between the render manager and the element instance. `jQuery.sap.focus` (defined in `jquery.sap.com.js`) and `sap.ui.core.Core.getCurrentFocusedControlId` provide further functions for focus handling.

Related Information

API Reference: `jQuery.sap.focus`

API Reference: `getCurrentFocusedControlId`

Item Navigation - Supporting Keyboard Handling in List-like Controls

The helper class `sap.ui.core.delegate.ItemNavigation` supports item navigation in lists.

The helper class can be used for keyboard events in controls that need the ability to navigate with arrow keys over a one- or two-dimensional list of DOM nodes. The delegate hooks into the browser events for arrow up/down/left/right, page up/down and home/end keys. With a list of DOM nodes provided by the control, it sets the focus to the relevant DOM node in the list while handling the events.

For item navigation handling, the control has to provide a DOM node that surrounds the DOM nodes of all items and a list of the DOM nodes of the items. When the control is entered, the initial focus should be set on the surrounding DOM node. The `setCycling` method determines whether the focus automatically moves to the top after the end of the list was reached. To use the page up/down keys, a page size must be set via the `setPageSize` method on the delegate.

To specify a preselected item for the delegate, use the `setSelectedIndex` method. On reentering a control with a selected item, the method sets the focus on the list item that had been selected before the control was ended. If no selected index is given, the method sets the focus on the first item when the control is entered again.

If item navigation has to trigger a control before a focus is set, the `BeforeFocus` and `AfterFocus` events can be used to do, for example, preparation tasks for the controls visibility.

> **i Note**
>
> Using the item navigation does not prevent you from reacting on the events handled by the delegate in your control.

Integrating Item Navigation

To integrate the item navigation in your control, apply the delegate in the `onAfterRendering` hook of your control.

```
sap.ui.commons.ListBox.prototype.onAfterRendering = function () {
        //Collect the dom references of the items
        var oFocusRef = this.getDomRef(),
                 aRows = oFocusRef.getElementsByTagName("TR"),
                 aDomRefs = [];
        for (var i=0;i<aRows.length;i++) {
                 aDomRefs.push(aRows[i].firstChild);
        }
        //initialize the delegate and apply it to the control (only once)
         if (!this.oItemNavigation) {
                  this.oItemNavigation = new
sap.ui.core.delegate.ItemNavigation();
                  this.addDelegate(this.oItemNavigation);
        }
       // After each rendering the delegate needs to be initialized as well.
       //set the root dom node that surrounds the items
         this.oItemNavigation.setRootDomRef(oFocusRef);
       //set the array of dom nodes representing the items.
         this.oItemNavigation.setItemDomRefs(aDomRefs);
       //turn of the cycling
```

```
        this.oItemNavigation.setCycling(false);
       //set the selected index
        this.oItemNavigation.setSelectedIndex(this.getSelectedIndex());
      //set the page size
       this.oItemNavigation.setPageSize(this.getVisibleItems()); };
```

After the control is destroyed, ensure that the delegate is correctly removed. Otherwise, memory will leak because DOM nodes are still referenced by the delegate.

```
sap.m.List.prototype.destroy = function() {
                  if (this.oItemNavigation) {
                          this.removeDelegate(this.oItemNavigation);
                          this.oItemNavigation.destroy();
                  }
 };
```

Right-to-Left Support in Controls

SAPUI5 supports right-to-left directionality (RTL) in controls.

Unicode defines the direction in which a browser arranges characters to form words. CSS 2.1 also provides a `direction` property. The `dir` attribute in HTML overrides the overall direction of blocks and influences the text alignment, if not set explicitly. The `lang` attribute does **not** influence the text direction.

It is possible to use `document.dir` for text direction. The browser supports it and it can be set in the bootstrap. The `<bdo>` tag in HTML is used to control the bidirectional algorithm. This means that the character order is then not reversed if RTL and LTR words are mixed.

In a nutshell, this means the following:

- Each character inherently belongs to an RTL or LTR script (defined by Unicode). Some characters like parentheses and dots have no inherent directionality.
- Single words are interpreted by the browser as character sequences with the same directionality. For these, the browser knows the text direction and handles them as blocks that get their internal text direction **only** from the used characters.
- The words themselves are separated by the direction-neutral characters like parentheses, spaces and dots. This makes it possible for a single sentence to contain words with either directionality.

  > **i Note**
  >
  > This behavior can be overridden by using the `<bdo>` tag or CSS `unicode-bidi:bidi-override`. This is done when the order of characters must follow the base direction regardless of the inherent character direction.

- The overall direction and how the blocks are put next to each other depends on the base direction of the whole HTML content.
- The default base direction of HTML is left-to-right (LTR), but can be inverted by setting the attribute `"dir='rtl'"`, either on the `<html>` tag or on any sub-region which should have a different base direction.
- This base direction also determines the default text alignment, the order of columns in tables and the presentation of some direction-neutral characters. For example, opening parentheses are still opening parentheses when RTL mode is switched.

- The algorithm for ordering blocks according to the base direction only covers one level of mixed directionality. To achieve deeper nesting, spans with a `dir` attribute can be used to define a sub-context with a different base direction.

General Algorithm

If SAPUI5 is configured for RTL mode, the SAPUI5 core performs the following steps:

1. `dir="rtl"` is set on the HTML tag.

   > **i Note**
   >
   > The W3C officially recommends using the HTML attribute instead of the CSS properties as directionality is determined by content and has nothing to do with the presentation. Another reason is that CSS properties can be ignored. They also recommend using the `<HTML>` tag instead of the `<BODY>` tag.

2. The respective `library-RTL.css` files are loaded.
3. The CSS generator includes an RTL flipping algorithm. This algorithm performs the following changes:
   - `border-left:` is converted to `border-right:`, `padding-left:` is converted to `padding-right:`, `float:left` is converted to `float:right` and so on.
   - All images in the `img` folder are mirrored. If images don't need to be flipped, you need to manually provide the correct RTL version of the image in the corresponding folder.

Right-to-Left Support Guidelines for Control Development

SAPUI5 developers have to consider the text directionality when implementing new controls. The following guidelines explain how this can be done and highlight what you need to focus on.

General Guidelines

You should develop the control as usual, with only left-to-right (LTR) direction in mind.

- You don't need any RTL-specific CSS classes and you shouldn't write RTL-specific styles into a CSS file.
- You shouldn't use CSS properties related to RTL.
- You need to consider the semantics of the control properties. Controls that have directional properties like `left` or `right` need to be changed to `begin` or `end` respectively.
- Think about the RTL behavior according to the items below when writing JavaScript code that relates to positions.

You can find more detailed guidelines and specifics in the *Related Information* section.

Turning on RTL Mode with the URL Parameter

You can test your control by setting the URL parameter `sap-ui-rtl` to `true`. This will display your control in RTL mode. The automatically converted stylesheets and mirrored images are used, and `dir=rtl` is set on the `<html>` tag.

RTL Mode in Text-Displaying Controls

Languages that have RTL text directionality keep the default directionality of numeric values and texts in LTR mode. In order to handle these cases, SAPUI5 uses two additional API properties - `textDirection` and `textAlign`. You can find detailed information on how to use these properties in the section *Related Information*.

Related Information

API Properties for Right-to-Left Support in Text-Displaying Controls [page 1483]

Programmatic Access to RTL

Some controls need to provide specific coding for right-to-left mode (RTL), for example, because they position or animate elements programmatically, and not via CSS. To read the SAPUI5 RTL configuration, use the following function call:

```
var bRtl = sap.ui.getCore().getConfiguration().getRTL();
```

Troubleshooting Common RTL Issues

The following table outlines some common issues that occur when implementing right-to-left (RTL) support for SAPUI5 controls and their solutions.

Table 110: RTL Issues and Solutions

Issue	Solution
For mirrored images, the mirroring does not show a correct RTL image, or animations are removed from GIF images when mirroring.	Create the correct RTL version of the image manually and put it into the `img-RTL` folder, using the same name and path. In most cases, this means just copying the original LTR image. In rare cases, an image may have some content that needs mirroring and other content that does not. In this case, the graphic needs to be adapted manually.
Image mirroring is only supported for GIF, PNG and JPEG images. Other types like .cur, .ico and .svg are not supported.	Create the correct RTL version of the image manually and put it into the `img-RTL` folder, using the same name and path.
The background position in CSS is correctly mirrored, but the LTR version of the control works fine with the default background position: `left top`. This is not explicitly written in the CSS and is therefore not mirrored.	Specify the background position explicitly to display the RTL version correctly.

Text is incorrectly aligned because the CSS `text-align` property is not converted.	Do not use `text-align:left` if you want the text to change sides in RTL mode, but use `text-align:start` instead. `start` and `end` are handled automatically by the browser. Only use `right` and `left` if you want the text to stay on the same side in RTL mode.
	⚠ Caution
	`start` and `end` are not supported by Internet Explorer 9, 10 and 11. These browser versions use the default alignment, which is the same as `begin`. You need to add specific rules for the LTR and RTL case that specify `right` and `left` respectively. For example, for alignment to `end`:
	Sample Code
	`html[data-sap-ui-browser^=ie] .sapUiTableEndAlign{text-align: right;}`
	`html[dir=rtl][data-sap-ui-browser^=ie] .sapUiTableEndAlign{text-align: left;}`

Issue	Solution
If style is set using JavaScript (for example, in the renderer or behavior of a control), the conversion does not take place and the result looks incorrect.	Consider the RTL mode in your calculations, or when possible, use the CSS file instead (which is automatically handled) and write a CSS class.
The alignment of popups with the parent element is unaffected by RTL mode and is therefore often incorrect.	`sap.ui.core.Popup.Dock` has been extended by adding `Begin*` and`End*`. Those will change sides in RTL mode. Use these instead of `Left*` and `Right*` if the popup alignment should change sides.
When JavaScript calculations are used to determine positions or dimensions, existing implementations might imply LTR mode and result in an incorrect layout.	Make these algorithms RTL-compliant by checking the SAPUI5 RTL configuration.
Some text elements inside the control may look incorrect, for instance parentheses may be shown in the wrong position, pointing to the wrong direction. For example **(very) short text** might be rendered as **very) short text)** in RTL mode.	This is a result of the browser's `bidi` algorithm considering the directionality of the characters used. As soon as there is LTR text in the control, the parentheses will be fine again. For controls that have mixed contents, see API Properties for Right-to-Left Support in Text-Displaying Controls [page 1483]
When images are added as `CSS generated content` (with `:before` or `:after` selector), Internet Explorer 9 and 10 automatically mirror the image. Double-mirroring results in an incorrectly mirrored image.	The LTR image needs to be provided without mirroring (use `html[dir=rtl]` and the respective browser selector).
When a control has a `textAlign` property (or something similar), you need to use additional API properties to ensure the correct alignment of the text according to the directionality.	Use the API Properties for Right-to-Left Support in Text-Displaying Controls [page 1483] . Additionally, the static helper method `sap.ui.core.Renderer.getTextAlign(oTextAlign, oTextDirection)` is available. This method calculates the effective value of the CSS `text-align` property considering the property setting and the current or given RTL mode.

Defining Groups for Fast Navigation (F6)

Adjacent controls within the tab chain can be grouped. Within such a group, F6 skips all controls of the group and moves the focus to the first control in the tab chain of the next group. Shift + F6 moves the focus to the first control of the previous group. Adjacent tab chain elements between groups are automatically handled as one group. For nested groups, the most concrete group is used.

Basically, an F6 group is defined via the attribute `data-sap-ui-fastnavgroup="true"` on a DOM element. Several options exist to implement fast navigation support in controls.

> **Note**
>
> We recommend that you do **not** provide fast navigation support for small controls such as `Button` or `InputField`. The fast navigation feature is intended for large, more complex controls containing multiple "tab-able" elements to enable the user to quickly jump over controls if needed.

Defining an `F6` Group on Control or Element Root Level

This is the preferred option and can be used for many use cases. If a control or an element with a DOM representation wants to define an F6 group on its root element, use the `CustomData` mechanism in the `init` function of the control or element to set the attribute.

```
init = function(){
  //...
  this.data("sap-ui-fastnavgroup", "true", true/*Write into DOM*/);
  //...
};
```

The `RenderManager` writes the attribute automatically during rendering when the `openStart` method is called (new rendering API) or when the `writeControlData` or `writeElementData` is called (legacy rendering API). The application can also change the custom data if desired.

Defining the F6 Group Within a Control

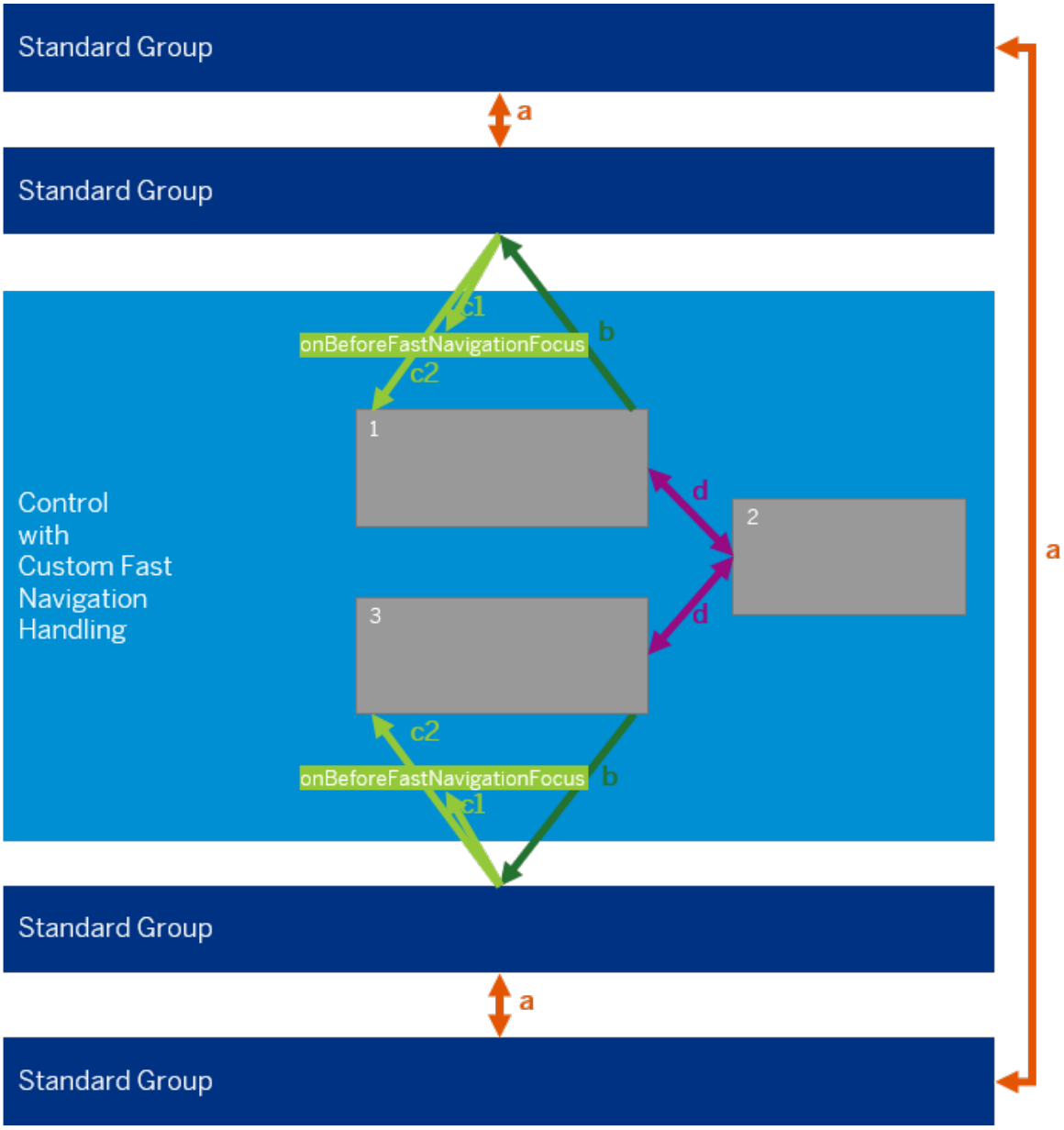
During rendering of a control, the attribute can also be written to any arbitrary DOM element of the control.

```
// assuming a renderer that uses the new rendering API
render = function(oRm, oControl){
  //...
  oRm.attr("data-sap-ui-fastnavgroup", "true");
  //...
};
```

> **Note**
>
> In this case it is difficult for an application to adapt the behavior.

Custom F6 Handling

It may be necessary that a control has to provide a custom fast navigation handling, for example, if the DOM structure of the control does not allow to define suitable navigation groups with one of the options described above. The following picture shows how the central fast navigation handling (a) outside the control collaborates with the custom handling inside the control.



To implement custom fast navigation handling, start with flagging the control as a custom handling area:

```
render = function(oRm, oControl){
  //...
  oRm.openStart("div", oControl);
  oRm.attr("data-sap-ui-customfastnavgroup", "true"); //Attribute must be on the
root element of the control.
  //...
};
```

To implement the custom F6 behavior within the control (d), use the event handlers `onsapskipforward` (F6) and `onsapskipback` (Shift + F6). When `preventDefault` is called on the provided event, the central fast navigation handling ignores the event.

The interesting point is the collaboration (b, c) between the control and the central fast navigation handling.

```
onsapskipforward = function(oEvent){ //F6
  var oTarget = findNextDomRefToFocus(oEvent.target); //Search for the next DOM
element within the control which should be focused.
  if(!oTarget){
    //target is in the last group -> focus should jump to the first group after
the control (done by the central handling, preventDefault not called)
  }else{
    oEvent.preventDefault();
    oTarget.focus();
  }
};
onsapskipback = function(oEvent){ //Shift+F6
  var oTarget = findPreviousDomRefToFocus(oEvent.target); //Search for the
previous DOM element within the control which should be focused.
  if (!oTarget) {
    //target is in the first group -> focus should jump to the first group
before the control (done by the central handling, preventDefault not called)
  } else {
    oEvent.preventDefault();
    oTarget.focus();
  }
};
```

If the focus resides within the control and jumps out of the control (b) when pressing F6 or Shift + F6, the `onsapskipforward` and `onsapskipback` events should not be handled (no `preventDefault` call).

If the focus resides outside the control and the central fast navigation handling calculates a target to focus within the control, the central handling first calls the event handler `onBeforeFastNavigationFocus` (if available) on the control (c1, c2) that is flagged as a custom handling area. The provided event has the following attributes:

- `target`: Specifies the DOM element that the central handling tries to focus within the custom handling area
- `source`: Specifies the DOM element which is the starting point for the calculation of the next/previous element to focus; this is usually the element that is currently focused
- `forward`: Specifies whether forward (F6) or backward (Shift + F6) navigation is used

If `preventDefault` is called on `BeforeFastNavigationFocus`, setting the focus on the target by the central handling is skipped.

```
onBeforeFastNavigationFocus = function(oEvent) {
  var oTarget;
  if (jQuery.contains(this.getDomRef(), oEvent.source)) {
    //The source is within the custom area (e.g. might happen when the focus is
on a popup which is attached to an element within the custom area)
    oTarget = oEvent.forward ? findNextDomRefToFocus(oEvent.source) :
findPreviousDomRefToFocus(oEvent.source);
  } else {
    //The source is outside of the custom area
    oTarget = oEvent.forward ? findFirstDomRefToFocus() :
findLastDomRefToFocus();
  }
  if (oTarget) {
    oEvent.preventDefault();
    oTarget.focus();
  }
};
```

Related Information

Fast Navigation [page 1491]

Composite Controls

Composite controls are implemented by reusing other controls.

Standard Composite Controls

Composite controls are a means to save time and effort by reusing existing controls for the implementation.

For application developers, the composite control is a black box, therefore, an application developer cannot distinguish a composite control from native (non-composite) controls. As the application developer can not distinguish the controls, the control developer can change the implementation later and avoid composition (or the other way around). For existing uses of the respective control, this change is fully compatible.

> **i** Note
>
> If you do **not** intend to re-use a control in several places, a composite control may not be your best choice. Composite controls are best suited for (massive) re-use and for a public API that shields the application developer from its inner workings. If these are not your requirements, consider to use other techniques of factoring out common parts within your application. You can, for example, simply write an XML fragment or a function returning the root of some control tree.

Simple Example: Search Field

To create a composite control, you start with crafting its API including properties, events, aggregations, and so on as you do it for any other control. Choose either element or control as base type. The following simple example combines an input field with a button that we call "search field". To the outside world, it offers an editable value and can fire a search event.

API

As any other control, you can describe composite controls via the JavaScript control definition API, see Developing Controls [page 2158] and the following example.

```
// "Control" required from "sap/ui/core/Control"
var SearchField = Control.extend("SearchField", {
  metadata : {
    properties : {
       "value" : "string"
```

```
    },
    aggregations: {
       "_input" : {type : "sap.m.Input", multiple : false, visibility: "hidden"},
       "_btn" : {type : "sap.m.Button", multiple : false, visibility: "hidden"}
    },
    events: {
       "search" : {}
    }
  }
});
```

The two aggregations with visibility set to `hidden` are defined in the code snippets above. These aggregations are used to hold the inner controls. Aggregations are used to define a parent-child relationship between a parent control and its children (controls or elements). The knowledge about this relationship is, for example, relevant for the SAPUI5 core to dispatch events properly, or to cleanup the children when the parent is destroyed. Hidden aggregations are control internal and are used especially to register the inner controls within the control hierarchy without making them publicly available. Because hidden aggregations are only used internally within a composite control for hidden aggregations, no typed `accessor` functions are generated, they are not cloned, and data binding is not enabled.

Behavior

The control implementation, that is, its behavior, contains the code for initialization and clean-up hooks as well as glue code for properties and events.

Init

The `init` function contains the composite's parts and stores references to them. If you want to hide the composite parts, you should **not** assign an ID to those parts, but rather let the framework compute the IDs automatically. This reduces the possibility that a composite's parts are accessed from outside via the `sap.ui.getCore().byId(...)` function.

If you have to assign IDs to the composite parts, then you should create those IDs by concatenating the main control ID (ID of your composite instance) with a single dash (`-`) and an additional ID for the part like in the following example:

```
mySearchField-input
mySearchField-btn
```

To avoid conflicts with the internal IDs of parts, the part ID (`input` or `btn` in the example) must be prefix-free. That means, it should not contain another dash (for example, don't use parts `input-label` and `input` at the same time). If the control that is used as part `input` also is a composite control and accidentally uses part `label`, then you'll have a conflict between `mySearchField-input-label` (`label` part of the `input`) and your `mySearchField-input-label` artifact (`input-label` part of your composite).

> **i** Note
>
> SAPUI5 reserves the single dash (`-`) for composite controls and their parts, a double dash (`--`) is used to combine the ID of views and their contained controls and a triple dash (`---`) is used to combine component IDs and the IDs of their owned controls or views.

During the `init` function, the settings of the composite only have their default values. If the application developer has provided some values to the constructor, these values will only be set later on. It is, therefore,

crucial for the correct behavior of your composite control that you implement one of the synchronization mechanisms described below.

```
// "Button" required from "sap/m/Button"
// "Input" required from "sap/m/Input"
/**
 * Initialization hook... creating composite parts
 */
SearchField.prototype.init = function(){
  var that = this;
  this.setAggregation("_input", new Input({
    change: function(oEvent){
      that.setProperty("value", oEvent.getParameter("Value"), true /*no re-
rendering needed, change originates in HTML*/); //see section Properties for
explanation
    }
  }));
  this.setAggregation("_btn", new Button({
    text: "Search",
    press: function(){
      that.fireSearch();
    }
  }));
};
```

Exit

You can use the `exit` function to clean up your control when it is destroyed. You do not need to destroy the inner controls. This is done automatically by the framework because the inner controls are kept in hidden aggregations.

```
/**
 * Clean-up hook... destroying composite parts.
 */
SearchField.prototype.exit = function() {
  //nothing to do here
};
```

Properties

Changes to settings in the API of a composite control are usually reflected in its parts. In the following example, the value property is propagated to the input part. To do so, the generated setter for that property is overwritten. Make sure that you include the proper implementation which generically sets a property inside the element base class, else you would have to override the getter also.

Note how the input's change event is used to update the composite's value property. Because the change originated in the HTML input field, no re-rendering is needed. This is expressed by the third parameter of the `setProperty` call. This trick is applicable whenever a property change does not require a re-rendering on this control level.

> **i Note**
>
> Changing the input part's value triggers a re-rendering of the input.

```
/**
 * Propagate value to input.
 */
SearchField.prototype.setValue = function(sValue){
    this.setProperty("value", sValue, true /*no re-rendering of whole search
field needed*/);
```

```
    this.getAggregation("_input").setValue(sValue); // Note: this triggers re-
rendering of input!
};
```

Propagating the API settings to the parts is usually not as straightforward as shown in the example above. If intercepting the changes by overriding the setters is not sufficient or too complicated, an alternative approach might be to implement a single `updateAllParts` method and call it at the beginning of the renderer of the composite control or in the `onBeforeRendering` hook of the control itself..

Renderer

You can use markup for layouting in the renderer implementation. But at the heart of it, you simply delegate (via the render manager) to the composite parts' renderers. This is where you really benefit from re-using other controls with non-trivial renderers. If you have chosen the `updateAllParts` approach to keep the composite API settings and the settings of the parts in sync, make sure that you call `updateAllParts` before the real rendering starts.

```
SearchFieldRenderer.render = function(oRm, oSearchField) {
  // oSearchField.updateAllParts(); // called depending on your 'sync' approach
  oRm.openStart("div", oSearchField);
  oRm.class("SearchField");
  oRm.openEnd();
  oRm.renderControl(oSearchField.getAggregation("_input"));
  oRm.renderControl(oSearchField.getAggregation("_btn"));
  oRm.close("div");
};
```

XML Composite Controls

An XML composite control allows you to define a composite control that clearly separates the behavior of the control from the visual part.

Overview

An XML composite control consists of two parts, a JavaScript part, which contains the behavior and the interface of the control, and an XML part, which contains the visual representation and the structure of the control. The XML part can also define the binding to the properties and aggregations specified in the JavaScript part, so you do not need to specify a model in the binding.

Related Information

Standard Composite Controls [page 2226]

API Reference: `XMLcomposite`

Samples for `XMLcomposite`

Example of a Simple XML Composite Control

This section shows you an example of a simple XML composite control.

> **i Note**
>
> The code samples in this section reflect examples of possible use cases and might not always be suitable for your purposes. Therefore, we recommend that you do not copy and use them directly.

Here is an example of a simple XML composite control, with the following JS part:

```
sap.ui.define([
    'sap/ui/core/XMLComposite'],
    function( XMLComposite ) {
    "use strict";
    var SimpleText = XMLComposite.extend("fragments.SimpleText", {
        metadata: {
            properties: {
                text: { type: "string", defaultValue: "Default Text"}
            }
        }
    });
    return SimpleText;
}, /* bExport= */true);
```

And here comes the visual part, the fragment definition XML file:

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
        <m:Text text="{$this>/text}" />
</core:FragmentDefinition>
```

There is no specific `renderer` method; the rendering is handled generically by the framework.

If you want to use this XML composite control, you can set up a page like this:

```
<!DOCTYPE HTML>
<html>
<head>
    .
    .
    .
    <script>
        sap.ui.require([
            "fragments/SimpleText"
        ], function(SimpleText) {
        var oSimpleText = new SimpleText({text : "Hello World"});
        oSimpleText.placeAt("content");
    });
    </script>
</head>
<body id="content" class="sapUiBody">
</body>
</html>
```

Properties and Property Bindings

You can use various types of binding for the properties in XML composite controls.

The fragment definition XML file defined in the previous section looks a lot like any other fragment that you might already be using in XML views. In this particular case, it defines only one nested `sap.m.Text` element. The text property is bound with the normal binding syntax to a special model, `$this`. `$this` refers to the interface of the `SimpleText` XML composite control.

Similar to other controls, you have the following options to use binding for the XML definition of an XML composite control:

Table 111: Property Binding

Binding	Sample	Use	Comments
Simple property binding	`text="{$this>/text}"`	Maps a property of the inner control interface to a property of your XML composite control interface.	
Expression binding	`text="{=${$this>/text} + 'additionalText'}"`	Adds 'additionalText' to the value of the property of the inner control.	One-way binding only
Composite binding	`text="{$this>/text} - {$this>/text1}"`	Concatenation of the two properties `text` and `text1`	One-way binding only

Events

This sections shows an example of a `press`event.

You can enhance the XML composite control created in the first step by adding a button. It's as simple as doing it in an XML view.

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
    <m:HBox>
        <m:Text text="{$this>/text}" />
        <m:Button text="Press Me" />
    </m:HBox>
</core:FragmentDefinition>
```

Now the sample should look like this:

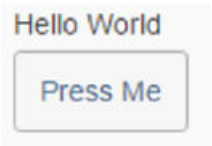


Figure 338: Sample UI

To handle the `press` event of the new button in the interface, first define the handler in the XML file, and then add a method in the JS file:

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
    <m:HBox>
        <m:Text text="{$this>/text}" />
        <m:Button text="Press Me" press="_handlePress"/>
    </m:HBox>
</core:FragmentDefinition>
```

```
...
    var SimpleText = XMLComposite.extend("fragments.SimpleText", {
        ...
    });
    SimpleText.prototype._handlePress = function () {
        this.setText("You pressed the button");
    }
    return SimpleText;
```

If you click the button now, the text should change:

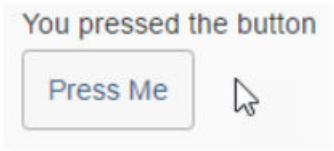


Figure 339: Changed Sample UI

> **i Note**
>
> When accessing controls within the fragment definition XML file, for example, the text or the button, you should use the `byId` method of `XMLComposite` and not `sap.ui.getCore().byId`. If the text has an `id="myText"`, you could get the text instance in the `_handlePress` method via `this.byId("myText")`.

Aggregations

This sections shows you a use case for aggregations in the XML composite control.

If you would like to define an aggregation within a control used in the XML composite control, you proceed as usual:

```
sap.ui.define([
    'sap/ui/core/XMLComposite'],
    function( XMLComposite ) {
    "use strict";
    var TextList = XMLComposite.extend("fragments.TextList", {
        metadata: {
            aggregations: {
                texts: {
                    type: "sap.ui.core.Item",
                    multiple: true
                }
            }
        }
    })
    return TextList;
```

```
}, /* bExport= */true);
```

For this use case, the fragment definition XML file might now look like this:

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
    <m:VBox items="{$this>/texts}">
        <m:Text text="{$this>text}" />
    </m:VBox>
</core:FragmentDefinition>
```

In the `items` template of the `VBox` (in our case an `sap.m.Text`), we bind to the `texts` aggregation. As a result, a list of text items is rendered within a `VBox`.

You can see an example of how you can use the XML composite control in the following HTML file:

```
//add the TextList and place it in the page
var oTextList = new fragments.TextList();
oTextList.addText(new sap.ui.core.Item({text: "Text Item 1"}));
oTextList.addText(new sap.ui.core.Item({text: "Text Item 2"}));
oTextList.addText(new sap.ui.core.Item({text: "Text Item 3"}));
oTextList.addText(new sap.ui.core.Item({text: "Item 4"}));
oTextList.placeAt("body");
```

You can also use advanced features of `ListBinding` to display your data. For example, you can sort or filter your items, as the following examples show:

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
    <m:VBox items="{path:'$this>/texts', sorter:{path:'text', descending:true}}">
        <m:Text text="{$this>text}" />
    </m:VBox>
</core:FragmentDefinition>
```

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
    <m:VBox items="{path:'$this>/texts', filters:{path:'text',
operator:'Contains', value1:'Text'}, sorter:{path:'text', descending:true}}">
        <m:Text text="{$this>text}" />
    </m:VBox>
</core:FragmentDefinition>
```

The result will look like this:

Figure 340: Sorting and Filtering UI

Aggregation Forwarding

A mechanism used for aggregations of composite controls.

Overview

Aggregation forwarding is used when application developers want to add child controls to an aggregation of a composite control, but the composite control does not keep these controls as direct children. Instead, it moves or forwards them to an aggregation of one of its internal controls.

This technique is often used when a control with an aggregation is wrapped by a composite control to add functionality, but the application still has control over the content of the wrapped control. In other cases, the composite control uses layout controls internally to define the placement of aggregated children.

> **i Note**
>
> While aggregation forwarding as described here is mainly aimed at composite controls, it has also been implemented for the `ManagedObject` base class. The forwarding can also be used for other entities which are not controls, but inherit directly from `ManagedObject` or one of its subclasses.
>
> For more information about this class, see the API Reference: `ManagedObject`.

> **i Note**
>
> Sometimes the controls that have been added to an aggregation of a composite control have to be transformed into different controls, which are then added to an aggregation of an internal control. This is a different use case and not covered by aggregation forwarding. With aggregation forwarding, aggregated child controls are moved **without** transforming them.

Configuration

Aggregation forwarding requires a simple additional setting in the definition of a control aggregation. SAPUI5 needs to know to which internal control all aggregated children need to be forwarded and to which aggregation of this internal control.

Aggregation forwarding is defined in the aggregation definition inside the control metadata.

The `forwarding` property can be set as an object defining the following:

- `getter` or `idSuffix`: A way how SAPUI5 determines the internal control instance to which the aggregation is forwarded at runtime, which is what you could call the target control. This can either be done by specifying the `getter`, the name of a function of the composite control which always returns the target control instance, or the `idSuffix`, a string which is appended to the ID of the composite control to construct the ID of the target control.
- `aggregation`: The name of the aggregation of the target control to which this aggregation is forwarded

- `forwardBinding` (optional): Determines whether any binding is done at the target control or only at the outer composite control. This can be crucial if the forwarding target control has functionality that requires the aggregation to be bound.

When such a forwarding definition is done, SAPUI5 moves all aggregated child controls to the target control. All calls to `addAggregation`, `removeAggregation`, `indexOfAggregation` and so on are forwarded. When asked for the forwarded child control, both the composite control and the forwarding target act like the child control belongs to their aggregation. However, the inner forwarding target control is the actual parent of all forwarded children.

Examples

Here is an example that demonstrates aggregation forwarding: The new `FilterableList` control is supposed to display a list of items with an input field above the list. The list items are filtered while the user is entering the input. This `FilterableList` control can be implemented as a composite control, using the `sap.m.List` and `sap.m.Input` controls as inner controls to take advantage of their existing implementation, design, and set of features. Application developers using `FilterableList` cannot change all attributesof the inner `List` control. However, they should be able to provide the actual list items. Hence, the new `FilterableList` composite control has an `items` aggregation and forwards all items to the inner `sap.m.List` control, so, for example, the layouting, events, and selection can be handled there.

```
aggregations: {
     // The items forwarded from the FilterableList to the internal sap.m.List
     items : {type: "sap.m.ListItemBase", multiple: true, forwarding: {
           idSuffix: "-myInternalList",
           aggregation: "items"
     }},
```

Another example would be a new `ButtonList` control that is supposed to contain and display an arbitrary number of `sap.m.Button` controls in a grid. Hence it has a `buttons` aggregation. For this control, control-specific HTML could be written that provides screen-size-dependent CSS for a proper grid layout of the buttons. However, this effort can be avoided, and a `sap.ui.layout.Grid` control used internally instead to do the layouting. The buttons given to the `ButtonList` control then need to be forwarded to the `content` aggregation of the `Grid` control.

```
aggregations: {
     // The items forwarded from the ButtonList to the internal
sap.ui.layout.Grid
     buttons: {type: "sap.m.Button", multiple: true, forwarding: {
           getter: "_getInternalGrid",
           aggregation: "content"
     }},
```

Aggregation Forwarding in XML Composite Controls

If you use aggregation forwarding with `idSuffix` for an XML composite control, you define this as follows:

```
sap.ui.define([
    "sap/ui/core/XMLComposite"],
```

```
    function( XMLComposite ) {
    "use strict";
    var TextList = XMLComposite.extend("fragments.TextList", {
        metadata: {
            aggregations: {
                texts: {
                    type: "sap.ui.core.Item",
                    multiple: true, forwarding: {
                           idSuffix: "--myInternalVBox",
                           aggregation: "items"
                     }
                }
            }
        }
    })
    return TextList;
}, /* bExport= */true);
```

In this case, the fragment definition XML file looks like this:

```
<core:FragmentDefinition xmlns:m="sap.m" xmlns:core="sap.ui.core">
    <m:VBox id="myInternalVBox"/>
</core:FragmentDefinition>
```

> **i** Note
>
> `myInternalVBox` is prefixed with `--`. Other than that, the coding looks exactly the same as the one for aggregation forwarding for standard composite controls.

Dos and Don'ts

If you use aggregation forwarding, you have to keep the following in mind:

- Do not call any methods (such as `add`, `insert`, `remove` , or `destroy`) that modify the aggregation in the forwarding target, but call them in the control that defines the forwarding.
  For example, if you create something like a `CustomList` control that uses forwarding for its `items` aggregation to an internal `List` control, do not call `this._internalList.destroyItems()`, but call `this.destroyItems()`.
- Aggregations can only be forwarded to non-hidden aggregations of the same or a greater multiplicity (single-to-single, single-to-multi, multi-to-multi).
- The target aggregation and the source aggregation have to be compatible: Any child elements given to the source aggregation must be valid in the target aggregation as well (otherwise the target element will throw a validation error).
- The aggregation target control for a particular instance of a composite control must stay the same across the entire lifecycle of the composite control.
- If the content in the target aggregation is modified by other entities or actions, such as the target control itself or another forwarding from a different source aggregation, this will lead to an unexpected behavior of the aggregation forwarding. Hence, these modifications are not allowed.
- Forwarded child controls always have the same models that were also available at their original location **before** the forwarding. They will not use any models that are only set for the inner control to which they are forwarded. This way, models set by an application will not be overridden.

Also, this is in accordance with what application developers would expect regarding the models set for the child control: Any bindings they define should work regardless of how aggregation forwarding is used within the controls.

- Never clone children in public aggregations even if the aggregation is forwarded to an inner control. They are cloned automatically by the framework.
  Also, do not clone inner controls created by your composite control, for example, inside the `init()` method: If your control is cloned, the `init()` method of the clone is called, and the inner control is created as well.

Accessibility Aspects

If you are developing SAPUI5 controls, you have to be aware of the accessibility aspects. A deeper understanding is needed, so that all accessibility requirements are met.

In the following topics, we provide additional information for control developers on keyboard handling, high-contrast theming and other important accessibility aspects.

For information on colors and theming, follow the guidelines listed under *Related Information*.

Related Information

Product Standards and Acceptance Criteria [page 2169]
Colors and Theming [page 1489]

Keyboard Handling for SAPUI5 Controls for Developers

As an application developer, you need to be aware of how the various accessibility aspects, like keyboard handling, are implemented and used.

Keyboard Handling for Basic Navigation

The following keys and key combinations are used for navigation between controls within an application.

Standard Navigation

Navigation between controls is done using the `TAB` key. `TAB` moves the focus from one control to the next one inside the application. The tab order is defined by the placement of the control within the DOM tree, therefore apps have a large influence on it.

> **i Note**
>
> Controls are in the tab order, if they are interactive, enabled and visible. This includes read-only controls. Disabled or hidden controls are taken out of the tab order. Non-interactive controls (for example, layout container) can never be reached with `TAB`.

Key combination	Behavior
`TAB`	Forward Navigation: On enter, move focus to the control. On leave, move focus to the next control in the application.
`SHIFT` + `TAB`	Backward Navigation: On enter, move focus to the control. On leave, move focus to the previous control in the tab order.

Group Navigation

Controls which are adjacent within the application can be grouped. Within a group, `F6` skips all controls of the group and moves the focus to the first control in the application within the **next** group. `SHIFT` + `F6` moves the focus to the first control of the **previous** group.

Key combination	Behavior
`F6`	Forward Navigation: Move focus to the next control in the tab order after the group
`SHIFT` + `F6`	Backward Navigation: Move focus to the previous control in the tab order before the group

Keyboard Handling for One-Dimensional Navigation

The following keys and key combinations are used for navigation in one-dimensional item containers (for example, lists and drop-downs).

Key combination	Behavior
`UP`, `LEFT`	If focus is on an item, move focus to the **previous** item. If focus is on the first item, do nothing.

Key combination	Behavior
DOWN, RIGHT	If focus is on an item, move focus to the **next** item. If focus is on the last item, do nothing.
PAGE UP	If focus is on an item, move focus **up/ left** by page size. **i Note** Page size can be set by the application; default page size is 10 items If focus is on the last item, do nothing.
PAGE DOWN	If focus is on an item, move focus **down/ right** by page size. **i Note** Page size can be set by the application; default page size is 10 items If focus is on the first item, do nothing.
HOME	If focus is on an item, move focus to the **first** item.
END	If focus is on an item, move focus to the **last** item.

Keyboard Handling for Two-Dimensional Navigation

The following keys and key combinations are used for navigation in two-dimensional item containers (for example, calendars and tables).

Key combination	Behavior
LEFT	If focus is on an item, move focus one item to the **left**. If focus is on the first item of a row, move focus to the last item of the previous row. If focus is on the first item, do nothing.
RIGHT	If focus is on an item, move focus one item to the **right**. If focus is on the last item of a row, move focus to the first item of the next row. If focus is on the last item, do nothing.
UP	If focus is on an item, move focus to the item **above**. If focus is on the first item of a column, do nothing.

Key combination	Behavior
DOWN	If focus is on an item, move focus to the item **below**. If focus is on the last item of a column, do nothing.
PAGE UP	If focus is on an item, move focus **up** by page size. **i Note** Page size can be set by apps; default page size is 5 rows. If there are less items available than page size, move focus to the first item. If focus is on the first item, do nothing.
PAGE DOWN	If focus is on an item, move focus **down** by page size. **i Note** Page size can be set by apps; default page size is 5 rows. If there are less items available than page size, move focus to the first item. If focus is on the last item, do nothing.
ALT + PAGE UP	If focus is on an item, move focus **left** by page size. **i Note** Page size can be set by apps; default page size is 5 columns. If there are less items available than page size, move focus to the first item. If focus is on the first item, do nothing.
ALT + PAGE DOWN	If focus is on an item, move focus **right** by page size. **i Note** Page size can be set by apps; default page size is 5 columns. If there are less items available than page size, move focus to the first item. If focus is on the first item, do nothing.

Key combination	Behavior
HOME	If focus is on an item, move focus to the **first** item on the same row. If focus is on the first item of a row, move focus to the first item.
END	If focus is on an item, move focus to the **last** item of the same row. If focus is on the last item of a row, move focus to the last item.
CTRL + HOME	If focus is on an item, move focus to the **first** item of the same column. If focus is on the first item of a column, move focus to the first item.
CTRL + END	If focus is on an item, move focus to the **last** item of the same column. If focus is on the last item of a column, move focus to the last item.

Keyboard Handling for Triggering Actions on Item Level

The following keys and key combinations are used for triggering events of clickable elements.

Key combination	Behavior
SPACE	If items are **not selectable** and focus is on an item, trigger the item event. → Tip If you press and hold the key, you can cancel the trigger action by pressing Shift. If items are selectable, select/deselect the item.
ENTER	If focus is on an item, trigger the item event immediately after the key press.

Use the following keys to trigger additional actions (if supported):

Key combination	Behavior
DELETE	If deletion of items supported: If focus is on an item, delete the item. Move focus to the next item. If the deleted item is the last item, move focus to the previous item. If the deleted item is the last remaining item, move focus to the next control in the tab order.
F2	If *Detail* of items is supported: If focus is on an item, trigger the click event for the *Detail* button.

Keyboard Handling for Item Selection

The following keys and key combinations are used for selecting one or multiple items from a list.

Single Selection

Key combination	Behavior
SPACE	If focus is on an item, select the item and deselect all others.

Multi Selection

Key combination	Behavior
SPACE	If focus is on an item, select the item and deselect all others.
CTRL + SPACE	If focus is on an item, select the item in addition to an existing selection..
SHIFT + SPACE	If focus is on an item, select all items from the previous selected item to the now focused item (included). Previous selection: all kinds of selection except SHIFT SPACE selections

Key combination	Behavior
SHIFT + UP	If focus is on an item, change selection state (selected/ not selected) to the item above.
SHIFT + DOWN	If focus is on an item, change selection state (selected/ not selected) to the item below.
CTRL + A	Selects all items which the user can reach in the current view by scrolling or paging. If all items are selected, deselect all items.

Keyboard Handling for Value Help and Auto-Complete

The following keys and key combinations are used for triggering and using the value help and auto-complete features.

Auto-complete

Auto-complete is available for one dimensional editing only.

Key combination	Behavior
Any printable character	Adds the corresponding character. If text is selected, it gets overwritten. Triggers autocomplete, if available.
RIGHT or DOWN	Move caret on position to the right. If text is selected, move caret to the end of the selection and remove selection. If caret is at the rightmost position, do nothing. If autocomplete is currently available, take over changes. Move caret to the right of the changed text.
ENTER	If autocomplete is currently available, take over changes. Move caret to the right of the changed text.
TAB	Move focus to next element. Take over autocomplete, if available.

Value Help

Use the following keys and key combinations to trigger value help:

Key combination	Behavior
ALT + DOWN or ALT + UP or F4	Open the value help dialog.

Screen Reader Support for SAPUI5 Controls

SAPUI5 offers screen reader support in order to aid people with visual impairments. The implementation is based on the ARIA (Accessible Rich Internet Applications) standard.

General Information

Currently, the following libraries have screen reader support based on the ARIA standard:

- sap.f
- sap.m
- sap.suite.ui.commons
- sap.tnt
- sap.ui.commons
- sap.ui.comp
- sap.ui.core
- sap.ui.generic
- sap.ui.layout
- sap.ui.suite
- sap.ui.table
- sap.ui.unified
- sap.ui.ux3
- sap.uxap
- sap.viz

SAPUI5 controls provide the prerequisites for screen reader support based on the ARIA standard. All screen readers that implement this standard should work fine. If there are deviations in the interpretation, these need to be addressed to the screen reader vendor. If you need more information on our testing environment, see SAP Note 2564165.

> **i Note**
>
> - No screen reader activation settings are necessary since the accessibility mode in SAPUI5 is switched on by default.

ARIA Attribute Mapping

Navigation with the keyboard and screen reader have to both work properly at the same time. In order for this to happen, you need to use the correct ARIA attributes and to map them to their HTML counterparts.

Attribute Mapping

The ARIA `role=application` is added to the body of each page by SAPUI5 Core to ensure that the page can be properly navigated using the keyboard. If this is not the case, the SAPUI5 JavaScript key handler code may get overridden by the screen reader and this will hinder keyboard handling.

The mapping of HTML attributes to ARIA attributes is described in the following table:

Table 112: Attribute Mapping

HTML Attribute	ARIA Attribute
`editable`	`aria-readonly`
`enabled`	`aria-disabled`
`visible`	`aria-hidden`
`required`	`aria-required`
`checked`	`aria-checked`
`selected`	`aria-selected`

For custom controls, not part of the ARIA 1.0 role definitions, mapping to similar and existing ARIA base role concepts is applied. In special cases, custom role names can be added by the SAPUI5 framework using `aria-describedby` or `aria-labelledby` references.

Additional API Associations

In order to ease the setting of ARIA attributes, we have introduced two new associations to the SAPUI5 API:

1. • ariaLabelledBy - holds a reference to the control that has the `aria-labelledby` attribute set
2. • ariaDescribedBy - holds a reference to the control that has the `aria-describedby` attribute set

These associations have the following structure:

Source Code

```
ariaLabelledBy : {
type : "sap.ui.core.Control",
multiple : true,
singularName : "ariaLabelledBy"
}
```

Source Code

```
ariaDescribedBy : {
type : "sap.ui.core.Control",
multiple : true,
singularName : "ariaDescribedBy"
}
```

Keyboard Usage of ARIA Role Mapped Controls

Screen readers offer list features, that ease the app navigation, by grouping and extracting all elements with similar behaviors. This leads to additional requirements when creating SAPUI5 controls. Control developers need to make sure that their controls are marked with the correct ARIA role.

Keyboard Handling for Role Mapped Controls

The applied role names define implicitly the keyboard usage. For example an element with `role=button` can be activated with `SPACE` and `ENTER` keys, navigation between controls with `role=menuitem` is expected using `Arrow keys` and so on.

Note

If you develop new SAPUI5 controls please note the following:

- Navigation with the cursor of the screen reader cannot be the only option. Keyboard navigation has to be implemented as well.
- All information about roles, states, and properties must be present at the keyboard focus position and be spoken when the focus moves to the control.

Additional Requirements for Some Control Types

Screen readers offer lists that group certain types of elements. These lists ease and speed up the navigation. There are some requirements, that specific control types have to comply to, in order to be properly used by the screen reader lists.

- **Landmarks/Regions**
  - Special regions must be indicated and labeled (Navigation, Page Header, Main etc.) in order to be part of the landmark list.
- **Headings**
  - The headings in an SAPUI5 app must have either ARIA `role=heading` + `aria-level` or use `<H1 – H6>` HTML tags.

- The panel heading hierarchy must be logical (for example, nested panels must have higher hierarchy levels).
- Headings must be referenced by the containers they belong to (using `aria-labelledby = "HEADING_ID"`).

- **Links**
  - Links in an SAPUI5 app must either be true `<a>` HTML elements or have ARIA `role=link`.
- **Form Controls**
  - Form fields must be correctly labeled and their list indicators have to be distinctive.
  - Form fields will show up as Input, Radio Buttons and so on in the form list of the screen reader.
- **Lists**
  - Lists have to implement a navigation concept that allows using both arrow keys to go through list item entities, and also be able to focus individual active sub-parts of a list item.
  - Lists and list items should always be identified using correctly nested `<ul>` and `<li>` markup or alternatively by ARIA `role=list` and `role=listitem` roles.
  - Sometimes list items may need special roles (for example `menuitem`).
  - For lists with a specific number of items, speech output should always be "*Current item number of N total items*". No matter how many items are visible, if the total number is not known, speech output should always be "*Current item number*".
- **Tables**
  - Data tables must be coded with valid HTML.
  - Tables must have titles.
  - Layout tables for presentational purposes must be coded as such (using ARIA `role="presentation"`). Then they are not displayed in the table list.
  - Editable and active cells may require special interaction models. During navigation, all screen reader relevant information must be available at the focus location.
  - Tables have to properly associate and identify (even hierarchical) row/column headers. They have to be announced for every cell.
  - In case there are no visible headers, but information is organized in a table-like layout with individual columns/cells, a respective row/column identifier has to be provided.
  - For tables with a specific number of rows, speech output should always be "*Current row of N total rows*". No matter how many rows are visible, if the total number is not known, speech output should always be "*Current row*".

ARIA Mapping for Tooltips and Textual Alternatives

Tooltips and semantic colors are important aspects in apps. They have to be interpreted correctly by the screen reader and require some special ARIA labeling.

Tooltips

Currently tooltips have to be disabled for all controls. An exception is made for images (stand-alone or as part of controls, such as buttons with icons/images but without text on the button itself).

Graphics and Colors

All images and icons must have a textual explanation that the screen reader can read. This is done with the attributes `alt` or `aria-label`. Text and content colored with semantic colors need to have a textual alternative describing the semantics. Interactive graphics, like charts, need to follow these requirements:

- An editable color scheme and possibility of color modification
- Navigable using the keyboard
- The screen reader information must be available at the focus points (axis values, dialog info, legend)

You should hide image controls or controls with background CSS images used for pure decoration purposes by using the `aria-hidden="true"` property.

ARIA Event Handling

When the UI of an application is changing or loading information, these state transitions and updates need to be passed on to the screen reader as well. You need to set the correct ARIA attributes (for example, aria-live or aria-busy) for the corresponding areas in your application.

UI Updates

In some cases an app needs to stay non-responsive for longer periods of time. This may be caused by the app fetching data or updating the UI. In these cases, busy indicators are shown to inform the user that the app is processing data.

The affected regions of the UI should have the property `aria-live` set to `true`. This informs the screen reader that the region's DOM structure is currently subject to change and therefore internal processing should be applied.

Event Handling

Accessible events are designed as a signaling mechanism for screen readers. An example for this is when parts of the UI update, either as a result of a direct user action (such as performing a selection) or by software (such as "*incoming mail*" messages or popups). Events like a dialog showing up or content updates of parts of the screen are handled by assigning specific roles and properties of the UI elements (for example, an HTML `<input type=checkbox>` element, would need `role="dialog"` or property `aria-live` set accordingly). Screen readers then listen to accessible events raised by browsers for these UI parts and react accordingly.

When the app is loading or fetching information, the ARIA property `aria-busy="true"` should be set for the corresponding region.

ARIA Labeling

Proper labeling of all UI elements is needed in order to ensure the screen reader announces everything correctly. Here we describe the available options and how and when they should be used.

Labeling

There are several options for labeling:

1. `Label` element with `labelFor` attribute
   - Single ID reference to the labeled control
   - Reference is maintained on the label, not on the labeled control
   - Multiple references are not possible

   Example

   ```
   <Label text="Product price" required="true" labelFor="productPriceInput"/>
   ```
2. The `aria-label` attribute
   - Text is directly provided in the attribute, no extra HTML control needed
   - Attribute is maintained on the labeled control
   - Only indirect support for multiple texts
3. The `aria-labelledBy` attribute
   - Whitespace separated list of ID references to controls
   - Attribute is maintained on the labeled control
   - Explicitly designed for multiple references

Caution

The different options cannot be used in conjunction. There is a precedence rule, which determines how the labeling attributes are prioritized and read by the screen reader. As an application developer, you need to be aware of the order in which the labeling is read by your screen reader.

Related Information

Best Practices for ARIA Labeling [page 2250]

Best Practices for ARIA Labeling

Sometimes the UI and the control usage may not allow standard ARIA labeling. Here we introduce some best practices on handling the labels in these cases.

Adding additional labeling to existing controls

Use Case:

There are two fields in a form, but there is only one label for both of them. For example, street and street no. share the same label - Street.

Solution:

Introduce the following association to controls:

Source Code

```
ariaLabelledBy : {
type : "sap.ui.core.Control",
multiple : true,
singularName : "ariaLabelledBy"
}
```

This association can be used to point to other controls that provide the needed labeling using the `aria-labelledby` property.

No suitable labeling text available on the UI which can be used with `aria-labelledby`

Use Case:

In some cases a suitable labeling text may not be available on the UI or it is hard for the application to reference it (text is contained in an inner control of a control, so the application would need to know the internals of the control).

Solution:

Use the new control `sap.ui.core.InvisibleText` which provides a hidden text and can be referenced in the `ariaLabelledBy` association.

Using the `labelFor` attribute together with `aria-labelledby`

Use Case:

The `labelFor` attribute provides additional benefits besides the pure labeling (for example, focus handling). When the label which is referenced with the `labelFor` attribute also has an `aria-labelledby` attribute of a referenced control, it is not read by the screen reader.

Solution:

A mapping table is introduced. The table is ID-based and matches label and labeled control. The `writeAccessibilityState` function of the `RenderManager` takes the mappings into account and adds the label to the `aria-labelledby` attribute of the labeled control (only when an `ariaLabelledBy` association is also present).

Internal labeling within a control

Use Case:

In some cases controls need to add additional label texts by themselves for a proper screen reader announcement, for example value states, messages, table headers and further descriptions.

Solution:

1. If the control already provides an `ariaLabelledBy` association, the additional texts must be referenced in other means (for example, with hidden texts within the control).

   > **i** Note
   >
   > It's not possible to combine `aria-labelledby` with `aria-label`.

2. If the control does not provide an `ariaLabelledBy` association, but it could, then the association should be added to follow the option above.

Writing a Control: FAQ

Why does `onmousemove` not work in my control?

SAPUI5 does not by default register an event handler for this event because of performance reasons. For example, how to register this event, see the `Dialog` control.

How can my control remember a state?

This can either be done in public properties, or in private member variables. The latter is usually defined in the `init()` method of the control and start with an underscore.

Why is my control renderer called while the control is already on the screen?

Whenever the control state changes because, for example, a property is changed, the default behavior is to rerender the control. SAPUI5 calls the control renderer with the updated state and takes care of replacing the HTML in the page. It is also possible to implement the control change explicitly in the control, which then adapts the HTML to represent the new state. In this case, the default rerendering can be suppressed (see below).

How can I avoid rerendering of my control when a property is changed?

If you call the property setter in your own code, like `this.setText("xy")`, you can instead use the generic setter defined in `Element.js` which also has the optional parameter `"bSuppressRerendering": this.setProperty("text", "xy", true)`. If the property change is done from the application, but you still want to avoid rerendering, for example, because you only need to toggle one CSS class or because the control DOM elements may not be removed and replaced, you need to overwrite the generated setter method. The generated method looks like this:

```
sap.m.Button.prototype.setText = function(sText) {
    this.setProperty("text", sText);
};
```

You also need to add the flag in your overwriting implementation:

```
sap.m.Button.prototype.setText = function(sText) {
    this.setProperty("text", sText, true);
};
```

Usually you then need to handle the visualization of the change yourself; in this case you might want to find the DOM element where your control text is located and exchange the text.

More About Controls

SAPUI5 contains controls that are provided with multiple libraries. This section contains detailed information about some of the controls beyond the information provided in the API reference.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]

Browser and Platform Support [page 20]

Busy Indicators

You use busy indicators to inform users that something is going on in the background, for example, some data is being fetched from the back end and the user has to wait. As long as the busy indicator is shown, either all or a specific part of the UI is blocked, and no user interaction is possible.

Whenever busy indication is triggered, the default delay until the busy indicator is displayed on the UI is 1000 ms (1 second). If this delay were not in place, the busy indicator would always be displayed, even if there is no negotiable waiting time.

You can choose between the following busy indicators, depending on your use case:

- `sap.ui.core.BusyIndicator`
- `sap.m.BusyDialog`
- `sap.m.BusyIndicator`

Blocking the Whole UI

You can use the `sap.ui.core.BusyIndicator` busy indicator to block the whole UI. You can set the delay in ms by specifying the number:

```
sap.ui.core.BusyIndicator.show(<number>);
```

To release the UI again, the busy indication must be hidden again. This function call hides the busy indication immediately:

```
sap.ui.core.BusyIndicator.hide();
```

API Reference: `sap.ui.core.BusyIndicator`

Busy Indication with Dialog

With the `sap.m.BusyDialog` busy indicator, you can block the whole UI like you do with `sap.ui.core.BusyIndicator`, but you can also show a dialog box. In this dialog box, you can also include a *Cancel* button that users can choose to stop the activity that's running in the background.

API Reference: `sap.m.BusyDialog`

API Overview and Samples: `sap.m.BusyDialog`

Busy Indication on Control Level

The `sap.m.BusyIndicator` busy indicator blocks specific UI areas that are defined by a control. For example, if a table in a complex UI is loading, only the table is blocked - the user can still carry on working with the rest of the UI.

If a control is set to busy, the complete control will be covered with a block layer, so no mouse events or keyboard interaction with the control are possible. If keyboard navigation is being used to step through the controls, controls that are set to `busy` are skipped and the focus jumps to the next control.

Here's how to do it:

```
    var oInput = new sap.m.Input({
        value:'Hello World'
    });
...
    oInput.setBusy(true);
```

The following code shows how you define the default state of a control as `busy` so that it will be displayed as busy when it has been rendered:

```
    var oInput = new sap.m.Input({
        value:'Hello World',
        busy: true
    });
```

To release the control's busy state again, the same API can be used. This has to be done by the application after some data has been loaded, for example with the following command:

```
oMyListBox.setBusy(false);
```

To change the default delay of the local busy indicator, use:

```
oMyListBox.setBusyIndicatorDelay(<number>);
```

API Reference: `sap.m.BusyIndicator`

API Overview and Samples: `sap.m.BusyIndicator`

API Overview and Samples: `sap.ui.core.Control`

Cards

A card is a design pattern that displays the most concise pieces of information in a limited-space container. Similar to a tile, it helps users structure their work in an intuitive and dynamic way.

Overview

Cards are composite controls that follow a predefined structure and offer content in a specific context. Cards contain the most important information for a given object (usually a task, or a list of business entities). You can

use cards for presenting information, which can be displayed in flexible layouts that work well across a variety of screens and window sizes.

With the use of cards, you can group information, link to details, or present a summary. As a result, your users get direct insights without the need to leave the current screen and choose further navigation options.

> **i Note**
>
> Keep in mind, that the hereby described card controls are not related to the Overview Page Card, which is intended to be used in the context of SAP Fiori Elements. For more information, see Developing Apps with SAP Fiori Elements [page 1535] and Overview Page Card [page 1934].

The following table provides an overview of the two main types of card controls in the SAPUI5 framework:

Table 113: Overview of Cards and Supported Features

Feature Supported	(Integration Card) `sap.ui.integration.widgets.Card` [page 2255]		(Freestyle Card) `sap.f.Card` [page 2257]
	Analytical, List, Object, Table, and Timeline cards	Component card	
Fiori 3 card UX	✓	✓[1]	✓
Cross product integration	✓	✓	✗
Cross HTML product integration	✓	✓	✓
FLP / cFLP integration	✓	✓	✓[2]
Independent of SAPUI5 runtime	✓	✗	✗
Can implement application logic	✗	✓	✓

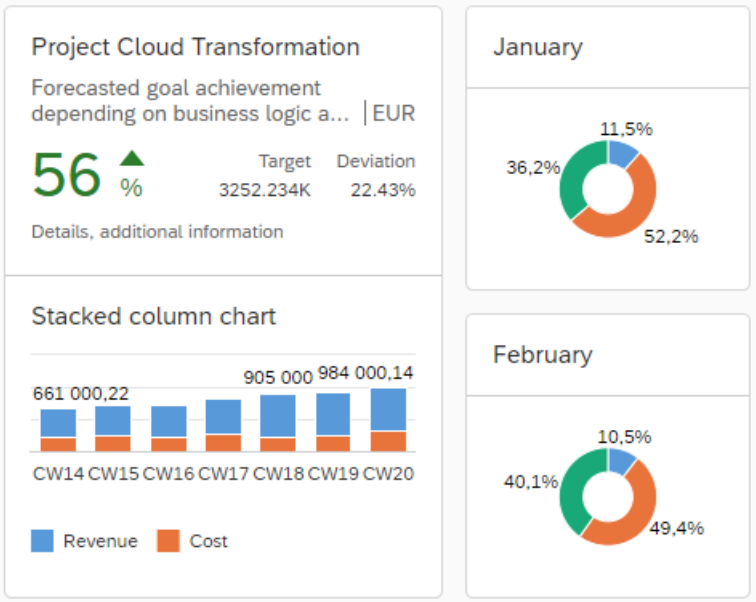
1) Depending on the implementation.
2) With `CustomElement`.

`sap.ui.integration.widgets.Card` (Integration Card)

The `sap.ui.integration.widgets.Card` is a self-contained user interface element, connected to a manifest, and used as a widget. It consists of three elements: card container, card header area, and content area.

- The container is the base and subdivides in a header and content area. It has a white background color and a border with radius.

- The header area indicates what the card is about and can function as a navigation area for opening the underlying source. It also serves as a counter showing the number of items on the card in relation to the total number of relevant items (in case multiple items are shown in the content area). It can have a title, a subtitle, an icon, and a status indicator. If the header is of type `Numeric`, it may have different attributes, describing a KPI.
- The content area represents data from the underlying source. It uses the interaction and visualization from the embedded controls. It depends on the card type.



Usage

The integration card is defined in a declarative way, using a `manifest.json` so that it can be easy to integrate and reuse it.

As a card developer, you describe it in its `manifest.json` file by defining its header, content, data source, and possible actions. As an app developer, you integrate the card in your app and define its dimensions (`height` and `width` properties) and behaviour for the declared actions (`action` event).

To use the `sap.ui.integration.widgets.Card`, you should pass the path to the `manifest.json` file of the card:

```
<mvc:View xmlns:w="sap.ui.integration.widgets">
<w:Card manifest="./demo/manifest.json" />
```

For more information and examples on the usage, see the API Reference and the Samples.

Content Types

Cards can be transactional (list, table, object, unstructured content, timeline) and analytical (line, donut, [stacked] column, stacked bar chart). Each card is designed in a different style and contains various content formats.

Card Type	Description
Analytical card[1]	Used for data visualization. Typically, it is defined by a numeric header and analytical data content. For example, chart types, such as line chart, donut chart, (stacked) column chart, (stacked) bar chart. For more information on the analytical card, see the SAP Fiori Design Guidelines.
Component card (Experimental)	Displays multiple controls and is used as a custom approach for use cases which do not fit in other card types and structures. The content area of the unstructured content card can be moved to the top. **i Note** In contrast to the other integration card types, the structure and behavior of the Component card are custom-definable and are following the established SAPUI5 Component model. For more information, see Components [page 720].
List card	Displays multiple list items of all kind. Aggregated information can also be visualized with a line item. The counter in the header area is required for this type of card.
Object card	Displays the basic details for an object, for example, a person or a sales order.
Table card	Displays a set of items in table format. For more information on the table card, see the SAP Fiori Design Guidelines.
Timeline card[2]	Displays time-related content.

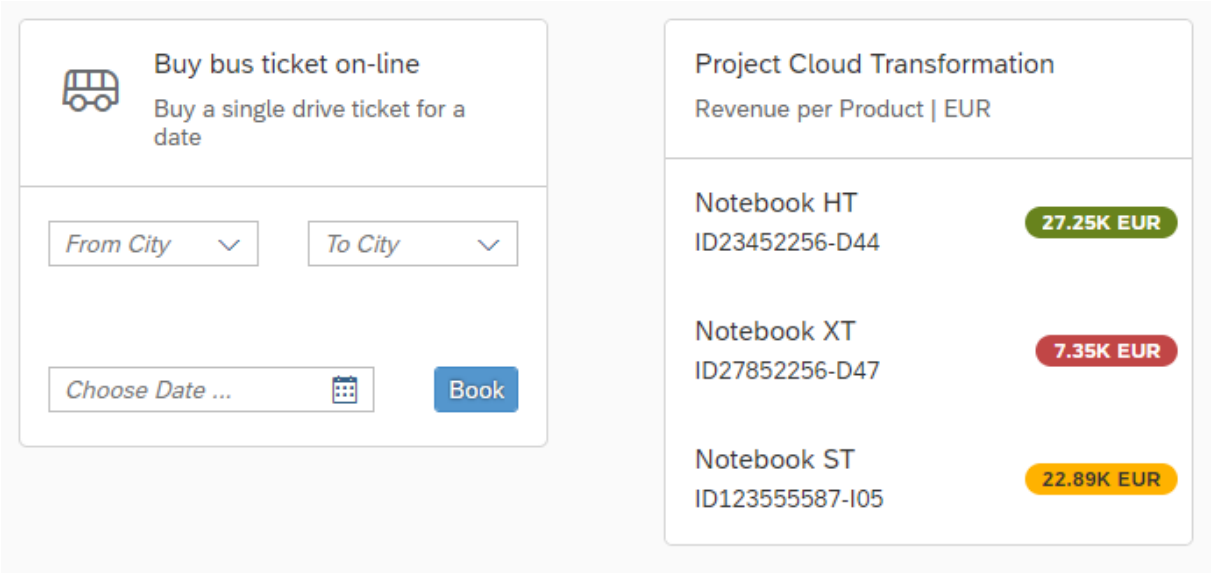
Limitations:
1) The analytical card is using `sap.viz.ui5.controls.VizFrame` charts which are part of SAPUI5 and are not available in OpenUI5.
2) The timeline card is using the `sap.suite.ui.commons.Timeline` control which is part of SAPUI5 and is not available in OpenUI5.

`sap.f.Card` (Freestyle Card)

The `sap.f.Card` control provides more freedom in choosing the structure and the controls which you can include. The data is provided by the application with the use of data binding, thus the control can be used just as a renderer.

In contrast to the self-contained and manifest-driven Integration Card (`sap.ui.integration.widgets.Card`), where the design, data retrieval, and rendering are predefined and encapsulated for better integration and reuse, with `sap.f.Card` you can decide and compose the card content area according to your needs.

The `sap.f.Card` consists of three elements: a container with background color and rounded corners, a header, and content.



The header is predefined and can be an instance of either `sap.f.cards.Header` or `sap.f.cards.NumericHeader`. The content area can be built with a desired combination of the standard SAPUI5 controls.

Usage

Considering the general purpose of the cards (to represent most important assets of a particular business object in a limited size UI element), it is not recommended to have very complex designs and heavy application-like interactions implemented in a `sap.f.Card` control.

Example:

```
<f:Card
    class="sapUiMediumMargin"
    width="300px">
    <f:header>
        <card:Header
            title="Project Cloud Transformation"
            subtitle="Revenue per Product | EUR"/>
    </f:header>
    <f:content>
        <List
            showSeparators="None"
            items="{path: '/productItems'}" >
            <CustomListItem>
            <HBox
                 alignItems="Center"
                justifyContent="SpaceBetween">
                <VBox class="sapUiSmallMarginBegin sapUiSmallMarginTopBottom" >
                    <Title level="H3" text="{title}" />
                        <Text text="{subtitle}"/>
                </VBox>
                <tnt:InfoLabel
                    class="sapUiTinyMargin"
                    text="{revenue}"
                    colorScheme= "{statusSchema}"/>
            </HBox>
            </CustomListItem>
        </List>
    </f:content>
</f:Card>
```

Related Information

Components [page 720]
Descriptor for Applications, Components, and Libraries [page 734]
Grid Controls [page 2261]
API Reference: `sap.f.Card`
API Reference: `sap.ui.integration.widgets.Card`

Date and Time Related Controls: Data Binding

Date and time related controls can be bound to an `OData` service.

Introduction

According to the `OData Version 2.0` specification, the following date and time related primitive data types exist:

1. `Edm.Time` - represents the time as, for example, PT17H15M, which corresponds to 17:15:00.
2. `Edm.DateTime` - represents the date and time as, for example, 2001-12-21T12:00, which corresponds to 12:00 PM on Dec 21, 2001.
3. `Edm.DateTimeOffset` - represents the date and time as an offset in minutes from GMT, with values from 12:00:00 midnight, January 1, 1753 A.D. through to 11:59:59 P.M, December 9999 A.D. For example, 1999-01-01T23:01:00Z corresponds to 11:01:00 PM on January 1, 1999.

OData Version 2.0 Binding Types

Binding of time values to `TimePicker`

1. With a dedicated `Edm.Time`:

```
<TimePicker displayFormat="short"
            value="{ path: 'EntryTime', type:
'sap.ui.model.odata.type.Time'}"/>
```

   The sample shows how to bind time values at the backend (for example PT11H33M55S for 11:33:55). Note that there is a dedicated data binding type that recognizes the `Edm.Time` format and can handle time conversions in both directions - from and to the backend. When you are working with this data binding type and you choose a value using the `TimePicker` control, the same value will be sent to the backend, that is, no timezone conversions will be applied to the value.
2. As a string:

```
<TimePicker value="{EntryTimeString}" valueFormat="HH:mm:ss"/>
```

In the above example, the apps give the `TimePicker` information about the exact format with which the time values are stored in the backend. The whole coding is string-based (the data field is a string, and the EDM type is an `Edm.String`) and no conversion is done.
With the introduction of the property `support2400` in version 1.54, this option may be used also if the apps need to differentiate between the beginning of a day (00:00:00) and the end of a day (24:00:00). For more information, see the API Reference and the Samples.

Binding of date values to `DatePicker`

```
<DatePicker value="{
    path: 'EntryDate',
    type: 'sap.ui.model.odata.type.DateTime',
    constraints: {
        displayFormat: 'Date'
    }
}" />
```

If you have date values at the backend, you should bind them as shown in the sample above. When exposing its database field (`EntryDate`) via `OData`, date values can be exposed both as `Edm.DateTime` and `Edm.DateTimeOffset`. The syntax for binding the `DatePicker` value property is the same in both cases.

Above you can see an example of the `displayFormat` constraint. It specifies if the given value should be interpreted as Date or DateTime (default). If Date is specified, the binding type performs the UTC conversion, which is always on. In other words, any local date chosen by the end user will be considered as a UTC date and sent as a UTC date to the backend.

Binding of date and time values to `DateTimePicker`

```
<DateTimePicker
        value="{ path: 'EntryDateTime', type:
'sap.ui.model.odata.type.DateTime' }"/>
```

In this example, the binding type specifies that the backend data will be interpreted as a date and time field.

> **i Note**
>
> JavaScript provides only one object for working with dates and times – Date, which contains both date and time information. Currently, all dates that are API properties in the `DatePicker`, `TimePicker`, `DateTimePicker`, `PlanningCalendar` and `Calendar` controls use local time. For example, if a user chooses 19.02.2018 as a date from the `DatePicker`, the app developer calls the `getDateValue()` method. In this case they will get 19.02.2018 00:00:00 local time. The disadvantage here is that by default this value will be sent to the backend in UTC, which may change the date by +/- one day.

Binding of date ranges

1. With a dedicated EDM type (`Edm.DateTime`, `Edm.DateTimeOffset`):

   ```
   <DateRangeSelection value="{parts: [{path: 'EntryDate'}, {path:
   'EntryDateTimeOffset'}], type: 'sap.ui.model.type.DateInterval',
   formatOptions: { UTC: true, format: 'yMd' }}" />
   ```

   If you have two dates in the backend represented as `Emd.DateTime` and/or `Edm.DateTimeOffset`, you can bind them to the `DateRangeSelection` control as shown above.

2. As a string:
   If you have the date range provided with a single date formatted string field in the backend (for example, July 29, 2015 - July 31, 2015), use the value property in the following way:

```
<DateRangeSelection
        value="{path: 'EntryDateRange', valueFormat: 'MMM d, y'}"/>
```

Note that you need to tell the `DateRangeSelection` the format of the dates stored in the backend. The format should denote the format of a single date. If the delimiter is different than "-", you need to additionally specify it in the property delimiter.

Related Information

Primitive Data Types in the OData Documentation
Formatting, Parsing, and Validating Data [page 854]
Step 10: Property Formatting Using Data Types [page 239]

Grid Controls

SAPUI5 provides several different grid layouts that are suitable for different use cases.

`sap.f.GridContainer`

The `sap.f.GridContainer` is a layout control used to align tiles, cards, or other controls in configuration, such as an overview page. It provides a regular grid system in which all rows have the same height and all columns have the same width. Each item can be configured to take different number of rows and columns inside that mesh. If rows span is unknown for an item, it is automatically calculated by the `GridContainer`, based on the height of the item.

`sap.f.GridList`

The `sap.f.GridList` is a layout control that provides the flexibility to display list items in a two-dimensional grid. It extends the `sap.m.ListBase` control and therefore receives all of its features. The layout used is based on the CSS display grid and the control has a default configuration that displays the list items in a grid.

`sap.ui.layout.cssgrid.CSSGrid`

The `sap.ui.layout.cssgrid.CSSGrid` is a layout control, used to create full-page layouts or user interface elements. It is based on the browser-native CSS display grid and works by the HTML standard specification of a

grid. This grid is two-dimensional, meaning that it is possible to specify both rows and columns. The dimensions and position of a single item can be configured, for example, an item can take two rows and two columns from the grid. The control can be used along with the `sap.m.FlexBox` control as a one-dimensional alternative for layouting.

`sap.ui.layout.Grid`

The `sap.ui.layout.Grid` control defines how many columns are displayed depending on the available screen size with a maximum of 12. The height of a single row is always based on the content of the highest item in that row.

Overview of Grids and Supported Features

Feature Supported?	`sap.f.GridContainer`	`sap.f.GridList`	`sap.ui.layout.cssgrid.CSSGrid`	`sap.ui.layout.Grid`
Complies with the grid specification according to the HTML Standard	✗	✓	✓	✗
Supports Internet Explorer and version 15 or below of Microsoft Edge	✓	✗[1]	✗[1]	✓
Number of columns supported	Unlimited	Unlimited	Unlimited	Up to 12
Can configure row height, column width, and gap dimensions	✓	✓	✓	✗
Supports auto calculation of rows per item	✓	✗	✗	✗
Can fill empty spaces in the grid	✗	✓	✓	✗
Items flow direction	Horizontal only	Horizontal and vertical	Horizontal and vertical	Horizontal only
Can configure item position	✗	✓	✓	✗
Can configure item dimensions The ability to define how many rows and columns an item should take.	✓	✓	✓	✗
Supports columns breathing	✗	✓	✓	✓

Feature Supported?	sap.f.GridContainer	sap.f.GridList	sap.ui.layout.cssgrid.CSSGrid	sap.ui.layout.Grid
Supports templating The possibility to use one of the predefined layout templates or to create a custom template for specific layouts.	✗	✓ 2	✓ 2	✗
Supports screen-size breakpoints	✓	✓ 2	✓ 2	✓
Supports container-size breakpoints	✗	✓ 2	✓ 2	✓
Supports indentation	✗	✗	✗	✓
Can control items visibility based on breakpoints	✗	✗	✗	✓
Supports keyboard handling	✗	✓	✗	✗
Supports growing	✗	✓	✗	✗
Supports sorting, filtering and grouping	✗	✓	✗	✗
Supports headers and footers	✗	✓	✗	✗
Supports selection and highlighting	✗	✓	✗	✗

1) Microsoft Internet Explorer 11 has limited support for the underlying `display:grid` technology. Specific layouts are enabled for Internet Explorer 11. Microsoft Edge supports `display:grid` for versions above 16.
2) By using the `customLayout` aggregation.

Related Information

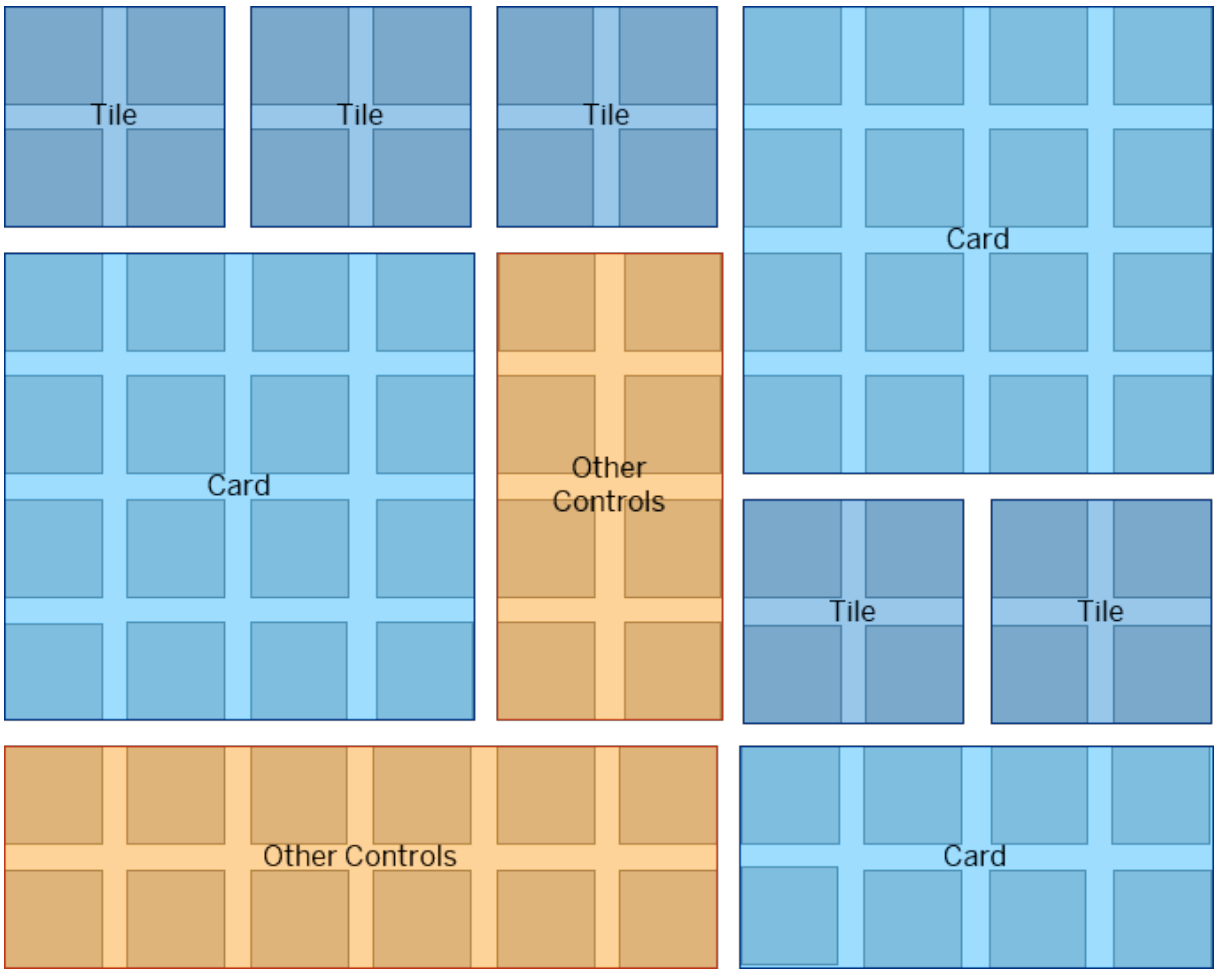
API Reference: `sap.ui.layout.cssgrid.CSSGrid`
Samples: `sap.ui.layout.cssgrid.CSSGrid`
API Reference: `sap.ui.layout.Grid`
Samples: `sap.ui.layout.Grid`
API Reference: `sap.f.GridContainer`
Samples: `sap.f.GridContainer`
API Reference: `sap.f.GridList`
Samples: `sap.f.GridList`

sap.f.GridContainer

The `GridContainer` allows you to align tiles, cards or other controls in configuration, such as an overview page.

Overview

The `GridContainer` allows the positioning of items (Tiles, Cards, or others) in a two-dimensional mesh. The mesh consists of rows with the same height and columns with the same width. Those height and width sizes along with the gap size are configurable.



The control provides responsiveness and automatically aligns the items depending on the available space.

- In contrast to the `sap.ui.layout.cssgrid.CSSGrid`, the `GridContainer` allows the rows per item to be automatically increased, if the item does not fit and is cut off.
- The `GridContainer` provides control over the behavior of the items if they are smaller in height than the given space. For example, if an item has a width of 4 rows, but its height is only 3.5 rows, then the item could either remain 3.5 rows or stretch to 4 rows. This behavior can be controlled through the `snapToRow` property.

Although the `display: grid` CSS property is not supported by Internet Explorer 11 and Microsoft Edge version 15 and below, there is a polyfill implemented for `GridContainer` and it works in these browsers.

The `GridContainer` also supports layout breakpoints based on the screen size. As a result, on smaller screens, the gaps, rows, and columns can be smaller. You can configure them through different `GridContainerSettings` for the different layouts.

Related Information

API Reference: `sap.f.GridContainer`
Samples: `sap.f.GridContainer`

sap.f.GridList

The `GridList` allows you to use different types of layouts responsible for the positioning and responsiveness of the content.

Overview

A grid is a two-dimensional structure composed of a series of intersecting vertical and horizontal guidelines used to structure content. The grid serves as a framework in which you can organize controls in a consistent way throughout the design. Dividing a design space into a grid can help position individual elements in a visually appealing manner, facilitate the representation of a user flow, and make information more comprehensible and accessible.

With the new `sap.f.GridList`control, you can easily organize and align your content according to your preferences.

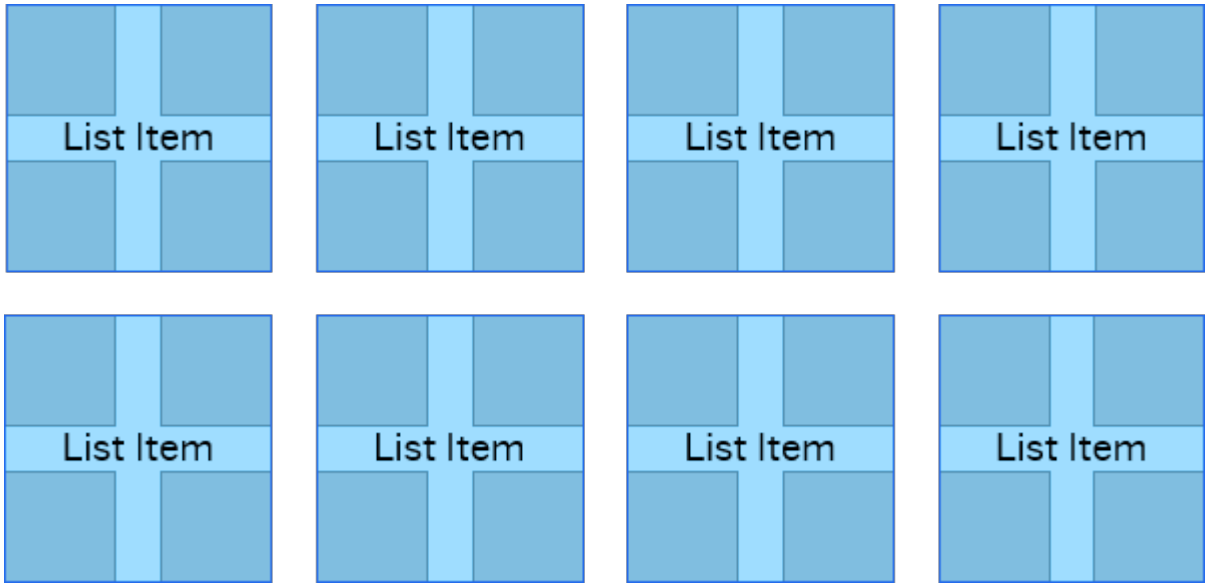


Figure 341: An example of a uniform grid layout, where all the grid items in a grout take the same dimensions

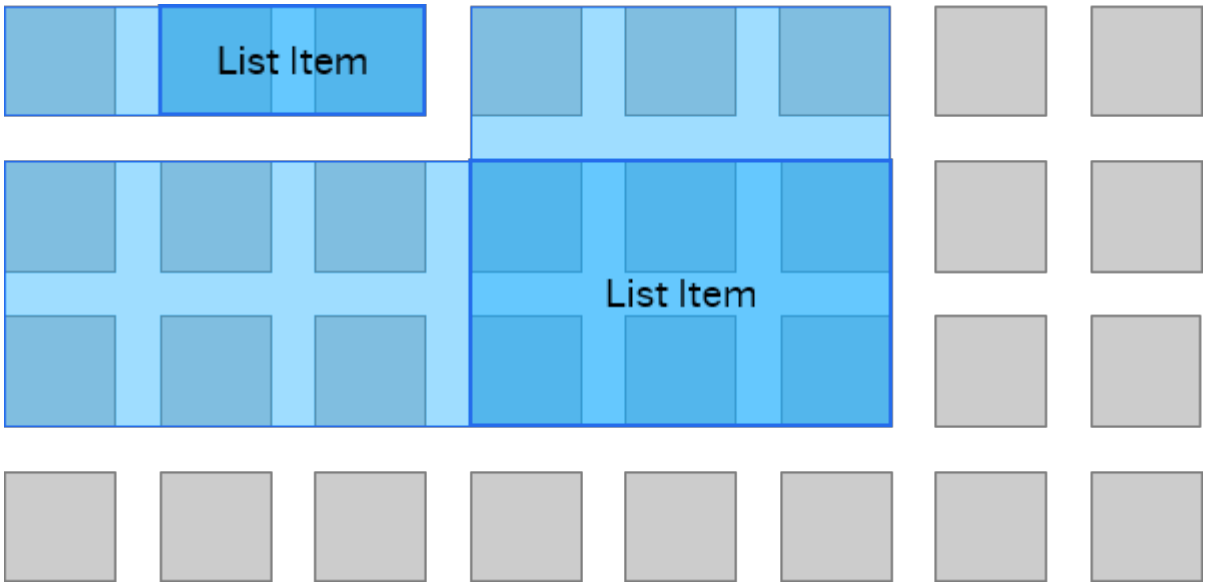


Figure 342: GridList allows for high flexibility layouts, where the app developer can decide on the specific placement and sizing of the grid items

Layouts

`sap.f.GridList` allows application developers to display list items in a two-dimensional grid where the visual layout/display options can be configured flexibly using predefined and custom templates. The layout used is based on the CSS display grid and has a default configuration that displays the list items in a grid.

`sap.ui.layout.GridBoxLayout`

`sap.ui.layout.GridBoxLayout` is a layout that allows you to position controls in a grid, relative to one another, using constraints defined by its `boxWidth`, `boxMinWidh` or `boxesPerRowConfig` properties.

- `boxMinWidth` allows the items inside `sap.ui.layout.GridBoxLayout` to accommodate the available width without allowing them to be smaller than the specified `boxMinWidth`.
- `boxWidth` sets the exact width of the items inside `sap.ui.layout.GridBoxLayout` regardless of the remaining space available in the row.
- `boxesPerRowConfig` allows the alignment and specification of the number of items in a row, depending on the browser viewport size.

This particular layout works with Internet Explorer 11, due to an implemented regressive enhancement (polyfill) in place of `display:grid`.

> **i Note**
>
> - The height of all items is set to the height of the highest item.
> - If the `boxWidth` property is set, `boxMinWidth` and `boxesPerRowConfig` properties are ignored. If the `boxMinWidth` property is set, `boxesPerRowConfig` property is ignored.

Related Information

API Reference: `sap.f.GridList`
Samples: `sap.f.GridList`

sap.ui.layout.cssgrid.CSSGrid

The `CSSGrid` is a flexible layout, based on the two-dimensional, browser-native grid system, using the standard CSS property `display` with value `grid`. It allows for flexible positioning of user interface elements, inside horizontal and vertical grid structures.

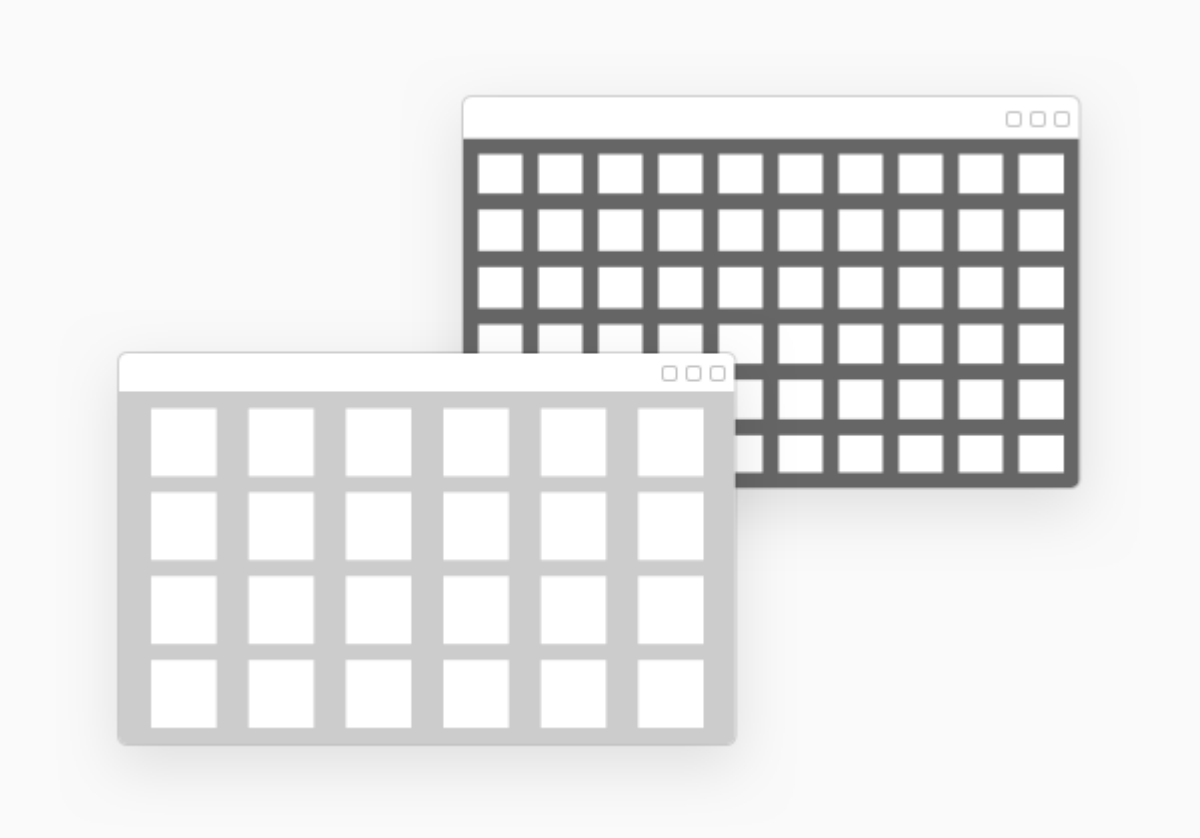
Overview

The `CSSGrid` is a powerful grid system that can be configured to display a variable number of columns and rows, depending on the available space. You can achieve complex but flexible layouts for both full-page and in-container page layouts. With the `CSSGrid` control, you can define a logical two-dimensional grid structure.

> **i Note**
>
> You can use the `CSSGrid` layout to divide a page into regions or define the size, position, and layer between parts of a control, and easily align elements into columns and rows.
>
> For more information, see https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout
>
> The display: grid layout is not fully supported by all browser platforms. It does not work with Internet Explorer 11.
>
> For more information, go to www.caniuse.com and search for: **`CSS grid layout`**

You can achieve the desired layout, using a flexible set of configuration properties, including responsive positioning of grid cells along different container break-point sizes.

The grid cells created by the grid layout are empty containers that can be filled with controls.

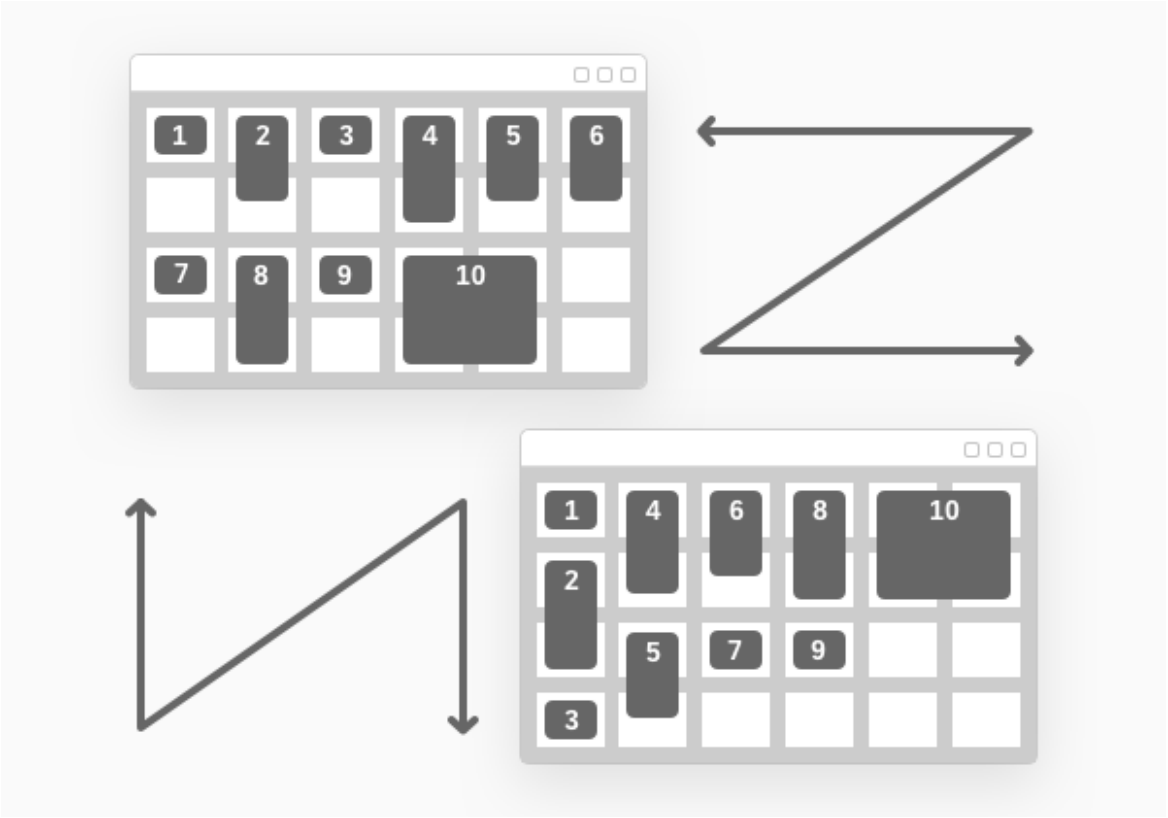


Usage

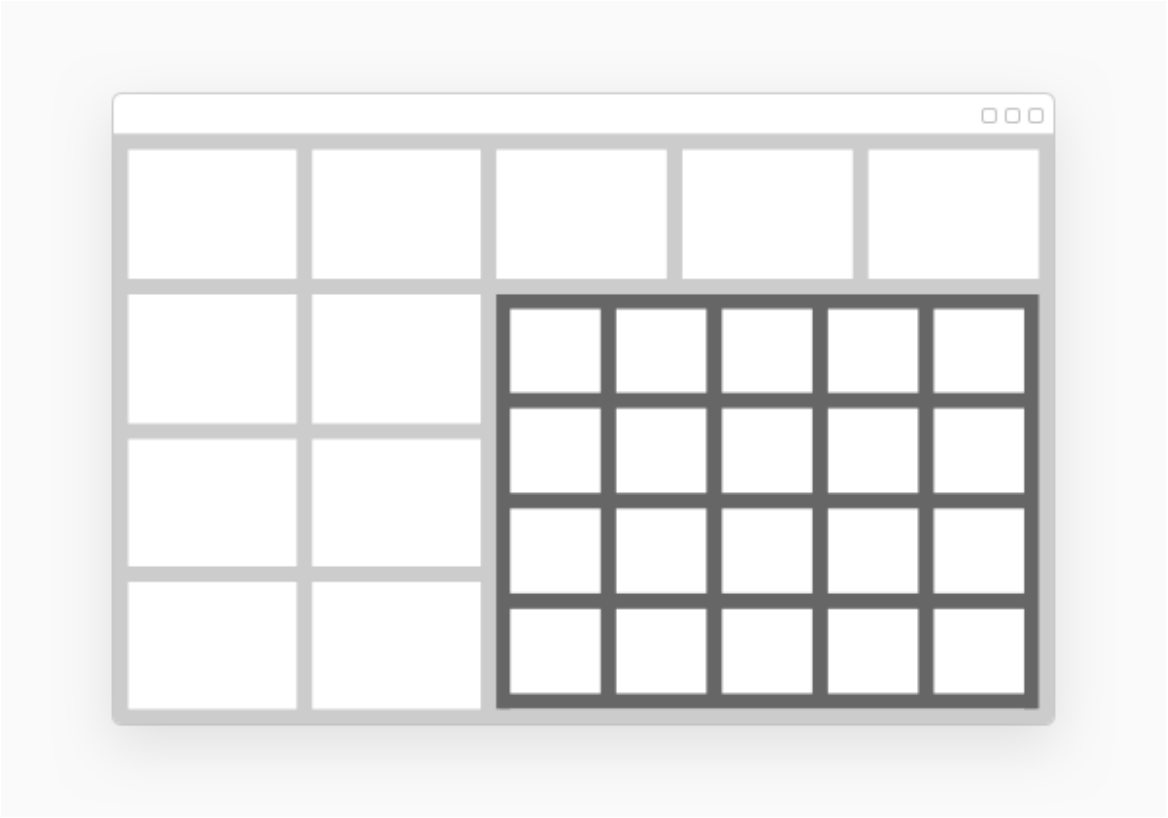
The grid layout allows for the placement of multiple elements on the user interface to display structured content. This helps to maintain one coherent experience within pages as well as across all pages and/or layouts.

Features and behaviors are configurable to enable the grid layout for a variety of use cases:

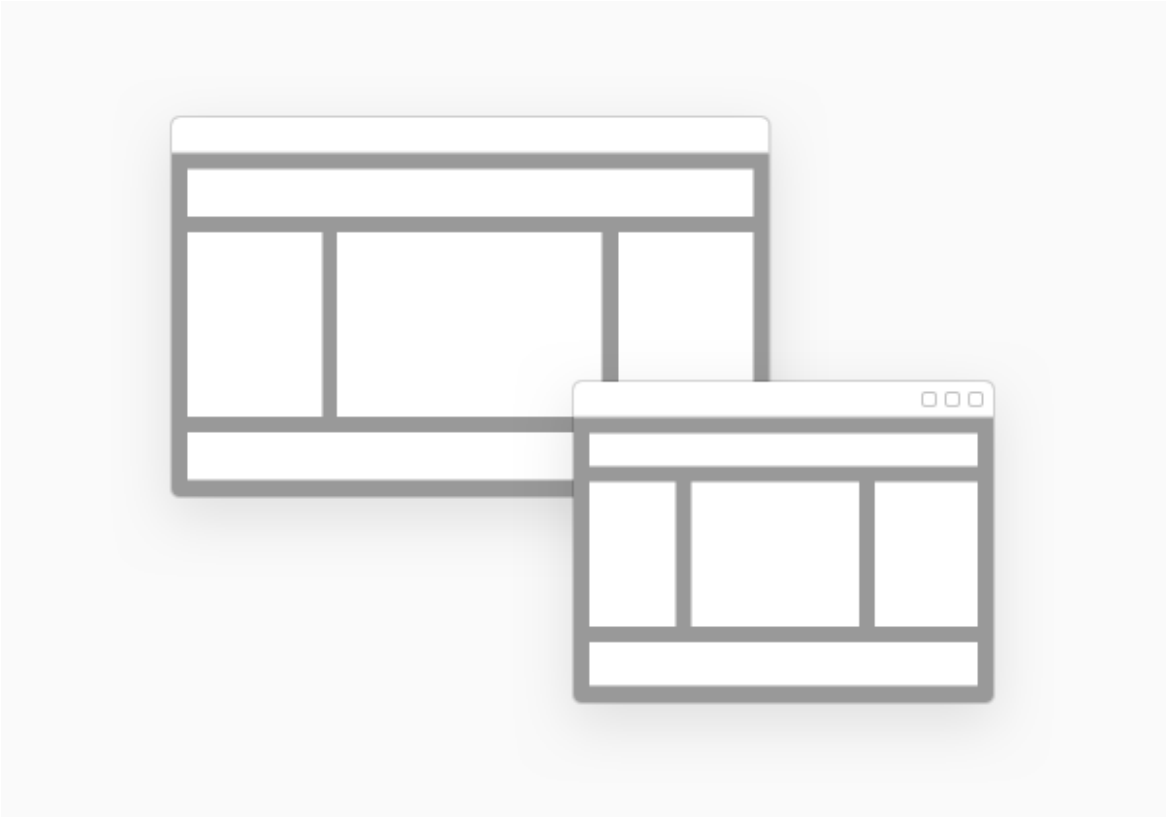
- You can define and control the elements flow, positioning the elements either horizontally in rows or vertically in columns.



- Nesting:
  - The grid layout can be used standalone or inside other layout containers, such as another page, a header or a dialog.
  - The grid layout supports nesting, which allows for the placement of a grid layout inside another grid layout.



- Size/dimensions configuration: you can configure either every row and column specifically, or using a template definition (`gridTemplateRows`, `gridTemplateColumns`), and you can configure the gaps too by using the `gridGap`, `gridRowGap`, `gridColumnGap` properties.
- Responsiveness: The ability to adjust grid size and reorganize the grid content, depending on pre-defined break points. (S, M, L, XL).



Related Information

API Reference: `sap.ui.layout.cssgrid.CSSGrid`
Samples: `sap.ui.layout.cssgrid.CSSGrid`

sap.ui.layout.Grid

A flexible layout that positions its items in a 12-column flow layout.

Overview

The `sap.ui.layout.Grid` is a powerful grid system that can be configured to display a variable number of columns depending on the available screen size. You can achieve complex but flexible layouts and line breaks for extra large, large, medium, and small-sized screens, such as desktop, tablet, and mobile.

With this control, you can define how many items are displayed per row depending on the available screen size, with a maximum of 12. The height of a single row is always based on the content of the highest item in that row.

The flow direction of the items is horizontal only. For example, if the control is configured to display six items per row, the seventh item is displayed on the next row.

Usage

You can use the `sap.ui.layout.Grid` on its own or in combination with the `sap.ui.layout.GridData`. The APIs in the `sap.ui.layout.Grid` apply for all grid items while with the `sap.ui.layout.GridData` you can manipulate individual grid items.

Using `sap.ui.layout.Grid` on its own

`defaultSpan` and `defaultIndent` are the main two properties that enable you to define a specific layout for the grid. The number of grid columns is always 12 but the span and indentation of the items determine how many are displayed in one row.

Property	Example Values	Description

`defaultSpan`	`"XL3 L4 M6 S12"`
	`"XL6 M6"`
	`"S2"`
		Determines the span of the items for the different screen/container sizes (XL, L, M, and S). Each item can be set to span over several grid columns (from 1 to 12).

`defaultIndent`	`"XL4 L3 M2 S1"`
	`"XL2 L1"`
	`"S6"`
		Defines the number of columns with which each item is indented for the different screen/container sizes (XL, L, M, and S). Each item can be indented with several grid columns (from 0 to 11).

More properties are available for defining the horizontal and vertical spacing between the grid items, setting a specific width, defining the position for the grid as a whole, and so on. For more information, see the available properties in the API Reference.

Using `sap.ui.layout.GridData` to manipulate individual grid items

To achieve a layout where the individual grid items have a different configuration, use `sap.ui.layout.GridData`.

In addition to individual span and indentation, there is a large number of properties that allow for setting the visibility of items and adding line breaks. For more information, see the available properties in the API Reference.

Examples

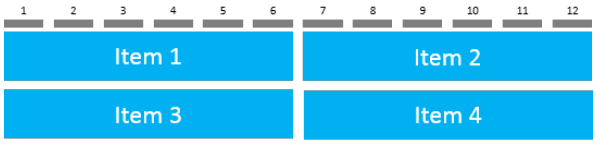
```
<l:Grid
        defaultSpan="XL3 L4 M6 S12">
    <Image src="/item1.png"
width="100%"></Image>
    <Image src="/item2.png"
width="100%"></Image>
    <Image src="/item3.png"
width="100%"></Image>
```

XL container/screen size (one item spans over 3 columns)

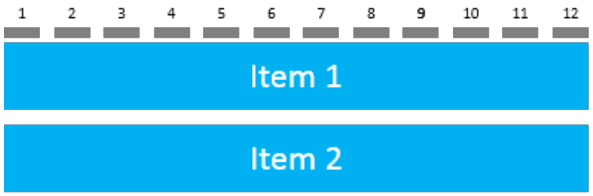
L container/screen size (one item spans over 4 columns)

```
    <Image src="/item4.png"
width="100%"></Image>
</l:Grid>
```

M container/screen size (one item spans over 6 columns)



S container/screen size (one item spans over 12 columns)



```
<l:Grid
        defaultSpan="XL3 L4 M6 S12"
        defaultIndent="XL1 L1 M1 S1">
    <Image src="/item1.png"
width="100%"></Image>
    <Image src="/item2.png"
width="100%"></Image>
    <Image src="/item3.png"
width="100%"></Image>
    <Image src="/item4.png"
width="100%"></Image>
</l:Grid>
```

XL container/screen size (items are indented with 1 column)

```
<l:Grid
        defaultSpan="XL3 L4 M6 S6">
    <Image src="/item1.png"
width="100%"></Image>
    <Image src="/item2.png"
width="100%">
        <layoutData>
            <l:GridData span="XL4"/>
        </layoutData>
    </Image>
    <Image src="/item3.png"
width="100%"></Image>
    <Image src="/item4.png"
width="100%">
        <layoutData>
            <l:GridData span="XL2"/>
        </layoutData>
    </Image>
</l:Grid>
```

Individual span for item2 and item4 using `sap.ui.layout.GridData`

Related Information

API Reference: `sap.ui.layout.Grid`
API Reference: `sap.ui.layout.GridData`
Samples: `sap.ui.layout.Grid`

Hyphenation for Text Controls

SAPUI5 offers the possibility to hyphenate words in multiline texts when controls are in wrapping mode.

Overview

The hyphenation feature is an intelligent wrapping capability for optimal visual rendering of multiline text. It is especially useful for longer text instances in any type of container.

It is enabled through the `wrappingType` property for the following controls:

- `sap.m.Text`
- `sap.m.Title`
- `sap.m.Label`

Use Cases

Using the integrated hyphenation through the `wrappingType` property directly in the text controls

All three controls have a `wrapping` property that determines text wrapping. By default, it is set to `true` for the `sap.m.Text` control and to `false` for `sap.m.Label` and `sap.m.Title`. Setting the `wrapping` property of these controls to `true` allows you to use the `wrappingType` property which enables hyphenation. It's an `enum` property with two possible values:

- `WrappingType.Normal` – The text wraps on several lines keeping the words in their entirety.
- `WrappingType.Hyphenated` – The text wraps on several lines separating words into syllables and marking the syllabification with a hyphen.

Example:

```
new sap.m.Label({
  text: "Liquiditätspositionshierarchie Datenänderungsbelege",
  wrapping: true,
  wrappingType: sap.m.WrappingType.Hyphenated
});
```

Using the `sap.ui.core.hyphenation.Hyphenation` API

This class provides methods to evaluate the possibility of using browser-native hyphenation or to initialize and use a third-party hyphenation module. Using this API you can check if browser-native hyphenation is supported for a particular language.

If browser-native hyphenation is not supported, you can directly use this API to hyphenate texts. A third-party library named Hyphenopoly is used in that case.

As the `sap.ui.core.hyphenation.Hyphenation` class is a singleton, an instance should be acquired from the `getInstance` method.

Example:

```
var oHyphenationApi = sap.ui.core.hyphenation.Hyphenation.getInstance();
if (!oHyphenationApi.canUseNativeHyphenation("en")) {
  oHyphenationApi.initialize("en").then(function() {
    console.log(
      oHyphenationApi.hyphenate("An example text to hyphenate.", "en")
    );
  });
}
```

Manual control of hyphenation when third-party resources are loaded.

By default, the text controls load any required third-party resources at a later state which can lead to flickering of the first visible text control between its unhyphenated and hyphenated states. To prevent this, you can prepare the third-party library before rendering your app.

Example:

```
sap.ui.core.hyphenation.Hyphenation.getInstance()
  .initialize()
  .then(function() {
    // continue with application initialization/rendering
  });
```

How It Works for Text Controls

We've taken the following dynamic approach to hyphenation:

- We use browser-native hyphenation when possible;
- If browser hyphenation is not possible – we use a third-party tool called Hyphenopoly through the `sap.ui.core.hyphenation.Hyphenation` API.

Once you've set the control property to `WrappingType.Hyphenated`, the control instance checks dynamically whether the browser you're using supports hyphenation. If yes, it enables the CSS hyphenation and lets the browser perform it. If it doesn't, the process is redirected to a third-party tool and the hyphenation module is asynchronously loaded together with the specific resources per language. This is done through the `sap.ui.core.hyphenation.Hyphenation` API, which is responsible for loading all resources in an async mode and for the dynamic initialization of the third-party library with the language resources and some required configurations. It also caches the rules internally for future use.

When the framework makes the choice whether browser-native hyphenation or third-party hyphenation should be used, it logs a message in the console for more information about what was decided.

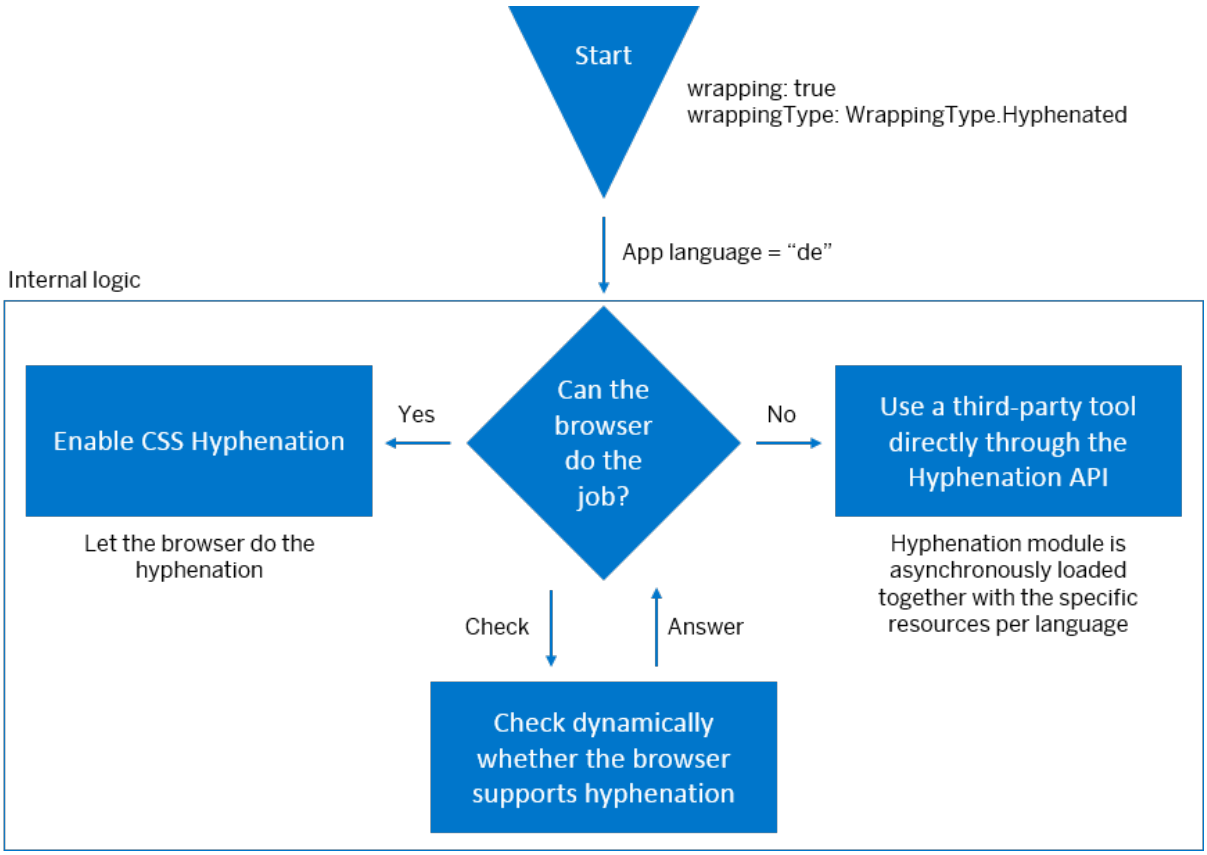


Figure 343: Hyphenation Workflow

Supported Languages

> ⚠ Caution
>
> Note that as the hyphenation feature uses third-party and browser-native tools, we are not responsible for any grammatical incorrectness or inconsistencies of the hyphenation. Also, the variety of supported languages is outside the scope of our control and may be subject to future changes.

SAPUI5 provides hyphenation through the `hyphens` CSS property or the third-party tool Hyphenopoly.

The following table provides a list of languages supported by the third-party tool Hyphenopoly (version 2.4.0). Texts in all other languages are hyphenated only if the used browser supports the `hyphens` CSS property for the specified language.

Language	Code
Bulgarian	bg
Catalan	ca
Croatian	hr
Danish	da

Language	Code
Dutch	nl
English (US)	en
Estonian	et
Finnish	fi
French (FR)	fr
German	de
Greek	el
Hindi	hi
Hungarian	hu
Italian	it
Lithuanian	lt
Norwegian	no
Portuguese (BR)	pt
Russian	ru
Slovenian	sl
Spanish (ES)	es
Swedish	sv
Thai	th
Turkish	tr
Ukrainian	uk

Related Information

API Reference: sap.m.Text
API Reference: sap.m.Title
API Reference: sap.m.Label
API Reference: sap.m.WrappingType
API Reference: sap.ui.core.hyphenation.Hyphenation
Hyphenopoly on GitHub

`hyphens` CSS property and browser compatibility on MDN Web Docs

Semantic Pages

The semantic page controls help the app designers and developers implement and comply with the SAP Fiori design guidelines more easily.

The benefit of these controls is to separate the design requirements from the actual app implementations, allowing these requirements to be adjusted centrally for the controls without the apps having to reimplement them with each change.

The main functionality of the semantic pages is to predefine the placement, behavior and styles of the page elements, such as titles and actions. Content specified in the aggregations is automatically positioned in dedicated sections of the header or the footer of the pages.

There are two separate and non-dependent `semantic` namespaces in the main SAPUI5 libraries:

- `sap.f.semantic` (implements SAP Fiori 2.0)
- `sap.m.semantic`

Both of them extend and enhance the base pages of the library they are in - `sap.m.Page` and `sap.f.DynamicPage`. They allow app developers to quickly add controls to the page that correspond to common operations, such as add, delete and filter. They only have to specify the action type, and the required styling and positioning is automatically added internally.

Semantic Page (sap.f)

The `sap.f.semantic.SemanticPage` is an enhanced `sap.f.DynamicPage` that implements the SAP Fiori 2.0 design guidelines.

For more information about this control, see the API Reference and the samples.

Features

The `SemanticPage` exposes all the `DynamicPage`'s API, which means you can do everything that you are currently able to do with the `DynamicPage`.

Title

The following aggregations are available to control the semantic in the title of the page:

- Semantic text actions:
  - `titleMainAction`
  - `deleteAction`
  - `copyAction`

- `addAction`

- Semantic icon actions:
  - `favoriteAction`
  - `flagAction`
- Semantic navigation actions:
  - `fullScreenAction`
  - `exitFullScreenAction`
  - `closeAction`
- Custom actions:
  - `titleCustomTextActions`
  - `titleCustomIconActions`

The actions in the title are grouped to text actions or icon actions. When an aggregation is set, the action appears in the following predefined order (from left to right) as follows:

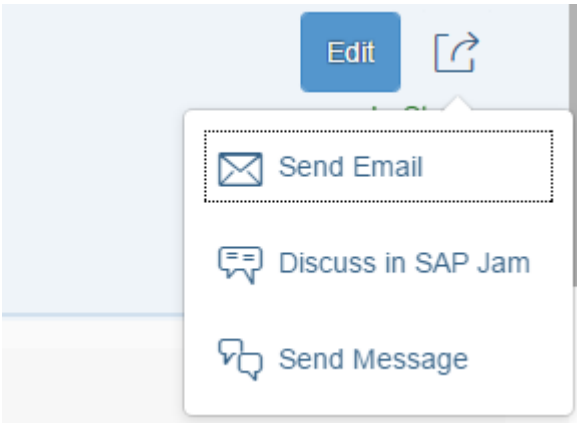
`titleMainAction`, `titleCustomTextActions`, semantic text actions (`deleteAction`, `copyAction`, `addAction`), `customIconActions`, semantic icon actions (`favoriteAction`, `flagAction`), share menu action, navigation actions (`fullScreenAction`, `exitFullScreenAction`, `closeAction`).

Share Menu

The following aggregations are available to control the semantic in the share menu. They are positioned vertically in this order:

- `sendEmailAction`
- `discussInJamAction`
- `shareInJamAction`
- `sendMessageAction`
- `printAction`
- `customShareActions`

The actions in the share menu icon appear in the title when a related aggregation is used.



Footer

The following aggregations are available to control the semantic in the page footer:

- `footerMainAction`
- `messagesIndicator`
- `draftIndicator`
- `positiveAction`
- `negativeAction`
- `footerCustomActions`

The actions in the footer are positioned either on its left or right area and have the following predefined order from left to right:

The left side contains the `messagesIndicator`, and the right side - `draftIndicator`, `footerMainAction`, `positiveAction`, `negativeAction` and `footerCustomActions`:

Examples

Initialization

Definition in an XML view:

```
<core:View
        xmlns:semantic="sap.f.semantic"
            controllerName="mycompany.myController"
            height="100%">
    <semantic:SemanticPage id="mySemanticPage">
        <!—semantic  page content specified  here -->
    </semantic:SemanticPage >
</core:View>
```

Definition in JavaScript:

```
var oSemanticPage = new sap.f.semantic.SemanticPage("mySemanticPage");
oView.addContent(oSemanticPage);
```

Adding content

Adding semantic content:

```
<mvc:View
   height="100%"
   xmlns:mvc="sap.ui.core.mvc"
   xmlns="sap.m"
   xmlns:semantic="sap.f.semantic">
   <semantic:SemanticPage>
      ...
      <!— will automatically create a button with “email” icon and style and
position it in accord with the underlying semantics -->
      <semantic:sendEmailAction press=”onSendEmailPress >
```

```
         <semantic:SendEmailAction />
      </semantic:sendEmailAction>
     <!– will automatically create a button with icon, styling and positioning
in accord with the underlying semantics -->
      <semantic:discussInJamAction press=”onDiscussInJamPress”>
         <semantic:DiscussInJamAction />
      </semantic:discussInJamAction>

     ...
   </semantic:SemanticPage>
</mvc:View>
```

Adding custom (non-semantic) content

```
<mvc:View
   height="100%"
   xmlns:mvc="sap.ui.core.mvc"
   xmlns="sap.m"
   xmlns:semantic="sap.f.semantic">
   <semantic:SemanticPage>
       ...
<!-- Title Heading-->
<semantic:titleHeading>
   <Title text="{/title}" />
</semantic:titleHeading>
<!-- Header Content -->
<semantic:headerContent>
      <!–custom header Content goes here -->
<FlexBox
      alignItems="Start"
      justifyContent="SpaceBetween">
   <items>
      <Panel backgroundDesign="Transparent">
         <content>
            <ObjectAttribute title="Functional Area" text="{/objectDescription/
category}"/>
            <ObjectAttribute title="Cost Center" text="{/objectDescription/
center}"/>
            <ObjectAttribute title="Email" text="{/objectDescription/email}"/>
         </content>
      </Panel>
      <ObjectStatus text="In Stock" state="{/objectDescription/status}" />
   </items>
</FlexBox>
</semantic:headerContent>
<!–Content -->
<semantic:content>
    <!–Custom page-body content gies here -->
   <Table
      id="idProductsTable"
      inset="false"
      items="{path:'/ProductCollection'}">
     ...
   </Table>
</semantic:content>
   </semantic:SemanticPage>
</mvc:View>
```

Related Information

API Reference: `sap.f.DynamicPage`

Semantic Page (sap.m)

The `sap.m.semantic.SemanticPage` is an enhanced `sap.m.Page` that implements the SAP Fiori 1.0 design guidelines.

For more information about this control, see the API Reference and the samples.

Features

The following categories give an overview of the internally defined semantic content that is supported:

- Visual properties - content is styled in a certain way, for example an `AddAction` is displayed as an icon button.
- Position in the page - some buttons are displayed in the page header only, while others are in the footer or in the share menu.
- Sequence order - there is a specific sequence order of semantic controls with respect to each other.
- Tooltip - there is a default localized tooltip for the icon-only buttons.
- Overflow behavior - some of the buttons are allowed to go to the overflow area of the toolbar when the screen becomes narrower. For icon buttons, the text label of the button appears when the button is in the overflow area.
- Screen reader support - there is invisible text for reading the semantic type.

The available semantic content is different buttons and selects. Each one can correspond to a common action, such as Add, Edit, Save, and Sort.

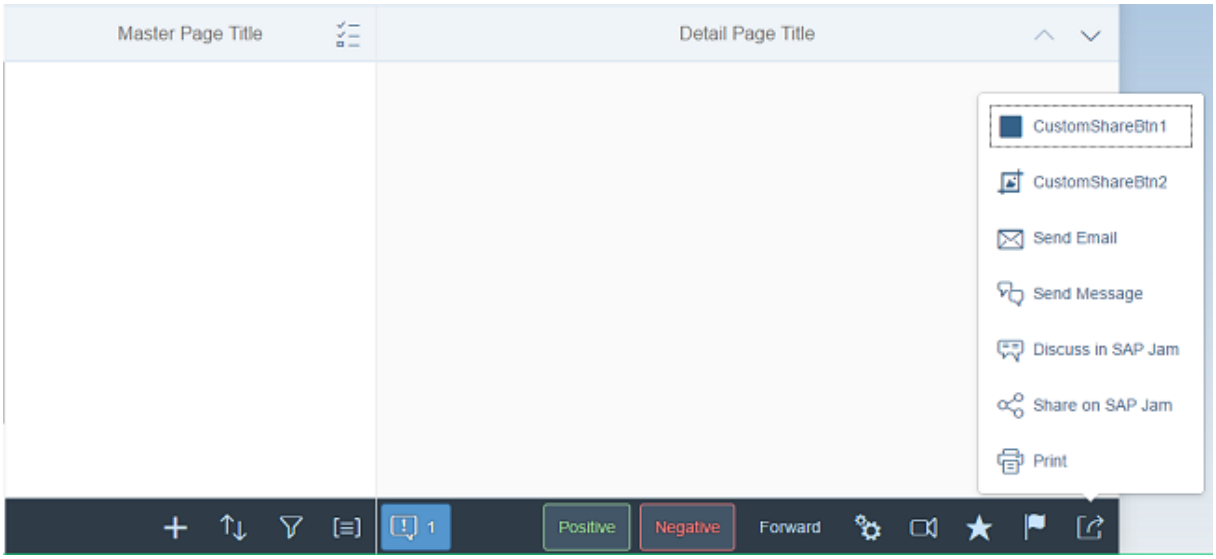
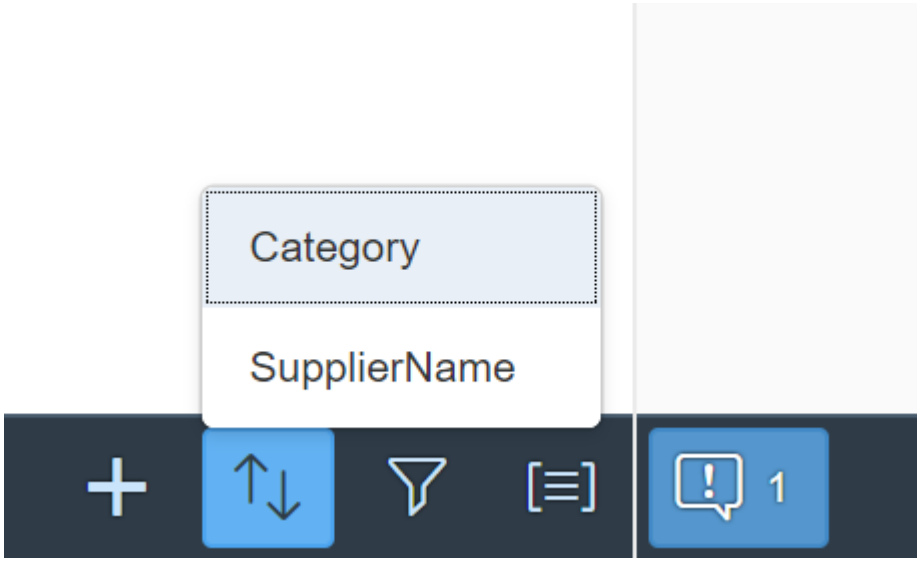


Figure 344: Semantic content at its default positions on the page.

Several different selects are supported for displaying a list of selectable items - `SortSelect`, `FilterSelect`, and `GroupSelect`.

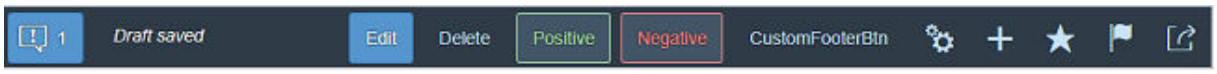


The following aggregations are available and enable the apps to add their own custom content to the different areas of the page:

- `customHeaderContent`
- `customFooterContent`
- `customShareMenuContent`
- `content` - for content in the body of the page

The ordering logic of custom and semantic content is as follows from left to right:

- Left area:
  - `messagesIndicator`
  - `draftIndicator`
- Right area:
  - `mainAction`
  - Semantic text-only buttons, such as `deleteAction`, `positiveAction`, and `negativeAction`
  - Custom content that the app fully controls with no automatic reordering
  - Semantic icon-only buttons, such as `favoriteAction`, and `flagAction`



Examples

Initialization

In the `sap.m` library, the semantic page controls are`sap.m.semantic.FullscreenPage`,`sap.m.semantic.MasterPage`, and `sap.m.semantic.DetailPage`. They have different purpose depending on the context:

For split-screen (Master-Detail) scenarios, apps should use `sap.m.semantic.MasterPage` together with `sap.m.semantic.DetailPage`:

```
<mvc:View
height="100%"
xmlns:mvc="sap.ui.core.mvc"
xmlns="sap.m"
controllerName="mycompany.myController"
xmlns:semantic="sap.m.semantic"
displayBlock="true">
<SplitContainer>
   <masterPages>
      <semantic:MasterPage>
         <!-- master page content goes here -->
      </semantic:MasterPage>
   </masterPages>
   <detailPages>
      <semantic:DetailPage>
         <!-- detail page content goes here -->
      </semantic:DetailPage>
   </detailPages>
</SplitContainer>
</mvc:View>
```

For fullscreen scenarios (where the page should always take the entire screen), apps should use `sap.m.semantic.FullscreenPage`:

```
<mvc:View
height="100%"
xmlns:mvc="sap.ui.core.mvc"
xmlns="sap.m"
controllerName="mycompany.myController"
xmlns:semantic="sap.m.semantic"
displayBlock="true">
<App>
   <pages>
      <semantic:FullscreenPage>
         <!-- page content goes here -->
      </semantic: FullscreenPage >
   </pages>
</App>
</mvc:View>
```

Adding semantic content:

The three semantic pages inherit from the abstract `sap.m.semantic.SemanticPage` control and each supports content that semantically belongs to its master/detail/fullscreen context.

For example, as the master part usually contains a list of items to be selected, so `sap.m.semantic.MasterPage` supports semantic controls for common operations on a list of items, such as sort, filter, group and multiselect:

```
...
```

```
<SplitContainer>
<masterPages>
   <semantic:MasterPage>
      <semantic:sort>
         <semantic:SortSelect change="onSortChange"
                         items="{
                      path: '/ProductCollectionStats/Filters',
                      sorter: { path: 'Name' }
                      }">
            <core:Item key="{type}" text="{type}" />
         </semantic:SortSelect>
      </semantic:sort>
      <semantic:filter>
         <semantic:FilterAction press="onFilterPress"/>
      </semantic:filter>
      <semantic:group>
         <semantic:GroupAction press="onGroupPress"/>
      </semantic:group>
   </semantic:MasterPage>
</masterPages>
<detailPages>
   ...
</detailPages>
</SplitContainer>
```

The `DetailPage` usually displays extended information for the item that was selected in the master part, therefore the `sap.m.semantic.DetailPage` also supports controls for operations like editing and sharing:

```
...
<SplitContainer>
       <masterPages>
              ...
        </masterPages>
        <detailPages>
              <semantic:DetailPage">
            <semantic:forwardAction>
                <semantic:ForwardAction press="onForwardPress"/>
            </semantic:forwardAction>
   <semantic:shareInJamAction>
      <semantic:ShareInJamAction press="onShareInJamPress"/>
   </semantic:shareInJamAction>
   <semantic:messagesIndicator>
      <semantic:MessagesIndicator press="onMessagesIndicatorPress"/>
   </semantic:messagesIndicator>
</semantic:DetailPage>
</detailPages>
</SplitContainer>
```

Adding custom (non-semantic) content:

```
...
<semantic:FullscreenPage title="FullScreen Page Title"
                           showNavButton="true"
                           navButtonPress="onNavButtonPress">
                         <semantic:customHeaderContent>
                  <!—custom header controls go here -->
                  <Button text="CustomHeaderBtn" press="onHeaderBtnPress"/>
               </semantic:customHeaderContent>
               <semantic:content>
                   <!—custom page content goes here -->
                  <semantic:AddAction press="onSemanticButtonPress"/>
               </semantic:content>
                         <semantic:customFooterContent>
                   <!—custom footer controls go here -->
                  <Button text="CustomFooterBtn" press="onFooterBtnPress"/>
```

```
                    <OverflowToolbarButton icon="sap-icon://settings"
text="Settings" press="onSettingsPress"/>
                </semantic:customFooterContent>
                            <semantic:customShareMenuContent>
                     <!—custom share-menu controls go here -->
                    <Button text="CustomShareMenuBtn" press="onShareMenuBtnPress"/>
                </semantic: customShareMenuContent >
</semantic:FullscreenPage>
```

Related Information

API Reference: `sap.m.Page`

Tables: Which One Should I Choose?

The libraries provided by SAPUI5 contain various different table controls that are suitable for different use cases. The table below outlines which table controls are available, and what features are supported by each one.

Table 114: Overview of Tables and Supported Features

Feature Supported?	Responsive Table (sap.m.Table)	Grid Table (sap.ui.table.Table)	Analytical Table (sap.ui.table.AnalyticalTable)	Tree Table (sap.ui.table.TreeTable)
Desktop	✓	✓	✓	✓
Tablet	✓	✓	✓	✓
Phone	✓	✗	✗	✗
Responsive (hide column, popin support)	✓	✗	✗	✗
Compact density	✓	✓	✓	✓
Condensed density	✗	✓	✓	✓
Cozy density	✓	✓	✓	✓
Summarized cell	✗	✗	✓	✗
Hierarchical data	✗	✗	✗	✓

Feature Supported?	Responsive Table (sap.m.Table)	Grid Table (sap.ui.table.Table)	Analytical Table (sap.ui.table.AnalyticalTable)	Tree Table (sap.ui.table.TreeTable)
Large number of rows (> 200)[2]	⚠	✓	✓	✓
Grouping	✓	✗	✓	✗
Freeze columns	✗	✓	✓	✓
Horizontal scrolling	✗	✓	✓	✓
Merge duplicates	✓	✗	✗	✗
Supported controls	Supports all kinds of controls inside a line item	Supports a limited set of controls[1]	Supports a limited set of controls[1]	Supports a limited set of controls[1]
Row-based	✓	✗	✗	✗
Column-based	✗	✓	✓	✓
OData-based	Requires manual modifications	Requires manual modifications	Requires manual modifications	Requires manual modifications

1) `Text`, `Label`, `ObjectStatus`, `Icon`, `Button`, `Input`, `DatePicker`, `Select`, `ComboBox`, `MultiComboBox`, `CheckBox`, `Link`, `Currency`, `RatingIndicator`, `ProgressIndicator` as well as the `StackedBarMicroChart`, `ComparisonMicroChart` , and `BulletMicroChart` controls (of responsive or extra small size); To keep the control height always stable, the `wrapping` and `renderWhitespace` properties in the `sap.m.Text` control, for example, must be set to `false`. For more information, search for cell level in the SAP Fiori Design Guidelines.

2) To optimize perfomance, we recommend to show no more than 200 items at once in the responsive table. For a larger number of items (up to 1000), use the growing feature to limit the number of displayed items and make sure the user can filter the data. For more information, see the API Reference for the growing* properties.

> ⚠ Caution
>
> The limits mentioned are only recommendations. For a specific app context, the actual number of manageable items might be higher or lower.
>
> The actual limits depend on what your scenario looks like, for example:
>
> - The number of rows in the table
> - The number of columns that are visible
> - The complexity of the cell content and/or the page (for example, multiple pages in a flexible column layout, or depending on how much binding is done)
> - The browser being used

For more information, search for loading items and performance in the responsive table section in the SAP Fiori Design Guidelines.

Smart Table

(sap.ui.comp.smarttable.SmartTable)

The `SmartTable` control is an OData-based control that automatically generates the required table (based on the `tableType` property) if the required annotations are maintained, and the suitable configuration is provided. All attributes mentioned in the table above depend on which table type is used for `SmartTable`. Refer to the respective table column for more information about the individual attributes.

sap.f

This library contains controls that were built based on the SAP Fiori 2.0 design guidelines.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]
API Reference: `sap.f`

Building an App with the Flexible Column Layout and Related Classes

The following sections provide you with best practices and details that help you develop SAP Fiori 2.0 apps with the `sap.f.FlexibleColumnLayout` control.

The simplest way to build an app with the `sap.f.FlexibleColumnLayout` is to create an instance of the control as a top-level container in the root view of your component and then use the `sap.f.routing.Router` to manipulate the control upon user interaction.

On each significant user action, query the `sap.f.FlexibleColumnLayoutSemanticHelper` class for the recommended layout and action buttons to show.

Flexible Column Layout

The `sap.f.FlexibleColumnLayout` control implements the master-detail-detail paradigm by displaying up to 3 pages in separate columns.

The control is suitable for apps that need to display several logical levels of related information side by side, for example, a list of items, item, sub-item. It is flexible in a sense that the app can focus the user's attention on one particular column by making it larger or even fullscreen.

Related Information

API Reference: `sap.f.FlexibleColumnLayout`

Control Structure

The `sap.f.FlexibleColumnLayout` contains 3 instances of `sap.m.NavContainer` – one for each column.

It is logically similar to `sap.m.SplitContainer`, however, there are two main differences:

- Displays up to 3 columns side by side (as opposed to 2 columns)
- The width of the columns is not fixed, but flexible (determined by the `layout` property)

The following table respresents how the `FlexibleColumnLayout` roughly relates to the `NavContainer` and `SplitContainer` controls.

Control	API	Number of Pages Displayed
`sap.m.NavContainer`	`pages` (aggregation)	1 page at a time
`sap.m.SplitContainer`	`masterPages` (aggregation) `detailPages` (aggregation)	Up to 2 pages at a time (2 instances of `NavContainer`)
`sap.f.FlexibleColumnLayout`	`beginColumnPages` (aggregation) `midColumnPages` (aggregation) `endColumnPages` (aggregation) `layout` (property of type `sap.f.LayoutType`, determining the relative widths of the 3 `NavContainers`)	Up to 3 pages at a time (3 instances of `NavContainer`)

Types of Layout

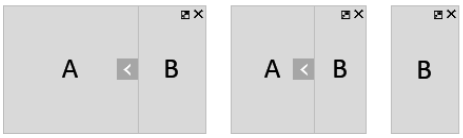
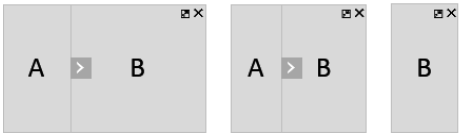
Overview of the possible layouts for an `sap.f.FlexibleColumnLayout`, as defined in the `sap.f.LayoutType` enumeration.

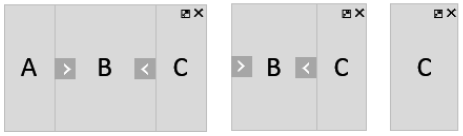
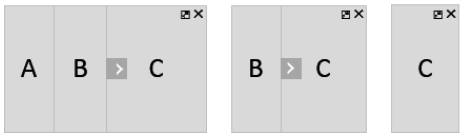
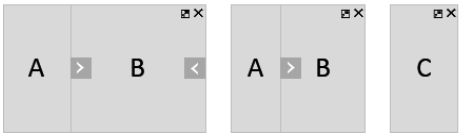
Although the `FlexibleColumnLayout` can display 2 or 3 pages at one time, they can never have equal width (50%/50% or 33%/33%/33%). One of the pages is always larger (expanded) or even takes up the full width of the control (fullscreen). This is intentional because users should have a clear indication of what to focus their attention on at any given moment, for example, a list of items, one particular item, one item's details.

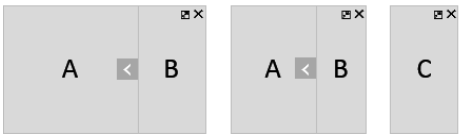
Transitioning from a one-column layout to any two-column layout, and then to any three-column layout is seen by the user as new columns appearing/disappearing on desktop/tablet or the next column replacing the previous one - on phone (small screen size). The app does not need to provide separate logic for the different screen sizes, but only change the layout based on the user input and desired behavior. The `FlexibleColumnLayout` will internally manage column visibility and resizing. If at any time, the browser size changes, this will be reflected by the control automatically.

There are 9 allowed variations of width and visibility for the 3 columns. They are represented with the values of the `layout` property, which is of type `sap.f.LayoutType`.

The table below shows how each of the 9 layout types affects the column visibility and width, based on the control size:

Layout Types	Desktop / Tablet / Phone
`OneColumn` This is the default layout type for a `FlexibleColumnLayout`. Only the `Begin` column is displayed, no matter the control size. Use to show the first logical information level (master page) only.	
`TwoColumnsBeginExpanded` Use this layout type to display two logical levels of information (master and detail pages, master being expanded) on desktop/tablet, or the second logical level of information only (detail page) on phone. On desktop/tablet the transition from `OneColumn` to `TwoColumnsBeginExpanded` is seen by the user as the `Begin` column shrinking and `Mid` column opening, while on phone it's seen as the `Mid` column replacing the `Begin` column.	
`TwoColumnsMidExpanded` Similar to `TwoColumnsBeginExpanded`, but this time the `Mid` column is the wider one.	

Layout Types	Desktop / Tablet / Phone
`MidColumnFullScreen` Use this layout to display the second logical level of information (detail page) only. The `Mid` column takes up the whole available control width for all control sizes. **i Note** On small screen sizes, the layouts `TwoColumnsBeginExpanded`, `TwoColumnsMidExpanded` and `MidColumnFullScreen` all lead to the same result for the user – the `Mid` column taking up the whole control width. However, if the user resizes the browser and makes space, the control will automatically apply the rules of the current layout for the new width.	
`ThreeColumnsMidExpanded` Use this layout to display up to three logical levels of information side by side (master, detail, and detail-detail pages), when the user should focus primarily on the content of the `Mid` column (as it is wider than the other two). On desktop, all three columns will be displayed, on tablet – the `Mid` and `End` columns only (with a layout arrow to access the `Begin` column), and on phone – the `End` column only.	
`ThreeColumnsEndExpanded` Similar to `ThreeColumnsMidExpanded`, but this time the `End` column is expanded, and this is where the user is supposed to focus their attention.	
`ThreeColumnsMidExpandedEndHidden` Use this layout to show the `Begin` and `Mid` columns only (`Mid` being expanded), while the `End` column is hidden but accessible with a layout arrow. The difference between this layout and the `TwoColumnsMidExpanded` layout is that for `TwoColumnsMidExpanded` the user cannot access the `End` column at all, as only two logical levels of information are offered (the third may not be loaded yet), while for `ThreeColumnsMidExpandedEndHidden` the `End` column is not empty, but just not currently displayed. Its content is loaded and easily accessible (most commonly already seen by the user, and now hidden so more space can be freed for the other columns).	

Layout Types	Desktop / Tablet / Phone
`ThreeColumnsBeginExpandedEndHidden` Similar to `ThreeColumnsMidExpandedEndHidden`, but this time the `Begin` column is expanded. A layout arrow is provided to shrink the `Begin` column, thus transitioning the layout to `ThreeColumnsMidExpandedEndHidden`.	
`EndColumnFullScreen` Use this layout to display the third logical level of information (detail-detail page) only. The `End` column takes up the whole available control width for all screen sizes. **i Note** On small screen sizes all three-column layouts and `EndColumnFullScreen` lead to the same result for the user – the `End` column taking up the whole control width. However, if the user resizes the browser and makes space, the control will automatically apply the rules of the current layout for the new width.	

For each value of the layout property, the `FlexibleColumnLayout` displays a different combination of columns, based on the available control width (or screen width, if the control takes up the whole screen).

Control size	Columns
Up to 960px (Phone)	1 column max
960px – 1280px (Tablet)	2 column max
1280px and above (Desktop)	3 column max

Related Information

API Reference: `sap.f.LayoutType`

Layout Arrows

Layout arrows are used to alter the current layout of the `FlexibleColumnLayout`.

The `FlexibleColumnLayout` provides layout arrows on screen sizes where more than one column may be displayed at one time (desktop and tablet). They allow the user to alter the current layout.

Layout arrows are the only way for the layout to change on control level, without the app changing it explicitly. Layout changes due to interaction with the layout arrows are only from one type of two-column layout to

another type of two-column layout, or from one type of three-column layout to another type of three-column layout. Changing the number of active columns (for example, from a two-column layout to a three-column layout) can only happen on app level by directly manipulating the `layout` property from your controller.

Examples for layout changes due to layout arrows interaction:

- In a `TwoColumsBeginExpanded` layout there is one arrow that changes it to `TwoColumnsMidExpanded`.
- In a `TwoColumnsMidExpanded` layout there is one arrow that changes it to `TwoColumnsBeginExpanded`.
- In a `ThreeColumnsMidExpanded` layout there are two arrows that allow changing the layout either to a `ThreeColumnsEndExpanded` or to `ThreeColumnsMidExpandedEndHidden` layout.

Reacting to Layout Changes

The `FlexibleColumnLayout` control provides the `stateChange` event.

The event is fired in the following cases:

- The layout changes because the user chose a layout arrow.
- The user resizes the browser beyond the 960px/1280px thresholds, which doesn't change the layout, but changes the number of columns that can be displayed at one time.

The app can subscribe to this event to be able to react to the layout change, for example to show/hide action buttons, such as fullscreen button or close column button.

> **i** Note
>
> The event is NOT fired if the app changes the `layout` property explicitly. It is only fired if it was changed internally, due to the user interacting with the layout arrows or columns appearing/disappearing.

Changing the Layout and Loading Views (Best Practices)

The app can load controls (usually views) in the three columns with the standard means (`beginColumnPages`, `midColumnPages`, `endColumnPages` aggregations) and navigate between them with the `.to()`, `.backToPage()` or any other public method, similar to the `sap.m.NavContainer` or `sap.m.SplitContainer` controls.

At the same time, the app can change the layout of the control by modifying the `layout` property.

If both a layout change and the loading of a new view need to happen as a result of one user action, the best practice is to change the layout first, and to load the views (or navigate to already loaded views) second. This would ensure that the columns are resized first (as layout change is done synchronously), and only then views are placed inside the already resized columns. This would eliminate the need for controls in the views to readapt to the new size after they are placed and rendered initially.

Flexible Column Layout Semantic Helper

A helper class, designed to facilitate the implementation of apps based on the `sap.f.FlexibleColumnLayout` control and the SAP Fiori 2.0 design guidelines.

`FlexibleColumnLayout` gives you the freedom to implement any app logic that involves changing the layout (showing/hiding columns) as a result of the user's actions. However, there are certain UX patterns that are considered as optimal and are recommended for SAP Fiori 2.0 apps. The `FlexibleColumnLayoutSemanticHelper` class helps you implement them by giving you tips about what layout to display when.

> **i Note**
>
> Using this class is NOT mandatory in order to build an app with the `FlexibleColumnLayout`, but makes it easier to achieve the optimal UX recommended in the SAP Fiori 2.0 design guidelines.

The first 3 logical levels of information are displayed in the three columns side by side, and the forth (and others) are displayed in the `End` column in fullscreen. If at any time the user opens a page in fullscreen, all subsequent levels are also displayed in fullscreen.

Here is a short overview of some of the rules of the default rule set:

- The control starts with `OneColumn` layout (usually to display a list of items).
- When the user selects an item from the `Begin` column, the item's details are displayed in the `Mid` column and the layout should change to `TwoColumnsBeginExpanded`. Two action buttons are displayed on the page in the `Mid` column – *Fullscreen* and *Close*.
  - Choosing the *Close* button changes the layout back to `OneColumn`.
  - Choosing the *Fullscreen* button changes the layout to `MidColumnFullScreen`. The *Fullscreen* button is then replaced with an *ExitFullscreen* button, which restores the layout to `TwoColumnsBeginExpanded`.
- If the user selects an item from the `Mid` column:
  - If the current layout is `TwoColumnsBeginExpanded`/`TwoColumnsMidExpanded`, the layout changes to `ThreeColumnsMidExpanded`. The *Close* and *Fullscreen* buttons should now be displayed only in the `End` column.
  - If the current layout is `MidColumnFullScreen`, the layout changes to `EndColumnFullScreen`.
- If the user selects an item from the `End` column (which represents the third logical level of information), the layout should change to `EndColumnFullScreen` and the forth (then fifth, etc.) logical level should again be displayed in the `End` column.

Main Methods

Overview of the two main methods used in the `sap.f.FlexibleColumnLayoutSemanticHelper` class.

`getCurrentUIState()`

This method returns an object, providing the following detailed information about the current state of the `FlexibleColumnLayout`:

- The current layout of the control
- The relative sizes and visibility of the control's columns
- Whether the control currently displays a page in fullscreen (and is fullscreen explicitly set, or is this due to device size constraints?)
- What action buttons the app should display, in which column, and what control layout appears when these buttons are chosen

The example below shows a sample value of the `actionButtonsInfo` field of this object:

```
"actionButtonsInfo":{
       "midColumn":{
         "fullScreen":null,
         "exitFullScreen":null,
         "closeColumn":null
       },
       "endColumn":{
         "fullScreen":"EndColumnFullScreen",
         "exitFullScreen":null,
         "closeColumn":"TwoColumnsBeginExpanded"
       }
   }
```

This means that currently the app should not display any action buttons in the `Mid` column. In the `End` column however, there should be two action buttons – *Fullscreen* and *Close*. If the user chooses the *Fullscreen* button, the layout should change to `EndColumnFullScreen`, and if the user chooses the *Close* button, the layout should change to `TwoColumnsBeginExpanded`.

`getNextUIState(iNextLevel)`

This method tells you to what layout the control should transition, and what kinds of action buttons would be displayed to represent a different logical level. The argument `iNextLevel` is a zero-based int. 0 stands for the initial logical level, 1 - for master and detail, 2 - for master, detail, and detail-detail, 3 and above – for the subsequent logical levels.

If the current layout is `OneColumn` (only the `Begin` column is displayed), and the user selects an item from it, both master and detail should be displayed. To determine the recommended layout to display them, call `getNextUIState(1)`. It will return an object with a layout field with value `TwoColumnsBeginExpanded`. Later, if the user selects an item from the `Mid` column, call `getNextUIState(2)` to determine the layout to use for the detail-detail level. Alternatively, if the user choses an item in the `Begin` column, call again `getNextUIState(1)` to determine the proper two-column layout to switch to.

Note that the `getNextUIState` method is not static – its return value depends on the current state of the control. For example, calling `getNextUIState(2)` will return `ThreeColumnsMidExpanded` if currently a two-column layout is displayed, but `EndColumnFullScreen` - if a fullscreen layout is displayed, etc.

Related Information

API Reference: `sap.f.FlexibleColumnLayoutSemanticHelper`

Router

The `sap.f.routing.Router` has built-in support for routing-capable controls in the `sap.f` library.

The `sap.f.routing.Router` currently supports the `FlexibleColumnLayout` control only, similarly to how the `sap.m.routing.Router` has specific knowledge of the `sap.m.NavContainer` and `sap.m.SplitContainer`. Therefore, the `sap.f.routing.Router` should be used for apps based on the `FlexibleColumnLayout`.

The `sap.f.routing.Router` offers the `beforeRouteMatched` event. This event is fired before views are loaded into the respective columns. Use this event to manually change the layout (if you didn't specify a layout on route level).

The `sap.f.routing.Router` has the following main differences to the `sap.m.routing.Router`:

- The `sap.f.routing.Router` can have up to 3 targets per route (one for each column).
- The `sap.f.routing.Router` supports the `layout` setting on route level. Effectively, this means that you can specify a layout along with the targets of a route. This layout will be applied to the root control (which is meant to be a `sap.f.FlexibleColumnLayout`) before views are loaded.
- The `sap.f.routing.Router` view loading is exclusively asynchronous.

Sample Code

A sample route of a `sap.f.routing.Router`-based app:

```
{
   "pattern": "itemInfo",
   "name": "itemInfo",
   "target": [
      "master",
      "detail",
      "detailDetail"
   ],
   "layout": "ThreeColumnsMidExpanded"
}
```

Sample Code

A sample of its targets definition:

```
"targets": {
   "master": {
      "viewName": "Master",
      "controlAggregation": "beginColumnPages"
   },
   "detail": {
      "viewName": "Detail",
      "controlAggregation": "midColumnPages"
   },
   "detailDetail": {
      "viewName": "DetailDetail",
      "controlAggregation": "endColumnPages"
   }
}
```

If the route is matched, the router will first apply the `layout` property to the `sap.f.FlexibleColumnLayout` root control, and will then start loading the views asynchronously in its aggregations (and will navigate to each of them in the respective column).

Another way to achieve this, when the layout cannot be known in advance, is to just specify the targets (as you do when using `sap.m.routing.Router`), without supplying a `layout` setting:

```
{
   "pattern": "itemInfo",
   "name": "itemInfo",
   "target": [
      "master",
      "detail",
      "detailDetail"
   ]
}
```

And then change the layout manually. The recommended lifecycle event to do this is a `beforeRouteMatched` event handler in your controller. In theory, you could change the layout as response of `routeMatched` too, but `beforeRouteMatched` is better for the purpose, as views aren't loaded yet, which ensures that when they are, columns will be already resized, and there will be no need for the controls in the views to readapt (which would be the case if views were loaded first, and only then columns resized).

Related Information

API Reference: `sap.f.routing.Router`

sap.m

This library contains the most important controls for building a user interface that is responsive to any device.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]
Browser and Platform Support [page 20]
API Reference: `sap.m`

App and Nav Container

Apps are often composed of several pages and the user can drill-down to detail pages and go back up again. This is often visualized by horizontal slide animations. SAPUI5 supports this pattern by providing the `sap.m.App` and `sap.m.NavContainer` controls, which handle the navigation between the pages.

`sap.m.App` inherits the navigation capabilities from the `sap.m.NavContainer` control. Thus, both controls are equal with regard to navigation and navigation events. The following sections refer to the `sap.m.NavContainer`, but the same also applies to the `sap.m.App` control.

An app can control the navigation flow centrally and directly trigger the initialization of the pages. To support modularization of the app, however, it may also be beneficial to control the navigation flow non-centrally. In this case, code which constructs a page is also the code that handles, for example, the data load in this page.

To support this, SAPUI5 provides two types of events:

- Events fired by the App or by the NavContainer whenever it navigates.
- Events fired **on** the pages when they get shown or hidden by navigation.

API References

- sap.m.App
- sap.m.NavContainer

Events Fired Centrally by the App or the NavContainer

When `NavContainer.to(...)` or `NavContainer.back(...)` are called, the `NavContainer` triggers events and the application can register for this events. The `navigate` event is fired before the transition animation starts, and the `afterNavigate` event is fired when the animation has been completed.

The events contain a lot of information about the page that is left and the target page of the navigation, as well as what kind of navigation is happening.

Example:

```
app.attachNavigate(function(evt) {
   var isBack = !evt.getParameter("isTo"); // there are several types of back
animation, but we want the general direction only
   alert("Navigating " + (isBack ? "back " : "") + " to page " +
evt.getParameter("toId"));
});
```

API References

- event:afterNavigate

- event:navigate

Events Fired on the Pages

Events fired on the pages allow a decentral reaction to navigation.

The `NavContainer` fires events to its child controls when they are displayed or when they are hidden.

> **i** Note
>
> Although this documentation calls them "pages" and a `sap.m.Page` control may be the typical child of a `NavContainer`, any full screen control can be used, for example, a `sap.m.Carousel` control or a custom control. The direct child controls often may be views. In this case, the events will be fired on the views, and not on a page control that may be contained in the view. Thus, the event is not fired **by** the child control, but by the `NavContainer` **on** the child control (whatever type it is). This causes the different registration compared to normal control events.

Before the navigation animation starts, the `NavContainer` fires the following events:

- `beforeHide` on the page which is about to be left
- `beforeFirstShow` on the page which is about to be shown; this event is only fired if the respective page has not been shown before
- `beforeShow` on the page which is about to be shown

These events can be used to create or update the user interface of the new page and to stop any activity, such as animations or repeated data polling, on the page which is left.

After the navigation has been completed and the new page has covered the whole screen, the following events are fired:

- `afterShow` on the page which is now shown
- `afterHide` on the page which has been left

You can destroy the hidden page, and the now active page can start its activity.

You can use the `addEventDelegate` function to register to these events. This function is available on every control.

```
page1.addEventDelegate({
   onBeforeShow: function(evt) {
      // page1 is about to be shown; act accordingly - if required you can read
event information from the evt object
   },
   onAfterHide: function(evt) {
      // ...
   }
});
```

API Reference

sap.m.NavContainerChild

Passing Data when Navigating

When you use the `to(...)` and `back(...)` methods of the `NavContainer` to trigger navigation, you can also give an optional payload data object.

This object is then available in the page events, for example `beforeShow` and `afterShow`. You can also use this mechanism to decouple page implementations.

Example:

```
app.to("detailPage", {id:"42"}); // trigger navigation and hand over a data
object
                                 // this data object could also be a binding
context when dealing with data binding
...
// and where the detail page is implemented:
myDetailPage.addEventDelegate({
   onBeforeShow: function(evt) {
      var idToRetrieve = evt.data.id;
      // ...now retrieve the data element with the given ID and update the page
UI
   }
});
```

When you navigate back to a page, it receives the original data object which has been given when you first navigated to the page, but you can also give an additional data object with the back navigation.

API References

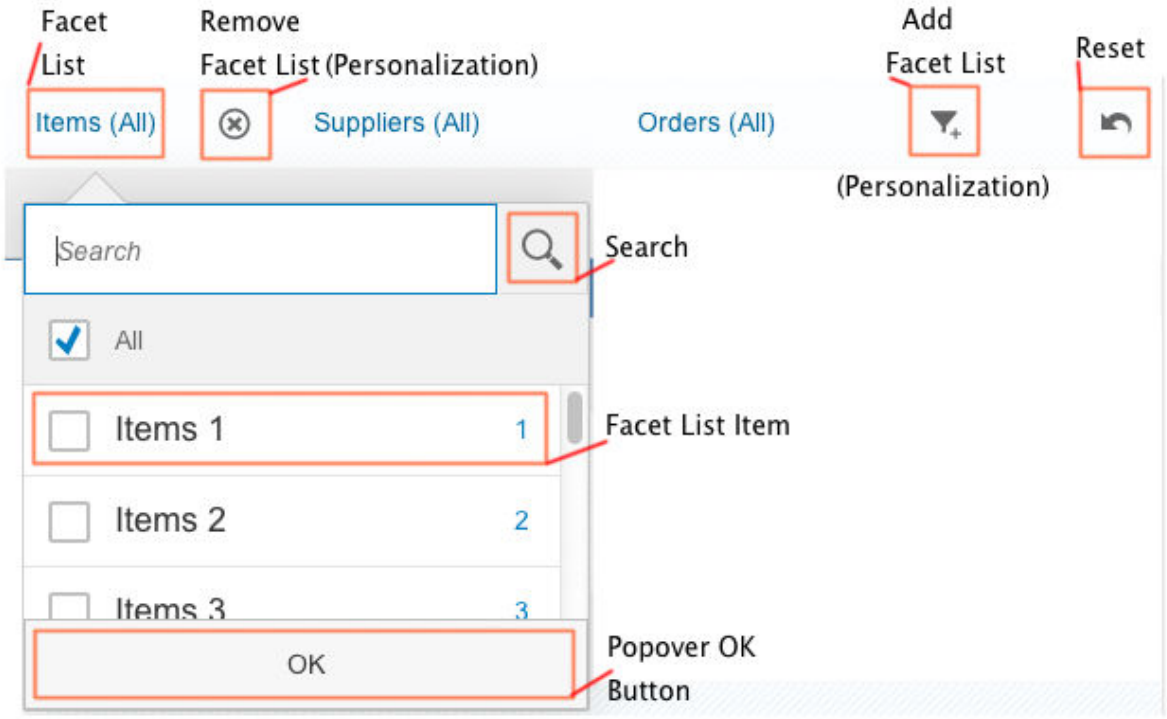
- sap.m.NavContainer
- sap.m.NavContainerChild

Facet Filter

Facet filters (`sap.m.FacetFilter`) support users in finding the information they need from potentially very large data sets.

With the facet filter, users can explore a data collection by applying multiple filters along certain discrete attributes or facets of the overall data collection.

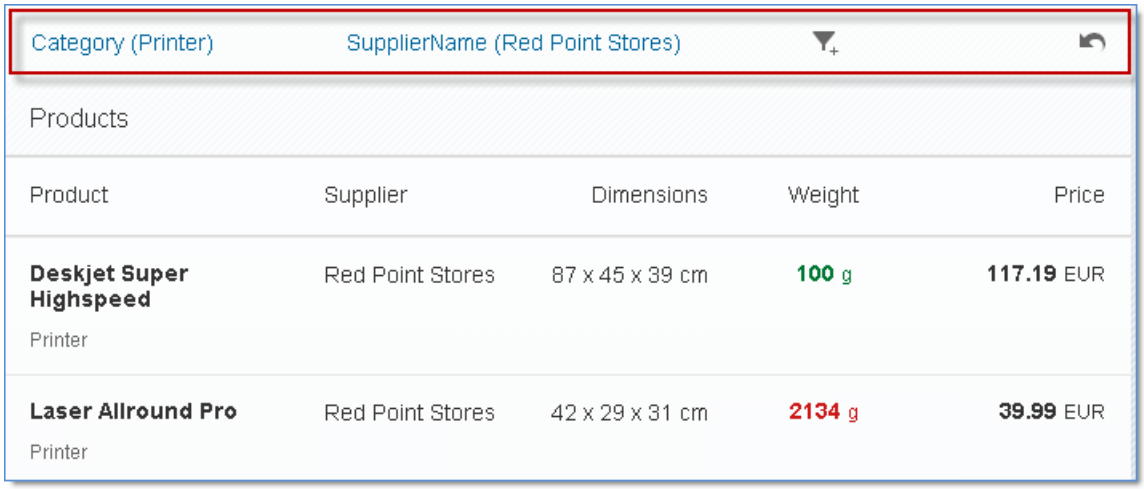
The following figure shows the structure of the facet filter.



Example

Your application displays a large list of products that can be grouped by category and supplier. With the facet filter, you allow users to dynamically filter the list so it only displays products from the categories and suppliers they want to see. In the following figure, the `FacetFilter` control is outlined in red and will be referred to as the 'toolbar' for the user. In the example, the user has set the following filters:

- Category: Printer
- Supplier: Red Point Stores



The facet filter supports the following two typeswhich can be configures using the control's `type` property:

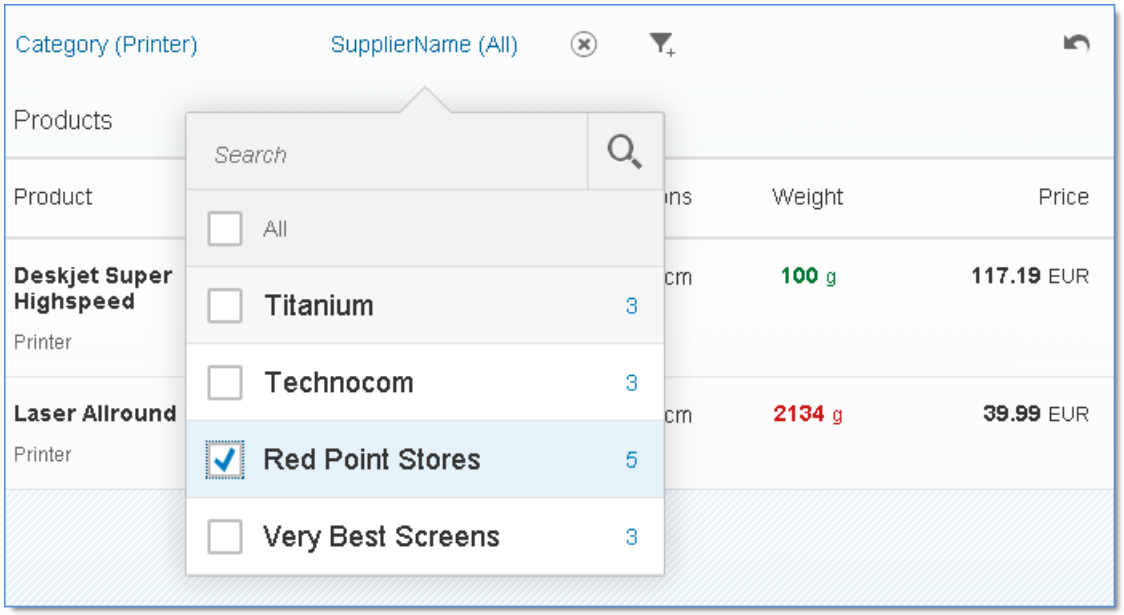
- Simple type
  The simple type is the default type and available for desktop and tablets.
- Light type
  The light type is automatically enabled on smart phone sized devices, but is also available for desktop and tablets.

Facet Filter: Simple Type

The simple type of the `FacetFilter` control is only available for desktop and tablets.

The active facets are displayed as individually selectable buttons on the toolbar as shown in the following figure.

If the user selects a facet in the toolbar, a popover list of the available filters for the selected facet is displayed.



The simple type provides the following functions:

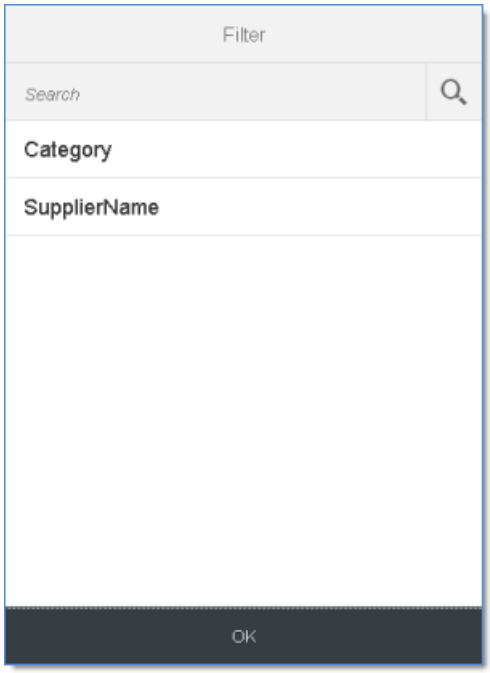
- With the `showPopoverOkButton` property of the `FacetFilter` control you can display an *OK* button in the popover. The *OK* button enables the user to close the popover in addition to the standard behavior of `sap.m.Popover`.
- With the `showPersonalization` property you enable the user to add facets to the toolbar by selecting the *Add Facet* icon. Personalization is disabled by default.
- With the `showSummaryBar` property you can display the active facets as a non-selectable summary bar. You use this property if you preset facet filters and the user is not allowed to change them.

Facet Filter: Light Type

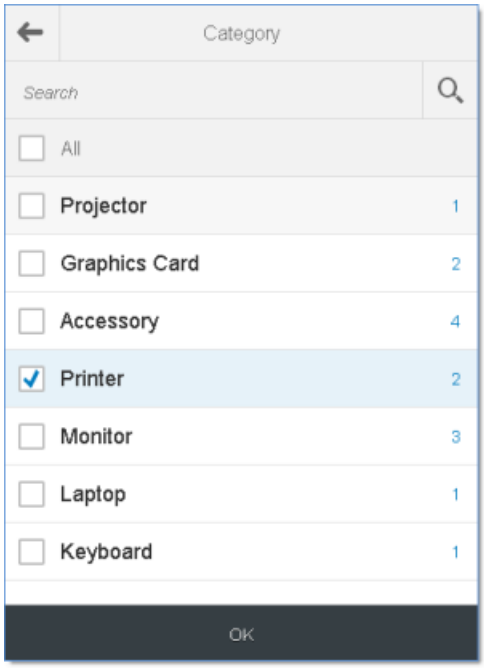
The light type of the `FacetFilter` control is automatically enabled on smart phone devices and is also available for desktop and tablets.

The active facets and selected filter items are displayed in the summary bar.

When the user selects the summary bar, a navigable dialog list of all facets is displayed. When the user selects a facet, the dialog scrolls to show the list of filters that are available for the selected facet.



By selecting any of its associated filters in the dialog, the user can add a facet to the summary bar.



Facet Filter List and Facet Filter Item

The `sap.m.FacetFilter` control uses the `FacetFilterList` and the `FacetFilterItem` controls to model facets and their associated filters.

The facet filter list aggregation is a collection of facet filter list objects, where each element in the collection represents a different facet. Likewise, the facet filter list items aggregation is a collection of facet filter item objects where each element in the collection represents a different filter items available for the facet.

Facet Filter List

The `FacetFilterList` control extends and supports all the features of the `sap.m.List` control, for example swipe for action, growing feature, remember selections and grouping.

The following properties of `FacetFilterList` affect the display of lists in the `FacetFilter` control.

Property	Description
`title`	Facet name
`mode`	Controls the selection mode for the list This property is overridden from `ListBase` and only allows `SingleSelectMaster` or `MultiSelect` settings. `MultiSelect` is the default: This setting displays the *All* checkbox above the filter list to allow the

Property	Description
	user to select all filters. This does not actually select individual filters to avoid performance overhead for lists with a large number of items.
`sequence`	Controls the order in which the facets are displayed in the toolbar Lists appear in the toolbar in ascending order according to sequence (assuming left to right). Lists with a sequence less than 0 are placed last, not before facets with sequence of zero. Only active lists are displayed regardless of sequence setting.
`active`	Indicates if a facet filter list is active and should appear on the toolbar; this is only applicable for the simple type as all facet filter lists are active in the light type
`allCount`	The `allCount` value can be set to the number of filter matches in the target data set given the currently selected filters for the facet filter list.

> **i** Note
>
> The list of properties is not complete. For a complete list, refer to the API documentation.

Facet Filter Item

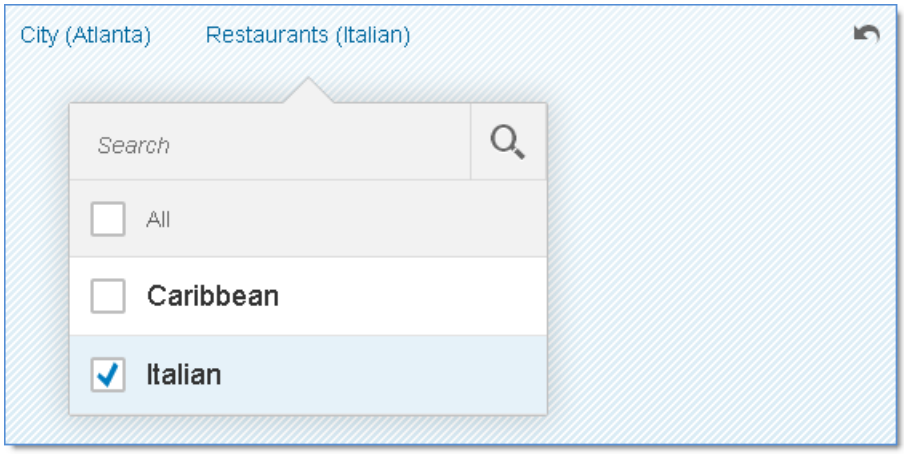
The `FacetFilterItem` control extends and supports all features of `sap.m.ListItemBase`, for example item selection and counter. `FacetFilterItem` provides the following properties:

Property	Description
`text`	Filter item name
`key`	Unique identifier of the filter item; used to filter the target data set If `key` is not set, `text` is used as the key value.

> **i** Note
>
> You must either set the `text` or the `key` property. Otherwise, the facet filter list can not properly maintain the selected state of the item and an error message is logged to the console.

Example

The following example shows how you use the controls. To build the face filter in the figure, use the code below the figure:




```
var oFacetFilter = new sap.m.FacetFilter({ // define FacetFilter Control
        lists : [ new sap.m.FacetFilterList({ // city facet
            title : "City",
            items : [ new sap.m.FacetFilterItem({
                text : "Waldorf",
                key : "WDF"
            }), new sap.m.FacetFilterItem({
                selected : true, // filter is selected (from ListItemBase)
                text : "Atlanta",
                key : "ATL"
            }) ]
        }), new sap.m.FacetFilterList({ // restaurant facet
            title : "Restaurants",
            items : [ new sap.m.FacetFilterItem({
                text : "Caribbean",
                key : "CRB"
            }), new sap.m.FacetFilterItem({
                selected : true, // filter is selected (from ListItemBase)
                text : "Italian",
                key : "ITL"
            }) ]
        }) ]
    });
```

> **i Note**
>
> The example does not have a model binding. A binding to the filter items is required for the search.

Events for Facet Filters

Facet filters support several events, such as reset and list open.

As the user interacts with the `FacetFilter` control, the following key events are fired for event handling in the application:

- `reset` event (`FacetFilter` control)
  The `reset` event is fired when the user presses the *Reset* icon on the toolbar or summary bar.
  You can use the `showReset` property of the `FacetFilter` control to disable the *Reset* icon so that it is no longer displayed. The application is responsible for implementing the reset logic.
  To remove all selected filters from a facet filter list, call `removeSelections(true)`.
- `listOpen` event (`FacetFilterList` control)
  The `listOpen` event is fired when the user selects a facet from the toolbar or when a facet is selected in the dialog.
  You can use this event to load the list with data the first time the user accesses it as opposed to loading all the lists with data when the application is initialized.
- `listClose` event (`FacetFilterList` control)
  The `listClose` event is fired when the user closes a popover, navigates back from the filter items page in the dialog, or closes the dialog. You use this event to handle any processing that needs to occur based on facet filter item selections, such as filtering the target data set.

Data Binding for Facet Filters

`FacetFilter` fully supports the SAPUI5 data binding concept.

As the information for the facet filter usually resides in the application backend, data binding is commonly used to populate the `FacetFilter` properties and aggregations.

Applications using OData do not bind the `FacetFilterItem.selected` property since this is not available from the backend service. Fortunately, selections are maintained internally on the client by the `FacetFilterList` until the list is destroyed.

Related Information

Facet Filter Selection [page 2308]

Filter Search

The popover and dialog displayed by `FacetFilter` contain an `sap.m.SearchField` control. This enables the user to search for specific items in the list.

`FacetFilterList` internally handles the `search` and `liveChange` events by filtering the underlying data model: The search only works when the `FacetFilterList` items aggregation is bound to a model.

If you enable the `liveSearch` property of the `FacetFilter` control, keep the performance in mind as this will result in a backend request for each search character typed by the user.

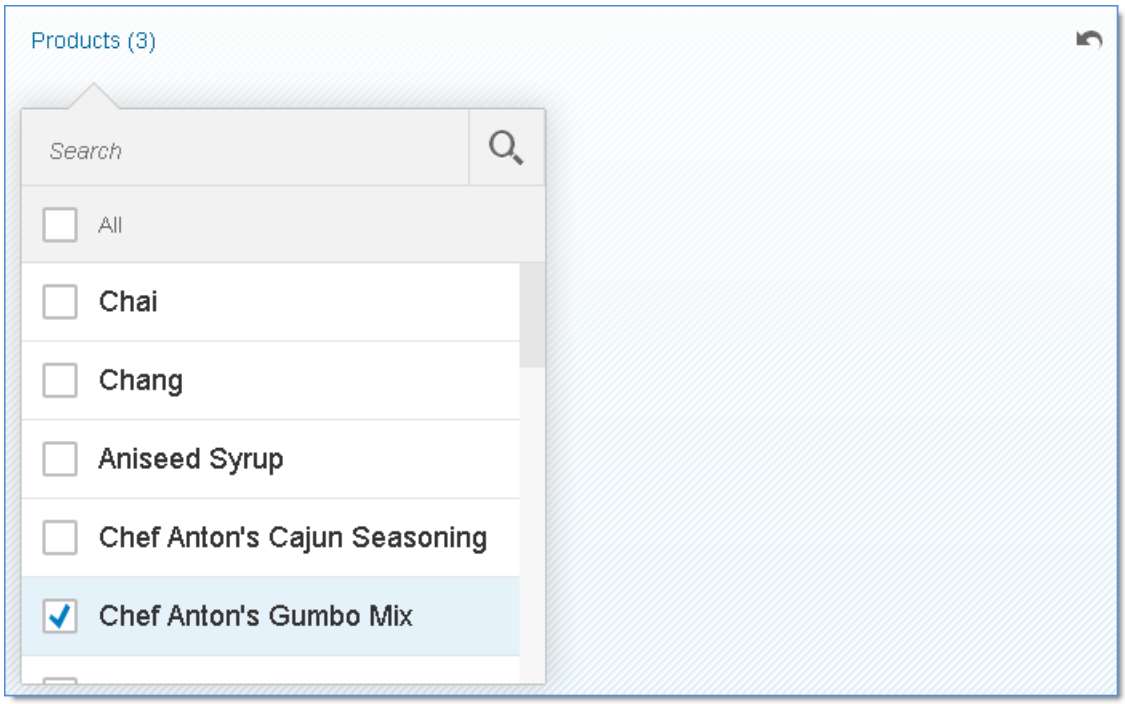
Facet Filter Selection

The `FacetFilterList.getSelectedItems()` method returns a copy of each selected facet filter item. You use the method to get the selected filter items when filtering the target data set.

Therefore, you should not attempt to modify any of the item's properties.

Variants

An application can support the personalization of settings and allow the user to save the facet filter list selections as well as other properties by means of a variant. For example, you can use `getSelectedKeys` to retrieve an object containing all selected items and use JSON.stringify to marshall and JSON.parse to unmarshall. After unmarshalling, you can use `setSelectedKeys` to apply the selections to the list. The following figure and code snippet give an example.




```
var oDataModel = new sap.ui.model.odata.v2.ODataModel("/uilib-sample/proxy/http/
services.odata.org/V3/Northwind/Northwind.svc");
// create the FacetFilterList and bind the filter items
var oFacetFilterList = new sap.m.FacetFilterList({
    title : "Products",
    growing : true,
    items : {
        path : "/Products",
        template : new sap.m.FacetFilterItem({
```

```
            text : "{ProductName}",
            key : "{ProductID}"
        })
    },
    listOpen : function(oEvent) {
        if(!this.hasModel()) {
            this.setModel(oDataModel);
        }
    },
});
// getSelectionsFromVariant() is an application method to retrieve
// selected keys from the backend. Selections were saved to the variant by
persisting
// the result of 'getSelectedKeys' for each list. This is an object
// containing Product keys as properties and Product text as property values,
for example:
/*
{
    '5' : "Chef Anton's Gumbo Mix",
    '17' : "Alice Mutton",
    '21' : "Sir Rodney's Scones"
}
*/
var oSelectedKeys = getSelectionsFromVariant();
// Now preselect these items
oFacetFilterList.setSelectedKeys(oSelectedKeys);
var oFacetFilter = new sap.m.FacetFilter({
    lists : [ oFacetFilterList ]
});
```

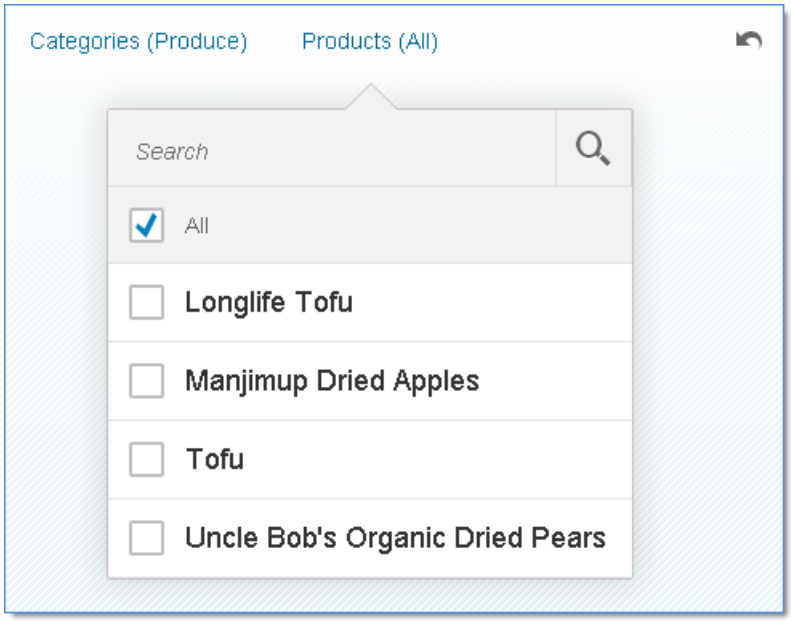
Dependent Facets

Applications can have dependencies between facets where selection of filter items in one facet list limits the list of valid filters in another facet list.

For example, an application displays a list of products and uses a facet filter with two facets: Categories and Products. If users select a category filter, they should only be able to filter products from that selected category. Facet filter does **not** explicitly handle dependencies between facets. Instead, use `FacetFilterList` events in the application.

Example

In this example, only products from the selected category are displayed.




```
var oCategoriesModel = new sap.ui.model.odata.v2.ODataModel("/uilib-sample/proxy/
http/services.odata.org/V3/Northwind/Northwind.svc");
var oCategoriesFFL = new sap.m.FacetFilterList({ // create the categories facet
list
    title : "Categories",
    mode : sap.m.ListMode.SingleSelectMaster, // restrict to one selection for
simplicity
    key : "Categories",
    items : {
        path : "/Categories",
        template : new sap.m.FacetFilterItem({
            text : "{CategoryName}",
            key : "{CategoryID}"
        })
    }
});
oCategoriesFFL.setModel(oCategoriesModel); // set the data model
// create the data model for the products facet list
var oProductsModel = new sap.ui.model.odata.v2.ODataModel("/uilib-sample/proxy/
http/services.odata.org/V3/Northwind/Northwind.svc");
var oProductsFFL = new sap.m.FacetFilterList({
    title : "Products",
    key : "Products",
    items : {
        path : "/Products_by_Categories",
        template : new sap.m.FacetFilterItem({
            text : "{ProductName}",
            key : "{ProductID}"
        })
    },
    listOpen : function(oEvent) {
        // only display products from the selected category (if any)
        var aSelectedKeys =
Object.getOwnPropertyNames(oCategoriesFFL.getSelectedKeys());
        if(aSelectedKeys.length > 0) {

            var oBinding = this.getBinding("items");
            var oUserFilter = new sap.ui.model.Filter(
            "CategoryName",
            sap.ui.model.FilterOperator.Contains,
            oCategoriesFFL.getSelectedKeys()[aSelectedKeys[0]]);
            var oFinalFilter = new sap.ui.model.Filter([ oUserFilter ], true);
```

```
            oBinding.filter(oFinalFilter, sap.ui.model.FilterType.Control);
        }
    },
});
oProductsFFL.setModel(oProductsModel);
// create the facet filter control
var oFF = new sap.m.FacetFilter(genId(), {
    lists : [ oCategoriesFFL, oProductsFFL ],
});
```

Feed Input

With this control you can enter and post text for a new feed entry.

The `sap.m.FeedInput` control allows users to enter and send plain text. It complements the `sap.m.FeedListItem` control to create a simple feed. You can use the `FeedInput` control if you need to input small amounts of text without formatting.

Overview

- Responsiveness
  The feed input control can be used in small and large containers due to its responsive behavior.
- Layout
  The feeder (that is, feed input) is used to write plain text and (depending on the context) to then create a note or feed entry by choosing the *Send* pushbutton. The `FeedInput` control provides the possibility to display a picture of the current user.
  If the user image has not yet been set, a generic placeholder is shown. If the app does not support user images at all, the image can be omitted.
  For more information about the layout options, see the Samples in the Demo Kit.
- Behavior
  Initially, the feeder contains an input prompt, and the *Send* pushbutton is disabled. You can click into the input field to focus on it.
  When the user starts to type, the input prompt disappears and the *Send* pushbutton is enabled and becomes more prominent.
  If the available width is less than 25 rem (for example, portrait mode on a mobile phone), the picture is removed. For more information, see the API Reference Documentation.

More Information

The `sap.m.FeedInput` control can be used in combination with the `sap.m.FeedListItem` control as a feed or notes control. For more information, see the Feed List Item [page 2312] control documentation.

> **i Note**
>
> If you need only one single text box instance, use the `sap.m.TextArea` control for multi-line text. For more information, see the Samples in the Demo Kit.

> Or you use the `sap.m.Text` control for single-line text. For more information, see the Samples in the Demo Kit.

Feed List Item

This control provides a set of properties for a feed, such as text, sender information and timestamp.

The `sap.m.FeedListItem` control is capable of displaying text accompanied by an optional user image.

For more information about this control, see the API Reference and the Samples in the Demo Kit.

Overview

- **Responsiveness**
  The `sap.m.FeedListItem` control can be used in both small and large containers.
- **Layout**
  The `FeedListItem` control consists of the user's name and an optional picture of the user who wrote the note or the update (optional). The name can contain a link that triggers a quick overview of the user's profile data. The actual text written by the user follows the name. The `FeedListItem` control includes the `sap.m.FormattedText` control that enables you to use HTML formatted text.
  For more information about about the `FormattedText` control, see the API Reference and the Samples in the Demo Kit.
- **Behavior**
  When the text exceeds a certain number of characters (default value can be overwritten by the consuming application), the rest of the text is truncated and a *MORE* link appears. Clicking the link shows the entire text, and the link is renamed to *LESS*. Clicking *LESS* collapses the text back to its original truncated length.
- **Actions**
  Each item in the feed may also include an optional *More* button that provides access to additional actions that the user can perform on this feed item. The actions available for a feed item are specified in its `actions` aggregation and are defined using `FeedListItemAction` elements.
  For more information about about the `FeedListItemAction` element, see the API Reference in the Demo Kit.

More Information

The `sap.m.FeedListItem` control can be used in combination with the `sap.m.FeedInput` control as a feed or notes control. For more information, see Feed Input [page 2311].

Flex Box

The `sap.m.FlexBox` control allows to develop layouts which adjust to the available space and avoid unused space or overflow.

User interfaces often have to adapt to different screen sizes. Therefore, building user interfaces in a way that a single layout reliably fits the available screen real estate is challenging. `FlexBox` controls can be nested to create more complex layouts.

The two main uses of a FlexBox control are:

- Two-dimensional layouting
- Flexible layouts

Getting Started With FlexBox

For a flexible box layout, create a `FlexBox` control and add any kind of controls to it.

You can either use the `addItem` method (see option 1), or the `items` aggregation of a configuration object (see option 2).

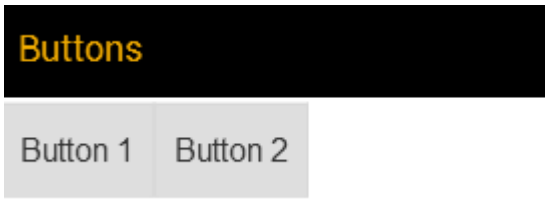
Option 1

```
var oMyFlexbox = new sap.m.FlexBox();
oMyFlexbox.addItem( new sap.m.Button({text: "Button 1"}) );
oMyFlexbox.addItem( new sap.m.Button({text: "Button 2"}) );
```

Option 2

```
var oMyFlexbox = new sap.m.FlexBox({
  items: [
    new sap.m.Button({text: "Button 1"}),
    new sap.m.Button({text: "Button 2"})
  ]
});
```

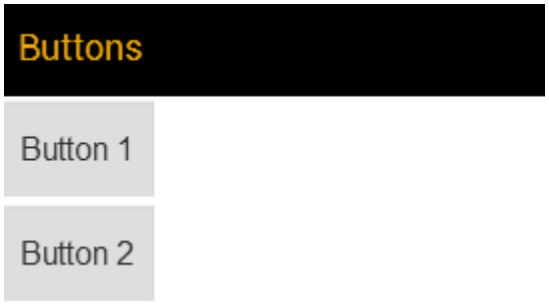
The following figure gives an example how the result looks like if used inside a mobile app page. The necessary code is not shown here.



Layout properties

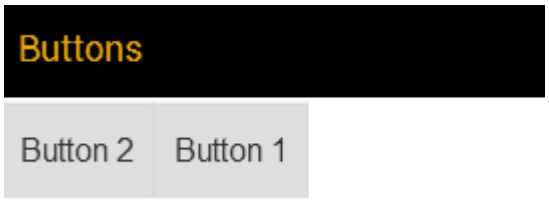
Some properties that affect the layout need to be set in the `FlexBox` control. Other properties can be attached to the controls which are placed inside the `FlexBox` by means of the `layoutData` aggregation. The layout direction, for example is set in the `FlexBox` as follows:

```
var oMyFlexbox = new sap.m.FlexBox({
  items: [
    new sap.m.Button({text: "Button 1"}),
    new sap.m.Button({text: "Button 2"})
  ],
  direction: "Column"
});
```



The order is attached to the button inside a `FlexItemData` object as follows:

```
var oMyFlexbox = new sap.m.FlexBox({
  items: [
    new sap.m.Button({
      text: "Button 1",
      layoutData: new FlexItemData({order: 2})
    }),
    new sap.m.Button({text: "Button 2"})
  ]
});
```



> **i Note**
>
> The `FlexBox` control is a wrapper for the flexible box layout properties in CSS. The control renderer sets the CSS properties (including prefixed versions where necessary) on the appropriate HTML elements. The actual layouting is done by the browser.

The controls that you place in the `FlexBox` control are each wrapped in a `DIV` or `LI` element, depending on the `renderType` property. All elements are placed inside another `DIV` or `UL` container, again depending on the `renderType`. If you use `Bare` as `renderType`, elements will be rendered without a wrapping HTML tag. The

outermost element represents the so-called *flex container* while its child elements are *flex items*. The HTML structure resulting from all of the examples above looks as follows:

```
<div class="sapMFlexBox">
       <div class="sapMFlexItem">
              <button id="__button1">Button 1</button>
       </div>
       <div class="sapMFlexItem">

              <button id="__button2">Button 2</button>
       </div>
</div>
```

> **i** Note
>
> The `layoutData` properties that you can attach to a control are applied to its wrapper element with *sapMFlexItem* class. This is because browsers currently only support these properties on some elements, for example `DIV`.

The two additional controls `HBox` and `VBox` are `FlexBoxes` that are fixed to horizontally or vertically layout their children.

Important FlexBox Layout Concepts

Introduction of important concepts for FlexBox layouts.

Main Axis and Cross Axis

A FlexBox layout has a direction in which child elements are laid out. The default direction is **Row** and rows are laid out horizontally in reading direction. This defines the **main axis**. The **cross axis** in this case is vertical.

You can change the layout direction property to **Column**, which results in a vertical main axis and a horizontal cross axis. This is important for the align properties.

> **i** Note
>
> If browsers support vertical text flows, the direction of a row can also be vertical. For now this is not an issue and can be ignored.

In addition to Row and Column, the flex direction can be set to **RowReverse** and **ColumnReverse** which will reverse the layout direction.

Two-Dimensional Alignment

You can determine where the flex items are aligned in a FlexBox layout. For the alignment you use the following two properties: **justifyContent** and **alignItems**. The `justifyContent` property sets the alignment along the main axis while `alignItems` acts on the cross axis.

Both properties accept the values **Start**, **Center** and **End**. This results in nine possible combinations, for example

- *justifyContent = End* and *alignItems: Start* places the items in the upper right corner of a horizonzal FlexBox
- If you set the direction property to *Column*, the main axis would be vertical. Combined with *justifyContent = End* and *alignItems: Start*, the items are placed in the lower left corner.
- By reversing the main axis with *direction = ColumnReverse* the layout starts from the bottom. In this case, *justifyContent = End* refers to the top of the FlexBox.
- *justifyContent* has the additional value *SpaceBetween*. This setting places the first and the last item at the extremes of the main axis. Any other items will be distributed evenly between these two.

For *alignItems* two additional values exist: *Baseline* and *Stretch*. *Baseline* takes the first line of text of each flex item and aligns their baselines. This can be useful if different font sizes are used. *Stretch* makes the flex items take up the whole space along the cross axis of the FlexBox. This is useful if all items should have the same size regardless of the amount of content.

Flexibility

You can let the browser handle the distribution of elements. This ensures that they always fill the available space along the main axis. To do this, set a flexibility factor on the flex items.

The property to control the flexibility is called `growFactor`. It is set on a flex item object by means of `FlexItemData` on the `layoutData` aggregation. The flex layout algorithm determines the "natural" width of the flex items. If there is space left, this space is distributed among the flex items according to their relative `growFactor`. If, for example, a horizonzal flex container is 300px wide and contains two elements of 100px each, 100px would remain. If the `growFactor` for both flex items is set to 1, both get 50px extra, thus making them 150px wide. If the `growFactor` is set to 3 for one item and to 1 for the other item, the first item gets additional 75px (¾ of 100px) of the remaining space and the second item 25px (¼ of 100px). If the `growFactor` is set to its default value of 0, the item is inflexible and both items would keep their width of 100px.

> **i** Note
>
> The flex algorithm distributes the **remaining** space, and **not** the whole space in the FlexBox. Therefore, the resulting widths of the items are not necessarily proportional to the `growFactor`.

To achieve a proportional width according to the `growFactor`, set the width of all items to 0 via CSS. The sum of the "natural" widths of all items is then also 0. The remaining space, however, now equals the full space of the `FlexBox`. This space is then distributed based on the `growFactor`. For the example above with `growFactor` set to 3 and 1, setting the width of the flex items to 0 via CSS results in a width of 225px (¾ of 300px) for the first item and 75px (¼ of 300px) for the second item.

> **⚠** Caution
>
> Once you set a `growFactor` for any item, the flex layout algorithm ignores the `justifyContent` property of the FlexBox because the items take up all available space anyway. There would be no difference between the different values.

Generic Tile

The `GenericTile` control is the basic concept that displays any kind of content within a tile comprising for example news, feeds, images, micro charts, or numeric content.

Overview

`GenericTile` controls are responsive and adapt their size to the size of the devices they are used on: the control is available in small and large size and depending on the media screen size, the appropriate size is chosen automatically. Therefore, the `size` property is deprecated and should not be used anymore.

Layout

The `GenericTile` control provides a header area, a content area, and a status area. Within these areas, different data can be displayed depending on the app the tile belongs to, for example, text in the header area, charts or an icon in the content area, or text only in the footer area.

- Header area
  The headline and a subtitle can be displayed. The header area is mandatory, but if the `NewsContent` control is used in the content area, no header and subheader should be used.
  If you use `HeaderMode`, up to five lines of header can be displayed and the content area is not displayed.
- Content area
  The `TileContent` control that represents the content area is the universal container for different content types. This means that the `TileContent` control can contain one of the following controls:
  - `NumericContent` (`frameType` 1x1): numeric value and icon
  - `FeedContent` (`frameType` 2x1): up to two lines of text
  - `NewsContent` (`frameType` 2x1): text and background image
  - `MicroCharts` (`frameType` 1x1): charts
  - `ImageContent` (`frameType` 1x1): images and icons
- Status area
  A status can be displayed. The status area is not mandatory.

SlideTile

`GenericTile` controls of the 2x1 `frameType` can be implemented as `SlideTile` control. The `GenericTile` control added to the `SlideTile` control appears alternately, showing one content at a time. The animated content of the slide tile includes a navigation option to pause the slide show or to navigate forward or backward to the slide. You can use `SlideTile` controls, for example, in day-to-day business to display news.

Tooltip Rendering

The `GenericTile` control generates a tooltip only if the header or subheader are truncated. In that case, the tooltip contains the full text of the truncated header or subheader.

If the `GenericTile` control contains a `MicroChart` control in its content area, the generated tooltip contains information about the embedded `MicroChart` content.

If an application provides an own tooltip, the generated tooltip of the `GenericTile` control is overwritten and the tooltip of the application is displayed. You can suppress the tooltip for your application by providing a tooltip consisting of whitespaces only. The ARIA label is not changed.

Line Mode

With this mode, you can switch the visual representation of the `GenericTile` from the rectangular format to an in-line format only by changing the value of the `mode` property and keeping all other settings as already set. After the switch, the API for the tiles and links stays consistent, and the control's ID and the contents stay the same. Only the header and subtitle are rendered.

API Reference/Sample

For more information about the `GenericTile` control, see the API Reference in the Demo Kit and the sample in the Demo Kit.

Constraints

With SAPUI5 1.34, the following controls are moved from the `sap.suite.ui.commons` library to the `sap.m` library:

- `GenericTile`
- `TileContent`
- `FeedContent` (formerly `JamContent`)
- `NewsContent`
- `NumericContent`
- `SlideTile` (formerly `DynamicContainer`)

The following controls have not been transferred to the `sap.m` library and are no longer used:

- `GenericTile2X2`
- `TileContent2X2`

If you have already included one of these controls before SAPUI5 1.34, a wrapper ensures that the embedding still works for each control. To benefit from all enhancements and new features for one of these controls as of SAPUI5 1.34, you need to switch to the controls in the `sap.m` library. With SAPUI5 1.34, all these controls in the `sap.suite.ui.commons` library are marked as **deprecated**.

Image

Additional information on `sap.m.Image`

Supporting Different Pixel Densities

Some mobile devices, starting with iPhone4 and iPad3, have a display with a high density of pixels (four pixels where older models would only have one pixel). They're called Retina Displays by Apple to suggest they're as

crisp and clear as the eye can see. They use four physical pixels to display one logical CSS pixel. This way images can be displayed much sharper when given twice as large as required because internally the device can use many more pixels to display all details of the image. Browsers on those displays do this automatically when images are scaled down.

Some devices support higher resolution images while others don't. We therefore recommend that SAPUI5 app developers provide image resources for all relevant densities to provide a crisp and clear display of images on devices with Retina Display.

The `sap.m.Image` control automatically chooses the right density depending on the device on which it's displayed. If an image of a certain density isn't available, the image control falls back to a default image, which must be provided as well.

The image control is also used implicitly by other controls, for example:

- `sap.m.Button`
- `sap.m.SegmentedButton`
- `sap.m.StandardListItem`

> ⚠ Caution
>
> If you don't have higher resolution images, you must set the `densityAware` property to `false` to avoid unnecessary roundtrips.

Example

Assume that the following controls are displayed on a device with high-density screen (`window.devicePixelRatio` is 2):

```
new sap.m.Image({
        densityAware: false, // tells the image control that there are no
different optimized image variants
        src : "first.png"   // therefore Image control will directly load
first.png
})
new sap.m.Image({
        src : "second.png"    // Image control will first look for second@2.png,
then fall back to second.png
})
```

The first image control is told that there are no image files for the different densities, so it directly loads `first.png`. This image looks as good as other images on Retina Displays.

The second image control first attempts to load `second@2.png`, which is twice as large as the normal image and is scaled down for display to look crisp on Retina Displays. If this file doesn't exist, it falls back to `second.png`, but this fallback causes an additional server request.

Naming Conventions

Density-related images are loaded if you provide an image name with density awareness in the following format:

```
[imageName]@[densityValue].[extension]
```

Supported densities are 1.5 and 2. The following example shows a set of images with different densities:

- `detail.png` (default)
- `detail@1.5.png`
- `detail@2.png`

> **i Note**
>
> `detail@0.75.png` isn't supported and uses the standard image for a such low density device.

Related Information

API Reference: `sap.m.Image`
Samples: `sap.m.Image`

List, List Item, and Table

`sap.m.List` and `sap.m.Table` both inherit from the abstract `sap.m.ListBase` and provide the features used for lists and tables.

`sap.m.List` and `sap.m.Table` provide the following features:

- Selection modes, such as `Single` and `Multi`
- Navigation types such as Navigation, Active or Detail
- Swipe for Action
- Growing feature
- Grouping

Related Information

Samples: `sap.m.List`
Samples: `sap.m.Table`

Lists

Lists have properties and events and they contain list items that inherit from `sap.m.ListItemBase`, which provides navigation, selection and event features. The list item type determines the way the list item interacts by providing additional features.

List Properties

Lists can have the following properties:

- The `mode` property defines the appearance of the left area of a list item. You can show a single selection, multi selection, delete buttons, or none of these. The `mode` property can have the following values:
  - `None` (default)
  - `SingleSelect` (on the right side)
  - `SingleSelectLeft` (on the left side)
  - `SingleSelectMaster` (without select control for use cases like the split app, by default the ! includeItemInSelection = true)
  - `MultiSelect`
  - `Delete`
- The `includeItemInSelection` property (default: `false` defines the tap handling of a list item. By default, you can select an item by tapping the radio button or check box. To use the whole list item tap for selecting an item, change the property value to `true`. This property is only relevant in selection mode.
- The `showUnread` property (default: `false` decides whether an 'unread' indicator is added to each list item. When active, it shows a blue bubble for unread list items.
- The `showNoData` property (default: `true` shows a text to the user if a list has no content. The default text is 'No data'. You can use the `noDataText` property to change the default text.
- The `noDataText` property (default: 'No data') can be used to change the text that is displayed when the list has no content and the `showNoData` property is set to `true`.

Swipe For Action

A user can swipe left on a list item to bring in a control, such as a button, and initiate an action for this item. This control is defined through the `swipeContent` aggregation of the list and is displayed to the right or center of the list item. For more information, see Swipe List for Action [page 2327].

List Events

Events are available for selecting, deleting and swiping in lists. The selection mode fires a `select` event and the deletion mode a `delete` event. A swipe left fires a `swipe` event. These events contain information about the list item that caused the event.

- `select (listItem)`
- `delete (listItem)`
- `swipe (listItem)`, see Events [page 2328]

Rerendering

A list is rerendered together with all of its list items when the data of a bound model is changed. Due to the limited hardware resources of mobile devices, this may cause delays for lists that contain many list items. For this reason, we do **not** recommend using a list for these use cases.

List Items

All list items inherit from `ListItemBase`, which contains the features for navigation, selection, and event.

Five different types of list items are available, which determine the way a list item interacts. A list item has a content area (main area), which may fire a `press` event, and a navigation area on the right hand side, which may fire a `press` or a `detailPress` event. The `type` property for each list item defines the events that are fired. You need to define a type to decide which visual feedback if given by a list item after it has been touched. The five available types are as follows:

- `StandardListItem` provides an image, title, and description
- `ActionListItem` provides a center-aligned text and is used to trigger actions
- `DisplayListItem` has a label and a value
- `InputListItem` provides a label and allows you to embed controls enabling user input, such as: input button, radio button, checkbox, slider, select, search
- `CustomListItem` can be used for all list items that are not provided by SAPUI5. You can use it to build your own content and aggregate it.

Except for `ActionListItem`, list items do **not** by default fire an event unless it is configured with a type that defines how events are fired. The following table shows the different combinations of list item types and events:

Type	Press Event	detailPress Event	Icon	Active Feedback
Inactive (default)	--	--	--	--
Active	yes	--	--	yes
Navigation	yes	--	>	yes
Detail	--	yes	(>)	--
DetailAndActive	yes	yes	(>)	yes (content only)

As mentioned above, `ListItemBase` has an unread indicator property, which shows a blue bubble. This has to be enabled by the lists `showUnread` property. For selections on each list item a selected property (default: `false`) exists. Another feature is the `counter` property (default: `null`), which shows integer numbers except zero. If the number is zero, the counter is hidden. Properties for `ListItemBase`:

- `unread` (default: `false`)

- `selected` (default: `false`)
- `counter` (default: `undefined`)

The following events are available for `ListItemBase`:

- `press`: This event is fired when the content of a list item is tapped.
- `detailPress`: This event is fired when the detail icon of a list item is tapped.

List Item Types

The following types are available for list items:

- `ActionListItem`
  In addition to the features inherited from `ListItemBase`, this type provides the `text` feature, which enables you to set a center aligned text. This is a simple list item for triggering actions. The following code snippet shows an example:

  ```
  <List headerText="Actions">
    <ActionListItem text="Reject" />
    <ActionListItem text="Accept" />
  </List>
  ```

- `DisplayListItem`
  In addition to the features inherited from `ListItemBase`, this type provides the `label` feature to set a label and the `value` feature to set a value. The following code snippet shows an example:

  ```
  <DisplayListItem
    label="Name"
    value="{SupplierName}" />
  ```

- `InputListItem`
  In addition to the features inherited from `ListItemBase`, this type provides the `label` feature to set a label and the `content` feature that can be aggregated with controls, for example a radio button or a search control. The following code snippet shows an example:

  ```
  <List>
    <InputListItem label="WLAN">
      <Switch state="true" />
    </InputListItem>
    <InputListItem label="Price (EUR)">
      <Input
        placeholder="Price"
        value="799">
        type="Number" />
    </InputListItem>
</List>
  ```

- `StandardListItem`
  In addition to the features inherited from `ListItemBase`, this type provides the following additional features:
  - `title`
  - `description`
  - `icon`: The icon is displayed on the left hand side of the list item and can be shown with or without an inset.

`label` feature to set a label and the `content` feature that can be aggregated with controls, for example a radio button or a search control. The following code snippet shows an example:

```
<List headerText="Products"
    items="{/ProductCollection}">
    <items>
      <StandardListItem
        title="{Name}" />
    </items>
  </List>
```

- `CustomListItem`
  In addition to the features inherited from `ListItemBase`, this type provides the option to aggregate content. You can use the `CustomListItem` for all list items that are not available in SAPUI5 standard, build your own content, and aggregate it.

```
<CustomListItem type="Inactive">
    <Label text="A first custom list item ..."  class="content"/>
    <Button text="Press me!" class="content"/>
</CustomListItem>
```

API References

- sap.m.List
- sap.m.ListItemBase

Custom List Item

You can use the `sap.m.CustomListItem` control to create your own layout if the other list items available in SAPUI5 do not fit your needs.

Available List Items in SAPUI5

SAPUI5 contains several list items that are used with the `sap.m.List` control to serve different standard scenarios. These are outlined in the table below, along with `sap.m.ColumnListItem`, which is used together with the `sap.m.Table` control:

List Item	Used for...
`sap.m.StandardListItem`	Displaying list content with a title description, icon and info
`sap.m.DisplayListItem`	Displaying name/value pairs
`sap.m.InputListItem`	Building a form-like user interface on phones

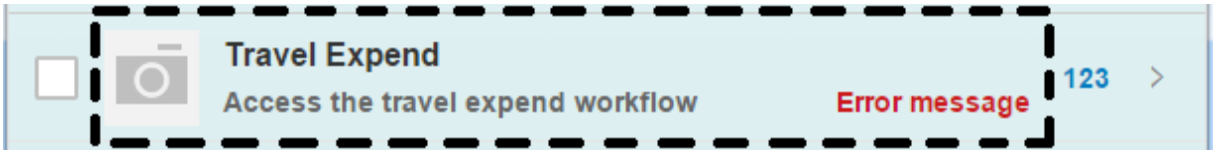
List Item	Used for...
`sap.m.ActionListItem`	Triggering an action directly from a list
`sap.m.FeedListItem`	Displaying a standard UI for feeds. For more information, see Feed List Item [page 2312]
`sap.m.ObjectListItem`	Providing a quick overview for an object within a list
`sap.m.ColumnListItem`	Providing responsive table design in combination with `sap.m.Table`
`sap.m.CustomListItem`	Creating custom list items if none of the list items above are suitable

If none of the predefined list items (the first seven entries in the table above) fit your scenario, you can also create your own layout by using `sap.m.CustomListItem` directly, or create a new control that inherits from `sap.m.CustomListItem`.

For more information about the different list items, refer to the corresponding API documentation.

Structure of a List Item

A list item can be split into three parts, as shown in the following graphic:



The parts that are to the left and right of the dotted area are part of the `ListItemBase` and are used to display the selection and deletion mode, as well as different list item type indicators such as navigation, details, and counter. The `Unread` indicator also comes from the `ListItemBase` and when it is set, any unread text will be displayed in bold format.

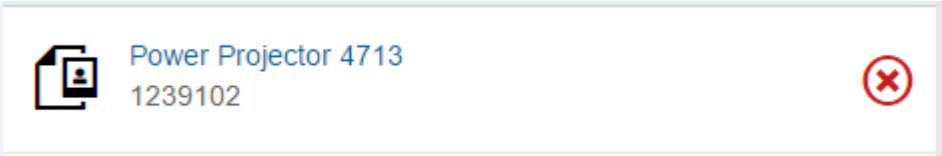
The dotted area is the area in which the content of a list item is placed. If you are using `sap.m.CustomListItem`, **all** of the content will be placed there. The section below explains how to use `sap.m.CustomListItem` in more detail.

Using the `sap.m.CustomListItem` Control

As mentioned above, you can either use `sap.m.CustomListItem` directly by adding any content via content aggregation, or you can create your own control that inherits from `sap.m.CustomListItem` if you need a more sophisticated list item featuring your own properties, styling, and complex layout. Below is an example showing how you can use the `sap.m.CustomListItem` together with `sap.m` controls.

```
<List headerText="Custom Content" mode="Delete" items="{path: '/
ProductCollection'}" >
    <CustomListItem>
        <HBox>
            <core:Icon size="2rem" src="sap-icon://attachment-photo"
class="sapUiSmallMarginBegin sapUiSmallMarginTopBottom" />
            <VBox  class="sapUiSmallMarginBegin sapUiSmallMarginTopBottom" >
                <Link text="{Name}" target="{ProductPicUrl}"
press="handlePress"/>
                <Label text="{ProductId}"/>
            </VBox>
        </HBox>
    </CustomListItem>
</List>
```

The example above creates an attachment list item that displays an attachment title as a link, as shown in the graphic below. Clicking on the link will open the attachment. Below the attachment title, we want to display the details of the attachment, so we have used `sap.m.HBox` and `sap.m.VBox` for basic layouting. Data binding is also supported, and here it assumes that a model featuring `ProductPicUrl` and `ProductId` properties is used.



The following example shows how to use a notepad control as a reusable control in an `sap.m.CustomListItem`. It assumes you want to build a product list item that shows an image of the product and displays its details:

```
sap.ui.define(["sap/ui/core/Control", "sap/m/Image"], function (Control, Image) {
    var MyListItemContent = Control.extend("my.control.ListItemContent", {
        metadata: {
            properties : {
                "name": {type: "string", defaultValue: ""},
                "description": {type: "string", defaultValue: ""},
                "price": {type: "string", defaultValue: ""},
                "currency": {type: "string", defaultValue: ""},
                "image": {type: "string", defaultValue: ""}
            },
            events: {
                "myTap": {},
            },
        },
        init: function(){
            this._image = new
Image({src:"<myImageSrc>"}).addStyleClass("myImageCSS").setParent(this);
        },
 renderer: {
            apiVersion: 2,  // see 'Renderer Methods' for an explanation of this
flag
            render: function(oRm, oControl) {
                oRm.openStart("div", oControl);
                oRm.class("listItemCSS");
                oRm.openEnd();
                    oRm.renderControl(oControl._image);
                    oRm.openStart("div").class("descCSS").openEnd();
                    oRm.text(oControl.getDescription());
                    oRm.close("div");
                    oRm.openStart("div").class("priceCSS").openEnd();
                    oRm.text(oControl.getPrice());
                    oRm.close("div");
                    oRm.openStart("div").class("curCSS").openEnd();
```

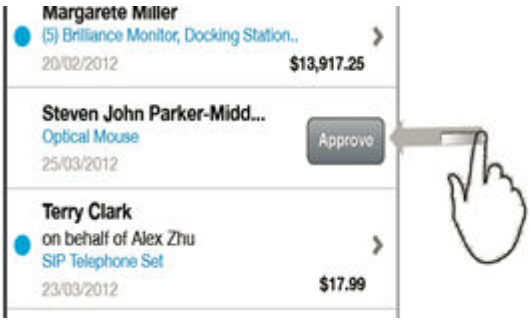
```
                    oRm.text(oControl.getCurrency());
                    oRm.close("div");
                    oRm.openStart("div").class("nameCSS").openEnd();
                    oRm.text(oControl.getName());
                    oRm.close("div");
                oRm.close("div");
            }
        }
    });
    //example how to react on browser events and convert them to control events
    ListItemContent.prototype.ontap = function(){
        //your own tap logic
        this.fireMyTap({});
    };
    return ListItemContent;
});
```

After we've created this notepad control above, we consume it in the `sap.m.CustomListItem` as a content aggregation, as shown here:

```
// "CustomListItem" required from "sap/m/CustomListItem"
var oCustomListItem = new CustomListItem({content: [new MyListItemContent({
    //usual control setup
})]});
```

Swipe for Action

If a user swipes left on a list item within a list or table, you can bring in a control, for example a button, to initiate an action for this item. The button is displayed on the right-hand side of the list item. An example is shown in the following graphic:



Aggregation

This control is defined by the `swipeContent` aggregation of the list or table. You can add any control as `swipeContent`, but keep in mind that your content cannot be higher than a list item. See the following examples:

- Button `swipeContent`

  ```
  <List
  ```

```
    headerText="Products"
    items="{/ProductCollection}" >
    <StandardListItem
      title="{Name}"
      description="{ProductId}"
      icon="{ProductPicUrl}"
      iconDensityAware="false"
      iconInset="false" />
    <swipeContent>
      <Button
        text="Delete Item"
        type="Reject"
        press="handleReject" />
    </swipeContent>
  </List>
```

- Control combination as `swipeContent`

```
new sap.m.List({
    swipeContent : new sap.m.HBox({
        items : [
      <List
    headerText="Products"
    items="{/ProductCollection}" >
    <StandardListItem
      title="{Name}"
      description="{ProductId}"
      icon="{ProductPicUrl}"
      iconDensityAware="false"
      iconInset="false" />
    <swipeContent>
      <Button
        text="Delete Item"
        type="Reject"
        press="handleReject" />
    </swipeContent>
  </List>
```

Events

`List` and `Table` provide a `swipe` event when a user swipes left to bring in a control on the right-hand side of the list item within a table or list. This event is fired before the `swipeContent` is shown and contains the following three important parameters:

- `listItem`: List item that fired the swipe event
- `swipeContent`: Specifies the `swipeContent` control to be shown on the right-hand side of a list item
- `srcControl`: Specifies the control that fired the swipe event

This means that you can dynamically change the swipe content according to the respective list item. If a list item has not yet been approved, for example, then the *Approve* button is shown. After approval or if it is already approved, the *Disapprove* button is shown. See the following example:

```
<List
    id=...
    swipe="handleSwipe" ... >
    ...
    <swipeContent>
      <Button
```

```
        text="Approve Item"
        type="Accept"
        press="handleApprove" />
    </swipeContent>
  </List>
```

And this is how it looks in the controller:

```
handleSwipe: function(e) {   // register swipe event
        var oSwipeListItem = e.getParameter("listItem"),    // get swiped list
item from event
            oSwipeContent = e.getParameter("swipeContent"); // get swiped
content from event
        // Check swiped list item if it is already approved or not
        if (oSwipeListItem.data("approved")) {
            // List item is approved, change swipeContent(button) text to
Disapprove and type to Reject
            oSwipeContent.setText("Disapprove").setType("Reject");
        } else  {
            // List item is not approved, change swipeContent(button) text to
Approve and type to Accept
            oSwipeContent.setText("Approve").setType("Accept");
        }
    },
```

Swipe events can be cancelled. The built-in controls that work with swipe left events like `Switch` or `Slider` cancel a swipe event by default. If you also want to disable swipe events for your custom use case, you can call the `preventDefault` method of the event object, as shown in the following example:

```
handleSwipe : function(e) {
         // get which control inside the list item fired swipe event
        var oSrcControl = e.getParameter("srcControl");
        // check if the event is coming from Input
        if (oSrcControl instanceof sap.m.Input) {
            e.preventDefault();   // cancel swipe
        }
    },
    ...
```

Methods

`List` and `Table` provide the following two swipe methods:

- `swipeOut( [callback] )`: After `swipeContent` is shown, the user can interact with the control, for example tap it. After this interaction, for example on tap event, you can use this method to hide `swipeContent` from the screen. By default, swipe for action works in auto-hide mode. This means that if a user tries to tap inside the list but outside the `swipeContent`, then the `swipeContent` hides automatically. After you call this method, `swipeContent` hides with animation and if you need to run code after the animation you can simply add a callback function to this method as a first parameter.
- `getSwipedItem()`: This method returns the currently swiped list item. When no item is swiped, null is returned. The `swipeContent` events, for example tap, are a good place to use this method to get information for which list item `swipeContent` is shown.

The following example shows a delete scenario:

```
tap : function(e) {
```

```
   var oList = this.getView().byId("myList"); // get the list using its Id
   var oSwipedItem = oList.getSwipedItem(); // Get which list item is swiped to
delete
   oList.removeAggregation("items", oSwipedItem); // Remove this aggregation to
delete list item from list
   oList.swipeOut(); // we are done, hide the swipeContent from screen
}
....
```

Properties

The `swipeDirection` property for lists and tables is used to configure the direction of the swipe event. This property accepts an enumeration from `sap.m.SwipeDirection?` with the following values:

- `LeftToRight?`: Swipe from left to right
- `RightToLeft?`: Swipe from right to left
- `Both`: Both directions (left to right, or right to left)

The default value is `Both`, but in some use cases we recommend that you change this property, for example to prevent swipe conflicts.

Creating Tables

A Table basically consists of columns (`sap.m.Column`) and rows. The rows, defines as `sap.m.ColumnListItems` consist of cells.

Procedure

1. To build a table, we first need to define columns. For this purpose you can define the Column control with the following properties or aggregations:

Property or Aggregation	Description
header	Defines column header. Any control can be used but most likely **Label** or **Text** control. If any column has header definition then header line gets visible for all columns.
footer	Any control can be assigned to be displayed in the column footer. If at least one column has a footer definition, then the footer line is displayed for all columns.
width	Defines the width of the column. If you leave it empty then this column covers the remaining space.
hAlign	Defines the horizontal alignment(Begin, Center, End, Left, Right) of the column content. Controls with a textAlign property inherit the horizontal alignment from Column hAlign property.

Property or Aggregation	Description
vAlign	Defines the vertical alignment of column cells. Possible values are Top, Middle, and Bottom. This property does not affect the vertical alignment of header and footer.
visible	Specifies whether the column is visible. Invisible columns are not rendered.

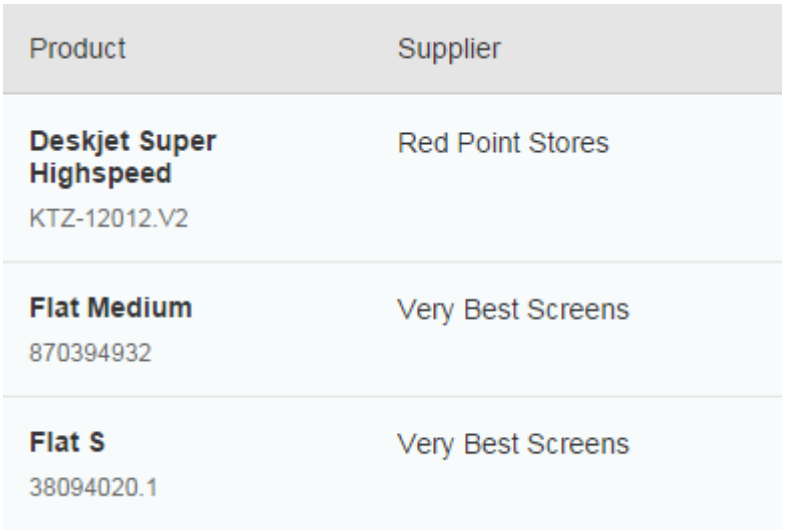
2. Sure, we have more properties to make it responsive and to change the design of a column, but we will explain this later. Now, let's see how we can define a right aligned column header:

```
<Table>
    <columns>
      <Column
        width="12em">
        <Text text="Product" />
      </Column>
       <Column
        minScreenWidth="Tablet"
        demandPopin="true">
        <Text text="Supplier" />
      </Column>
    </columns>
</Table>
```

3. To build a Table, you have to define table rows. For this purpose you use `ColumnListItems`. `ColumnListItems` have a cell aggregation (one to many) which defines cells in one row according to the column definition. Let's build a real table example to understand it better. Here is the implementation:

```
<Table>
    <columns>
      <Column
        width="12em">
        <Text text="Product" />
      </Column>
       <Column
        minScreenWidth="Tablet"
        demandPopin="true">
        <Text text="Supplier" />
      </Column>
    </columns>
<items>
      <ColumnListItem>
        <cells>
          <ObjectIdentifier
            title="{Name}"
            text="{ProductId}"
            class="sapMTableContentMargin" />
          <Text
            text="{SupplierName}" />
        </cells>
      </ColumnListItem>
    </items>
  </Table>
```

And that is what we have built:



The `mergeFunctionName` property holds the function that the column merge functionality uses to pull the property value to compare for duplicates. The default of `getText` can be used for the most common use cases, where an `sap.m.Label` or `sap.m.Text` control is used, but if you have another control with a different function to pull the comparison property value from, you can specify it as the `mergeFunctionName`. For example, the `sap.m.Icon` control has a `getSrc` getter function that returns the `src` property value - the icon's URI, which is a good candidate for comparison.

Next Steps

API References

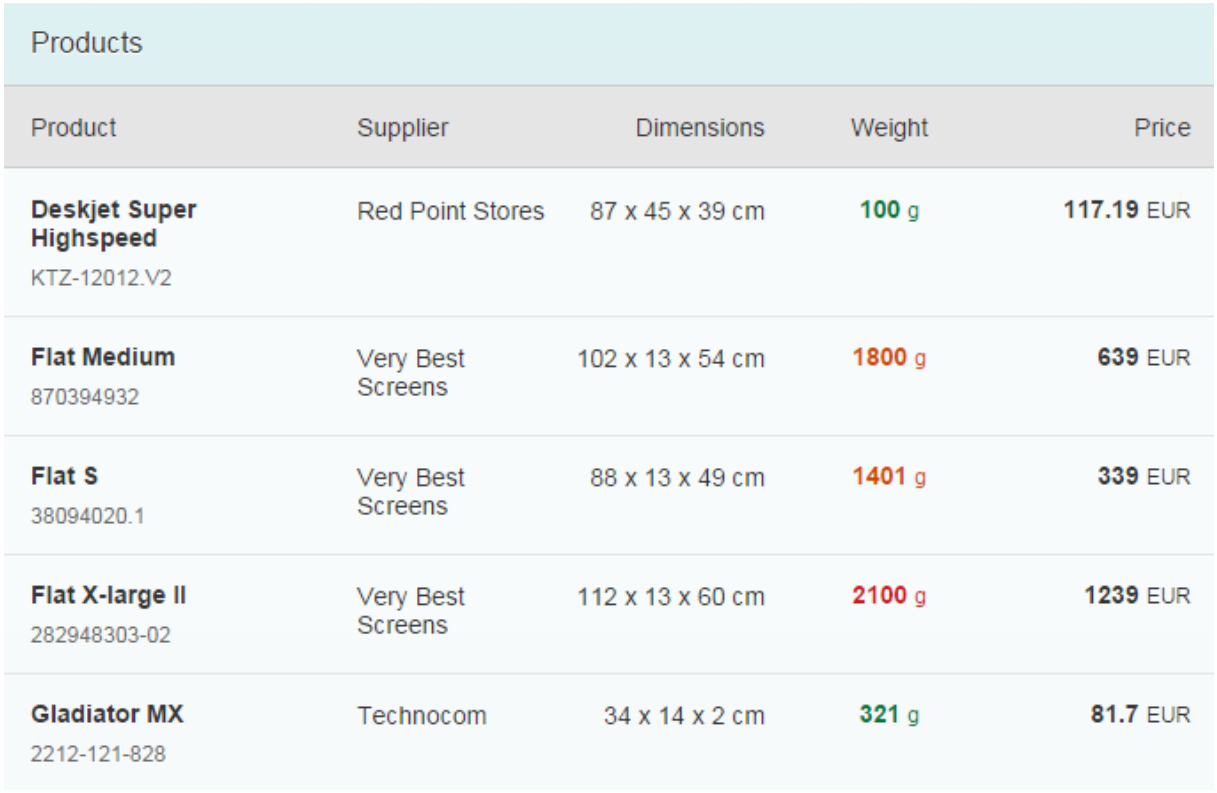
- sap.m.Column
- sap.m.ColumnListItem

Configuring Responsive Behavior of a Table

SAPUI5 supports column-based and row-based solutions to support flexible and clearly arranged tables.

One of the biggest challenges in responsive web design (RWD) is presenting tabular data. Large tables containing lots of columns simply don't fit on smaller screens and there is no easy way to reformat the table content with CSS and media queries for an acceptable visual display. To address this, our framework offers a column-based solution (column hiding) and row-based solution (pop-ins) for displaying tables responsively and both options are applicable at the same time. This may sound rather complicated, so let's look at an example.

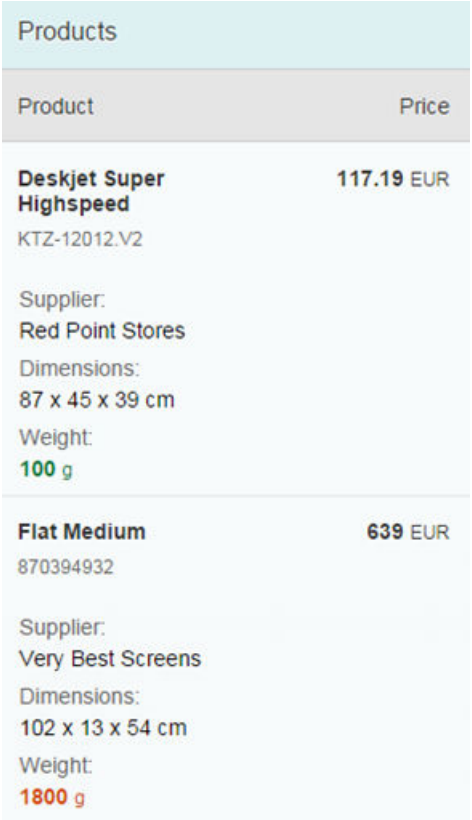
Say we want to build this nice table to display on a desktop:



On mobile devices, we know that we won't have enough space to show all these columns, so we need to ask ourselves which columns are most important. Let's say:

- *Product* and *Price* are most important. So they should never be hidden.
- *Supplier*, *Dimensions* and *Weight* are not particularly important, so we'll only show them as pop-ins.

If we apply these decisions we just made, our mobile devices should now look like this:



Responsive Column Control

You can control the responsive table design using the API of `sap.m.Column`. This control provides two properties to handle column hiding and pop-in.

1. `minScreenWidth`: This value defines the break point for the column visibility. For instance: An Apple iPhone 5 device has 568px x 320px resolution (dip/device-width), so if we assign 400px (or 25em based on 16px), then this column will not be visible for portrait mode (width 320px) but will be visible for landscape mode (width 568px). Instead of specifying in px or em, you can also assign one of the predefined `sap.m.ScreenSize` types like `Tablet` (for 600px) or `Desktop` (for 1024px). The default value for this property is an empty string, meaning this column will **always** be visible.
2. `demandPopin`: Depending on your `minScreenWidth`, the column can be hidden in different screen sizes. Setting this property to true shows this column as a pop-in instead of hiding it. The default value is false.

And that's it! All you need to know are these two variables for responsive tables. So if we go back to our original example for a minute:

- *Name* and *Status* columns should never be hidden. This is the default behavior of a column, so let's just let the default values (`minScreenWidth : ""` and `demandPopin: false`) do their job.
- *Model Number* column should be hidden for small devices, so our break point is `minScreenWidth : "Small"` and `demandPopin : false` (default value).
- *Quantity*, *Unit Price* and *Final Price* columns should go into our pop-in, so our break point is still `minScreenWidth : "Small"` but now with `demandPopin : true` to show the column in a pop-in.

- For example: On tablets and wider devices we'll have more space available, so we can show the *Final Price* column, but we'll revert to a pop-in for smaller devices. So here our break point should be `minScreenWidth : "Tablet"` and `demandPopin : true`.

> **i** Note
>
> Please note that in order to have a valid table design, at least one column should always be visible and should **not** go to the pop-in.

Defining Column Width

The `width` property of `sap.m.Column` can have any valid CSS size, for example, 100px, 6em, or 25%. The default value of the width is `empty`, which makes the column flexible by covering the available space.

There are a few things to keep in mind when defining the width of the column:

- You can use percentage values but you should be careful doing that: A value might be suitable for a desktop screen, but not for a mobile device. In this case, using an absolute width (for example, 200px or 4rem) can be a better option.
- Leave the most important column's width empty or set it to `auto` if your table contains columns that have the `demandPopin` property enabled.

  > → Tip
  >
  > Let's say you have a 100%-width table with four columns, each of which has a width of 200px and a viewport that is 800px wide. If you resize the viewport to 500px, you can still show two columns while the remaining two columns are rendered as pop-ins. The total width of the two main columns is 400px. However, the viewport is then 500px, and the table is 100%. In that case the browser takes over handling this. Google Chrome increases the width of the last column, and the Internet Explorer increases the width of the first column. If you configure `Selection` or `Navigation`, these are also rendered as columns. The width of these columns is then also changed by the browser, which can lead to unexpected results. So the best solution is leaving the most important column's width empty (or set to `auto`) so it can take up as much space as it needs. In our example, this will be 300px.
- Do not use percentage values for the width of all columns even if this adds up to 100% of the total column width.

  > → Tip
  >
  > What if there is a `Selection` (3rem width), `Navigation` (3rem width), or `Deletion` (3rem width)? In this case, the total width would be 100% plus 6rem. If the total width is less than 100%, for example, one column with 20% and the other column with 40%, the total width would be 60% plus 6rem. By default, `Table` itself is in fixed layout mode and has a width of 100%. The browser needs to split up the width as it does not fit a 100% width. In some cases, browsers might handle this correctly, but you should avoid it. As mentioned, leaving the most important column width empty or set to `auto` fixes this problem because then the column will be flexible and cover the available space.

For more information, see the Defining Column Width Sample.

API Reference

- sap.m.Column
- sap.m.ScreenSize

Table Design

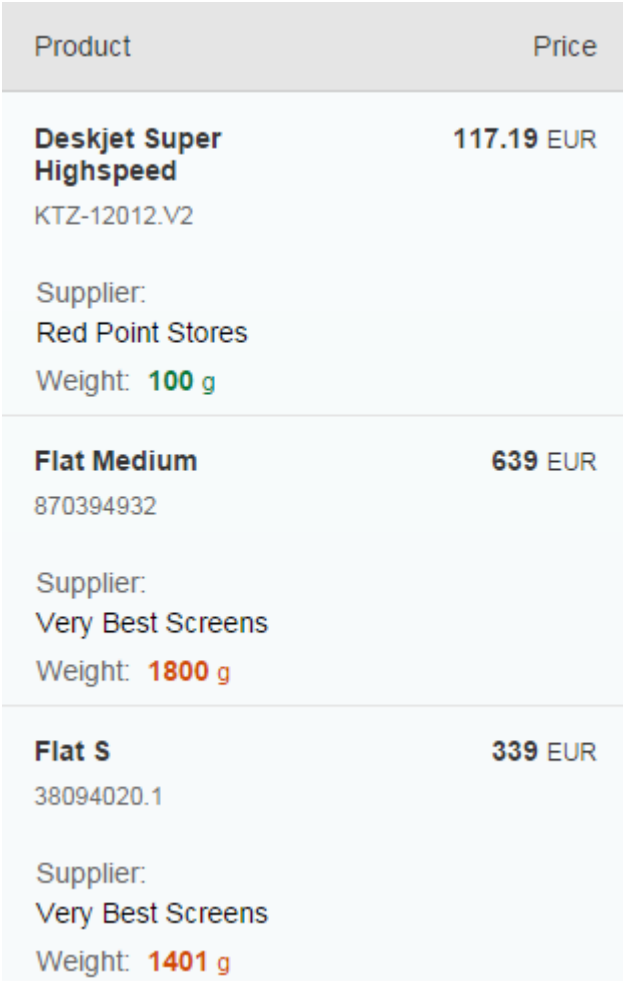
The table design in SAPUI5 can be changed by using various table and column features, such as the pop-in design.

Pop-in Design

When displaying information in a pop-in, the information, typically a column header and the column (cell) content, can be displayed in an `Inline` or in a `Block` display style. The difference between these two can be seen in the following example.

```
<columns>
   <Column
     width="12em">
     <Text text="Product" />
   </Column>
   <Column
     popinDisplay: "Block"
     minScreenWidth="Tablet"
     demandPopin="true">
     <Text text="Supplier" />
   </Column>
   <Column
     popinDisplay: "Inline"
     minScreenWidth="Tablet"
     hAlign="End">
     <Text text="Dimensions" />
   </Column>
   <Column
     popinDisplay: "Inline"
     minScreenWidth="Tablet"
     demandPopin="true"
     hAlign="Center">
     <Text text="Weight" />
   </Column>
   <Column
     hAlign="End">
     <Text text="Price" />
   </Column>
 </columns>
```

The *Dimension* column is hidden, the *Weight* column is displayed in a pop-in when the screen size is smaller than a desktop. The *Supplier* is to be displayed in `Block` mode, with the header and content arranged vertically, whereas the two prices are to be displayed in `Inline` mode, with the header and content arranged next to each other.



Merging Duplicate Values

When you have repeated values in your table, you can use the `mergeDuplicate` feature of the `sap.m.Column` control. There are two properties that are related to merging duplicate values:

- `mergeDuplicates`: Set this to true if you want duplicate values for the given column to be merged
- `mergeFunctionName`: Use this to specify the name of the getter function of the control in the column

Duplicate values will only be merged if they are adjacent. This means that you should sort your data first before binding it.

> **i** Note
>
> When using `sap.m.Column` in a table, the column merging feature is not supported when used in combination with two-way binding.

Here's an example of how to use the `mergeDuplicates` feature. We'll set up a table of sales data and display the data in a table, merging any duplicate regions. When the *Mix Up* button is pressed to lightly shuffle the

`salesFigures` array of objects, you will see that only adjacent duplicates are merged. Press the *Sort* button again to see the ideal merging.

```
<Table
    headerText="Products"
    items="{
      path:'/ProductCollection',
      sorter: {
        path: 'SupplierName',
        descending: false
      }
    }" >
    <columns>
      <Column mergeDuplicates="true">
        <header>
          <Text text="Supplier" />
        </header>
      </Column>
      <Column mergeDuplicates="true">
        <header>
          <Text text="Product" />
        </header>
      </Column>
      <Column
        minScreenWidth="Tablet"
        demandPopin="true"
        hAlign="End" >
        <header>
          <Text text="Dimensions" />
        </header>
      </Column>
      <Column
        minScreenWidth="Tablet"
        demandPopin="true"
        hAlign="Center" >
        <header>
          <Text text="Weight" />
        </header>
      </Column>
      <Column hAlign="End" >
        <header>
          <Text text="Price" />
        </header>
      </Column>
    </columns>
    <ColumnListItem>
      <Text text="{SupplierName}" />
      <ObjectIdentifier title="{Name}" text="{ProductId}"
class="sapMTableContentMargin" />
      <Text text="{Width} x {Depth} x {Height} {DimUnit}" />
      <ObjectNumber
        number="{WeightMeasure}" unit="{WeightUnit}"
        state="{
          path: 'WeightMeasure',
          formatter: 'sap.m.sample.TableMergeCells.Formatter.weightState'
        }" />
      <ObjectNumber
        number="{Price}"
        unit="{CurrencyCode}" />
    </ColumnListItem>
  </Table>
```

Products

Supplier	Product	Dimensions	Weight	Price
Red Point Stores	**Deskjet Super Highspeed** KTZ-12012.V2	87 x 45 x 39 cm	**100** g	**117.19** EUR
	Hurricane GX K47322.1	34 x 14 x 2 cm	**588** g	**219** EUR
	Laptop Case 214-121-828	53 x 34 x 7 cm	**1289** g	**78.99** EUR
	USB Stick 16 GByte XKP-312548	6 x 2 x 0.5 cm	**11** g	**17.19** EUR
	Laser Allround Pro 89932-922	42 x 29 x 31 cm	**2134** g	**39.99** EUR
Technocom	**Webcam** 22134T	18 x 19 x 21 cm	**700** g	**59** EUR
	Monitor Locking Cable P1239823	11 x 11 x 3 cm	**40** g	**13.49** EUR
	Gladiator MX 2212-121-828	34 x 14 x 2 cm	**321** g	**81.7** EUR

Highlighting Rows and Columns

You can use CSS to achieve striping for table rows, as you might do in other web-based applications, for example to highlight alternate rows in the table above so that it looks like this:

Product	Supplier	Dimensions	Weight	Price
Deskjet Super Highspeed KTZ-12012.V2	Red Point Stores	87 x 45 x 39 cm	**100** g	**117.19** EUR
Flat Medium 870394932	Very Best Screens	102 x 13 x 54 cm	**1800** g	**639** EUR
Flat S 38094020.1	Very Best Screens	88 x 13 x 49 cm	**1401** g	**339** EUR
Flat X-large II 282948303-02	Very Best Screens	112 x 13 x 60 cm	**2100** g	**1239** EUR
Gladiator MX 2212-121-828	Technocom	34 x 14 x 2 cm	**321** g	**81.7** EUR
Hardcore Hacker 977700-11	Titanium	53 x 24 x 6 cm	**651** g	**89** EUR
High End Laptop 2b OP-38800002	Titanium	64 x 34 x 4 cm	**1190** g	**939** EUR
Hurricane GX K47322.1	Red Point Stores	34 x 14 x 2 cm	**588** g	**219** EUR
Laptop Case 214-121-828	Red Point Stores	53 x 34 x 7 cm	**1289** g	**78.99** EUR
Laser Allround Pro 89932-922	Red Point Stores	42 x 29 x 31 cm	**2134** g	**39.99** EUR
Monitor Locking Cable P1239823	Technocom	11 x 11 x 3 cm	**40** g	**13.49** EUR

You just need to note the ID of the `sap.m.List` or `sap.m.Table` control (in this case it is `"salesdata"`) and then apply some appropriate CSS such as:

```
#ProductsView-ProductsTable tbody tr:nth-child(even) {
    background: rgb(245, 245, 245);
}
```

> **i Note**
>
> Use `<even>`, rather than `<odd>` for the sibling specification. This way, the pop-in highlighting will be correct.

It's also possible to highlight table columns by using the `styleClass` property of the `sap.m.Column` control. The value of this property is applied as a class to the whole column (header, cells and footer) and can be used

in the following way: specify a class name for the `styleClass` property of your column, and set the style as you wish:

```
<style>
#products .MyPrice {
  background: @sapUiNeutralBG;
}
</style>
...
<Column
   hAlign="End"
   class="myPrice">
   <Text text="Price" />
</Column>
```

This highlights the *Price* column in our example table, as shown below:

Product	Supplier	Dimensions	Weight	Price
Deskjet Super Highspeed KTZ-12012.V2	Red Point Stores	87 x 45 x 39 cm	**100** g	**117.19** EUR
Flat Medium 870394932	Very Best Screens	102 x 13 x 54 cm	**1800** g	**639** EUR
Flat S 38094020.1	Very Best Screens	88 x 13 x 49 cm	**1401** g	**339** EUR
Flat X-large II 282948303-02	Very Best Screens	112 x 13 x 60 cm	**2100** g	**1239** EUR
Gladiator MX 2212-121-828	Technocom	34 x 14 x 2 cm	**321** g	**81.7** EUR
Hardcore Hacker 977700-11	Titanium	53 x 24 x 6 cm	**651** g	**89** EUR
High End Laptop 2b OP-38800002	Titanium	64 x 34 x 4 cm	**1190** g	**939** EUR

List and Table Events

Both `sap.m.List` and `sap.m.Table` offer the same events, inheriting them from `sap.m.ListBase`.

The events are:

- `delete`
- `itemPress`
- `selectionChange`
- `swipe`

- `updateFinished`
- `updateStarted`

All of these events are handled in the same way: an event is fired, and the event object that is passed contains `listItem` data that tells you which items were affected.

To use these events, you can simply define handlers for them, as shown below:

```
<Table
   delete="deleteHandler"
   swipe="swipeHandler"
   selectionChange="selectionChangeHandler"
   itemPress="itemPressHandler"
   updateStart="updataeStartHandler"
   updateFinish="updateFinishHandler"
...
```

An example for `delete` would look like this:

```
<List
      id="list"
      mode="Delete"
      delete="handleDelete"
      enableBusyIndicator="true"
      headerText="Products"
      growing="true"
      items="{
           path: '/ProductCollection'
      }" >
...
```

An example for `selectionChange` would look like this:

```
<List
      id="idList"
      items="{/ProductCollection}"
      selectionChange="onSelectionChange"
      mode="MultiSelect"
      includeItemInSelection="true" >
…
```

> **i Note**
>
> Item press events are not fired for items that have the type `inactive`. As this is the default property of `listItemType`, change it to a different value if you want the event to be fired.
>
> To test swipe gestures on desktop devices, open the Google Chrome developer tools, and within Settings → Overrides, check the "Emulate touch events" checkbox and reload the page.

Growing Feature for Table and List

`sap.m.ListBase` provides growing-related properties, which can be used for tables and lists.

A growing list has a loading mechanism that requests data from the model in a lazy way. This enables the app to only fetch data from the server as and when necessary.

> **i Note**
>
> Before release 1.16, the `sap.m.GrowingList` control existed as an extension of the `sap.m.List` control. As this is now deprecated, use the properties as described here instead.

The growing-related properties of `sap.m.ListBase` are:

- `growing`: Boolean to set the growing feature to on or off
- `growingScrollToLoad`: If you want to allow more data to be fetched when the user scrolls down to the end of the current items, set this boolean property to true; otherwise a trigger button must be used
- `growingThreshold`: The number of items that are requested each time from the model
- `growingTriggerText`: The text on a trigger button used to cause a request for more data

> **i Note**
>
> For `sap.m.listbase`, the growing feature containing the growing property is not supported when used in combination with two-way binding for a table or list.
>
> Also, as the growing feature enables extended change detection for the binding, it only updates rows that are changed. This means that if the position of a particular row has not been changed, this row will **not** be updated.

To enable data for a table to be fetched on demand like this, you just need to set the values for these properties appropriately on your table control. For example, adding the highlighted lines as shown in the following code will cause five items to be displayed in the table initially along with a *More* button (this is the default text used if you don't set a different text using the `growingTriggerText` property), as shown below the code:

```
<List
   items="{/ProductCollection}"
   headerText="Products"
   growing="true"
   growingThreshold="4"
   growingScrollToLoad="false">
   <StandardListItem
     title="{Name}"
     description="{ProductId}"
     icon="{ProductPicUrl}"
     iconDensityAware="false"
     iconInset="false" />
 </List>
```

If you want the user to have to scroll down to see more items (by setting the `growingScrollToLoad` property to true), you must ensure that the control is within a container that has a scroll feature, such as an `sap.m.Page` in an `sap.m.App` control, like this:

```
<App>
   <Page title="Table Events">
      <Table>
         ...
      </Table>
   </Page>
<App>
```

Sample

For more information, see the sample in the Demo Kit.

Grouping in a Table

When you have repeated values in your table, you can use `mergeDuplicate` of the `sap.m.Column` control to sort your data based on the column to be merged.

There are two properties related to merging duplicate values:

- `mergeDuplicates`: set this to true if you want duplicate values for the given column to be merged

- `mergeFunctionName`: use this to specify the name of the getter function of the control in the column

Duplicate values will only be merged if they are contiguous. That means that you probably want to sort your data first before binding it.

Here's a simple example of using `mergeDuplicates`. We'll set up a table of sales data, and display the data in a table, merging any duplicate regions. When the *Mix Up* button is pressed to lightly shuffle the salesFigures array of objects, you will see that only adjacent duplicates are merged. Press the *Sort* button again to see the ideal merging.

```
// Sales Areas
var oAreas = {
    "North West": ["Manchester", "Liverpool", "Lancaster"],
    "South East": ["London", "Brighton"],
    "North East": ["Middlesbrough", "Newcastle", "Hull"]
};

// Generate some sales figures into a flat array of region/town/amount objects
var oSalesFigures = [];
oSalesFigures = oSalesFigures.concat.apply(oSalesFigures,
Object.keys(oAreas).map(function(region) {
    return oAreas[region].map(function(town) {
        return { "region": region, "town": town, "amount":
(Math.random()*1000000+1).toFixed(2) };
    });
}));
var oModel = new sap.ui.model.json.JSONModel({ "sales": oSalesFigures });
sap.ui.getCore().setModel(oModel);

var oTable = new sap.m.Table("salesdata", {
     inset: true,
     headerText: "Sales by Area",
     headerContent: [
        new sap.m.Button({
            text: "Sort",
            press: function() {
                var oData = oModel.getData();
                oData.sales.sort(function(a, b) {
                    if (a.region === b.region) return 0;
                    return a.region > b.region ? 1 : -1;
                });
                oModel.setData(oData);
            }
        }),
        new sap.m.Button({
            text: "Mix Up",
            press: function() {
                var oData = oModel.getData();
                oData.sales.sort(function() { return Math.random()-0.5; });
                oModel.setData(oData);
            }
        })
    ],
    columns: [
        new sap.m.Column({
            header: new sap.m.Label({ text: "Region" }),
            mergeDuplicates: true
        }),
        new sap.m.Column({ header: new sap.m.Label({ text: "Town/City" }) }),
        new sap.m.Column({
           header: new sap.m.Label({ text: "Sales (GBP)" }),
            hAlign: "End",
            minScreenWidth: sap.m.ScreenSize.Tablet,
            demandPopin: true,
            popinDisplay: "Block"
        }),
```

```
    ],
});
oTable.bindAggregation("items", {
    path: "/sales",
    template: new sap.m.ColumnListItem({
        cells: [
                new sap.m.Label({ text: "{region}" }),
                new sap.m.Label({ text: "{town}" }),
                new sap.m.Label({
                    text: {
                        path: "amount",
                        type: new sap.ui.model.type.Float({ minFractionDigits:
2, maxFractionDigits: 2 })
                    }
                })
        ]
    })
});

oTable.placeAt("content");
```

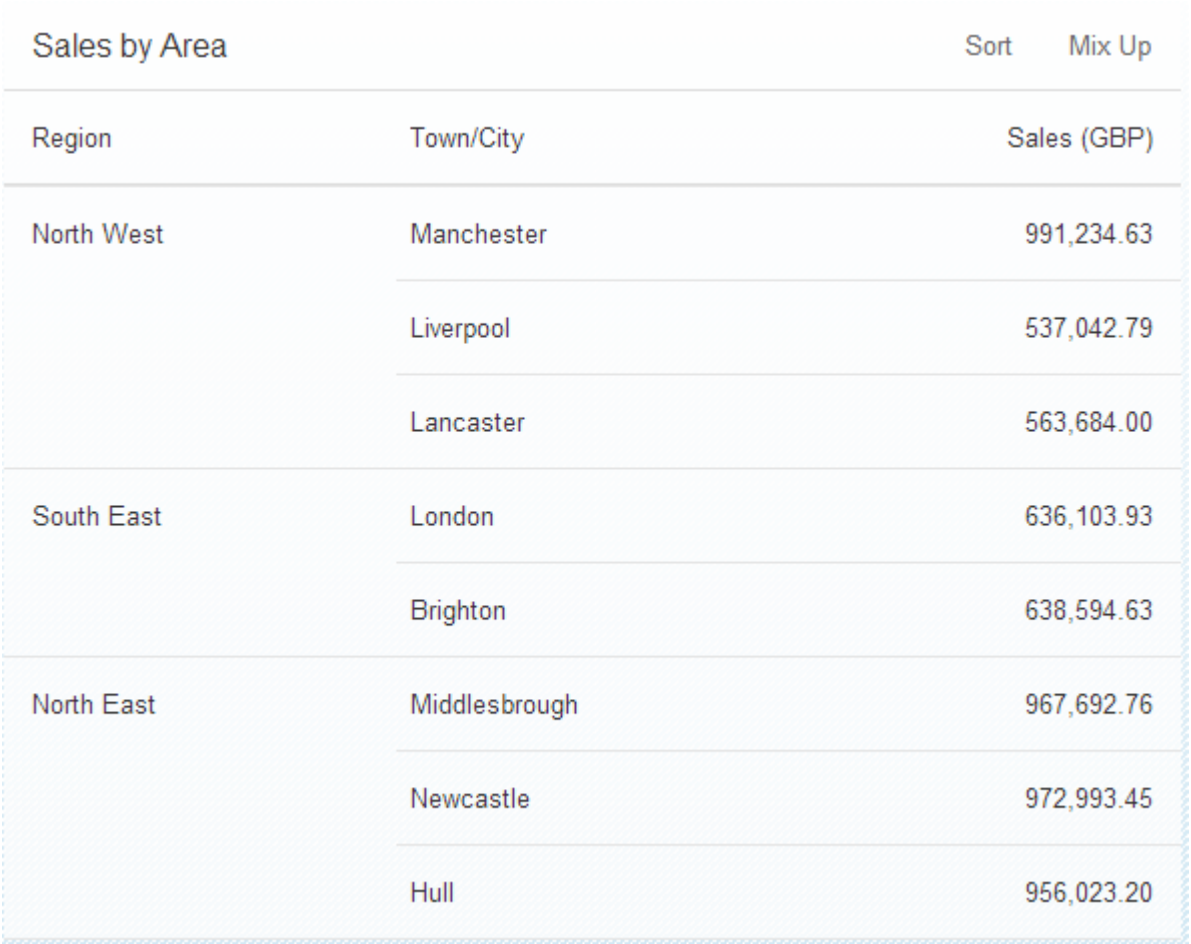


Table Personalization

The simple concept of table personalization allows the user to personalize a table and to persist these settings.

Personalization currently supports defining the order of columns and their visibility.

Components Defining Personalization

The table personalization concept is built upon three distinct artifacts:

- A table personalization dialog `sap.m.TablePersoDialog`
- A table personalization controller `sap.m.TablePersoController`
- A table personalization persistence service provider, based on abstract class `sap.m.TablePersoProvider`

Table Personalization Dialog

The table personalization dialog `sap.m.TablePersoDialog` is a visual control that can be invoked within the context of the table personalization controller. The dialog shows the list of columns in the table, the order in which they appear, and whether their visibility is set to on or off. The user can then use this dialog to adjust these details.

When a user closes the dialog, its table personalization controller automatically applies the column order and visibility settings to its table and it calls the table personalization persistence service provider's `setPersData` method and fires a `personalizationsDone` event.

You can also use the table personalization controller's `getTablePersoDialog` to directly access the table personalization dialog, and use its `retrievePersonalization` method to access its column order and visibility settings.

The personalization settings retrieved with this method take the form of an object that currently has one single property, `aColumns`, the value of which is an array of column objects each having the following properties:

Property	Type	Description
`id`	string	The ID of the column
`order`	integer	The order of the column. Starts with `0`
`text`	string	The text of the column header
`visible`	boolean	Specifies whether the column is visible (true) or not (false)

Table Personalization Controller

The table personalization controller `sap.m.TablePersoController` can be seen as a wrapper around three things:

- Your table (to be personalized)
- A table personalization dialog
- A table personalization persistence service provider

As an application developer, you most likely want to use a table personalization controller. The controller manages the instantiation of the table personalization dialog and the connection to the table persistence

service provider (see later). It also applies the personalizations to the table once the dialog has been closed by clicking *OK*.

Here is an example of how a table personalization controller can be used, assuming you are running in the context of the unified shell, which provides backend services such as persistence.

```
// Create a persistence key
var oPersId = {container: "mycontainer-1", item: "myitem-1"};
// Get a personalization service provider from the shell (or create your own)
var oProvider =
sap.ushell.Container.getService("Personalization").getPersonalizer(oPersId);
// Instantiate a controller connecting your table and the persistence service
var oTablePersoController = new sap.m.TablePersoController({
    table: oTable,
    persoService: oProvider
}).activate();
// Cause the dialog to open when the button is pressed
sap.ui.getCore().byId("idPersonalization").attachPress(function() {
    oTablePersoController.openDialog();
});
```

Once the user closes the table personalization dialog, the personalizations made are automatically applied to the table and persisted. Conversely, when the controller is instantiated, any existing personalizations are fetched and applied to the table. The link to the persistence service provider, the instantiation of a table personalization dialog object, and the automatic appliance of any personalization settings to the table is invoked with the activate method as shown above.

As well as the `openDialog` and `activate` methods shown, methods are also available to apply and save the personalizations: `applyPersonalizations` and `savePersonalizations` respectively. In most cases, you do not need to call the functions: the table personalization controller takes care of it all after it has been activated.

The table personalization controller also offers a 'refresh' function. It re-loads the personalization information from the table personalization provider, applies it to the controller's table, and updates the table personalization dialog.

Note the reference to "create your own" personalization service provider in the code example above. The unified shell provides a shell-based personalization persistence service provider (see below), but you are of course free to build your own. One simple example might be the use of browser local storage to read and write data.

Table Personalization Persistence Service Provider

The table personalization persistence service provider should be based on the abstract class `sap.m.TablePersoProvider`. Do not instantiate and use this `sap.m.TablePersoProvider` class directly as a provider; it merely describes the interface that a real persistence service provider should be built to.

The interface itself is simple: on instantiation, persistence identification can be supplied. This will then be used as the key for retrieving, saving and removing the personalization data. The following methods are available:

- `getPersData()`: Retrieves the personalizations.
- `setPersData(oPersonalizationData)`: Saves the given personalization data.
- `delPersData()`: Removes the personalization data.

- `getGroup(oColumn)`: Lets you specify to which group a column should belong. If you set the `hasGrouping` flag in the table personalization controller, the table personalization dialog will call this method to arrange the columns in groups.
- `getCaption(oColumn)`: Implement this method if you would like the table personalization dialog to display a different column name than the one displayed within the table, or if you would like to add any information to the standard column name.
  If present, the table personalization controller's `getCaption` method is asked for the column text when the table personalization dialog is opened. If it does not deliver a result, the column header texts are taken from the table.

After activation, the table personalization controller applies the personalization obtained through calling the provider's `getPersData()` to its table: it re-arranges the order of columns in the table and makes them invisible if required. To define a default visibility setting, you can either fill the persistence of your table personalization provider with default settings, or you can implement your table personalization provider's `getPersData` method so that it delivers a default visibility if no persisted personalization is available yet.

> **i Note**
>
> There are no keys specified in the calls to the `get`, `set` and `del` functions. These should be inherent from the original instantiation of the service and used implicitly. There may be a requirement to supply a variant style sub-key in future, but this is not yet implemented.

Shell Table Personalization Persistence Service Provider

A concrete implementation of the table personalization persistence service provider (`sap.m.TablePersoProvider`) is available from the Unified Shell services. The previous code example shows this persistence service provider being instantiated and utilized in the controller.

> **i Note**
>
> Please note that this is not part of the SAPUI5 framework. It is delivered separately, and you must check that this service is available in your application context.

Sample

For a detailed example of how table personalization works, see the sample.

API Reference

- sap.m.TablePersoDialog
- sap.m.TablePersoController
- sap.m.TablePersoProvider

Performance of Lists and Tables

Mobile devices usually have a limited memory and processing power. Complex web pages may therefore have a negative impact on the application performance. This also depends on the mobile device and affects most likely the `sap.m.List` and `sap.m.Table` controls.

These controls are often bound to data sources and often dynamically generate list items or table rows. The data may contain a large number of elements. Apart from the performance, creating long lists or tables on mobile devices also affects usability: navigating through large data sets on a mobile device is less convenient than on the desktop. We recommend to restrict the number of elements in lists or tables to 100 on mobile devices.

If a list contains too many items, the device needs more time for rendering and the user has to wait before he can see the list or table on the device. The scrolling behavior may also be affected. To reduce the number of items that must be rendered initially, you can use the `growing` feature for the list or table. With this feature, subsequent loads will not trigger a rerendering of the complete list and are more performant. However, if a re-rendering of the entire list is triggered for another reason, the rendering may have a reduced performance.

Message Handling

Recommended guidelines for message handling.

We recommend to invest care and energy in good message content:

- Provide short and crisp error messages to the user.
- A message should always contain a 'Call for Action'.
- To achieve the above, you need to map error messages from a back-end system.
- Focus on the most common error situations and improve the messages there.
- You need to detect all problems related to network connectivity and indicate them as such.

Messages Related to a Page

For showing messages to the user that are related to the currrent page, you have several possible controls. Each of these offers a different type of interaction from the user. Choose the control that fits best in you interaction pattern.

Message Dialog

- A message dialog interrupts the user's workflow by blocking the current page and needs to be closed by the user.
- Use a message dialog if the message is important and must be acknowledged by the user.
- The easiest way of showing a message dialog is to use the `sap.m.MessageBox`.
- If you want full control of the content you can also use `sap.m.Dialog` control and set the type to `sap.m.DialogType.Message`.
- As `MessageBox` is a static class, a `jQuery.sap.require("sap.m.MessageBox");` statement must be explicitly executed before the class can be used.

Example

```
    // load MessageBox asynchronously
    sap.ui.require(['sap/m/MessageBox'], function(MessageBox) {

        // and display message
        MessageBox.show(
            "This message should appear in the message box.", {
                icon: MessageBox.Icon.INFORMATION,
                title: "My message box title",
                actions: [MessageBox.Action.YES, MessageBox.Action.NO],
                onClose: function(oAction) { / * do something * / }
            }
        );
    });
```

Message Toast

- A message toast is an overlay that disappears after some time or if the user taps somewhere else. It does not block the user.
- The message will automatically fade out, unless it is selected by the user.
- Use this pattern if the message is less important and the user should not be blocked in his work.
- You can open a message toast easily with the `sap.m.MessageToast` API.

Example

```
    // add MessageToast as import
    sap.ui.define([..., 'sap/m/MessageTost', ...], function(...,
MessageToast, ...) {
        ...
        // show toast when needed
        MessageToast.show("Item deleted");
        ...
    });
```

MessageStrip

`MessageStrip` enables the embedding of short application-related messages in the application. There are four types of messages and each is color-coded and has an icon corresponding to its type: `Information`, `Success`, `Warning` and `Error`.

Example

```
    // add MessageStrip and MessageType as imports
    sap.ui.define([..., 'sap/m/MessageStrip', 'sap/ui/core/library', ...],
function(..., MessageToast, coreLibrary, ...) {

    ...
        var MessageType = coreLibrary.MessageType;

        var msg = new MessageStrip({
            id: "importantMessage",
            text: "This is a sample text",
            type: MessageType.Error,
            showIcon: true,
            showCloseButton: true
        });

    ...
```

```
    });
```

The `MessageStrip` is useful when you want to display short notices, for example of finished background tasks, that do not require further user interaction.

MessagePopover

`MessagePopover` displays a summarized list of different types of messages (errors, warnings, success and information). It provides a handy and systemized way to navigate and explore details for every message. You can find more information on `MessagePopover` here [page 2353].

MessageView

`MessageView` displays the same type of summarized messages list as the `MessagePopover`. The main difference between the controls is that the `MessageView` can be embedded in any suitable control (for example a `Dialog`). This allows displaying of the message summary in any part of the application. As of version 1.46, the `MessagePopover` has been refactored to automatically instantiate and use a `MessageView` for its content. All other controls need to instantiate it themselves. Here is a sample for a `MessageView` in a `Dialog`:

```
...
    sap.ui.require(['sap/m/Dialog', 'sap/m/MessageView', 'sap/m/Bar', 'sap/m/
Button', 'sap/m/Text'],
        function(Dialog, MessageView, Bar, Button, Text) {

            // create message view
            var oMessageView = new MessageView({
                showDetailsPageHeader: false,
                itemSelect: function () {
                    that._oBackButton.setVisible(true);
                },
                items: {
                    path: "/",
                    template: oMessageTemplate
                }
            });
            ...
            var oDialog = new Dialog({
                title: "Messages",
                resizable: true,
                content: oMessageView,
                state: 'Error',
                beginButton: new Button({
                    press: function () {
                        oDialog.close();
                    },
                    text: "Close"
                }),
                customHeader: new Bar({
                    contentMiddle: [
                        new Text({ text: "Error"})
                    ],
                    contentLeft: [
                        oBackButton
                    ]
                }),
                contentHeight: "300px",
                contentWidth: "500px",
                verticalScrolling: false
            });

            oDialog.open();
```

```
            ...

        }
    );
...
```

Messages Related to Elements of a Page

For showing messages to the user that are related to a specific element of a page there is no dedicated UI control available in `sap.m` in this version. We recommend to use the `sap.ui.core.HTML` control to show these error messages 'somewhere close to the input' or use some kind of overlay. Consider that the user will have the on screen keyboard open which might hide messages. Putting the message above an input field could help.

You can set the `ValueState` of the `sap.m.Input` control to `Error` to indicate that the content is not correct.

Multiple Messages

SAPUI5 Mobile does **not** support multiple messages at the same time. Mobile Designs recommend to be 'more sparse' with messages, that is, only show one message at a time. This can also be achieved by combining and reducing multiple messages.

Related Information

Message Popover [page 2353]

Message Popover

Message Popover is used to display a summarized list of different types of messages (errors, warnings, success and information). It provides a handy and systemized way to navigate and explore details for every message.

Control overview

The `MessagePopover` control displays a list of messages which can be further drilled down to reveal more details. Typically, it will be placed in the footer and can be expanded when clicking on its icon. As it inherits from the `Popover` control, it can also be placed relative to any other SAPUI5 control using the `placement` propery and its respective values:

- `sap.m.VerticalPlacementType.Top` - placed at the top of the reference control

- `sap.m.VerticalPlacementType.Bottom` - placed at the bottom of the reference control
- `sap.m.VerticalPlacementType.Vertical` (default) - placed at the top or bottom of the reference control

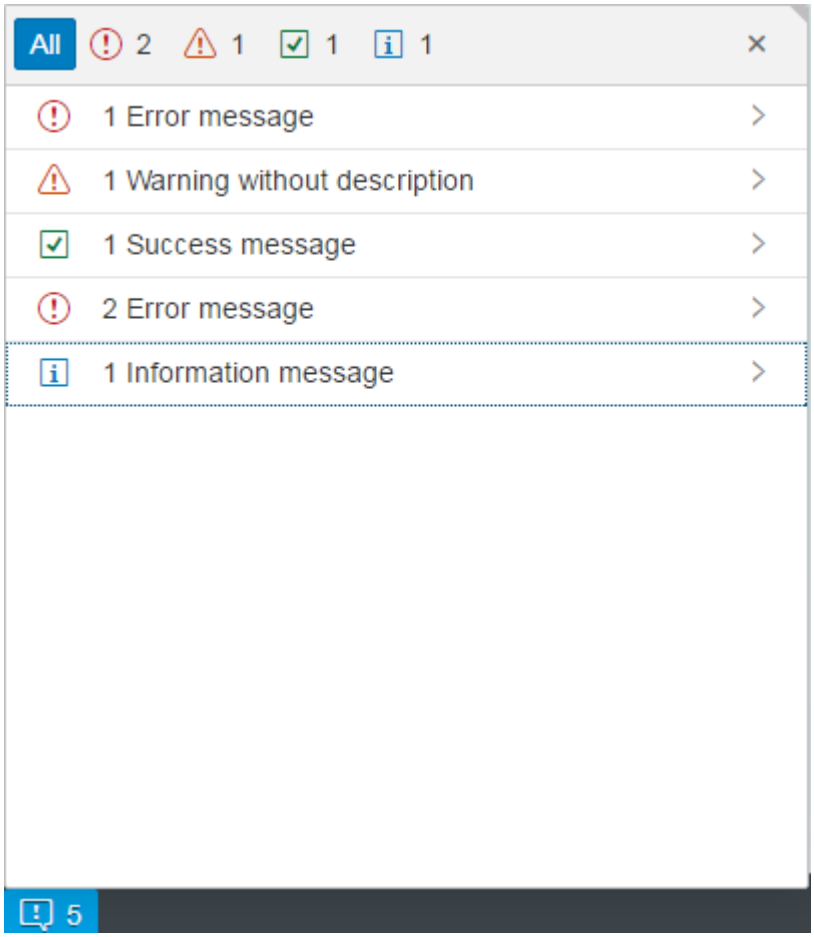


Figure 345: Message Popover control

The `MessagePopover` also features the modes - collapsed (showing only the type and number of messages) and expanded (showing the complete list). An example of the collapsed mode is shown in the screenshot below.

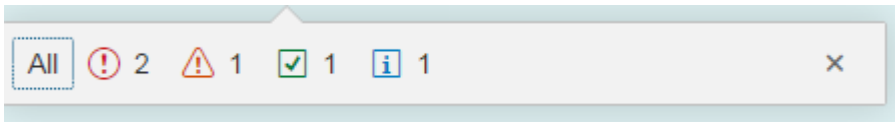


Figure 346: Message Popover collapsed

Handling links in long-text messages

The `MessagePopover` control allows app developers to provide a long-text description for a message, which can include markup and formatting of the content. When this is the case, the control will automatically, and by default, invalidate links and will not allow clicking on them. This is due to security reasons and it is part of the app developer's responsibility to check the links for possible vulnerabilities, exploits and access policies. App developers are provided with an asynchronous function property that should be used for this matter - `asyncURLHandler`.

Sometimes, you may need to validate all links by default. You can do this with the following function property:

```
asyncURLHandler: function(config){
    config.promise.resolve({
        allowed: true,
        id: config.id
    });
}
```

How does it work?

The required flag - `allowed` - is always set to **true** and the promise is resolved immediately. Therefore all of the links in the description will be automatically validated.

PDF Viewer

PDF viewer can be used to display PDF documents within your app, which enables your users to preview PDF documents before printing or downloading them.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The `PDFViewer` control displays PDF documents right inside your app. It can be embedded into your page layout, or you can set it to open in a popup dialog. In addition, this control allows you to download the PDF documents it displays.

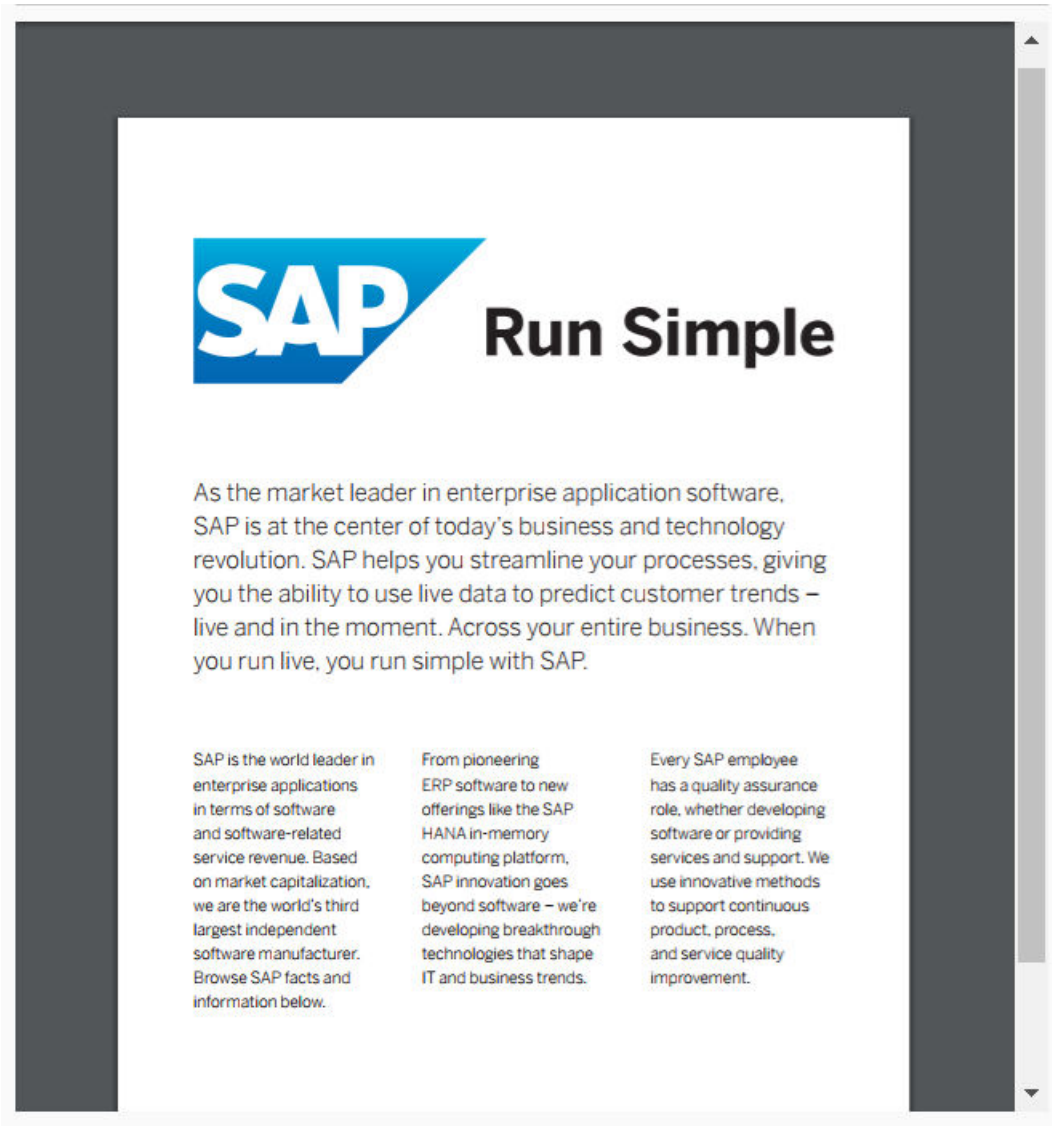


Figure 347: PDF Viewer Example

Details

PDF File Source

- You can specify the source of the PDF document that you want to display using the `source` property that points to a PDF file path. This property can be set to a relative or an absolute path.
  Optionally, you can set the `source` property to a data URI or a blob URL in all major web browsers except Internet Explorer and Microsoft Edge. If you want to use a data URI or a blob URL, you need to make sure that this data URI or blob URL is whitelisted in advance. For more information, see URL Whitelist Filtering [page 1477].

Content Caching

- PDF documents displayed in the PDF viewer may or may not be cached, depending on the app that uses the PDF viewer control. It's up to you to decide how often the content should be refreshed and whether to use caching or not.

Constraints

Supported Device Types

- The PDF Viewer control is fully displayed on desktop devices only.
- On mobile devices, the PDF document is not displayed. Only the toolbar with a download button is visible. Click the download button to download and open the PDF document.

Browser Limitations

- Internet Explorer
  - When a PDF document is loaded, the Internet Explorer browser may not allow its content to be validated. As the result, the `sourceValidationFailed` event may not be fired for loaded PDF documents.
  - When the PDF viewer is open in the Internet Explorer browser, the displayed PDF document may appear on top of all other page elements. To work around this issue, insert another `iframe` element between the PDF viewer and the rest of the elements on the page.
  - Data URI paths and blob URLs used as the PDF source are not supported in Internet Explorer.
- Microsoft Edge
  - When the PDF viewer is open in the Microsoft Edge browser, the displayed PDF document may appear on top of all other page elements. To work around this issue, insert another `iframe` element between the PDF viewer and the rest of the elements on the page.
  - Data URI paths and blob URLs used as the PDF source are not supported in Microsoft Edge.
- Mozilla Firefox
  - The sourceValidationFailed event is not fired for PDF documents loaded in the Mozilla Firefox browser.

Embedding the PDF Viewer into a Tab

- When the PDF viewer is embedded into the `sap.m.IconTabBar` control, the PDF documents may fail to reload when you switch tabs. To work around this issue, you can do either of the following:
  - Set the visibility of the PDF viewer to `false` when the user is switching between tabs.
  - Remove the PDF viewer `iframe` element from the DOM each time the user navigates to a different tab. The PDF viewer element can be removed by calling the `sap.m.PDFViewer#invalidate` method. For more information, see the API Reference: `sap.ui.core.Control.html#invalidate`.

Accessibility

- Accessibility features available to the user may vary, depending on the version of the Adobe Acrobat Reader installed.

Fillable PDF Forms

- Support for fillable PDF forms depends on the browser and device limitations.

Related Information

URL Whitelist Filtering [page 1477]

Personalization Dialog

The `sap.m.P13nDialog` control provides a dialog to personalize tables, such as adapting the order of columns or filtering table content.

For more information about this control, see the API Reference and the samples.

Overview

The `P13nDialog` control offers you a personalization dialog and allows the consuming application to define various settings for tables using panels. The panels represent the settings on the user interface. In order to register them to the control, the `sap.m.P13nPanel` aggregation is used.

Details

The following panels are available in the `P13nDialog` control:

- `sap.m.P13nSortPanel`
  Defines the sorting in a column in an ascending or descending order.
- `sap.m.P13nFilterPanel`
  Defines filter conditions for a column.
- `sap.m.P13nGroupPanel`
  Defines the grouping of columns.
- `sap.m.P13nDimMeasurePanel`
  Defines the chart-specific settings.

The following buttons are available in the dialog:

- *OK*
  Closes the personalization dialog.
  In some cases, the consuming application must transfer the settings of the end user to the table, for example, when sorting and filtering.
- *Cancel*
  Closes the personalization dialog.
  All changes made by the end user in the dialog that is currently open are rejected.
- *Restore*
  Personalization dialog remains open.
  All changes made by the end user are set back to the initial state.

> **i Note**
>
> The control only provides the visual representation of the table settings on the user interface. The consuming application must ensure the settings are actually changed in the table.

Scrolling

Because of limited size of mobile devices, scrolling is an essential topic in mobile user experience. Smooth and easy scrolling is important for user acceptance of mobile applications.

In general, application programmers do not need to take care of scrolling when using the sap.m control library. Scrolling is provided automatically by the following controls:

- `sap.m.Page`
- `sap.m.Dialog`
- `sap.m.Popover`
- `sap.m.ScrollContainer`

Scrolling: Implementation Details

SAPUI5 embeds the open source library iScroll4 that takes care of scrolling in the application.

Scrolling support in mobile browsers is weak and inconsistent. Only the latest platforms and browsers start to support partially usable scrolling functionality. To avoid this, SAPUI5 supports iScroll4. Though the library is globally available in a SAPUI5 application, programmers should not call it directly. The `sap.ui.core.delegate.ScrollEnablement` delegate provides all functionality and smooth integration of iScroll4 into the SAPUI5 library.

For more information, see sap.ui.core.delegate.ScrollEnablement

Do not use nested scrolling

We do not recommend to use nested levels of scrolling, for example, when a page with enabled vertical scrolling contains a scroll container that has vertical scrolling too. Such combinations may lead to behavior that is unexpected both for programmers and users.

Implement a custom scroll container

A custom control that needs to provide a scrollable area for its content should implement the following steps:

1. Instantiate a `sap.ui.core.delegate.ScrollEnablement` delegate, at best in the `.onAfterRendering` callback.
2. Implement a `.getScrollDelegate` method that returns the current instance of the delegate to other controls.
3. Destroy the `ScrollEnablement` delegate on exit.

Example:

```
myCustomScroller.prototype.onAfterRendering = function() {
  if(!this._oScroller){
```

```
    jQuery.sap.require("sap.ui.core.delegate.ScrollEnablement");
    // attach a scroller to the scrollable container DOM element
    this._oScroller = new sap.ui.core.delegate.ScrollEnablement(this,
this._scrollContainerId, {
      horizontal: false,
      vertical: true
    });
  }
};
myCustomScroller.prototype.getScrollDelegate = function() {
  return this._oScroller;
};
myCustomScroller.prototype.exit = function() {
  if(this._oScroller){
    this._oScroller.destroy();
    this._oScroller = null;
  }
};
```

> **i Note**
>
> The Zynga scroller that is included in the SAPUI5 library is deprecated. The configuration parameter `oConfig.zynga=true` of the scrolling delegate should not be used.

Interaction with the scroll containers

There are cases, when an embedded control controls scrolling of the parent container, if required. These are

- a `sap.m.ScrollContainer` inside a `sap.m.Page` may block parent scrolling, if it scrolls in the same direction itself;
- a `sap.m.TextArea` control in edit mode blocks parent scrolling, so that the user can scroll text contents during input;
- a `sap.m.GrowingList` control scrolls parent container to update positions of visible items after the new items have been loaded from the server.

When using a `sap.m.FlexBox` with `fitContainer:true` or `sap.m.TileContainer` inside a page, the `enableScrolling` property of the page needs to be set to false for the FlexBox or TileContainer to fit the viewport.

Scrolling: Pull to Refresh

The SAPUI5 mobile library supports the pull down to refresh functionality that allows users to refresh lists or page content with fresh data from server.

To implement it, create a `PullToRefresh` control and put it as the first control into a page or a scroll container that contains the list that needs to be refreshed.

Example:

```
var pullToRefresh = new sap.m.PullToRefresh({ description: getLastUpdatedTime(),
refresh: function(){
    pullToRefresh.setDescription("loading from server...");
```

```
  //request new data from server
  getNewData({
    // when data comes from server
    onSuccess: {
      pullToRefresh.hide();
      pullToRefresh.setDescription(getLastUpdatedTime());
      redrawList(); } }); };
```

The application should request new data on the refresh event and call the hide method when the data is received and the list is refreshed. You can provide a URL to a custom logo image with customIcon or switch display of logo of by setting showIcon to false. The first line of text "Pull to refresh" is standard and cannot be changed. However, you may set an optional description text to display, for example, the last update time.

> **i Note**
>
> `PullToRefresh` control is part of the scroll area and therefore its height is reflected in the scroll bar calculation and display. The user can see that the page can be scrolled down to reveal the pull-down area.

Carousel

Pull to Refresh does not work with a Carousel if both are contained in a page: in order to make Pull to Refresh work, the page has to enable scrolling which leads to problems with the Carousel (Carousel not visible). Suggested Workaround: Add a `sap.m.PullToRefresh` instance to each page that you add to your Carousel.

API References

sap.m.PullToRefresh

Sliders

Control Overview

A slider is a control that enables you to adjust values on a specified range. SAPUI5 has two controls of this type - `sap.m.Slider` and `sap.m.RangeSlider`. The slider allows you to choose a single value, whereas with the RangeSlider you can choose an interval with start and end within a given interval.

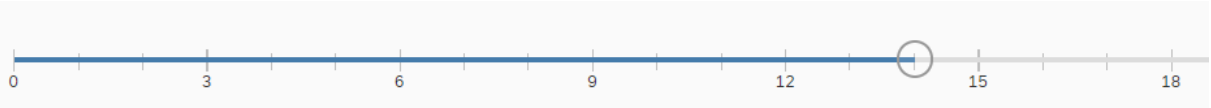


Figure 348: Slider

Figure 349: RangeSlider

Technically the RangeSlider extends `sap.m.Slider` and thus uses all its properties. It has two additional properties for the second slider value and the selected range.

Both versions of the control support features for tickmarks, labels and advanced tooltips. If you only need a simple native browser tooltip, you can enable `handleTooltip` and show it on mouse hover. For more advanced cases, you can customize the labels and the tooltips for the slider handles.

Custom Scale

In the background, the sliders operate on floats, but some usecases may require that the range consists of other values, for example dates. In order to properly match your values to floats, you need to add custom scale and implement the `Iscale` interface.

```
// "Element" required from "sap/ui/core/Element"
var CustomScale = Element.extend("sap.xx.custom.CustomScale", {
                interfaces: [
                    "sap.m.IScale"
                ],
                library: "sap.xx.custom",
            });
```

You need to implement the following methods of the IScale interface.

- `getTickmarksBetweenLabels` - Determine which tickmarks should have a label
- `calcNumberOfTickmarks` - Determine the number of tickmarks on the scale
- `handleResize` - Resize handler
- `getLabel` - Getter for the label

This way you have full control over the labels, their placement, density, and text. As your custom labels may be longer , you will also need to show less tickmarks in order to prevent cluttering of the scale values.

This custom scale is then passed to the control.

```
// "Slider" required from "sap/m/Slider"
// "CustomScale" required from "sap/xx/custom/CustomScale"
// "CustomTooltip" required from "sap/xx/custom/CustomTooltip"
var oSlider = new Slider({
                min: 0,
                max: 30,
                value: 15,
                width: "80%",
                enableTickmarks: true,
                showAdvancedTooltip: true,
                scale: new CustomScale(),
                customTooltips: [new CustomTooltip()]
            })
```

Custom Tooltips

In order to create a custom tooltip, you should extend the class `sap.m.SliderTooltipBase` and override some methods. If you want to define your own content for the tooltip, you should override just the `renderTooltipContent` method. If you want to be notified when the Slider's value has been changed, you need to implement the `sliderValueChanged` which takes as an argument the new value of the Slider, so you can adjust the value of the tooltip.

During rendering you can provide the content of the tooltips by writing directly to the DOM.

```
renderTooltipContent: function (oRm, oControl) {
                            // you can write any DOM here - render controls or
anything you want
                            // (inline elements are not recommended as you need
to style them on your own)
                            oRm.openStart("div", oControl.getId() + "-inner");
                            oRm.class("sapCustomSliderTooltip");
                            oRm.openEnd();
                            oRm.close("div")
```

Accessibility for Sliders

Depending on the type of slider, you may need different values to be read out by the screen reader. In the case of a simple numeric slider, the screen reader will read the current float value. If you have a slider with a custom scale with tickmarks, the screen reader will read the value returned by `getLabel()` of the scale. If you have custom tooltips, then the return value from the tooltip formatter will be read. The priority for these id the following: Custom tooltips overrule custom scale and custom scale overrules the generic slider.

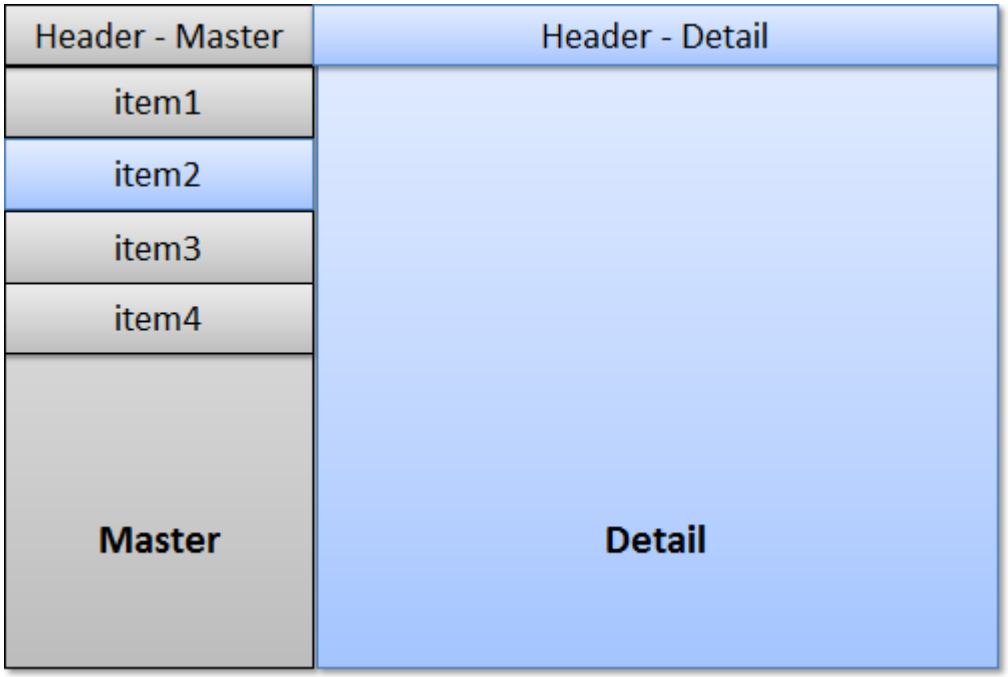
Split App

You can use the `sap.m.SplitApp` control to ensure that your UI automatically adapts to the size available on the respective device.

As tablets such as iPad or Google Nexus7 provide more space compared to smartphones, porting existing mobile apps to tablets leads to a lot of unused space.

A common pattern to address this is called master-detail, and is often used in native iOS and Android development. Good examples are the native *Settings* and *E-Mail* applications of iOS and Android tablets. This pattern can be used with the `SplitApp` control.

The figure shows the basic idea of the pattern. The app is divided into two views, the master and the detail view. The master view presents a list of items and is used as the main navigation within the application. The detail view shows detail information for the selected item.



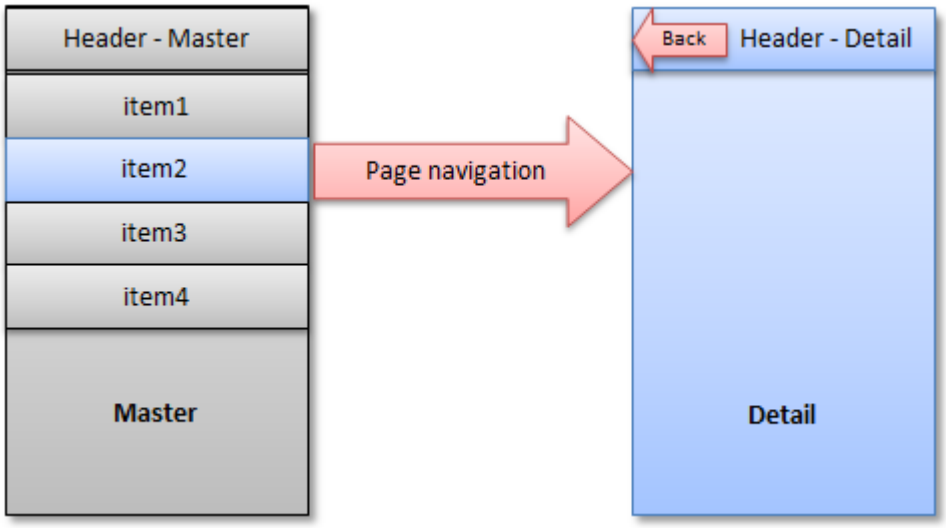
Whereas the selection of an item on a mobile devices navigates the user to the detail page, the user can see the list of items and the detail view at the same time on a tablet device.

If the tablet device is used in portrait mode it has less available width space. For this, the `SplitApp` control provides three different modes for displaying the master and detail view in portrait mode:

- **ShowHideMode**
  This is the default mode. This mode hides the master view automatically when the user turns the device into portrait mode. To display the master view, the user swipes right on the detail view or uses the button which is placed on the header of the detail view.
  The Master view slides in from the left hand side. The user can choose another list item which will update the detail view and automatically hides the master view again.
- **PopoverMode**
  This mode places the master view inside a popover which can be opened via the button in the header of the detail view.
- **StretchCompressMode**
  This mode displays the master view in both, the portrait and the landscape mode. In portrait mode, the detail view has less space available.

In landscape mode, all three modes described above display the master view.

If you run a `SplitApp` on a mobile device, it automatically behaves like a standard mobile application. The following figure shows the difference:



As only one page per screen can be displayed, the master and detail view are automatically displayed on separate pages and the standard page navigation is applied.

Text

The `sap.m.Text` control allows you to display longer texts in your app, with the possibility to manage the number of lines, the text wrapping, and the visualization of line breaks and spacing.

Multi-line Paragraphs and Text Wrapping

You can influence the maximum number of visible lines by setting the `maxLines` property to a numerical value. This option is only available when the `wrapping` property is set to true. For more information, see the sap.m.Text - Max Lines Sample.

Whitespace Handling

The Text control supports the rendering of new lines and the preservation of whitespace and tabs. Depending on whether you are using JavaScript or XML, or data binding, you need to consider specific aspects related to the definition of new lines and tabs.

Table 115: sap.m.Text: New lines and tabs definition in the different use cases

To visualize	JavaScript/Data binding defined string	XML View defined string
New line	`\n`	`
`

To visualize	JavaScript/Data binding defined string	XML View defined string
\n For example, if you want to visualize `C:\NewFolder`	`\\n`	`\n`
Tab	`\t`	`&#x9;`
\t For example, if you want to visualize `C:\TestFolder`	`\\t`	`\t`

For more information, see the API Reference and the sap.m.Text - Render Whitespace Sample.

Usage Guidelines and Limitations

- Use the Text control if you want to display text inside a form, table, or any other content area.
- Do not use the Text control if you need a Label. For more information, see the API Reference: `sap.m.Label`.
- If you need special text formatting, use Formatted Text or HTML. For more information, see the API Reference: `sap.m.FormattedText` and API Reference: `sap.ui.core.HTML`.

> **!** Restriction
>
> When you use wrapping, bear in mind that the multi-line overflow indicator depends on whether the browser supports line clamping. If the browser supports it, it shows the overflow as an ellipsis; if not, the overflow indicator is not displayed.

Upload Collection

`Upload Collection` is a list control for attachment management that provides the ability to upload, edit or delete attachments.

The `UploadCollection` control allows users to upload single or multiple files from their devices (desktop PC, tablet or phone) to an app. Due to its responsiveness, the upload collection can be used in containers of different sizes.

Overview

You can use the `UploadCollection` control to show a list of files, for example, attachments or uploads, that can be modified, or you can allow users to add or remove files to or from this list.

The `UploadCollection` control can be used in the following scenarios:

- **Instant Upload**
  Allows users to upload single or multiple files from their devices and attach them to the application immediately. The selection and upload process is completed in one step and is automatically triggered by the `UploadCollection` control.
- **Upload Pending**
  In this scenario, the upload process is divided into two steps.
  - Users select attachments and add them to the upload list (multiple selection is possible). Thereby, a user can delete a file of this selection again without canceling the whole upload of the other selected files.

  In a second step, the user uploads the selected files to the application. The upload is provided by the application, for example, with an upload button or similar. To trigger the upload, the control offers an event.

Details

Interaction and Behaviour

- **Upload files**
  The upload is triggered by the *Add* or *Upload* pushbutton (depending on the scenario). In the OS-native file picker dialog you can select one or multiple files. However, the consuming application need to enable the upload of multiple files by using the `multiple` property during the instantiation. During the upload process, a progress indicator provides information about the status. For upload, also Drag & Drop can be used.
- **Open files**
  To open a listed file, you can choose the icon or thumbnail of the attachment as well as the filename itself. Depending on the file extension, browser capabilities, and device type (Desktop or mobile), the file is handled differently, for example, it is opened in a new tab or downloaded, if no display capability is available in the browser.
- **Rename files**
  Choosing the *Rename* pushbutton (pen icon) turns the file name into an input field and allows modification. If the `sameFilenameAllowed` property is set to `false`, the new file name is validated and, if the file name already exists in the current list, an error occurs, which indicates that you need to change the file name.
- **Delete files**
  The user can delete files by using the *Delete* button. After choosing the *Delete* button, a dialog appears, asking the user to confirm the deletion of the respective file that removes the file from the upload list or application (the delete behaviour depends on the scenario that is used).
- **Sorting and filtering files**
  The application can provide the sorting and filtering feature for the Upload Collection list. After setting the filtering criteria, the users get the filtering information displayed in the info toolbar.
- For the instant upload scenario only, the following features are available:
  - **Download files**
    You can download a file item from the Upload Collection list and save it on your device. For example, you can provide a *Download* button in the `sap.m.OverflowToolbar` in the header, and once you marked one of the items the download feature can be used. To use this feature, API methods in the `UploadCollection` control and the `UploadCollectionItem` control are used. Using the `askForLocation` parameter, you can configure that the browser should ask for the location where to

store the file or not (depending on the browser). When this parameter is set to `false`, the download method acts in the same way as when clicking on the file link. If the `askForLocation` parameter is set to `true`, the browser opens a *Save As* dialog.

- **Upload a new version of a file**
  To upload a new version of a file to the Upload Collection list, the `openFileDialog` method is available. You can provide a button in the `sap.m.OverflowToolbar` in the header and if one entry in the Upload Collection list is selected, the API method shall be called. The uploading process of the new version depends on the settings for the optional `UploadCollectionItem` parameter:
  - If the parameter is provided, the method removes the selected item from the Upload Collection list automatically to upload the new version.
  - If the parameter is not provided, the select dialog of the operating system opens and the further steps have to be proceeded manually by the user.

  The detailed behavior of the feature depends on the app, in which the control is used.
- **Terminate uploading a file**
  During an upload process of a file, you can terminate the upload with the *Delete* pushbutton, for example, if the data file is very big and the uploading process takes too long. With the `terminationEnabled` property, you can decide for your application to make the button for termination invisible to prevent the user from terminating an instant upload in the `sap.m.UploadCollection` control.
- `uploadButtonInvisible` **property**
  With the `uploadButtonInvisible` property, you can decide for your application to make the *Upload* button in the instant upload scenario invisible to prevent the user from uploading a file.

Layout

- The *Rename* and *Delete* pushbuttons are displayed for each item and are active and visible by default. Both icons can be set to invisible or inactive.
- While most file types have generic icons (such as Word documents, Excel sheets and PDFs), graphic files can be displayed with a small thumbnail preview of its graphic if the respective URL is provided.
- The toolbar of the `UploadCollection` control can be customized:
  - To customize the toolbar, the `toolbar` aggregation can be set to the `sap.m.OverflowToolbar` control which can contain your preferred UI5 controls. To make the position of the upload (+) pushbutton configurable, the `sap.m.ToolbarSpacer` type is used. To configure the position of the button, you can set the placeholder element to the designated position.
  - If the toolbar is not customized it only provides the upload (+) button.
  - In case the customizable toolbar has been configured and the placeholder is missing, an exception will be thrown.
- The `mode` property of the `UploadCollection` control can be set. Therefore the `sap.m.ListMode` type is used.
  - The listmode `Delete` is not supported and will be set to listmode `None` if used.
  - In chase of an upload pending scenario, the listmode `MultiSelect` is not supported; in this case it will be set to listmode `None` automatically.

API Reference/Sample

For more information, see the API Reference and the sample in the Demo Kit.

Constraints and Dependencies

By using the `UploadCollection` control, you need to be aware of the following constraints:

- The `UploadCollection` control does not work with IE9 because of a missing header parameter handling needed for the upload.
- The file name of an upload item can contain any text and special characters but not a URL.

The `sap.m.UploadCollection` control uses the `sap.ui.unified.FileUploader` control and contains dependencies to this control. If you want to use the `UploadCollection` control, you need to be aware of these dependencies. For more information about the `FileUploader` control, see the API Reference Documentation.

URL Helper

With `sap.m.URLHelper` you can easily trigger native mobile phone applications such as e-mail, telephone, and text messages.

You can set predefined values for the application so that a user does not need to enter this information themselves. When personal information is displayed, for example phone numbers and e-mail addresses, you can initiate a phone call or e-mail with just one tap.

The URLHelper API contains the following triggers for telephone, texts, and e-mail applications:

- Trigger telephone application

```
sap.m.URLHelper.triggerTel( [Telephone Number] ); //Telephone number is
optional
```

- Trigger text messaging application

```
sap.m.URLHelper.triggerSms( [Telephone Number] ); //Telephone number is
optional
```

- Trigger e-mail application

```
sap.m.URLHelper.triggerEmail( [Destination Email], [Subject], [Default
Message Text], [CC], [BCC] );
// All parameters are optional
```

- Redirect To custom URL

```
sap.m.URLHelper.redirect( URL );
//URL is required and can be used for custom protocols (e.g http, ftp, ...)
```

> **i** Note
>
> - iOS does **not** trigger a phone call if the phone number contains "*" or "#".

- You can add multiple recipients for a text message in Android phones by separating recipient numbers with ";".
- According to RFC 2368 you can set multiple subscribers for the e-mail application by separating each with ","; however, this still depends on the application. Outlook, for example, uses ";" as separator.
- You can use the `sap.m.URLHelper.redirect` method to use custom URL schemes:
  - For iOS: http://developer.apple.com/library/safari/#featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html
  - For Android: http://developer.android.com/guide/appendix/g-app-intents.html
  - URI schemes: http://en.wikipedia.org/wiki/URI_scheme
- If you just want to get a URI back without a redirect, you can use normalize methods which have the same parameter as trigger methods, for example:

```
/*
 * These methods do not redirect but return URI scheme back as string.
 * All parameters are optional
 */
sap.m.URLHelper.normalizeTel( [Telephone Number] );
sap.m.URLHelper.normalizeSms( [Telephone Number] );
sap.m.URLHelper.normalizeEmail( [Destination Email], [Subject], [Default
Message Text], [CC], [BCC] );
```

API Reference

- sap.m.URLHelper

Examples for Triggering Telephone, Text and E-Mail Applications

Code samples for triggering telephone, text and e-mail applications.

Sample data used in the examples:

```
var person = {
    name : "John Smith",
    tel : "+49 62227
          747474",
    sms : "+49 173 123456",
    email : "john.smith@sap.com",
    website : "http://www.sap.com"
};
```

You can trigger an external application at any time, but it is usually triggered as a reaction to a UI control throwing an event. The next sections illustrate some of the most typical use cases.

Click Button To Trigger Phone Call

The following button can be used to place a call.

```
new sap.m.Button({
    text : person.tel,
    icon : "images/action.png", /* Depends where your images are located */
    tap : function() {
        sap.m.URLHelper.triggerTel(person.tel);
    }
});
```

Click Image To Trigger E-mail

The following code snippet gives an example for triggering an e-mail application. You can also set the subject and message of the e-mail application:

```
new sap.m.Image({
    src : "images/website.png", /* Depends where your images are located */
    tap : function() {
        sap.m.URLHelper.triggerEmail(person.website, "Info", "Dear " +
person.name + ",");
    }
});
```

Inside List

`DisplayListItem` with active feedback is the most popular use case for the following example.

```
new sap.m.DisplayListItem({
    label : "Sms",
    value : "( " + person.sms + " )",
    type : "Active",
    tap : function() {
        sap.m.URLHelper.triggerSms(person.sms);
    }
});
```

To use any other control inside the list, use `InputListItem`:

```
new sap.m.InputListItem({
    label : "Website",
    content : new sap.m.Button({
        text : person.website,
        tap : function() {
            sap.m.URLHelper.redirect(person.website);
        }
    })
});
```

sap.suite.ui.commons

This library contains various controls.

> **Note**
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]
Browser and Platform Support [page 20]
API Reference: `sap.suite.ui.commons`

Calculation Builder

The `CalculationBuilder` control enables you to perform arithmetic calculations on constants and variables, using standard arithmetic operators and most common logical operators and functions.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The calculation builder control enables you to perform arithmetic calculations on constants and variables, using standard arithmetic operators and most common logical operators and functions. You can customize the sets of variables and functions that are visible in the calculation builder and introduce your own custom functions, as needed. Arithmetic expressions can be entered using a touch-friendly visual editor or a textual editor that provides autocomplete suggestions for variables and checks the expression syntax as you type.

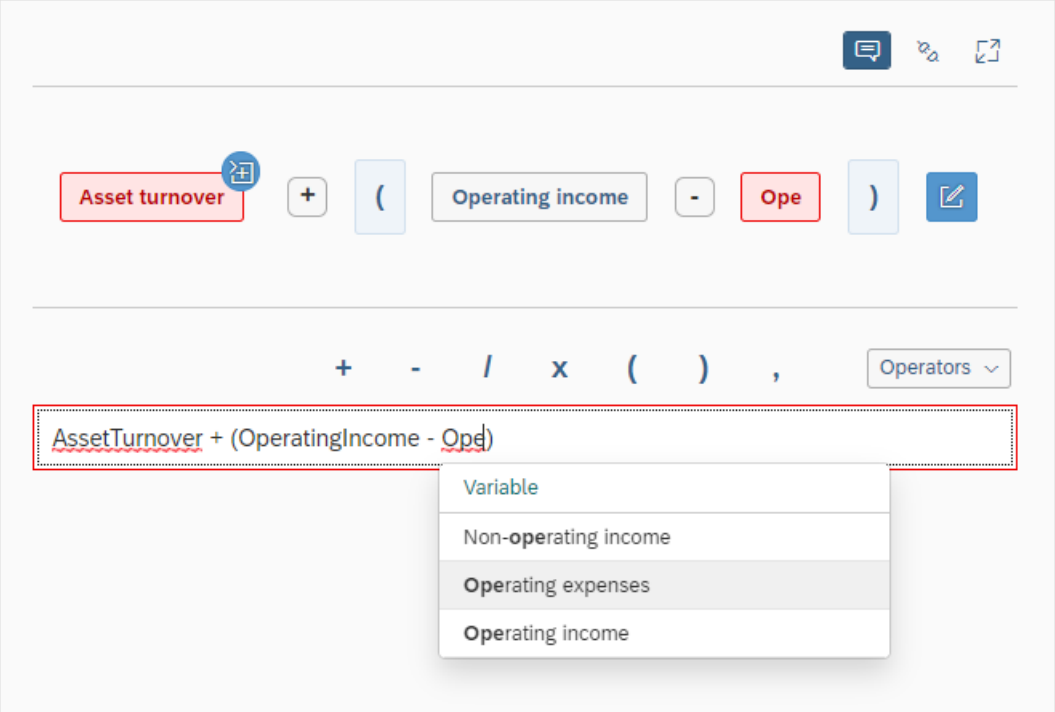


Chart Container

The `ChartContainer` control contains a toolbar and the control of the content aggregation displayed under the toolbar.

Overview

The `ChartContainer` supports the `sap.viz.ui5.controls.VizFrame` and `sap.m.Table` controls in its content area. With the toolbar, you can change the view of the content in different ways.

For more information about the `VizFrame` control, see the API Reference in the Demo Kit.

Details

- **Selection boxes in the toolbar**
  The control supports multiple selection boxes in the toolbar for changing dimensions, for example.
- **Content**
  You can switch between chart types or between a chart and a table view. `ChartContainer` supports different chart and table controls, for example, `vizFrame` and `sap.m.Table`. The `sap.suite.ui.commons.ChartContainerContent` control is a wrapper control for the content to be

displayed in the `ChartContainer` control. You need to provide the title and icon properties in this control when you define the content so that it is displayed in the `ChartContainer` control.

- **Fullscreen mode**
  A fullscreen toggle button is provided in the toolbar. You can switch between fullscreen and normal mode.
- **Personalization**
  The control provides a personalization icon. If you press the icon, a personalization event will be fired.
- **Selection details**
  The *Details* button provides a popover that displays the details of the items selected in the chart. You can include the button into the toolbar by setting the `showSelectionDetails` property to true. Please note that `SelectionDetails` is only available when `sap.viz.ui5.controls.VizFrame` is displayed in `ChartContainer`.
- **Custom toolbar**
  As an app developer, you can create your own toolbar inside a `ChartContainer` control in your application. To configure a custom toolbar, the toolbar aggregation can be set to the `sap.m.OverflowToolbar` control that contains your preferred SAPUI5 controls. The new `sap.suite.ui.commons.ChartContainerToolbarPlaceholder` type has been introduced to make the position of the embedded buttons configurable. To configure the position of the embedded buttons, you need to set the placeholder element to the required position in the aggregation.
  The order of the embedded buttons is set automatically by the control as shown in the image below:

  > **i Note**
  >
  > The displayed toolbars below are taken from an example of the `ChartContainer` toolbar with `VizFrame`.

  - legend, zoom in, zoom out, personalization, full screen, custom icons and segmented button for content selection.

  - If a dimension selector is displayed in the chart container, it will always be displayed in the first position in the toolbar. When a dimension selector is not visible, the chart container's title (`title` property of `ChartContainer`) will be displayed in the first position in the toolbar as shown below.

  If the custom toolbar has been configured and the placeholder is missing, an information log will be displayed and the custom toolbar is ignored. The default toolbar is used instead. If a custom toolbar has been configured, the embedded buttons are still present and integrated using a placeholder.
- **Custom Icons**
  You can also use custom icons. In this case, custom icons are rendered as `OverflowToolbar` buttons. But they offer less options as the custom toolbar, for example, there is no control option for the visibility.

API Reference/Samples

For more information about the `ChartContainer` control, see the API Reference Documentation and the sample in the Demo Kit.

Micro Process Flow

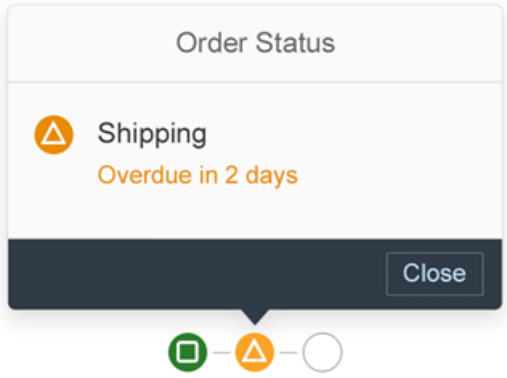
`MicroProcessFlow` is a linear process flow with circular nodes that serve as progress indicators.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The `MicroProcessFlow` control can be used to display the progress of linear workflows, such as order flows, document flows, and approval flows. It can be embedded into tiles, tables, and other types of parent containers.

It is best used for simple linear workflows that include up to seven progress steps. For more complex processes and workflows, consider using the `ProcessFlow` [page 2381] control instead.



Details

Layout and Content

- The `MicroProcessFlow` control consists of progress indicator nodes that are laid out horizontally and connected by connector lines.
  - The nodes can be defined using `MicroProcessFlowItem` controls that are added to the `content` aggregation in the `MicroProcessFlow` control.
  - The connector line that appears right after each node can be hidden using the `showSeparator` property of the `MicroProcessFlowItem` control that defines this node.
  - If you want to display additional objects between nodes, you can define them using the `intermediary` aggregation in the `MicroProcessFlowItem` control and use the `showIntermediary` property to make them visible.
    If you enable the `showIntermediary` property without defining the objects, the nodes are separated by red vertical bars.
- Default nodes use semantic colors that correspond to their `status` property values. Available statuses include: `Standard` (neutral), `Error` (negative), `Success` (positive), and `Warning` (critical).

- To display a custom control instead of a default node, you can use the `customControl` aggregation in the `MicroProcessFlowItem` control defining this node.
  For example, if you want to use a rectangular status indicator [page 2384] instead of a node, you can add the following to your view [page 76]:

```
<mvc:View
...
xmlns:si="sap.suite.ui.commons.statusindicator">
...
<MicroProcessFlow>
    <content>
    ...
        <MicroProcessFlowItem>
                    <customControl>
                <si:StatusIndicator id="indicator1" width="20px"
                            height="40px" viewBox="-1 -1 52 102" value="50">
                    <si:ShapeGroup>
                        <si:Rectangle x="0" y="0" width="50" height="100"
                                    fullAnimationDuration="2000"
fillColor="blue"
                                    strokeWidth="1" animationOnStartup="true"
                                    fillingType="Linear"/>
                    </si:ShapeGroup>
                </si:StatusIndicator>
            </customControl>
        </MicroProcessFlowItem>
    ...
    </content>
</MicroProcessFlow>
...
</mvc:View>
```

  For more ideas and examples, see the samples in the Demo Kit.

Click Events

- You can specify click events for each of the nodes in the micro process flow using the `press` event.
  For example, to display a popover element when the user clicks the node, consider adding the following (or similar) code to your view [page 76] and its controller [page 79].
  View

```
<MicroProcessFlowItem state="Success" press="itemPress">
    <customData>
        <core:CustomData key="title" value="Payment"/>
        <core:CustomData key="icon" value="sap-icon://accept"/>
        <core:CustomData key="subTitle" value="Payment successful"/>
    </customData>
</MicroProcessFlowItem>
```

  Controller

```
itemPress: function (oEvent) {
    var oItem = oEvent.getSource(),
        aCustomData = oItem.getCustomData(),
        sTitle = aCustomData[0].getValue(),
        sIcon = aCustomData[1].getValue(),
        sSubTitle = aCustomData[2].getValue();

    var oPopover = new sap.m.Popover({
        contentWidth: "300px",
        title: "Order status",
        content: [
            new sap.m.HBox({
                items: [
```

```
                    new sap.ui.core.Icon({
                        src: sIcon,
                        color: this._getColorByState(oItem)
                    }).addStyleClass("sapUiSmallMarginBegin
sapUiSmallMarginEnd"),
                    new sap.m.FlexBox({
                        width: "100%",
                        renderType: "Bare",
                        direction: "Column",
                        items: [new sap.m.Title({
                            level: sap.ui.core.TitleLevel.H1,
                            text: sTitle
                        }), new sap.m.Text({
                            text: sSubTitle
                        }).addStyleClass("sapUiSmallMarginBottom
sapUiSmallMarginTop"),
                            new sap.m.Text({
                                text: sDescription
                            })
                        ]
                    })
                ]
            }).addStyleClass("sapUiTinyMargin")
        ],
        footer: [
            new sap.m.Toolbar({
                content: [
                    new sap.m.ToolbarSpacer(),
                    new sap.m.Button({
                        text: "Close",
                        press: function() {
                            oPopover.close();
                        }
                    })]
            })
        ]
    });
    oPopover.openBy(oEvent.getParameter("item"));
}
```

For more ideas and examples, see the samples in the Demo Kit.

Network Graph

The `NetworkGraph` control displays objects as a network of nodes connected to one another by lines.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The `NetworkGraph` control can be used to illustrate how different objects are related. In a network graph, each object is represented by a node, and the relations between objects are represented by lines connecting the nodes. Nodes can be clustered into groups that can be expanded or collapsed to show or hide a portion of the graph.

This control supports both directed and undirected graphs, as well as graphs that contain cycles. It also provides broad customization options, enabling you to separate the graph layout from its rendering and to position individual graph elements freely, for example, when displaying geospatial data on top of a map.

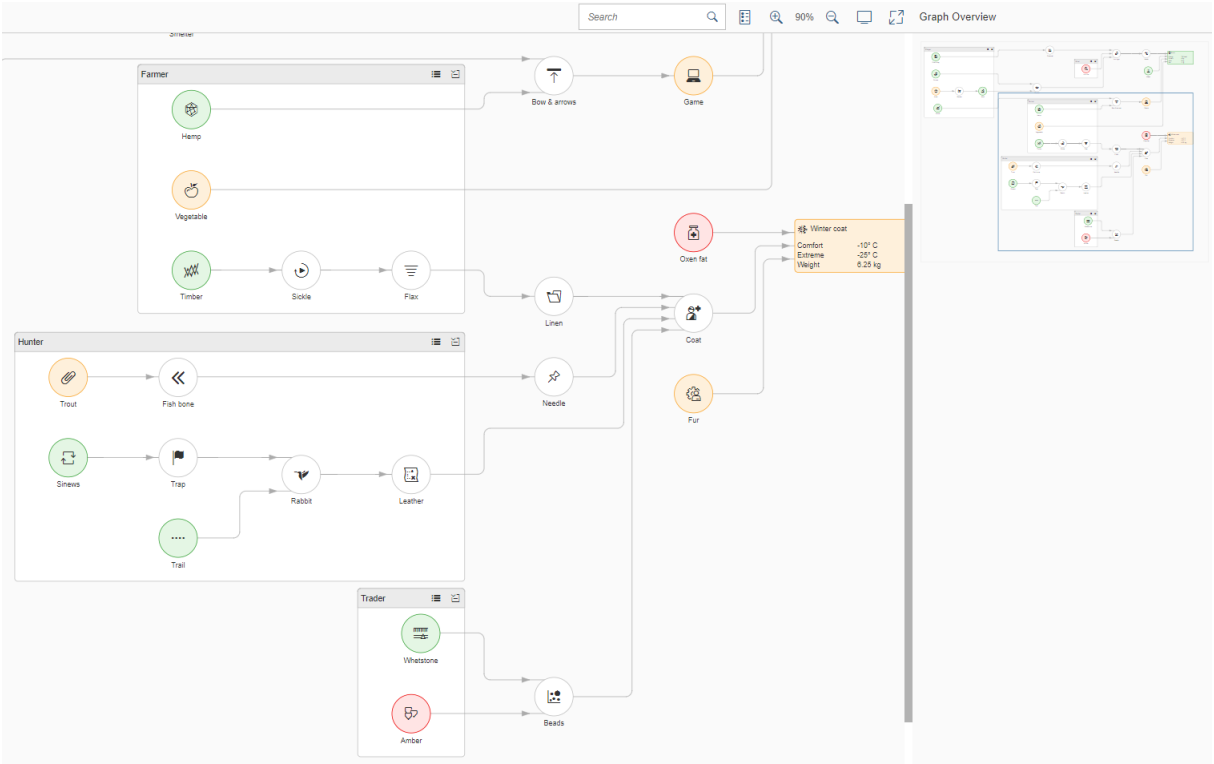


Figure 350: Network Graph Example

Details

Layout

- There are three layout options available:
  - Layered layout – The `LayeredLayout` algorithm arranges the nodes into a layered graph.
  - Force-based layout – The `ForceBasedLayout` algorithm arranges the nodes into a force-based (or force-directed) graph.
  - Free-form layout (`NoopLayout`) – No layout algorithm is applied, so the nodes can be placed anywhere within the graph.
  - Custom layout – You can define your own algorithm that will be used to lay out your graph.

  For details, see `sap.suite.ui.commons.networkgraph.layout`.
- The process of drawing the graph consists of two phases:
  - Computing the layout based on the specified layout algorithm
  - Visual rendering of the graph

  The first phase is marked by the `beforeLayouting` event that is fired just before the layout computation begins. The second phase starts with the `afterLayouting` event that is fired when the layout algorithm has finished arranging the graph and the visual rendering begins.
  Any change to nodes or other graph elements that happens after the layout has been computed may cause invalidation of the whole graph, making it impossible to render. It may lead to an infinite loop with graph invalidation triggering layout algorithms that trigger the event that caused the original invalidation of the graph, and so on.

To prevent unwanted invalidation, you can call the `preventInvalidation` method on the graph:

```
this._graph.attachEvent("beforeLayouting", function (oEvent) {
    this._graph.preventInvalidation(true);
    // Perform actions on the nodes that would normally trigger invalidation.

    // Check out the Org Chart sample for more inspiration.
    this._graph.preventInvalidation(false);
}.bind(this));
```

For more information, see `sap.suite.ui.commons.networkgraph.Graph#preventInvalidation`.

Appearance

- **Grouping** – You can join nodes into a group, so they are displayed closer to one another. A group can be collapsed to hide the nodes that are included in it. Please note that grouping is available only for graphs that use layered layout.
  For details, see `sap.suite.ui.commons.networkgraph.Group`.
- **Node Shape** – You can choose between two node shapes: circular or rectangular. Depending on the node shape, the `width` and `height` properties of the node are treated differently:
  - For rectangular nodes, only the `width` property is considered, while the `height` property is ignored.
  - For circular nodes, the `height` property determines the diameter of the circle, while the `width` property is used as the width of the node's title and description.

  For details, see `sap.suite.ui.commons.networkgraph.Node`.
  Other Properties – In addition to shape, you can define some other properties:
  - `icon` – The icon to be displayed inside the node shape.
  - `title` – The node's title.
  - `description` – The node's description.
  - Element attributes – Additional attributes of the node.
- **Line Customization** – You can choose among several connector line styles: dashed, dotted, or solid, as well as define where the arrow should be positioned and where it should point to.
  For details, see `sap.suite.ui.commons.networkgraph.LineType`.
- **Semantic Colors** – Nodes, groups of nodes, and connector lines may use semantic colors, based on their status. You can use any of the custom statuses defined by the `statuses` aggregation in the `sap.suite.ui.commons.networkgraph.Graph` control or use the default statuses provided by `sap.suite.ui.commons.networkgraph.ElementStatus`.
  For details, see `sap.suite.ui.commons.networkgraph.ElementStatus` and `sap.suite.ui.commons.networkgraph.Status`.

Customizing the Graph

- **Arbitrary Node Positions** – To be able to position the nodes freely, switch the layout algorithm to `NoopLayout`.

```
<Graph>
   <layoutAlgorithm>
      <layout:NoopLayout/>
   </layoutAlgorithm>
...
</Graph>
```

For details, see `sap.suite.ui.commons.networkgraph.layout`.

After that you can define the positions for each of your nodes using their `x` and `y` attributes that define the coordinates of the upper left corner of the node (or upper right corner in languages that have right-to-left writings systems).
To redefine the positions of the lines connecting the nodes, you can use the aggregation `Coordinates` for the `Line` control. Each coordinate contains x and y coordinates of one point. It is recommended that you use the following methods to add coordinates to the `Coordinates` aggregation:

- `setSource` – Sets the starting point of the line.
- `setTarget` – Sets the end point of the line.
- `addBend` – Adds a point between the start and the end points of the line.

These methods do not trigger invalidation.
For details, see `sap.suite.ui.commons.networkgraph.Line`.

- **Using Events for Graph Customization** – You can adjust the graph behavior through a variety of event calls. Such event calls may suppress the default behavior of certain events. For example, if you define the following function for a node, action buttons will no longer be displayed when the user clicks the node:

```
<Node press="nodePress">
...
nodePress: function (oEvent) {
      // Prevents the rendering of default action buttons
      oEvent.preventDefault();
};
```

Similarly, you can suppress the following events:

Object	Suppressed Event	Result
Node	`press`	Action buttons will not be displayed when the user clicks the node.
Node	`collapseExpand`	The node will not be expanded or collapsed when the user clicks the *Collapse/Expand* button.
Node	`hover`	Moving the mouse over the node will have no effect on the node appearance.
Line	`press`	Details popup will not be displayed when the user clicks the line.
Line	`hover`	Moving the mouse over the line will have no effect on the line appearance.
Group	`showDetail`	Details dialog will not be displayed when the user clicks the *Group Details* icon.

Zooming

- The predefined zooming scale used by the `NetworkGraph` control supports zooming levels ranging from 5 to 500 percent of the original graph size. The zooming level is updated when the user clicks the *Zoom In* or *Zoom Out* buttons or uses a mouse, a trackpad, or a touch screen to zoom in or out.
  - When the user clicks (*Zoom In*) or (*Zoom Out*), the graph is scaled up or down to the next zooming level.

- When the user clicks (*Zoom to Fit*), the graph is scaled up or down, so all nodes and lines it includes are visible on the screen. The optimal zooming level is selected from the predefined scale: 5%, 10%, 25%, 33%, and so on up to 500%.
  If the original graph size is out of scale, taking up less than 5 percent or more than 500 percent of the screen, clicking *Zoom to Fit* scales the graph up or down to the closest minimum or maximum zooming level, 5% or 500%. For example, if the original graph takes 1 percent of the screen, when the user clicks *Zoom to Fit*, the graph is scaled up to the 5% zooming level.

Process Flow

The `ProcessFlow` control allows you to show flows of multiple object types, for example, documents or approvals.

Overview

Process Flow is a complex control that enables you to display documents or other items in their flow. The items or documents are displayed as nodes in a lane containing a lane header with a donut chart. The donut chart reflects the status of the nodes in its lane. The chevron arrows in the lane header separate the lanes from one another. Connector lines show the process flow of the items between the nodes.

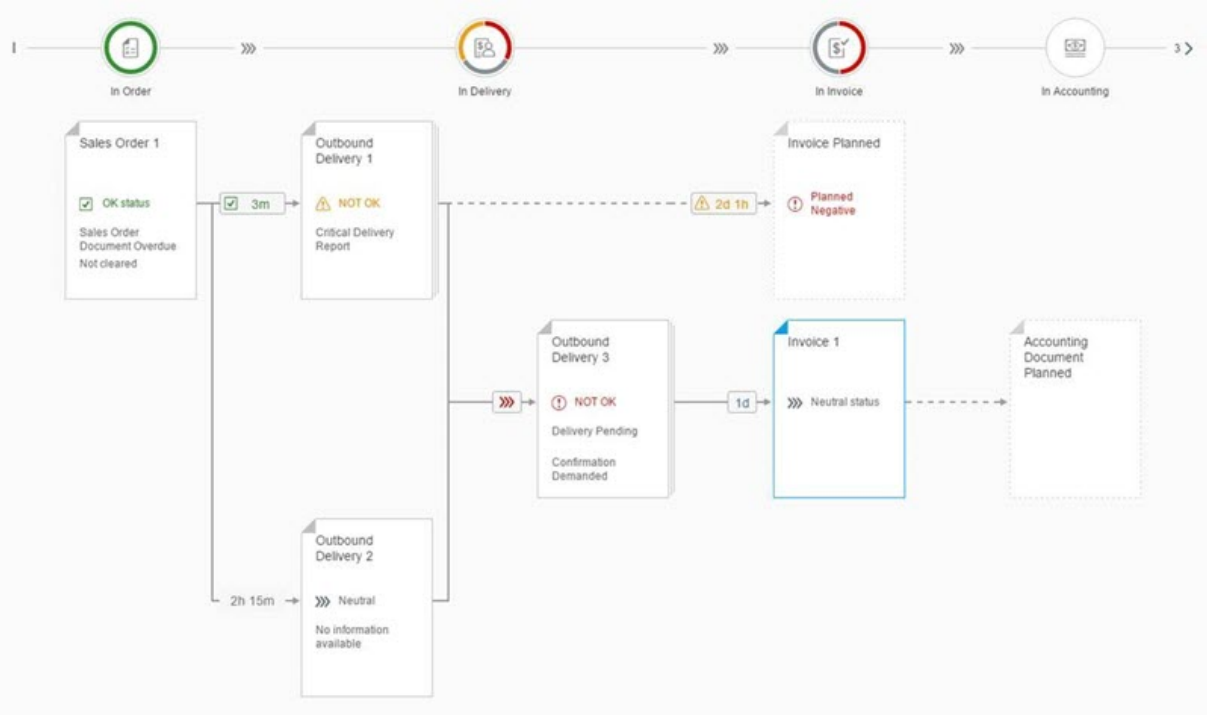


Figure 351: Process Flow Example

Details

Behavior and Interaction

- The control offers different **scrolling and zooming** behaviors:
  - You can move the whole flow by holding down the left mouse button as though you were navigating a street map in a web browser.
  - To zoom in or out, you can either use the mouse wheel or choose the respective buttons if the `ProcessFlow` control is used in combination with a toolbar. The zoom has a semantic effect: detailed information is added or removed depending on the zoom level.

  Both scrolling and zooming can be switched off by changing the values of the `scrollable` and `wheelZoomable` properties.

  > **i Note**
  >
  > Scrolling is only an option in containers smaller than the ProcessFlow control to be displayed. For more information, see the sample.

- To improve user experience, **clickable navigation arrows** (‹ and ›) are included in the header. They are visible once there is an overflow, for example, when the complete `ProcessFlow` control can't be displayed because the visible area is limited. A navigation arrow on either side of the process flow header indicates that some part of the horizontal content is hidden, and you need to click the arrow to display it.
  - On desktop computers, use navigation arrows to move the complete process flow over one lane.
  - On mobile devices, the navigation arrows can be treated just as indicators. You can scroll through the `ProcessFlow` control by swiping across your screen.

  > **i Note**
  >
  > The number of lanes that are hidden is displayed next to the navigation arrows.

- You use the **aggregated node** type in `sap.suite.ui.commons.ProcessFlowNode`, to display a group of nodes that are semantically equal or that have some properties in common. The possible values of the `type` property are `single` (default) or `aggregated`.
- To provide detailed information about a connection between process flow nodes, you can use **connection labels**. For example, if connections between nodes are not unique, the user can select a connection by clicking the respective label. To use this feature, the application needs to implement the popup, the content, and the actions required for the connection labels. You can hide or show these labels if you make the appropriate settings for the `showLabels` property.For more information, see the sample in the Demo Kit.
- The control provides various click **events**. You can use these events as follows:
  - Click event on the node:
    - To display a popover that has more information about a certain object type. From this quick overview, you can navigate to the object type.
    - If no additional information needs to be displayed, you can trigger an action sheet instead of the popover to allow the user to execute an action for the item.
  - Click event on the header:
    The `ProcessFlowLaneHeader` provides a `press` event **only** if the `ProcessFlow` is in Header mode. This happens if no nodes are defined. For more information, see the second sample on this sample page.
- The process flow offers a complete overview of structured data in their one-to-many (1:n) relationships. To focus on the important details in this flow, you can use the `highlighted` property to accentuate specific node relationships. If the `highlighted` property is set to `true`, the aggregation in the header node is also adapted (colors in donut charts and the lane header). For more information about highlighting , choose the *Toggle HighlightPath* button in the sample in the Demo Kit.

- In the process flow, you can set the **status** for the following objects:
  - **node**: Check the available statuses in `ProcessFlowNodeState`. The color of the donut chart is adjusted automatically based on the node status and cannot be overwritten.
  - **connection label**: Check the available statuses in `ProcessFlowConnectionLabelState`. You can set the status only when using connection labels.
  - **lane**: Check the available statuses in `ProcessFlowLaneState`.

  The control calculates all other states. For more information, see the API Reference.

Implementation Tips

- How to use connections and connection labels
  Connections are defined using the `children` property of a node:

```
id: "1", lane: "0", title: "Sales Order 1", titleAbbreviation: "SO 1",
children: [2, 3]
```

  If you want to use connection labels, you define the child node as an object consisting of a `nodeID` and a `connectionLabel` defined by the `ProcessFlowConnectionLabel` control:

```
    id: "1", lane: "0", title: "Sales Order 1", titleAbbreviation: "SO 1",
    children: [
    {
        nodeId: 2,
        connectionLabel: new sap.suite.ui.commons.ProcessFlowConnectionLabel({
            id: "myLabelId1To2",
            text: "my text",
            enabled: true,
            icon: "sap-icon://message-success",
            state:
sap.suite.ui.commons.ProcessFlowConnectionLabelState.Positive
        })
    }]
```

  Make sure that you also set the `showLabels` property on the `ProcessFlow` control to `true`. Otherwise, the labels are not visible.
- When to call the `updateModel()` method
  When you change the model that is bound to the `ProcessFlow` control, you need to call the `updateModel()` method, because changing the model affects the nodes and lanes in the process flow, so it must be recalculated. If you change only the content of the nodes, calling the `updateNodesOnly()` method is sufficient.
- Which binding mode to use
  When using an OData model, make sure that you set the binding mode to One-Way Data Binding as the default because the `ProcessFlow` control does **not** support Two-Way Data Binding. If you cannot set the binding mode of your OData model to One-Way Data Binding, you can bind a JSON model to the `ProcessFlow` filled with data from the OData model instead. For more information, see the API Reference Documentation in the Demo Kit and Setting the Default Binding Mode [page 999].

Size and Responsiveness

- The `ProcessFlow` control provides **four zoom levels**. It is responsive to the size of the container you put it in. Depending on the container size at the time of the initial load, the control chooses one of the four zoom levels. After loading, you can change the zoom levels as needed.
- In some cases, the Process Flow includes too much white space. You can reduce the white space by using the `optimizeLayout()` method. When the optimization mode is switched on (by calling the method),

layout optimization is triggered with every rendering or rerendering of the `ProcessFlow` control, for example, when zooming in or out.

> **i** Note
>
> The layout optimization process runs in parallel with other standard processes required for this control. This may lead to performance issues that can be avoided by testing your app before you use it in production.

API Reference and Samples

For more information about the `ProcessFlow` control, see the API Reference.

Status Indicator

The `StatusIndicator` control reflects a percentage value between 0 and 100.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The status indicator control can be used to display a percentage value in the form of a fillable shape or a group of shapes, translating plain numbers into meaningful visuals that convey the status of the items they represent.

Each status indicator may consist of a single group or multiple groups of shapes that display the value. The filling of the shapes can be proportional to the percentage value of the status indicator, or it can be based on thresholds that are specified using the `discreteThresholds` aggregation.

You can fully customize the control by setting fill direction and fill color and by picking a shape that matches the value measured. In addition to standard shapes, you can define your own custom SVG shapes. This allows you to create a powerful connection between your data and business by using symbols from a specific domain—for example, a half-filled truck shape may represent a real delivery truck filled up to 50% of its capacity.

The status indicator is best used in tiles, tables, and object pages.

Figure 352: Status Indicator Example

Details

Element Structure

- A status indicator must contain at least one group that includes at least one shape.
  - Groups can be defined using `ShapeGroup` elements joined in the `groups` aggregation.
  - Shapes are defined using `Shape` elements joined in the `shapes` aggregation.

Supported Shapes

- You can use any of the predefined shapes or create your own custom shapes:
  - Simple shapes that include circular shapes, rectangular shapes, and shapes defined as SVG paths.
  - Custom shapes that consist of an SVG definition, height, width, and other parameters defining the shape. The filling of a custom shape is defined by the `FillingOption` control.

  For more information about simple shapes, see `sap.suite.ui.commons.statusindicator.SimpleShape`.
  For more information about custom shapes, see `sap.suite.ui.commons.statusindicator.CustomShape`.

Shape Filling

- You can choose among the following filling types:
  - `Linear` – The shape is filled with a linear gradient.
  - `Radial` – The shape is filled with a radial gradient.
  - `None` – No filling is applied.

  For more information, see `sap.suite.ui.commons.statusindicator.FillingType`.
- The direction of the filling animation can be:
  - `Up` – The shape is filled from bottom upwards.
  - `Down` – The shape is filled from top to bottom.
  - `Left` – The shape is filled from right to left.
  - `Right` – The shape is filled from left to right.

  For more information, see `sap.suite.ui.commons.statusindicator.FillingDirectionType`.
- To define how the filling color should change based on the status indicator value, you can use the `PropertyThresholds` aggregation. For each threshold, you can define a filling color that will be applied until the status indicator value reaches this threshold.
  For more information, see `sap.suite.ui.commons.statusindicator.PropertyThreshold`.

Value Distribution

- By default, the filling of the status indicator shapes is proportional to the status indicator value. To adjust the filling, you can specify discrete thresholds using the `discreteThresholds` aggregation. When discrete thresholds are used, the displayed value may not exactly match the actual value of the status indicator.
  For more information, see `sap.suite.ui.commons.statusindicator.DiscreteThreshold`.
- When multiple groups of shapes are used, the percentage value is distributed in the following way: shapes in the first group in the `shapes` aggregation are filled first, shapes in the second group second, and so on.

T Account

The T account control displays debit and credit entries on a general ledger account.

The `TAccount` control can be used to display debit and credit entries on a general ledger account, visualizing the flow of transactions through the accounts where they are stored.

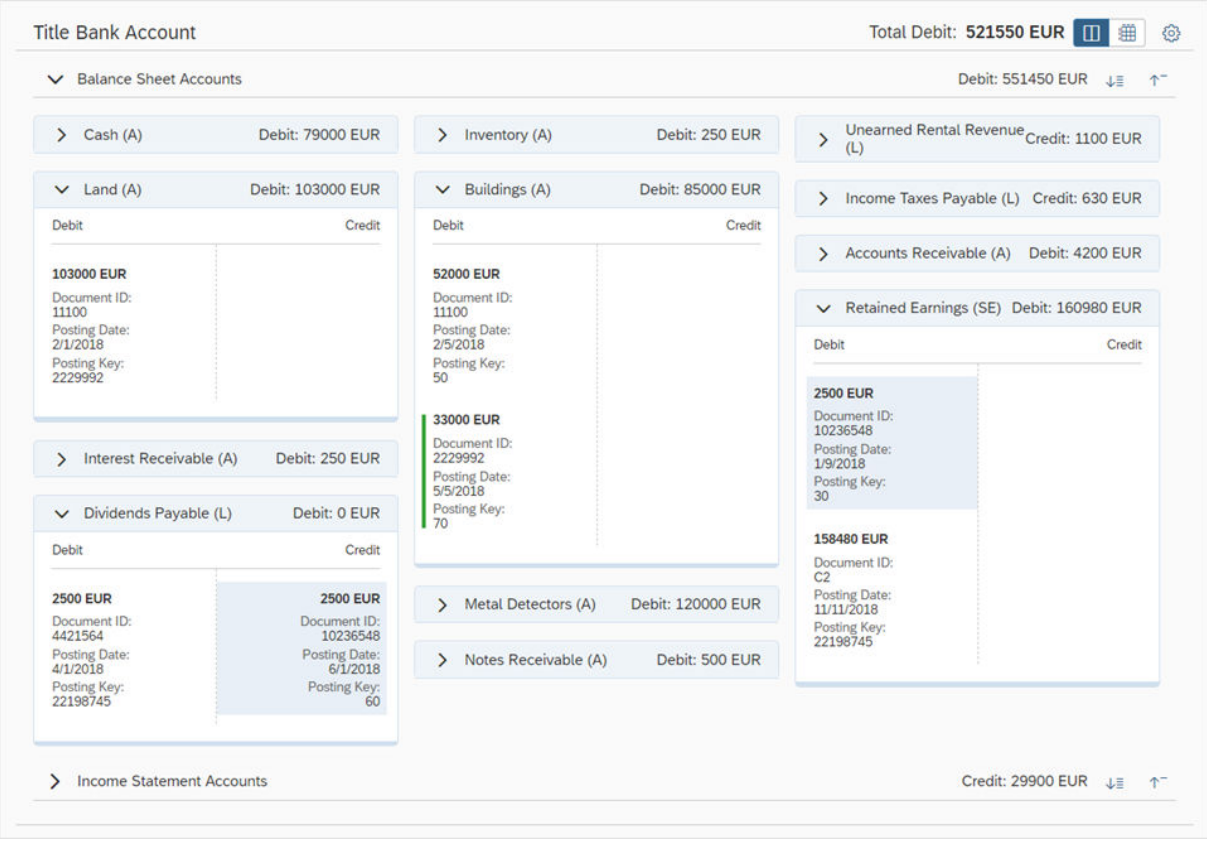
You can group T accounts by type, such as balance sheet T accounts and income statement T accounts, displaying multiple T accounts together. The T account group provides total balances for all T accounts included in this group. It also highlights matching debit and credit entries within the same group when an entry is clicked or tapped. In addition, you can display all T accounts and T account groups on a T account panel that provides a toolbar with total balances for all T account and T account groups included in the panel.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

In double-entry bookkeeping, journal entries are transferred to the general ledger by posting their debit and credit amounts on specific ledger accounts, which are often referred to as T accounts. A ledger account (or T account) is usually displayed in a format that resembles the letter T: with the account name above the T, debit entries to the left of the T, and credit entries to the right of the T.

T accounts are usually clustered together, so that accountants can analyze how individual line items from different journal entries affect the ledger balances.



Details

Layout and Grouping

- The T account control provides three subcontrols that can be used to create different layout combinations:
  - `TAccount` – A basic T account control with debit and credit entries arranged on either side of the T shape.
  - `TAccountGroup` – A group that includes multiple `TAccount` controls and displays them side by side. A T account group can be expanded or collapsed. It also provides total balance for the accounts included in the group.
  - `TAccountPanel` – A panel that combines multiple `TAccountGroup` and `TAccount` controls, providing total balance for all accounts and groups included in this panel. It also enables you to switch between T account view and table view and adjust display options.

Customizing

- You can add color indicators for debit and credit entries using the `color` property of the `TAccountItem` control. This can be an arbitrary color, but for consistency with other SAP Fiori apps, it is recommended that you use semantic colors defined by the sap.m.ColorPalette or sap.ui.unified.ColorPicker on the app level.
- You can define additional properties for debit and credit entries using `TAccountItemProperty` elements that can be added to the `properties` aggregation of the `TAccountItem` control. For details, see the API Reference.

Limitations

- It is recommended that all entries displayed in a single T account use the same currency.
  If debit and credit entries are in different currencies, the `TAccount` control does not convert the values to the currency specified in its `measureOfUnit` property. In such cases, the overall account balance is not displayed.
- Similarly, if T accounts in a T account group are in different currencies, the overall balance of the group cannot be calculated and is not displayed.

Timeline

The `Timeline` control displays a list of events, changes, or posts in chronological order.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The `Timeline` control shows information related to an object in chronological or reverse chronological order. This information may include entries, changes, or events related to an object. Every change, event, or entry is represented by a post on the timeline axis. The posts can be either generated by the system or added manually.

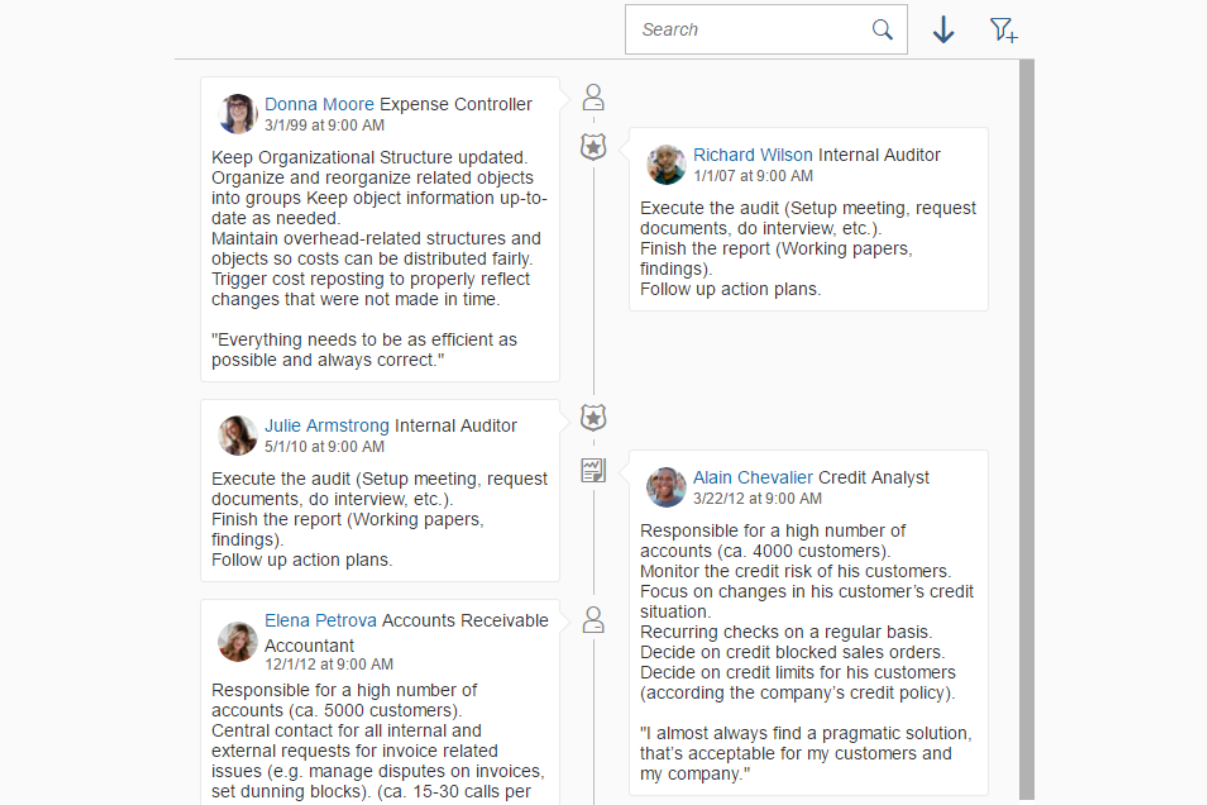


Figure 353: Timeline Example

Even though manual data entry is supported, it is recommended that you use data binding to pull data into the timeline.

The timeline does not have a fixed location on the UI. Where you place it depends on your use case. For example:

- If the timeline is closely related to the content and needs to be seen in parallel, you can use the dynamic side content floorplan. For more information, see the API Reference: `sap.ui.layout.DynamicSideContent`.
- If the timeline contains only secondary information, or needs to be accessed occasionally, you can embed it in a tab. For more information, see the API Reference: `sap.m.IconTabBar`.
- If you are using the object page floorplan, you can use the horizontal layout to integrate the timeline. For more information, see the API Reference: `sap.uxap.ObjectPageLayout`. See also *Details* > *Layout and Appearance* > below.

These are only some of the ways you can position the timeline on a page.

Details

Data Binding

- There are two types of data binding that you can use to pull data into timeline posts:
  - JSON binding

- OData binding

- If JSON data binding is used, it is recommended that you add a function that converts all date strings into dates. For example:

```
function convertData(oEvent) {
    var oModel = oEvent.getSource();
    if (!oEvent.getParameters().success) {
        return;
    }
    oModel.getData().Employees.forEach(function (oEmployee) {
        oEmployee.HireDate =
sap.suite.ui.commons.utilDateUtils.parseDate(oEmployee.HireDate);
    });
    oModel.updateBindings(true);
}
```

  If strings are not converted into dates, sorting of the timeline posts may fail.
- If your timeline provides filtering, it is important that you enable filtering directly on the binding level by setting the `enableModelFilter` property to `true`. This allows the filtering to be performed before any data is pulled into the timeline. If the `enableModelFilter` property is set to `false`, the timeline loads all data before performing the filtering, which may lead to performance issues.
  If you use filtering over a large amount of data (more than 100 timeline posts), you may need to increase the data model's size limit. The default limit is 100, which means that filtering is performed over the first 100 items in the data set. To increase the data model's size limit, set the model's `sizeLimit` property to a higher number. For details, see `sap.ui.model.Model.html#setSizeLimit`.

Initialization

The way you initialize a timeline depends on whether you want to use JSON model binding, OData model binding, or no binding at all.

- JSON Data Binding

```
// File data.json
{
    "Employees": [
        {
            "Name": "Laurent Dubois",
            "JobTitle": "Accounts Payable Manager",
            "JobResponsibilities": "Plans, organizes and manages the
operations and activities of an accounts payables.\nSupervises employees and
monitors activities.\nFinal check of accounts payable payments and sign off.
\nReporting to the head of finance.\n\n\"I am a diligent person. I put great
attention to detail.\"",
            "HireDate": "Date(1371020400000)"
        }
    ]
}
```

```
// controller.js
// a function that converts date strings into dates
function convertData(oEvent) {
    var oModel = oEvent.getSource();
    if (!oEvent.getParameters().success) {
        return;
    }
    oModel.getData().Employees.forEach(function (oEmployee) {
        oEmployee.HireDate =
sap.suite.ui.commons.utilDateUtils.parseDate(oEmployee.HireDate);
    });
    oModel.updateBindings(true);
```

```
}
// in onInit function
var oModel = new JSONModel("data.json");
oModel.attachRequestCompleted(convertData);
var oItem = new TimelineItem({
    dateTime: "{HireDate}",
    title: "{JobTitle}",
    text: "{JobResponsibilities}",
    userName: "{Name}"
});
var oTimeline = new Timeline({
    enableDoubleSided: true
});
oTimeline.bindAggregation("content", {
    path: "/Employees",
    template: oItem
});
oTimeline.setModel(oModel);
```

- OData Model Binding

```
var oModel = new ODataModel("http://services.odata.org/V3/Northwind/
Northwind.svc/", true);
var oItem = new TimelineItem({
    dateTime: "{HireDate}",
    title: "{Title}",
    text: "{Notes}",
    userName: "{FirstName} {LastName}"
});
var oTimeline = new Timeline({
    enableDoubleSided: true
});
oTimeline.bindAggregation("content", {
    path: "/Employees",
    template: oItem
});
oTimeline.setModel(oModel);
```

- No Data Binding

```
var oTimeline = new Timeline("myNiceTimeline", {
    content: [
        new TimelineItem({
            dateTime: new Date(2016, 1, 1),
            icon: "sap-icon://accept",
            title: "Title 1",
            text: "Some comment"
        }),
        new TimelineItem({
            dateTime: new Date(2015, 2, 2),
            icon: "sap-icon://decline",
            title: "Title 2",
            text: "Some comment"
        })
    ]
});
```

Layout and Appearance

- **Basic layout** – A timeline consists of a chronological axis, timeline posts, and an optional header. The header may include a search field, as well as sorting, filtering, and grouping options.
  You can set the timeline axis to be displayed vertically or horizontally, with the posts arranged on one or both sides of the axis. The posts can be displayed in chronological or reverse chronological order.

- **Scroll bar** – It is not recommended that you use the timeline control inside a scroll container. Please use the timeline's `enableScroll` property instead. When the timeline's `enableScroll` property is set to `true`, the timeline has its own scroll bar.
  If you still want to use your timeline inside a scroll container, make sure that your timeline meets the following requirements:
  - The timeline's `growingThreshold` property is set to `0`.
  - The `lazyLoading` property is set to `false`.
  - The `forceGrowing` property is set to `false`.
- **Node icons** – Timeline posts can have optional node icons displayed on the timeline axis itself. To define the icons used for posts, use the `icon` property of the `TimelineItem` object. If you want the icons to use semantic colors that indicate the status conveyed by the post, use the `status` property of the `TimelineItem` object.
  If you don't want to display any icons on the timeline axis itself, set the timeline's `showIcons` property to `false`.

Responsiveness

- The timeline control works with multiple screen sizes. If your timeline uses double-sided layout (the `enabledDoubleSided` property is set to `true`), the posts are displayed on one side of the timeline axis on smaller screens. The side depends on the `alignment` property settings.

Integration with Other Controls

- You can embed other SAPUI5 controls into timeline posts. However, please note that not all properties that work for ordinary posts can be applied to posts with embedded controls. For example, the `textHeight` property cannot be applied correctly to posts with embedded controls.

Related Information

Data Binding [page 815]
API Reference: `sap.m.ScrollContainer`

sap.suite.ui.microchart

This library contains controls to visualize charts and diagrams that are lightweight and easy to use.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

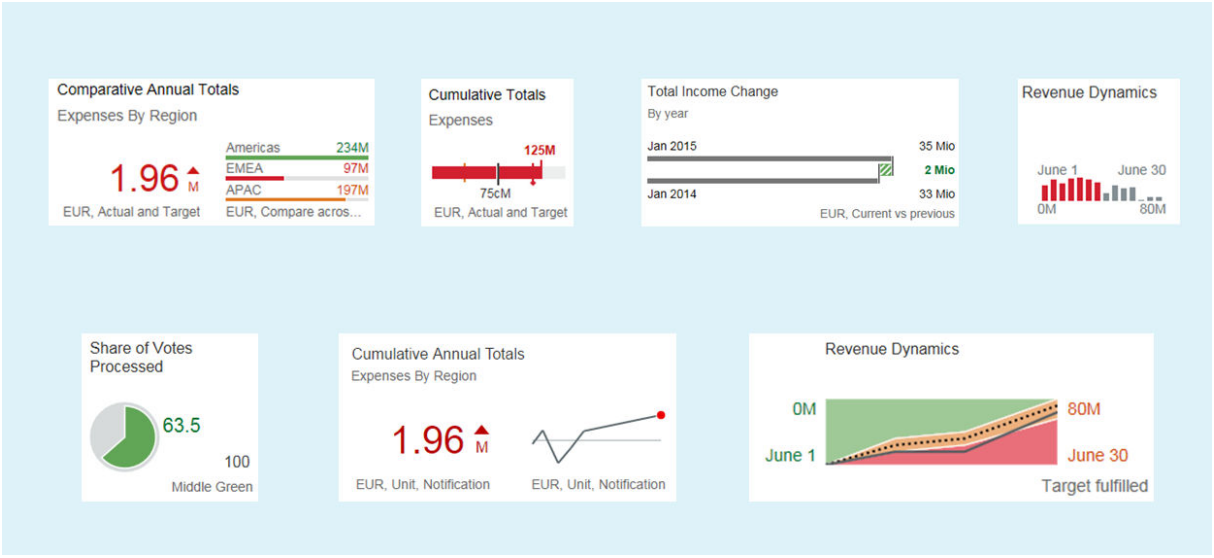
`MicroCharts` give a quick overview of customer-defined key figures like KPIs as graphical items and display the current status of defined key figures or thresholds.

The different values can be visualized in the charts by a semantically-colored representation and can be defined by the customer to correspond to the significance of the figures. For example, you can display critical

statuses in red or you can display thresholds in specific colors to represent the significant values of your business data.

`MicroCharts` have a responsive design and can adapt their appearance and functions to the screen size of the devices. In the responsive mode, if `MicroCharts` are included in further controls (for example generic tiles or flex boxes), the control adapts to the available space provided by the parent control.

> **i Note**
>
> You can include a `MicroChart` control into a `GenericTile` control in the responsive mode only. However, if you define a width, a height, and size properties, this data is overwritten and the responsive mode is used.



Prerequisites

With the new `sap.suite.ui.microchart` library, the available MicroChart controls of the `sap.suite.ui.commons` library have been moved to their own library with SAPUI5 version 1.34.

- If you have already included a `MicroChart` control before SAPUI5 version 1.34, a wrapper ensures that the embedding still works for each control. To benefit from all enhancements or new features for the `MicroChart` controls as of SAPUI5 1.34, you need to switch to the controls to the new library. With SAPUI5 1.34, all `MicroChart` controls in the `sap.suite.ui.commons` library are marked as deprecated. The respective controls are as follows:
  - `AreaMicroChart`
  - `BulletMicroChart`
  - `ColumnMicroChart`
  - `ComparisonMicroChart`
  - `DeltaMicroChart`
  - `HarveyBallMicroChart`

> **i Note**
>
> During the move, the following controls and their elements have been renamed:
>
> - `MicroAreaChart` to `AreaMicroChart`
> - `ComparisonChart` to `ComparisonMicroChart`
> - `BulletChart` to `BulletMicroChart`

Related Information

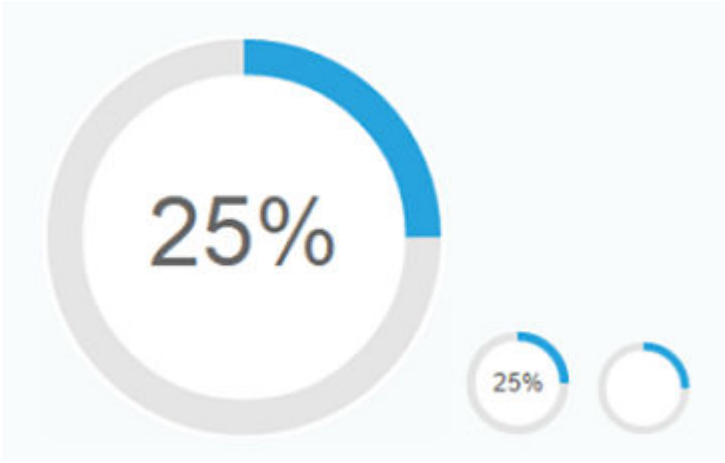
Browser and Platform Support [page 20]
Supported Library Combinations [page 26]
API Reference: `sap.suite.ui.microchart`

Radial Micro Chart

Displays a ring chart that represents a percentage value.

Overview

The `RadialMicroChart` control provides a ring chart that displays the percentage value in the center of the chart. To indicate the significance of the displayed percentage value, you can define the status as `Good`, `Neutral`, `Critical`, or `Error`.



Details

The percentage value in the `RadialMicroChart` control can be defined by:

- Inserting the percentage property directly.
- Setting the total and the fraction property and calculating the percentage value automatically.

> **i** Note
>
> If the total property is set to `0`, only a blank circle gets rendered because no valid value is returned. The error is described in the error log.

Colors

The color of the circle is defined by setting the `valueColor` property to one of the following types:

- `sap.m.ValueColor`: The possible value color settings are listed and show the status `Good`, `Neutral`, `Critical`, or `Error`.
- `sap.ui.core.CSSColor`: All CSS color values are allowed.

Sizes

The `RadialMicroChart` control is rendered in a responsive way. Thus, the size of the control adapts automatically to the size of the surrounding container and does not have a defined width or height. Alternatively, you can use the `size` property with the control, so that the Radial Micro Chart can also be rendered in fixed sizes that are available in `sap.m.Size`. There, you can choose between the sizes `S`, `M`, `L`, `Auto` and `Responsive`.

API Reference/Samples

For more information about the `RadialMicroChart` control, see the API Reference in the Demo Kit and the sample in the Demo Kit.

Line Micro Chart

Displays a series of values as segmented lines along a threshold line.

Overview

The `Line Micro Chart` control displays a series of values as segmented lines along a threshold line.

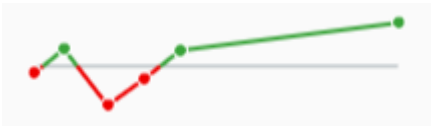
Details

You use this control primarily for embedded analytics applications. It is designed to display a set of ordered points. These points are connected by lines that show a data progression for a specific data range. If you want to add further details, such as decisive values or dimensions to this chart, you can add up to four labels.

Usage

With the line micro chart, you can visualize the diagram curve and show trends. If you want to stress values that are above or below a certain threshold, you can use data points.

By default, the line of the chart is blue. However, you can use different colors, such as semantic colors for the line, to mark positive and negative values.



You can also choose to use focus points instead of data points. Data points are useful if you want to concentrate on one or two special values, for example, the first and the last data point on the chart. For focus points, you can use any CSS color, but we recommend using semantic colors.

Properties

Color

With the sap.m.ValueCSSColor type, the Line Micro Chart can use regular points or emphasized points, with or without semantic colors. You can find more information in the API Reference in the Demo Kit.

Data Points

The *showPoints* property controls whether the points are displayed. The default value is false, which means that the points are not displayed and only a continuous line is visible. You can use this configuration to show trends and to visualize the data progression. If emphasized points are used, the chart's color and *showPoints* properties do not have an effect. These properties can be used for regular points only. Do not use these two data point types together in one chart.

These types of data points are available:

- LineMicroChartPoint for the *x* and *y* coordinates.
- LineMicroChartEmphasizedPoint consists of the *color* and *show* properties and *x* and *y* properties.

Scaling

The chart has a built-in automatic scale that is applied based on existing values, so that all values are visible. You can also set a manual scale, but only a part of the chart might be visible. You can set a manual scale by providing values for the following properties:

- minXValue
- maxXValue
- minYValue
- maxYValue

API Reference/Sample

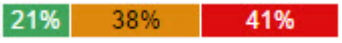
For more information about the `LineMicroChart` control, see the API Reference in the Demo Kit and the sample in the Demo Kit.

Stacked Bar Micro Chart

The control shows a progress chart that displays the overall progress and a summary of the items.

Various bars can have different value colors, for example for different statuses such as accepted, rejected, and open items. There is no limit to the number of bars that the `StackedBarMicroChart` control can contain.

The `sap.suite.ui.microchart.StackedBarMicroChart` control is designed to display key figures or numbers inside a set of bars defined by different lengths and colors. It can be displayed standalone or in a table.



Inspection Lot	Batch	Samples	Characteristics	Progress
40000000032001	25632596	2	30	67% 23%
40000000032002	25632597	2	23	23% 43% 33%

Details

Bar values

The progress status of each item can be defined as an absolute value or as a percentage (`value` property). Each stacked bar is part of the chart that is defined as 100% (if you are working with percentages) or as an absolute value (if you are working with a maximum value). If you do not set a maximum value (`maxValue` property), the sum of all bar values is considered as the maximum value and the displayed value is set as a percentage for each item.

The number of decimal places for percentage values is, by default, 1. This can be changed by the application using the `precision` property. If the percentage values are used as labels, the `precision` property affects the length of the labels. Take this into account when the length of the bar is limited or there is not enough space to display the whole label.

Bar colors

By default, the given value is displayed as a label inside the bar, but you can overwrite it in your application using the `displayValue` property. You can also overwrite the default color of the bar (`valueColor` property) using predefined `less` parameters for colors, predefined semantic parameters (for example, `Error`, `Critical`, `Good`, `Neutral`) or even hex values, such as `#fafafa`. If there is no color, the micro chart automatically chooses one of the chart colors (`sapUiChart1-11`).

Text and background colors

The text color adapts automatically depending on the corresponding bar background color. If there is a dark background, a light text color is used and if there is a light background, a dark text color is used.

Chart size

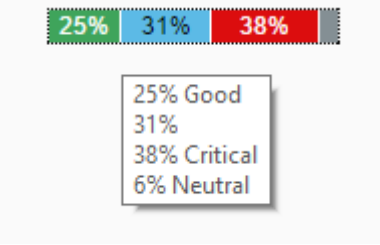
You can define the chart with the `size` property. If you use a fixed size, you can select predefined sizes (XS, S, M, L). If you use a responsive design, you can use the `responsive` property value. The chart size is then adjusted based on the surrounding container size, for example in a `sap.m.FlexBox` control. The maximum height of the chart is 1 rem. If there is not enough space to show the labels, they are not displayed. If there is not enough space to show the micro chart, it is not displayed either.

Integration and use

The `sap.suite.ui.microchart.StackedBarMicroChart` control is a chart control and aggregates the bars of type `sap.suite.ui.microchart.StackedBarMicroChartBar`. One chart can contain one or more bars. You can call it up by clicking the mouse button or by using the `Space` or `Enter` key.

Tooltip support

The chart can have a tooltip with predefined `displayValue` property values or calculated percent values and it can be set on bar level and on chart level. Each value of the chart is displayed in the tooltip in a separate line. When the bar has a semantic color, the text (for example Good, Critical) will be added behind the displayed value. In some cases the text in the tooltip has another name than the semantic color. The tooltip can be suppressed by setting its value to an empty string.



API Reference/Sample

For more information about the `StackedBarMicroChart` control, see the API Reference and the samples in the Demo Kit.

sap.tnt

This library contains controls that provide the basic structure of a tool app.

The goal of these controls is to ensure consistency of the user interfaces in the tools area and the implementation of a common design language of applications and tools on the basis of SAP Fiori.

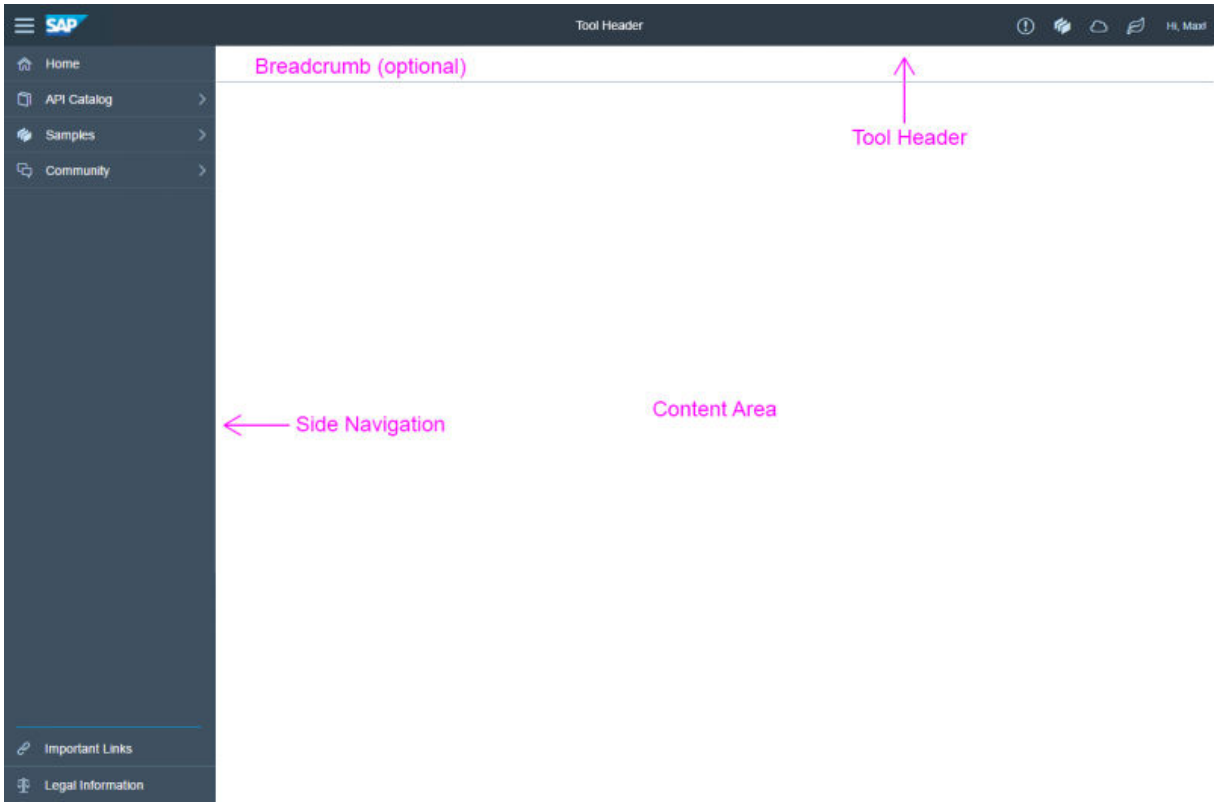


Figure 354: Basic Page Structure of a Tool

> **i Note**
>
> This set of controls described in this article has been designed exclusively for the SAP tool landscape for the SAP Cloud Platform.
>
> Do not use these controls in regular SAP Fiori applications. This highly specialized set of controls targets the specific needs of the tools user group (typically developers and administrators).

For more information, check out the concept guidelines at https://experience.sap.com/fiori-design-web/overview/.

Related Information

Supported Library Combinations [page 26]
API Reference: `sap.tnt`

sap.ui.codeeditor

This library contains the `CodeEditor` control.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Code Editor

The `sap.ui.codeeditor.CodeEditor` offers functionality to display and edit source code artifacts with syntax highlighting and code completion capabilities for various programming languages.

Overview

The `sap.ui.codeeditor.CodeEditor` provides simple SAPUI5 wrapper control that exposes a subset of API and functionality provided by the third-party ACE (Ajax.org Cloud9 Editor) implementation. You can find more information about ACE on the website https://ace.c9.io/ .

> **!** Restriction
>
> - If you use API calls to the native API of ACE, we cannot guarantee backwards compatibility after an upgrade to higher ACE library versions.
> - Accessibility features like high-contrast themes and keyboard handling are not fully available as they are for the rest of the SAPUI5 controls.
> - `CodeEditor` doesn't work within `IconTabBar` on Internet Explorer. However, there is a way to achieve the same functionality. For more information, see the Sample.

Details

Autocompletion

The sap.ui.codeeditor.CodeEditor control is enabled with two modes of autocompletion:

- Default autocompletion
  The default autocompletion options of the underlying ACE editor `enableBasicAutocompletion`, `enableSnippets`, `enableLiveAutocompletion` are always enabled.
- Custom autocompletion
  The users of CodeEditor can specify their own autocompletion handling. To do this, the following API method is provided: `CodeEditor.prototype.addCustomCompleter(oCustomCompleter)`.
  There are several characteristics you should bear in mind:
  - `oCustomCompleter` must contain implementation of a custom completer method called `getCompletions`.
  - The method accepts two parameters: `fnCallback` and `context` object. `Callback` should be called, as in the example, with the suggestions that you want to appear in the editor. The format is an array of objects. Each object should contain the following properties: `name`, `value`, `score` and `meta`.
  - The `context` object contains `oPos` and `sPrefix` as provided by the ACE editor.

Here is an example of custom autocompletion:

Sample Code

```
codeEditor.addCustomCompleter({
      getCompletions: function(callback, context) {
          // callback is provided to us by ACE so we can execute it as shown
          // below in order to display suggestions to the user
          // ideally, the array argument, provided to the following method
call
          // will be dynamically generated based on the content of the context
          // object
          // let's assume the context contains an sPrefix equal to 'read',
which
          // means the cursor in ACE is at the end of a 'read' word
          // by executing the following call, we can show a list of
suggestions
          // such as: readFile, readStream, readResponse
          callback(null, [{
            name: "foo",
            value: "foo",
            // name is not displayed on the screen
            name: "readFile",
            // value is displayed on the screen
            value: "readFile()",
            // score determines which suggestion goes first
            score: "1",
            meta: "rhyme"
              // meta is short info displayed on the right of
value                       meta: "function"
          }, {
            name: "bar",
            value: "bar",
            score: "1",
            meta: "rhyme"
              // name is not displayed on the screen
            name: "readStream",
            // value is displayed on the screen
            value: "readStream(input)",
            // score determines which suggestion goes first
            score: "3",
            // meta is short info displayed on the right of value
            meta: "params: input"
          }, {
            name: "baz",
            value: "baz",
            score: "1",
            meta: "rhyme"
```

```
            // name is not displayed on the screen
            name: "readStream",
            // value is displayed on the screen
            value: "readStream(input, encoding)",
            // score determines which suggestion goes first
            score: "2",
            // meta is short info displayed on the right of value
            meta: "params: input, encoding"
          }])
```

sap.ui.comp

This library contains composite controls.

> **i Note**
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]
Browser and Platform Support [page 20]
API Reference: `sap.ui.comp`

Filter Bar

The `sap.ui.comp.filterbar.FilterBar` control is used to provide a user-friendly interface for queries.

The frequently asked questions section below aims at answering some basic questions that you might have when using this control.

For more information about this control, see the API Reference and the samples.

Overview

The `FilterBar` control has been implemented as a toolbar with a collapsible filter area. Whereas the toolbar is always visible, the filter area can be hidden to reduce the space required.

> **i Note**
>
> This applies to desktop use. On tablets, the filter area is always collapsed but can be expanded by the user. On phones, the filter area is not displayed at all. You can access the filters through the `Filters` dialog.

The toolbar includes the `VariantManagement` control and contains the following buttons:

- *Hide Filter Bar* / *Show Filter Bar*
  Hides or displays the filter area.

  > **i Note**
  >
  > This function is available for desktop use only.

- *Filters*
  Lists the filters available and, if selected, displays the filter dialog.
- *Go*
  Executes the query.
  A `Search` event is raised, and the consuming control must respond to it. The `showGoButton` property is set to `True` by default and determines whether the button is visible.

> **i Note**
>
> You can also deactivate the toolbar using the `useToolbar=false` property setting. In this case the filter bar buttons are rendered in the filter area right behind the filters; on phones, right before the filters. The *Filters* dialog is then called *Adapt Filters* dialog.

> **i Note**
>
> The `FilterBar` control can be used on its own. However, we strongly recommend to use it in combination with the `SmartFilterBar` control.

Details

`FilterItem` and `FilterGroupItem`

Filters are similar to query parameters. A filter is represented either by a `FilterItem` element or its specialization, the `FilterGroupItem` element. They are populated via the `FilterItems` or `FilterGroupItems` aggregations. The main difference is that the `FilterGroupItem` has the `group` attribute. The `FilterItem` is internally processed as a `FilterGroupItem` that belongs to the `Basic` group.

The embedding component of the `FilterBar` control determines if a filter is mandatory and visible, defines its label, and whether the filter is visible in the filter area.

Filters Dialog

The *Filters* dialog provides all the functions that are available with the `FilterBar` control. You can define which filter is visible in the filter bar and whether a filter is shown in the current view at all. You can also clear and restore values by selecting *Clear* and *Restore* and trigger the query execution.

> **i Note**
>
> Some of the buttons are hidden and must be enabled first.

The information in this dialog is row-based. The first element is the `VariantManagement` control, followed by the filters. The filters are organized in groups, each starting with a group title. For each filter, you can select the *Add to Filter Bar* checkbox to make the filter visible in the filter bar.

> **i Note**
>
> This function is available for desktop use only.

The filters of the *Basic* group are always included in the current view. All other group filters are initially not assigned to the current view. You can assign filters other than the ones from the basic filter group by selecting the following for a specific group:

- *More Filters*
  Shows all filters of the relevant group that have not been assigned yet.
- *Change Filters*
  Displayed if all filters of the relevant group have already been assigned.

This takes you to the *Select Filters* dialog where you can define which filters are included in the current view. This dialog displays all the filters of a group.

> **i Note**
>
> The *Select Filters* dialog is available on its own and can be used in other scenarios, for example, in the value help dialog.

Once filters have been included in the current view, they can also be added to the filter bar. A filter can only be removed from the current view if it is not visible in the filter bar.

Mandatory filters can only be removed from the filter bar if they have a value. As long as a mandatory filter does not have a value, it will be treated as added to the filter bar.

The buttons in the dialog offer the following functions:

- *Save*
  Represents the *Save* function for the `VariantManagement` control on the user interface.
  It either saves the current view or provides the *Save* dialog for view management if a *Save As* function is required.
- *Clear*
  Clears all filters by raising the `Clear` event.
  The `showClearButton` property is set to `False` by default and determines whether the button is visible.
- *Restore*
  Reverts all changes of the current view by raising the `Reset` event and applies the current view again.
  The `showRestoreButton` property is set to `True` by default and determines whether the button is visible.
- *Go*
  Executes the query.
  The function of the button is the same as for the one in the filter bar. However, here the button is always visible.
- *Cancel*
  Reverts all changes made by the user since the dialog was opened and closes the dialog.

Integration with Value Help Dialog

The filter bar is also used in the *Value Help* dialog scenarios. The filter bar property `advancedMode` must be set to `true` to enable this function.

The embedding component has to provide a search field using the `FilterItems` aggregation and an advanced search using the `FilterGroupItems` aggregation. For the search, the `Search` event is raised.

Integration with Smart Variant Management

A view represents a set of filters. Views are handled and represented on the user interface by the `VariantManagement` control that is included in the toolbar of the `FilterBar` control. Views are stored in a backend system and also retrieved from there.

For views, the following is available:

- Shell service for personalization
- SAPUI5 flexibility

They can be accessed using the `SmartVariantManagement` or `SmartVariantManagementUI2` control.

For more information about the shell services, see the API Reference in the Demo Kit. For more information about SAPUI5 flexibility, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152].

The consuming control that is using the `SmartVariantManagement` control has to provide the following dedicated methods:

- `applyVariant (oVariantContent)`
- `var oVariantContent = fetchVariant ()`

The `fetchVariant` has to return a JSON object. This is the information that is stored along with some administrative information, such as the name of the view and the ID. Once the view has been retrieved from the backend system and transferred to the consuming control, the `applyVariant` is executed. During execution, the consuming control must know the format of `oVariantContent`, since the control must also provide the method and react on it.

The following table shows which controls are used by the filter bar controls to handle views:

Table 116: Controls for View Management Integration

Control	Uses
`FilterBar`	`SmartVariantManagementUI2`
`SmartFilterBar`	`SmartVariantManagement`

Shell Service for Personalization

The shell service for personalization is handled internally by the `FilterBar` control. The basic view handling is implemented by the `SmartVariantManagementUI2` control. It extends the `VariantManagement` control. The shell service for personalization only supports end user personalization.

The consuming control of the `FilterBar` has to provide two methods, one for fetching the data that must be stored as the content of the view, and one for applying this data, if the view is set. Both methods have to be registered using the corresponding methods of the `FilterBar` control:

- `registerApplyData (fApplyData)` – interface: `fApplyData (oVariantContent)`
- `registerFetchData (fFetchData)` – interface: `oVariantContent = fFetchData ()`

A persistence key `setPersistencyKey` (`sKey`) must be provided as well. This key identifies the storage area and saves and retrieves the views currently used.

To trigger the retrieval of the views, the consuming control must register to the `Initialise` event of the `FilterBar` and call the `fireInitialise()` method in the `FilterBar` control.

> **i Note**
>
> The retrieval of the initial views is done asynchronously. If the retrieval has been completed, the `SmartVariantManagementUI2` control will be populated with the view information, and the event handler for the `Initialise` event will be called.

SAPUI5 Flexibility

SAPUI5 flexibility is handled internally by the `SmartFilterBar` control. The basic view handling is implemented by the `SmartVariantManagement` control. It extends the `VariantManagement` control. SAPUI5 flexibility features support end user personalization and also allow you to create and store views in the VENDOR layer of the layered repository.

> **i Note**
>
> We recommend that you always use SAPUI5 flexibility rather than the shell service.

> **i Note**
>
> You can also hide `VariantManagement` if no `persistencyKey` is provided. Also, you can separate the `VariantManagement` control from the `FilterBar` control by using the page variant of the `SmartVariantManagement` control.

FAQ

How can I use SAPUI5 flexibility in the `FilterBar` control?

In general, the `FilterBar` control supports the shell service for personalization. The following steps describe how to enable the `FilterBar` control to use the SAPUI5 flexibility features provided the `FilterBar` is already used and the consuming control supports the shell service via the `FilterBar` control. If this is not the case, make sure the shell service is used as described above before you start.

After that, here is what you need to do:

1. Extend the `FilterBar` to create a new `FilterBar` control.

```
sap.ui.comp.filterbar.FilterBar.extend("my.ui5flex.FilterBar") {..}
```

2. Overwrite the internal `_initializeVariantManagement` function of the `FilterBar` control.

```
my.ui5flex.FilterBar.prototype._initializeVariantManagement = function() {
          if (this._oSmartVM && this.getPersistencyKey()) {
            var oPersInfo = new sap.ui.comp.smartvariants.PersonalizableInfo({
             type: "filterBar",
             keyName: "persistencyKey"
             });
         oPersInfo.addControl(this);
```

```
this._oSmartVM.addPersonalizableControl(oPersInfo);

sap.ui.comp.filterbar.FilterBar.prototype._initializeVariantManagement.apply(t
his, arguments);
     } else {
        this.fireInitialise();
 }
    };
```

> **i** Note
>
> Here, the `_initializeVariantManagement` function has to be called instead of the `fireInitialise()` method, as mentioned before. The main purpose of this function is to register the extended `FilterBar` control to the `VariantManagement` control. Once the `VariantManagement` control has been initialized, the `FilterBar` control triggers the `fireInitialise()` method call internally.

3. Overwrite the internal `_createVariantManagement` function of the `FilterBar` control.

```
my.ui5flex.FilterBar.prototype._createVariantManagement = function() {
         this._oSmartVM = new
sap.ui.comp.smartvariants.SmartVariantManagement({
                    showExecuteOnSelection: true,
                    showShare: true
                  });
                return this._oSmartVM;
};
```

Further Communication between `FilterBar`, `SmartFilterBar`, and Consuming Control

The following is also valid for the `SmartFilterBar` control: Before a view is saved, the `FilterBar` control triggers the `beforeVariantSave` event. This allows the consuming control to prepare for the `fetchData` call, if required. Right after this event is raised, the `FilterBar` control calls the method provided by the `registerFetchData` method. If you select a view from the `VariantManagement` control or the `SmartVariantManagement` respectively after the `FilterBar` has called the method provided by the `registerApplyData` method, the `afterVariantLoad` event is raised.

> **i** Note
>
> The `afterVariantLoad` event is also raised internally when you select *Cancel* or *Restore* in the *Filters* dialog of the filter bar.

Related Information

Smart Filter Bar [page 2413]

Smart Variant Management [page 2457]

Smart Chart

The `sap.ui.comp.smartchart.SmartChart` control can be used to create complex diagrams.

The frequently asked questions section below aims at answering some basic questions that you might have when using this control.

> **i** Note
>
> The code samples in this section reflect examples of possible use cases and might not always be suitable for your purposes. Therefore, we recommend that you do not copy and use them directly.

For more information about this control, see the API Reference and the samples.

For more information about annotations for this control, see the API Reference.

Overview

You can select a chart type, such as a pie chart, column chart or a chart with an x and a y-axis or two y-axes, and define the dimensions and measures for the chart and how you want to display them. A drilldown enables you to display even more information about a dimension.

On the UI, the `SmartChart` control consists of a toolbar and a chart area. The control uses the `sap.chart.Chart` control, which is shown in the chart area. The toolbar offers you various functions, such as the selection of various types of charts, the drilldown/drillup and maximize/minimize functions as well as zooming in and out of a chart, and a download. In addition, the control allows you to navigate to the related semantic object for the chart by clicking `Jump To` after selecting the relevant part of the chart, such as a column (or, alternatively, you can also use a similar feature by clicking the `Details` button if this feature is enabled).

Details

The `SmartChart` control offers further functions by integrating other smart controls: You can save a chart as a view using the `SmartVariantManagement` control or make chart-specific personalization settings using the `sap.m.P13nDialog` control.

In the personalization dialog, you can select a number of chart-specific features in the *Chart* panel, such as selecting various dimensions and measures. You can also sort the data in the chart or filter it based on the conditions you define here. The *Filter* panel shows the filter criteria that have been defined here manually. For more information about other, related smart controls, see sap.ui.comp [page 2401] and Personalization Dialog [page 2358].

If a chart is changed several times, the final outcome of the changes can be persisted as a view once the chart looks as required by the user. When the view is loaded the next time, the final outcome of the changes will be shown, but not each single step of the changes.

If the `showDownloadButton` property is set to `true`, you can download the part of the chart that is currently visible by clicking the `Download Chart` button. The chart will be downloaded in PNG format.

> **i Note**
>
> In Microsoft Internet Explorer, you can only download SVG formats. Here you will be notified in a notification bar at the bottom of the page that allows you to display or download the chart.

Annotations

The following table shows a selection of the annotations used by the `SmartChart` control:

Table 117: Annotations

Element	Annotation	Value	Mandatory	Description
Entity type	`sap:semantics`	aggregate	Yes	Enables the aggregation of dimensions and measures.
Dimensions	`sap:aggregation-role`	dimension	Yes	Defines the dimensions.
Measures	`sap:aggregation-role`	measure	Yes	Defines the measures.

A property can be annotated with attribute `sap:aggregation-role` if it has an aggregation role. The attribute can have the value `"dimension"` if the property represents the key of a dimension or `"measure"` if the property represents a measure whose values are aggregated according to the aggregation behavior of the entity type that contains the control. Both values are only valid for properties of an entity type that is annotated with `sap:semantics="aggregate"`.

```
<EntityType Name="Product" sap:service-schema-version="1" sap:service-
version="1" sap:semantics="aggregate" sap:content-version="1">
    <Property Name="Category" Type="Edm.String" Nullable="false" MaxLength="40"
sap:aggregation-role="dimension" sap:label="Product Category"
sap:creatable="false" sap:updatable="false" sap:sortable="true"
sap:filterable="true" />
    <Property Name="Quantity" Type="Edm.Decimal" Nullable="false" MaxLength="3"
sap:aggregation-role="measure" sap:label="Quantity"
     sap:creatable="false" sap:updatable="false" sap:sortable="true"
sap:filterable="true" />
      ...
</EntityType>
```

The `com.sap.vocabularies.UI.v1.Chart` annotation is used to specify the chart type and the visible measures and dimensions of the chart.

```
<Annotations Target="EPM_DEVELOPER_SCENARIO_SRV.Product"
      xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <Annotation Term="com.sap.vocabularies.UI.v1.Chart">
            <Record>
                  <PropertyValue Property="ChartType"
                        EnumMember="com.sap.vocabularies.UI.v1.ChartType/
Column" />
                  <PropertyValue Property="Dimensions">
                        <Collection>
```

```
                              <PropertyPath>Name</PropertyPath>
                              <PropertyPath>Category</PropertyPath>
                        </Collection>
                  </PropertyValue>
                  <PropertyValue Property="Measures">
                        <Collection>
                              <PropertyPath>Price</PropertyPath>
                              <PropertyPath>Quantity</PropertyPath>
                        </Collection>
                  </PropertyValue>
            </Record>
      </Annotation>
</Annotations>
```

FAQ

Can I use annotations with qualifiers in the `SmartChart` control? And, in particular, how can I use the annotations with a qualifier within the control?

As a general rule, the `SmartChart` control looks for annotations **without** a qualifier, the primary annotations. However, you can also use the `PresentationVariant` and `Chart` annotations with qualifiers as mentioned below.

We first look for `PresentationVariant` and try to get the `Chart` annotation from there. If no such annotation exists, we look for the `Chart` annotation directly on the entity.

You can use `chartQualifier` for the `SmartChart` control:

```
<SmartChart customData:chartQualifier=”Customer360” …>
```

Or use the following:

```
<SmartChart customData:presentationVariantQualifier=”Customer360” …>
```

If **no qualifier** has been defined for the presentation variant, you can use the fallback option and check if there is a `Chart` annotation with or without a qualifier, as specified by the application developer.

> **i** Note
>
> `customData` is the shortcut notation for specifying custom data for the control, provided you have added the following to the XML view: `customData="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"`.
>
> For more information on how to use custom data in XML views, see Custom Data - Attaching Data Objects to Controls [page 1042].

Why does the `initialise` event of `SmartTable` not get fired in my scenario?

The `SmartTable` control fires the `initialise` event just **once** after it has completed analyzing the metadata and has initialised its inner state for the first time. Therefore, using `attachInitialise` does not help. However, the `isInitialised` method can be used in such scenarios.

You can also use the following code sample to handle scenarios where you need to trigger some function after this control has been initialized. It should work in scenarios where the event has already been fired:

```
if (oSmartControl.isInitialised()){
  runSomeCodeAfterInit();
} else {
  oSmartControl.attachInitialise(runSomeCodeAfterInit);
}
```

Smart Field

The `sap.ui.comp.smartfield.SmartField` control offers a wrapper for other controls using OData metadata to determine which control has to be instantiated and makes it possible to add input-enabled fields to an application.

The frequently asked questions section below aims at answering some basic questions that you might have when using this control.

For more information about this control, see the API Reference and the samples.

For more information about annotations for this control, see the API Reference.

Overview

The `SmartField` control provides an efficient way to add input-enabled fields to an SAP Fiori application. The control automatically adjusts to the metadata of the underlying OData service, for example, by doing the following:

- Hosting controls for editing and displaying values of OData properties
- Providing value help automatically
- Performing input checks

In addition, the `SmartField` control implements field control and supports message handling.

The `SmartField` control can be used in the following ways:

- As a standalone control, for example, in XML views
- In combination with a `SmartForm` control
- In combination with a `SmartTable` control
  Used in particular as cell editor in editing scenarios.

The `SmartField` control selects a control for displaying and a control for editing the OData property to which they are bound. The main criterion for selecting nested controls is the EDM type of the OData property to which a `SmartField` control is bound.

Details

Binding

The OData property that a `SmartField` control manages is determined by the binding of the value property of the control. Properties of a complex type are not supported.

The `SmartField` control allows for binding of navigation properties.

The entity set to which the bindings are related is either specified in the `entitySet` attribute of the control or derived from the binding context at runtime.

When binding the `SmartField` control against the OData service property of type `Edm.Boolean`, and if the `SmartField` control is in read-only mode, static texts are used for visual representation. In addition, a configuration parameter in the `SmartField` control can define the properties of the static texts of the `CheckBox`, such as `Yes/No` or `True/False`. For the `SmartForm` control, the custom data can be used for this purpose.

Configuration

The `configuration` aggregation of `SmartField` provides the option to overwrite the default behavior of the `SmartField` control.

Using the `controlType` property, you can select the appropriate control for your use case.

The following control types are available:

- Check box
- Date picker
- Drop-down list (combo box)
- Input
- Select drop-down list (`sap.m.Select`)

It depends on the related data types **which** control types are supported, for example:

- If the relevant OData property is of type `Edm.String`, the `SmartField` control can be configured to render a combo box or a select drop-down list.
- If the relevant OData property is of type `Edm.Boolean`, the `SmartField` control can be configured to render a combo box.
- If the relevant OData property is of type `Edm.DateTime`, the `SmartField` control can be configured to render a date picker.

The table below shows which controls are used if you **don't** overwrite the control type.

Table 118: Control Selection

Editing Use Cases		Display Use Cases	
EDM Type	**Control**	**EDM Type**	**Control**
`Edm.Boolean`	`sap.m.CheckBox`	`Edm.Boolean`	`sap.m.CheckBox`
`Edm.Int16`	`sap.m.Input`	`Edm.Int16`	`sap.m.Text`

Editing Use Cases		Display Use Cases	
EDM Type	Control	EDM Type	Control
Edm.Int32		Edm.Int32	
Edm.Int64		Edm.Int64	
Edm.SByte		Edm.SByte	
Edm.Byte		Edm.Byte	
Edm.Single		Edm.Single	
Edm.Float		Edm.Float	
Edm.Double		Edm.Double	
Edm.Decimal		Edm.Decimal	
Edm.String		Edm.String	
Edm.DateTime	sap.m.DateTimePicker	Edm.DateTime	
Edm.DateTimeOffset		Edm.DateTimeOffset	

Using the `displayBehaviour` property, you can define how an ID and a description or Boolean values are represented in read-only mode.

You have the following options:

- `sap.ui.comp.smartfield.DisplayBehaviour.descriptionAndId`: Description and ID are displayed for available values.
- `sap.ui.comp.smartfield.DisplayBehaviour.descriptionOnly`: Only the description of the available values is displayed.
- `sap.ui.comp.smartfield.DisplayBehaviour.idAndDescription`: ID and description are displayed for available values.
- `sap.ui.comp.smartfield.DisplayBehaviour.idOnly`: Shows the ID only.
- `sap.ui.comp.smartfield.DisplayBehaviour.OnOff`: Shows Boolean value as `On/Off`
- `sap.ui.comp.smartfield.DisplayBehaviour.TrueFalse`: Shows Boolean value as `True/False`
- `sap.ui.comp.smartfield.DisplayBehaviour.YesNo`: Shows Boolean value as `Yes/No`

Using the `preventInitialDataFetchInValueHelpDialog` property, you can prevent the query from being fired immediately when the value help dialog is opened.

Field Control

The field control handles the visual representation of `SmartField` controls, such as:

- Whether input is mandatory
- Whether the controls are read-only
- Whether the controls are hidden as defined by the SAP Fiori user interface programming model

The following attributes are available to implement field control:

- `Enabled`
  Toggles from display to edit mode.
- `Visible`
  Hides the `SmartField` control.
- `Mandatory`
  Determines whether input is required.

Consumers of the `SmartField` control can further adapt the runtime behavior by binding these attributes.

The behavior can only be made more restrictive on client side, for example, if an OData property is mandatory, this cannot be overwritten on `SmartField` control level.

FAQ

How can I prevent OData requests from fetching value lists to display the description for a given key in read-only mode?

If the property `fetchValueListReadOnly` is set to `false`, no request is sent to fetch the value list, thus improving performance. However, if you would actually like to to see the description for a key, you have to use the `sap:text` v2 annotation or the `com.sap.vocabularies.Common.v1.Text` v4 annotation to define the path to a property containing the description. This target property has to be defined in the model, which means that you as the application developer have to take care of fetching the data.

I have defined custom formatters for `SmartField` within my application. Why are they ignored?

If you define custom formatters, they will not be taken into consideration. `SmartField` always uses its **own** formatters for the external representation of values. In addition, custom data types, which you define for your application, are supported.

If I use composite binding for the `SmartField` control, the field does not show all elements. How can I use this type of binding?

`SmartField` does not support composite binding. You can only use the standard one-way or two-way binding for the `SmartField` control.

Smart Filter Bar

The `sap.ui.comp.smartfilterbar.SmartFilterBar` control analyzes the `$metadata` document of an OData service and renders a `FilterBar` control that can be used to filter, for example, a table or a chart.

The frequently asked questions section below aims at answering some basic questions that you might have when using this control.

> **i** Note
>
> The code samples in this section reflect examples of possible use cases and might not always be suitable for your purposes. Therefore, we recommend that you do not copy and use them directly.

For more information about this control, see the API Reference and the samples.

For more information about annotations for this control, see the API Reference.

Overview

The `SmartFilterBar` control is a wrapper control that analyzes the metadata and annotations of an OData service. It renders a `FilterBar` control and provides integration with the `VariantManagement` control that is easy to configure.

OData annotations are used to:

- Determine the type of control (for example, whether a field is shown as `MultiInput` control or as `DatePicker`)
- Enable the `Suggest` feature
- Enable value help for filters

Details

In addition to the `$metadata` document, you can also have an additional configuration for `SmartFilterBar` in the XML view. This additional configuration can be either `sap.ui.comp.smartfilterbar.ControlConfiguration` or `sap.ui.comp.smartfilterbar.GroupConfiguration`. Using this additional configuration, you can override certain settings from the OData metadata, such as labels, indexes, or the type of control. You can also add custom fields or custom groups to the filter bar that are not part of the OData `$metadata` document at all.

The `FieldGroup` annotation is used by the `SmartFilterBar` control to create a grouping of the fields. The grouping is shown in the filter dialog. Any label specified in this dialog is used to override the default label of the property. Only `sap:filterable` fields are enabled in the `SmartFilterBar` control by default (default is `true` when `null`).

The `SmartFilterBar` control creates filters lazily. This is done because applications often declare a large number of filters, but then only use a subset of filters in the `SmartFilterBar` control. This way, only **visible** filters are created initially (the properties relevant for `FilterGroupItem` in the `FilterBar` control are `visibleInFilterBar` and `partOfCurrentVariant`). All other filters will be created at a later point in time, once they have been made visible or requested via the APIs.

> ⚠ Caution
>
> Calling `getFilterGroupItems` of the `FilterBar` control always leads to an instantiation of **all** filters that have been declared. If the application needs to react to specific filters only, it is recommended to use `determineFilterItemByName` to obtain a specific filter item instead of calling `getFilterGroupItems` and iterating through the filters.

Multi-value and unrestricted `Date` fields are supported if the annotation `sap:filter-restriction="multi-value"` is set for date properties.

For `MultiInput` filter fields, the `MultiLine` mode is active.

The SmartFilterBar control supports the Edm.Time OData type. The fields bound to OData properties of this type are represented by the sap.m.TimePicker control. The filter panel of the SmartFilterBar control containing the conditions allows filtering for time types using the TimePicker control.

Integration with Other Controls

Support of Selection Variants with SmartVariantManagement

You can use the com.sap.vocabularies.UI.v1.SelectionVariant annotation with your SmartFilterBar control in combination with the considerSelectionVariants property. SelectionVariant is based on OData and metadata-driven.

> **i** Note
>
> You can only use this annotation if you use the SmartVariantManagement control **without** page variants.

considerSelectionVariants is set to false by default. It is only taken into account during the initialization of the SmartFilterBar control.

If the function is active, the provided metadata and annotations are checked for SelectionVariant annotations. Each one of these annotations is then added as a single variant item to the SmartVariantManagement control. The qualifier property determines the internal variant key. The variant items are added once the initialization of SmartVariantManagement has been completed.

Use of Standard Views

If a SelectionVariant annotation entry is provided without a qualifier, it will be treated as the new standard view entry if there is no application-delivered standard view.

> **i** Note
>
> If an application-delivered standard view exists, the default SelectionVariant annotation will be completely ignored.

The new standard view has filter values based on the information provided in SelectionVariant and is enhanced by the _CUSTOM part of the existing standard view.

The filter visibility is also taken over from the existing standard view. However, all filters that are part of SelectionVariant are also treated as if defined in the partOfCurrentVariant property of the FilterBar control. So these filters will at least be visible in the *Filters* dialog.

> **→** Tip
>
> Replacing the standard view greatly influences all other views, since views always show a delta of visible filters in comparison to the standard view.

All further new views that are based on SelectionVariant are treated the same way: The filters in SelectionVariant are handled as if defined in partOfCurrentVariant.

FAQ

`TypeAhead` is not working. When I start typing, no http requests are sent.

Take a look at the `$metadata` document and make sure there are `ValueHelp` annotations for this field. The `Target` attribute must look like this: {`Namespace`}.{`EntityName`}/{`FieldName`}.

Make sure that the namespace in the `Target` attribute is correct.

Example of a `ValueHelp` annotation:

```
<Annotations Target="FAP_VENDOR_LINE_ITEMS_SRV.Item/Creditor" xmlns="http://
docs.oasis-open.org/odata/ns/edm">
    <Annotation Term="com.sap.vocabularies.Common.v1.ValueList">
        <Record>
            <PropertyValue Property="CollectionPath" String="Vendors"/>
            <PropertyValue Property="SearchSupported" Bool="true"/>
            <PropertyValue Property="Parameters">
                <Collection>
                    <Record
Type="com.sap.vocabularies.Common.v1.ValueListParameterInOut">
                        <PropertyValue Property="LocalDataProperty"
PropertyPath="Creditor"/>
                        <PropertyValue Property="ValueListProperty"
String="VendorID"/>
                    </Record>
                </Collection>
            </PropertyValue>
        </Record>
    </Annotation>
    </Annotations>
```

I have a field *Entered on* that's an `Input` field. It should be a `DatePicker`.

Take a look at the `$metadata` document and make sure that the property is of type `Edm.DateTime` and the property is annotated with `sap:display-format="Date"`.

I tried to set default values for a filter field in the control configuration in JavaScript. These default values don't have any effect.

The `ControlConfiguration` and `GroupConfiguration` are intended to be used to add static configuration in an XML view.

There are three properties that can be set dynamically:

- `visible`
- `label`
- `visibleInAdvancedArea`

All other properties and aggregations are not dynamic. This means they have to be set statically in the XML view, and not dynamically by JavaScript. Any changes made in the `ControlConfiguration` or `GroupConfiguration` after the `initialise` event has been fired do not have any effect.

If you have to set values of a filter field dynamically in JavaScript, you can use the `setFilterData` API.

The value help dialog for a filter field contains a table with multiple columns. How can I change the order of these columns?

The order of the columns is specified in the OData `$metadata` document in the `ValueHelp` annotation.

There is one column for each `ValueListParameterInOut` or `ValueListParameterOut` in the related annotation.

The order of the columns is the same as the order of the `InOut/Out` parameters in the `$metadata` document. You can't use configuration in the XML view to change this order. If you want to change the order, you can do it in the OData `$metadata` document.

I have added custom controls to the `SmartFilterBar`. If I save a view and load it again, the custom fields are initial. What do I have to do to enable custom fields for view management?

In general, custom fields cannot be handled automatically by the `SmartFilterBar` control. You have to implement this in the view's controller. The `SmartFilterBar` offers the following events that can be used to enable custom fields for view management:

- `beforeVariantSave` (deprecated)
- `afterVariantLoad`
- `beforeVariantFetch`
  `beforeVariantFetch` replaces the `beforeVariantSave` event since it is triggered at the same points in time. Contrary to `beforeVariantSave`, the `beforeVariantFetch` event is also called whenever the *Filters* dialog is opened. It allows you to restore the state of the custom filters in the *Filters* dialog once the *Restore* button has been pressed.

You can use the `beforeVariantSave` event to update the model of the `SmartFilterBar` with the values from the custom fields. Every value within the model is stored as a view. The values of custom fields should be stored under the property `_CUSTOM`, for example, `oSmartFilter.setFilterData({_ CUSTOM : {field1:"abc", field2:"123"}});`.

You can use the event `afterVariantLoad` to get the values from the model and use them to update the custom filter fields, for example:

```
oData = oSmartFilter.getFilterData();
var oCustomFieldData = oData["_CUSTOM"];
oCustomField1.setValue(oCustomFieldData.field1);
```

If both events are handled this way, custom fields are enabled for view management.

How can I set initial or default data in the `SmartFilterBar` control?

Static data can be set in the control using `ControlConfiguration` in the `view.xml`:

```
<smartFilterBar:SmartFilterBar id="smartFilterBar" ...>
…
<smartFilterBar:controlConfiguration>
              <smartFilterBar:ControlConfiguration key="CompanyCode"
visible="true" index="3"…>
                    <smartFilterBar:defaultFilterValues>
                          <smartFilterBar:SelectOption low="0001">
                          </smartFilterBar:SelectOption>
                    </smartFilterBar:defaultFilterValues>
              </smartFilterBar:ControlConfiguration>
```

How can I set dynamic data as initial or default data in the `SmartFilterBar` control, for example, for navigation parameters?

Dynamic data can be set as initial or default data in the control by registering to the `initialise` event and setting JSON/JSONstring using the `setFilterData` API in your `controller.js`.

```
…
onInitSmartFilter: function(oEvent) { //Assuming that this is the eventhandler
registered for the "initialise" event of the SmartFilterBar control in your
view.xml
              var    oSmartFilter = oView.byId("smartFilterBar");
              var oTodaysDate = new Date();
              //Sample Data
              var oJSONData = {
                           Company: {
                                  items: [ //MultiInput fields with filter-
restriction="multi-value" (Ex: shown as Tokens based on control type)
                                         {
                                            key:"0001",
                                            text:"SAP SE" //Display text on the
token --> not used for filtering!
                                          },
                                          {
                                             key:"0002",
                                             text:"SAP XYZ"
                                         }
                                          ]
                           },
                           SomeDate: { //DateRange field with filter-
restriction="interval"
                                  low: oTodaysDate, //Date fields require
JavaScript Date objects!
                                  high: oTodaysDate
                           },
                           YearInterval: {
                                  low: "2000-2014" //simple input field with
filter-restriction="interval" --> text separated by a single "-"
                           },
                           Ledger:"0L" //Single-value field --> Plain input
              };

              oSmartFilter.setFilterData(oJSONData); //Data will be updated with
existing data in the SmartFilter
       },
…
```

> **i Note**
>
> You can use the `setFilterData` API to set data in the `SmartFilterBar` control.

How does the `SmartFilterBar` determine if a filter has a value assigned to it?

The `SmartFilterBar` control handles the checks whether values are set for the OData-service-based filters, but has only a limited capability to do the same for custom fields. For checks like this, the custom field provider has to provide a Boolean value `(true/false)` as an indicator whether a value for the custom field exists via the custom data extension `hasValue`. If the custom data does not exist, the `SmartFilterBar` control analyzes if the custom control has either the method `getValue` or `getSelectedKey` and by using those tries to determine whether the value exists.

> **i Note**
>
> The method-based check is not very reliable, since, for example, `MultiComboBox` provides both methods mentioned, but the actual value is accessed via `getSelectedKeys`. It is strongly recommended to use the

custom data extension for such scenarios. The `SmartFilterBar` control can only react to an `onChange` event. Therefore, the application has to set the `hasValue` custom data while handling the `onChange` event.

I would like to use the `SmartFilterBar` control in an analytical scenario, for example, use it in combination with `SmartChart` or an analytical table. What do I have to bear in mind?

Analytical binding does not support filtering using `navigationProperties`. However, `SmartFilterBar` creates a group for each filterable property based on the navigation property of the bound `entitySet` and assigns filters to it. If you would like to prevent these filters from being created, since they cannot be used in an analytical scenario, set the property `useProvidedNavigationProperties` to `true` while leaving the provided list of navigation properties empty (property `navigationProperties` is not defined or has an empty value).

If you would like the `SmartFilterBar` control to create filters only for some of the navigation properties, set `useProvidedNavigationProperties` to `true` and list the navigation properties for the filters you require in `navigationProperties`, for example, `navigationProperties="to_CompanyCode"` takes only this specific navigation property into account.

Why does the `initialise` event of `SmartTable` not get fired in my scenario?

The `SmartTable` control fires the `initialise` event just **once** after it has completed analyzing the metadata and has initialised its inner state for the first time. Therefore, using `attachInitialise` does not help. However, the `isInitialised` method can be used in such scenarios.

You can also use the following code sample to handle scenarios where you need to trigger some function after this control has been initialized. It should work in scenarios where the event has already been fired:

```
if (oSmartControl.isInitialised()){
  runSomeCodeAfterInit();
} else {
  oSmartControl.attachInitialise(runSomeCodeAfterInit);
}
```

How does the `SmartFilterBar` control determine if a filter has a value?

The `SmartFilterBar` control handles the checks whether any values are set for the OData-service-based filters, but has only a limited capability to do the same for custom fields. For checks like this, the custom field provider has to provide a Boolean value (`true/false`) as an indicator whether a value for the custom field exists using the custom data extension `hasValue`. If there is no custom data, the `SmartFilterBar` control analyzes if the custom control has either the method `getValue` or `getSelectedKey` and, by using those, tries to determine whether any value exists.

Once `hasValue` has been set, the custom extension calls the `fireFilterChange` method of the `FilterBar` control (no parameters required for this method) to indicate that the count of assigned values has to be recalculated.

Related Information

Filter Bar [page 2401]

Smart Variant Management [page 2457]

Smart Form

The `sap.ui.comp.smartform.SmartForm` control makes it possible to render a form. Depending on user authorizations, the form enables users, for example, to switch from display to edit mode, add and group fields, rename field labels, and implement a user input check.

> **i Note**
>
> The code samples in this section reflect examples of possible use cases and might not always be suitable for your purposes. Therefore, we recommend that you do not copy and use them directly.

For more information about this control, see the API Reference and the samples.

Overview

The `SmartForm` control displays form content. If used in combination with the `SmartField` control and OData metadata annotations along with additional configuration, the control allows you to create a form with minimal effort.

The `SmartForm` control supports the following features:

- Adaptation settings
  A key user can adapt the form for all users in one client by doing the following:
  - Adding and hiding fields
  - Adding and hiding groups
  - Changing the order of fields and groups
  - Renaming field labels
- *Display/Edit* button
  This optional button allows the user to toggle from display to edit mode.

  > **i Note**
  >
  > Fields of type `SmartField` are automatically displayed with the appropriate control in the required mode, for example, texts on the user interface in display mode and user input in edit mode. If controls other than the `SmartField` control are used, the application in question has to handle the switch between display and edit mode.

- Field labels
  For fields of type `SmartField`, the `SmartForm` control automatically creates a label based on the OData metadata annotations.
- *Check* button
  This optional button allows the user to check the current user input.

  > **i Note**
  >
  > For fields of type `SmartField`, values will be checked based on the OData metadata annotations. Depending on which theme is defined, the fields with errors will be circled in red. When the user clicks on one of these fields, the relevant error message is displayed.

Details

Groups

A `SmartForm` control consists of groups (`sap.ui.comp.smartform.Group`) and group elements (`sap.ui.comp.smartform.GroupElement`).

A group element is a collection of controls that are displayed along with a label. Typically, a group element consists of exactly one control and the respective label. Multiple group elements can be grouped together. This group then also has a label.

The `SmartForm` control aggregates groups, and a group aggregates group elements. The group elements themselves aggregate elements of type `sap.ui.core.Control`.

The following example shows the `SmartForm` control and its entities:

```
<smartForm:SmartForm id="MainForm" title="General Data"
                entityType="Header, Tax" editTogglable="true" expandable="true"
                expanded="true" ignoredFields="AccountingDocumentCategory"
                checkButton="true">
    <smartForm:customData>
        <core:CustomData key="suppressUnit" value="false" />
        <core:CustomData key="dateFormatSettings" value='\{"style":"short"\}' />
        <core:CustomData key="defaultDropDownDisplayBehaviour"
value='descriptionAndId' />
    </smartForm:customData>
    <smartForm:customToolbar>
        <Toolbar height="3rem">
            <Text text="Custom Toolbar with a header text" />
            <ToolbarSpacer />
            <Button icon="sap-icon://settings" />
            <Button icon="sap-icon://drop-down-list" />
        </Toolbar>
    </smartForm:customToolbar>
    <smartForm:Group id="GeneralLedgerDocument" label="General Ledger Document"
                    expandable="true">
        <smartForm:layout>
            <layout:GridData span="L4 M4 S4" />
        </smartForm:layout>
        <smartForm:GroupElement id="GeneralLedgerDocument.CompanyCode">
            <smartField:SmartField value="{CompanyCode}"
                            enabled="true" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="GeneralLedgerDocument.AccountingDocument">
            <smartField:SmartField value="{AccountingDocument}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="GeneralLedgerDocument.FiscalYear">
            <smartField:SmartField value="{FiscalYear}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement
                        id="GeneralLedgerDocument.AccountingDocumentType">
            <smartField:SmartField value="{AccountingDocumentType}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement
                        id="GeneralLedgerDocument.AccountingDocumentHeaderText">
            <smartField:SmartField value="{AccountingDocumentHeaderText}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement
                        id="GeneralLedgerDocument.AccountingDocumentTypeName">
            <smartField:SmartField value="{AccountingDocumentTypeName}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="GeneralLedgerDocument.AmountInCoCodeCrcy">
            <smartField:SmartField value="{AmountInCoCodeCrcy}" />
        </smartForm:GroupElement>
```

```
        <smartForm:GroupElement id="GeneralLedgerDocument.CoCodeCurrency">
            <smartField:SmartField value="{CoCodeCurrency}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="GeneralLedgerDocument.LedgerGroup">
            <smartField:SmartField value="{LedgerGroup}" />
        </smartForm:GroupElement>
    </smartForm:Group>
    <smartForm:Group label="Dates" id="Dates">
        <smartForm:layout>
            <layout:GridData span="L3 M3 S3" />
        </smartForm:layout>
        <smartForm:GroupElement id="Dates.DocumentDate">
            <smartField:SmartField value="{DocumentDate}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="Dates.PostingDate">
            <smartField:SmartField value="{PostingDate}" />
        </smartForm:GroupElement>
    </smartForm:Group>
    <smartForm:Group label="Reversal" id="Reversal">
        <smartForm:layout>
            <layout:GridData span="L3 M3 S3" />
        </smartForm:layout>
        <smartForm:GroupElement id="Reversal.ReversalDocument">
            <smartField:SmartField value="{ReverseDocument}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="Reversal.ReversalReasonName">
            <smartField:SmartField value="{ReversalReasonName}" />
        </smartForm:GroupElement>
    </smartForm:Group>
    <smartForm:Group label="Administrative Data" id="AdministrativeData">
        <smartForm:layout>
            <layout:GridData span="L3 M3 S3" linebreak="true" />
        </smartForm:layout>
        <smartForm:GroupElement id="AdministrativeData.CreatedByUser">
            <smartField:SmartField value="{CreatedByUser}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="AdministrativeData.CreatedByUserName">
            <smartField:SmartField value="{CreatedByUserName}" />
        </smartForm:GroupElement>
        <smartForm:GroupElement id="AdministrativeData.CreationDate">
            <smartField:SmartField value="{CreationDate}" />
        </smartForm:GroupElement>
    </smartForm:Group>
</smartForm:SmartForm>
```

Layout

The `SmartForm` control uses a `ResponsiveGridLayout` that can be adjusted. The following properties are exposed in the aggregation layout:

- `labelSpanXL`, `labelSpanL`, `labelSpanM`, `labelSpanS`
- `emptySpanXL`, `emptySpanL` `emptySpanM`, `emptySpanS`
- `columnsXL`, `columnsL`, `columnsM`
- `breakpointXL`, `breakpointL`, `breakpointM`
- `gridDataSpan`

To display the fields next to each other with a label on top, you can use the `gridDataSpan` property of the `layout` element in combination with the `useHorizontalLayout` property.

The form will be embedded in an `sap.m.Panel` if the `expandable` property is set. Using this property, the form can also be collapsed and expanded.

Group Layout

The `layoutData` aggregation of a group can be used to define a `GridData` layout. With this layout, the default span of a single group can be changed to allocate the number of columns available to each group. The number of columns allocated to each group depends on how many groups there are. Moreover, a line break can be inserted to display the group in a new line.

Toolbar

The `SmartForm` control uses a toolbar for displaying the title of the form and the following buttons (if configured):

- *Display/Edit* (`editTogglable` property)
- *Check* (`checkButton` property)

Alternatively, the custom toolbar can be used (`customToolbar` aggregation). The `SmartForm` control will then replace the standard toolbar with the custom toolbar and add the title and the buttons if requested.

Key-User-Specific Adaptation

To enable key user adaptation settings, the following prerequisites have to be fulfilled:

- Use of stable IDs for every `Group`, `GroupElement`, and field in the XML view
  The adaptation settings use control IDs to identify the entities that can be modified.
- Use of the `entityType` property
  The property determines the entity type of the OData service defining the fields that can be added to the form.
  Fields that must not be available on the adaptation dialog can be listed in the `ignoredFields` property: Separate the property names by commas without using spaces.

Smart Link

The `sap.ui.comp.navpopover.SmartLink` control provides a popover with navigation links to related applications, for example, more detailed information about customer data.

For more information about this control, see the API Reference and the samples.

For more information about annotations for this control, see the API Reference.

Overview

The `SmartLink` control provides further navigation information for a certain entity and offers the following options for navigation:

- Main navigation target

- Additional content, which can be customized
- Links to related apps, for example

Navigation links can be personalized by selecting *Define Links* on the popover where users can select from a list which cross-application links they want to see. They can also restore the links they selected to the previous state.

Details

Within a `SmartTable` or `SmartForm` control, the `SmartLink` control is created automatically if the `SemanticObject` annotation has been defined in the metadata of the OData service used. Navigation targets are determined by the `SmartLink` control and its classes using the `CrossApplicationNavigation` service of the unified shell. For more information about this service, see CrossApplicationNavigation in the Demo Kit.

The events of the `SmartLink` control allow the consuming application to do the following in the popover:

- Add, edit, or remove
  - Texts
  - Parameters
  - Navigation targets
- Add an additional customized area

The `SmartLink` control can be created in an XML view or in the coding. However, it is recommended to use the `SemanticObject` definition of the OData metadata via the related annotation. This ensures that the `SmartLink` control is instantiated in a correct manner.

Semantic Object Controller

Within a `SmartTable` or `SmartForm` control, the `SemanticObjectController` class is used as a central instance that is exposing the automatically generated events of the `SmartLink` control.

All events of the `SmartLink` control and the features of the navigation popover connected to it are registered with the `SemanticObjectController` class and are thus available for use by the consuming application from one single source.

In addition, the `SemanticObjectController` class enables further features that influence the behavior of the registered `SmartLink` controls within the `SmartTable` or `SmartForm` controls.

The `SemanticObjectController` class can be set up in the XML view as well as in the coding of the `SmartTable` and the `SmartForm` controls. All `SmartLink` controls in a `SmartTable` or `SmartForm` control will automatically get registered with the `SemanticObjectController` class provided.

Smart Micro Chart

The `sap.ui.comp.smartmicrochart.SmartMicroChart` control is used to create different micro charts based on OData metadata.

For more information about this control, see the API Reference and the sample in the Demo Kit.

Overview

The `SmartMicroChart` control analyzes the metadata document of an OData service and renders a `MicroChart` control for a specified entitySet. If no annotations or metadata are provided, the chart will not be rendered.

According to the defined value in the `chartType` property of the *UI.Chart* annotation the corresponding `MicroChart` control is rendered. Currently, the following chart types are supported:

- *Area* (`sap.ui.comp.smartmicrochart.SmartAreaMicroChart` will be created) based on the `UI.ChartType` *Area* and *Line*.
- *Bullet* (`sap.ui.comp.smartmicrochart.SmartBulletMicroChart` will be created) based on `UI.ChartType` *Bullet*.
- *Radial* (`sap.ui.comp.smartmicrochart.SmartRadialMicroChart` will be created) based on the `UI.ChartType` *Donut*.

> **i** Note
>
> In case of a different *UI.ChartType* is provided than the mentioned above, the control does not render anything. The developer gets informed with a log statement.

The metadata should be annotated with the *UI.Chart* and the *UI.DataPoint* terms. Both terms need to annotate one and the same entityType (see also the `entitySet` property of the `SmartMicroChart` control in the API Reference).

Details

Property handling

All supported chart types of SmartMicroCharts need to handle the properties as follows:

- *DataPoint* property of the *Chart* annotation should point to the *DataPoint* annotation. Example:

  ```
  <Property="DataPoint" AnnotationPath="@UI.DataPoint#BulletChartDataPoint">
  ```

  In this example, the `BulletChartDataPoint` is the qualifier of the *DataPoint* annotation.
- *Measures* and *Measure* properties
  Measures property of the *Chart* annotation and *Measure* property of the *MeasureAttributes* property of the *Chart* annotation should point at the same property in the `entityType` (Revenue in the sample below). Example:

  ```
  <EntityType Name="ProductType" sap:label="Product Sales Data" sap:content-
  version="1">
                  <Key>
                      <PropertyRef Name="Product" />
                  </Key>
  ```

```
                <Property Name="Product" Type="Edm.String" />
                <Property Name="Revenue" Type="Edm.Decimal"/>
                <Property Name="TargetRevenue" Type="Edm.Decimal"/>
                <Property Name="ForecastRevenue" Type="Edm.Decimal"/>
                <Property Name="Criticality" Type="Edm.String"/>
                <Property Name="Currency" Type="Edm.String"/>
                <Property Name="ChartTitle" Type="Edm.String"/>
                <Property Name="ChartDescription" Type="Edm.String"/>
            </EntityType>
```

```
<Annotations Target="BmcNamespace.ProductType" xmlns="http://docs.oasis-
open.org/odata/ns/edm">
                <Annotation Term="UI.Chart">
                    <Record>
                        <PropertyValue EnumMember="UI.ChartType/Bullet"
                            Property="ChartType" />
                        <PropertyValue Property="Title" Path="ChartTitle" />
                        <PropertyValue Property="Description"
Path="ChartDescription" />
                        <PropertyValue Property="Measures">
                            <Collection>
                                <PropertyPath>Revenue</PropertyPath>
                            </Collection>
                        </PropertyValue>
                        <PropertyValue Property="MeasureAttributes">
                            <Collection>
                                <Record Type="UI.ChartMeasureAttributeType">
                                    <PropertyValue Property="Measure"
PropertyPath="Revenue" />
                                    <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                                    <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#BulletChartDataPoint" />
                                </Record>
                            </Collection>
                        </PropertyValue>
                    </Record>
                </Annotation>
                <Annotation Term="UI.DataPoint"
Qualifier="BulletChartDataPoint">
                    <Record>
                        <PropertyValue String="Product" Property="Title" />
                        <PropertyValue Path="Revenue" Property="Value" />
                        <PropertyValue Path="TargetRevenue"
Property="TargetValue" />
                        <PropertyValue Path="ForecastRevenue"
Property="ForecastValue" />
                        <PropertyValue Decimal="0" Property="MinimumValue" />
                        <PropertyValue Decimal="200"
Property="MaximumValue" />
                        <PropertyValue Path="Criticality"
Property="Criticality" />
                    </Record>
                </Annotation>
            </Annotations>
```

i Note

The same `entityType` property should be used as a path for the `Value` property of the *DataPoint* annotation.

Qualifiers

With the qualifier for the *UI.Chart* annotation term, the SmartMicroChart control can support multiple *UI.Chart* annotations for an OData service. Depending on the qualifiers, you can separate these multiple annotations and handle the different OData Annotations in this control.

You can provide the qualifier name through the `CustomData`aggregation of the corresponding SmartMicroChart control. A qualifier for *UI.Chart* annotation with Qualifier can look like this:

```
<Annotation Term="UI.Chart" Qualifier="BulletChartQualifier">
```

The "BulletChartQualifier" string is the qualifier and can be any kind of string. The SmartMicroChart control needs this string to find the corresponding annotation. This is handled by providing the qualifier string as a custom data on the control instance.

```
        <SmartMicroChart id="smartChartBullet" entitySet="Products"
enableAutoBinding="true" chartBindingPath="/Products('PC')" isResponsive="true">
            <customData>
            <core:CustomData key="chartQualifier" value="BulletChartQualifier" />
            </customData>
        </SmartMicroChart>
```

The custom data key is “chartQualifier”. There are three different options supported how custom data can be provided:

- as XML declaration
- by calling data function
- by calling addCustomData function

Smart Bullet Micro Chart

The `sap.ui.comp.smartmicrochart.SmartBulletMicroChart` control creates an `sap.suite.ui.microchart.BulletMicroChart` control based on OData metadata.

For more information about this control, see the API Reference and the sample in the Demo Kit.

By using the `chartType` property *Bullet* of the *UI.Chart* annotation the corresponding `SmartBulletMicroChart` control is rendered. The `entitySet` property of the control must be specified. This attribute is used to fetch metadata and annotation information from the given default OData model. Based on this, the Bullet Micro Chart is created.

> **i** Note
>
> The control can also be used directly (without creating a SmartMicroChart).

Binding

The `SmartBulletMicroChart` control should be bound to **one** entity, and not a collection of entities (entitySet). It supports both `enableAutoBinding` = **`false`** (no binding is done inside the control) and `enableAutoBinding` = **`true`** :

- If the `chartBindingPath` is provided, the control will be bound relatively to it. For example, the `chartBindingPath` can be a navigation property.
- If `chartBindingPath` is not provided, the control will be bound absolutely to the entitySet.

For information on how the control provides *Title*, *Description* and *UnitOfMeasure* values retrieved from the annotations, please see the *Title, Description and UnitOfMeasures* section in the Smart Area Micro Chart [page 2432] documentation.

> **i Note**
>
> If `enableAutoBinding` = **`true`** and `chartBindingPath` is set then the `bindingContext` for the *Title*, *Description* and *UnitOfMeasure* is set to the value as well.

Color of the Chart bar (Criticality)

The color of the chart bars can be controlled by the *Criticality* property either directly or by calculation.

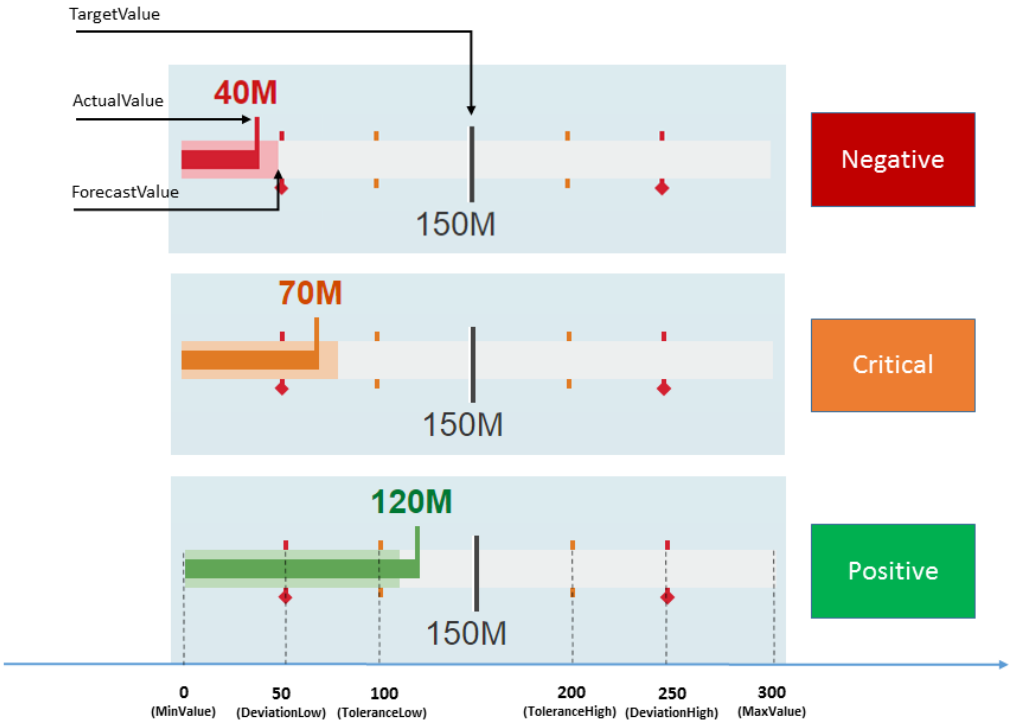
- Criticality can be set directly:

```
<EntityType Name="ProductType" >
…
<Property Name="Criticality" Type="Edm.String" />
</EntityType>
<Annotation Term="UI.DataPoint" Qualifier="BulletChartDataPoint">
<PropertyValue Path="Criticality" Property="Criticality" />
…
</Annotation>
```

- Criticality can be calculated by the `SmartBulletMicroChart` control based on the provided thresholds and the `ImprovementDirection` property of the *DataPoint* annotation.

  > **i Note**
  >
  > The target and forecast values are not taken into account for the color calculation.

  ImprovementDirection Property
  The following values for the `ImprovementDirection` property are supported:
  - **Target**

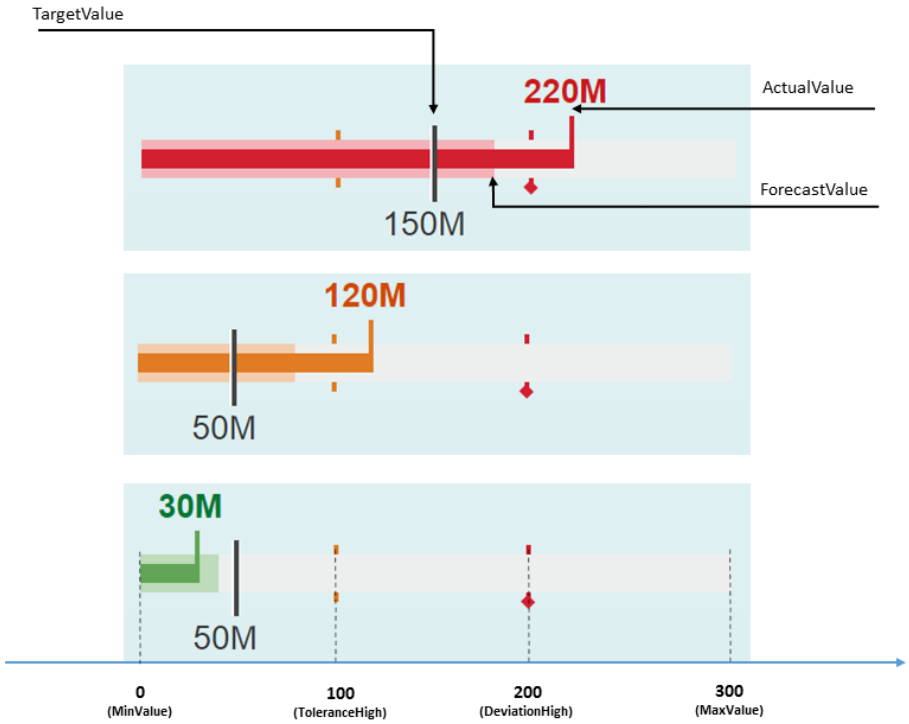


Code Example

```
<EntityType Name="ProductType"  >
         <Key>
                  <PropertyRef Name="Product" />
         </Key>
                  <Property Name="Product" Type="Edm.String"
Nullable="false"/>
                  <Property Name="Revenue" Type="Edm.Decimal"/>
                  <Property Name="TargetRevenue" Type="Edm.Decimal"/>
                           <Property Name="ForecastRevenue"
Type="Edm.Decimal"/>
                           <Property Name="ToleranceRangeLow"
Type="Edm.Decimal"/>
                           <Property Name="ToleranceRangeHigh"
Type="Edm.Decimal"/>
                           <Property Name="DeviationRangeLow"
Type="Edm.Decimal"/>
                           <Property Name="DeviationRangeHigh"
Type="Edm.Decimal"/>
</EntityType>
.....
 <Annotations Target="BmcNamespace.ProductType" xmlns="http://docs.oasis-
open.org/odata/ns/edm">
                  <Annotation Term="UI.Chart">
                     <Record>
                        <PropertyValue EnumMember="UI.ChartType/Bullet"
Property="ChartType" />
                        <PropertyValue Property="Measures">
                                 <Collection>
                                  <PropertyPath>Revenue</PropertyPath>
                                 </Collection>
                        </PropertyValue>
```

```
                        <PropertyValue Property="MeasureAttributes">
                                <Collection>
                         <Record Type="UI.ChartMeasureAttributeType">
                             <PropertyValue Property="Measure"
PropertyPath="Revenue" />
                              <PropertyValue Property="Role"
EnumMember="UI.ChartMeasureRoleType/Axis1" />
                           <PropertyValue Property="DataPoint"
AnnotationPath="@UI.DataPoint#BulletChartDataPoint" />
                         </Record>
                                </Collection>
                        </PropertyValue>
                    </Record>
                  </Annotation>
                  <Annotation Term="UI.DataPoint"
Qualifier="BulletChartDataPoint">
                    <Record>
                      <PropertyValue String="Product" Property="Title" />
                      <PropertyValue Path="Revenue" Property="Value" />
                      <PropertyValue Path="TargetRevenue"
Property="TargetValue" />
                      <PropertyValue Path="ForecastRevenue"
Property="ForecastValue" />
                      <PropertyValue Decimal="100"
Property="MinimumValue" />
                      <PropertyValue Decimal="300"
Property="MaximumValue" />
                      <PropertyValue Property="CriticalityCalculation">
                       <Record>
                          <PropertyValue Property="ImprovementDirection"
EnumMember="UI.ImprovementDirectionType/Target"/>
                          <PropertyValue Path="ToleranceRangeLow"
Property="ToleranceRangeLowValue" />
                          <PropertyValue Path="ToleranceRangeHigh"
Property="ToleranceRangeHighValue" />
                          <PropertyValue Path="DeviationRangeLow"
Property="DeviationRangeLowValue" />
                          <PropertyValue Path="DeviationRangeHigh"
Property="DeviationRangeHighValue" />
                        </Record>
                     </PropertyValue>
                    </Record>
              </Annotation>
 </Annotations>
```

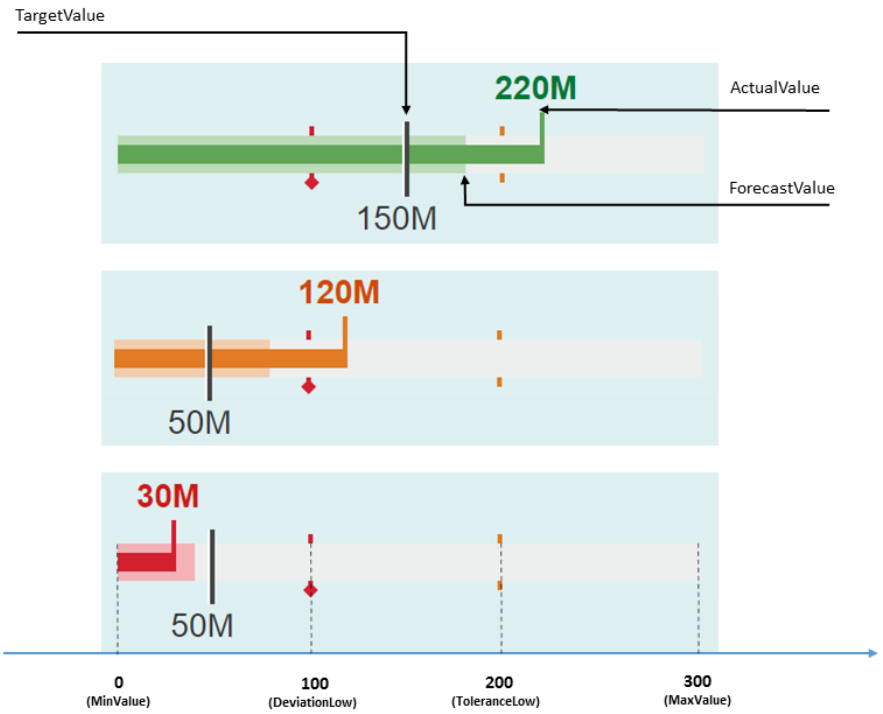
- **Minimize**



Related to the code example for *target*, the code line for the `ImprovementDirection` *minimize* would be as follows:

```
Property="ImprovementDirection" EnumMember="UI.ImprovementDirectionType/
Minimize"
```

- **Maximize**



Related to the code example for *target*, the code line for the `ImprovementDirection` *maximize* would be as follows:

```
Property="ImprovementDirection" EnumMember="UI.ImprovementDirectionType/
Maximize"
```

Smart Area Micro Chart

The `sap.ui.comp.smartmicrochart.SmartAreaMicroChart` control creates an `sap.suite.ui.microchart.AreaMicroChart` control based on OData metadata.

For more information about this control, see the API Reference and the sample in the Demo Kit.

By using the `chartType` property *Area* of the *UI.Chart* annotation the corresponding `SmartAreaMicroChart` control is rendered. The `entitySet` attribute must be specified to use the control. This control property is used to fetch the corresponding annotations. Based on this, the `AreaMicroChart` will be rendered; it can also be used to fetch the actual chart data.

> **i Note**
>
> The control can also be used directly (without creating a `SmartMicroChart` control).

Details

Binding

The `SmartAreaMicroChart` control supports only `enableAutoBinding` = **`true`**. This means it should be bound to only **one** entitySet, and should get a navigation property as a chartBindingPath to an entitySet (for the relative binding) or else the `entitySet` property will be used for the absolute binding.

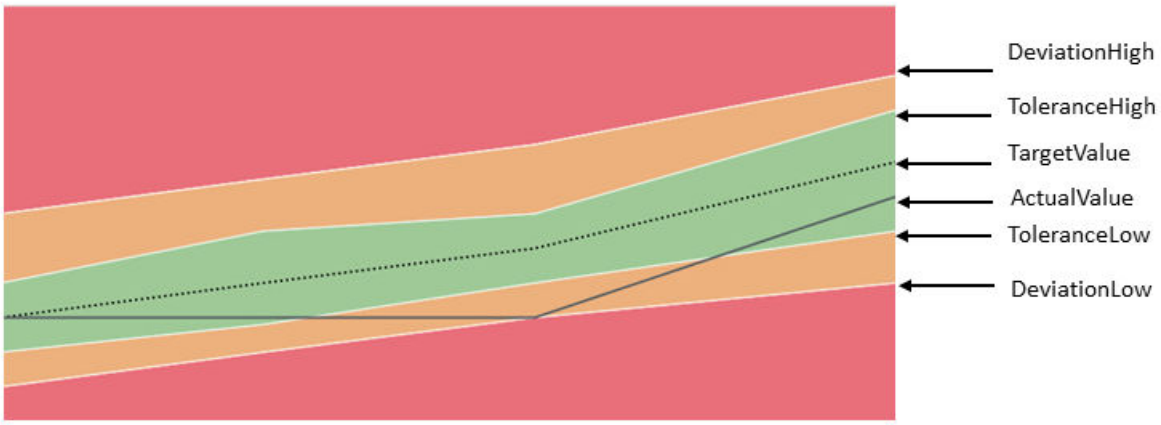
Colors of the Chart (Criticality)

The color of the chart is defined due to the thresholds by using the `CriticalityCalculation` property of the *UI.DataPoint* annotation and the `ImprovementDirection` property.

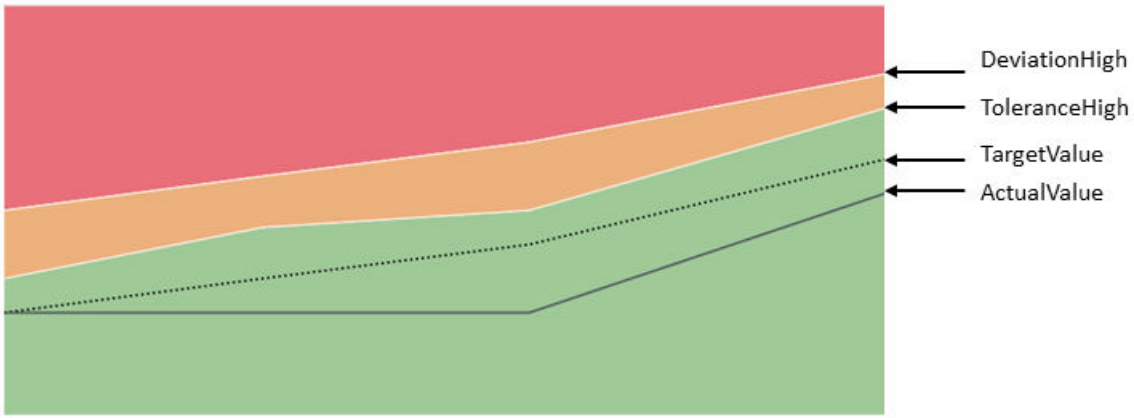
> **i** Note
>
> Setting the color directly via the `Criticality` property of the *UI.DataPoint* annotation is not supported by SmartAreaMicroChart.

The following directions are supported:

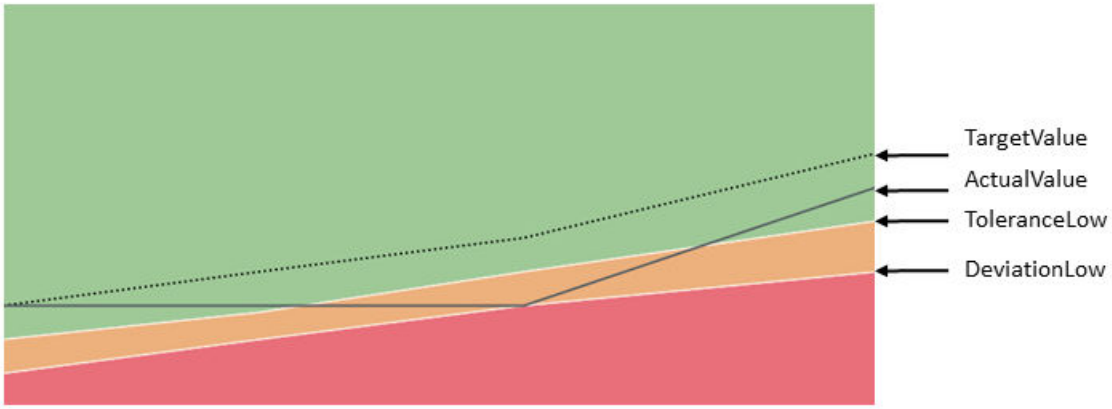
Target-oriented:



Minimize-oriented:



Maximize-oriented:



For more information about the colors in the chart, see the sample in the Demo Kit.

Labels, formatting and label colors of the chart

The control provides labels that are displayed at the top and bottom of the chart. You can control the labels themselves, as well as the color of the labels, by using annotations:

- Labels of the Chart [page 2436]
- Formatting [page 2437]
- Label Colors of the Chart [page 2438] (criticality)

Title, Description and UnitOfMeasure

The control provides the *Title*, *Description* or *UnitOfMeasure* values retrieved from the annotations. If needed, the application can create a corresponding `chartTitle`, `chartDescription` or `unitOfMeasure` (of type `sap.m.Label`) association. Then, the information will be read from the annotation document and it will be set as the `text` property of the `sap.m.Label`.

- **Title and Description**
  For the *Title* and *Description* properties of the *Chart* annotation both `String` and `Path` are supported:
  - `<PropertyValue Property="Title" Path="ChartTitle" />`
  - `<PropertyValue Property="Title" String="ChartTitle" />`

  > **i** Note
  >
  > Only the *Title* and *Description* properties of the *Chart* annotation are taken into account (not these of the *DataPoint* annotation).

  For the `SmartAreaMicroChart` control, the *Title*, *Description* and *UnitOfMeasure* have the same binding context as the chart itself (either `chartBindingPath` or `entitySet`).
- **UnitOfMeasure**
  For more information about the supported annotation terms *ISOCurrency* and *Unit*, see http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/Org.OData.Measures.V1.xml .
  The control takes into account only the *measure* annotation for the `entityType` property that is addressed by the `Value` property of the *DataPoint* annotation. In the following example, the `Value` property of the *DataPoint* points to the `Price` property in the `EntityType` (`Path="`**`Price`**`"`). This means that the control (`Path="`**`Currency`**`"`) uses the *measure* annotation with `Target="AmcNamespace.StockPrice/Price"`:

```
<EntityType Name="StockPrice">
....
<Property Name="Currency" Type="Edm.String" />
</EntityType>
<Annotation Term="UI.DataPoint" Qualifier="AreaChartDataPoint">
<PropertyValue Property="Value" Path="Price" />
</Annotation>
```

```
<Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm"
Target="AmcNamespace.StockPrice/Price" >
    <Annotation Term="MEASURES.ISOCurrency" Path="Currency">
    </Annotation>
</Annotations>
```

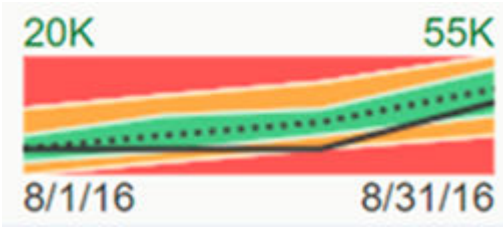
Labels of the Chart

You can display a maximum of four labels.

The relationships between these four labels and their annotations are as follows:

Table 119: Relationship of labels and their annotations

Chart point (sap.suite.ui.microchart.AreaMicroCharItem)	Coordinate	Label position	Annotation Term	Property
First point	x	Bottom left	UI.Chart	Dimensions
	y	Top left	UI.Chart	Measures
Last point	x	Bottom right	UI.Chart	Dimensions
	y	Top right	UI.Chart	Measures



The values of the x and y coordinates of the first chart point are retrieved from the first data entry in the bound model. The formatted x-value is displayed at the bottom left of the chart. It corresponds with the `dimensions` property of the UI.Chart annotation term. Depending on the data type of the property in the entity type, an appropriate formatter is chosen so that you can format the value as needed.

The formatted y-coordinate value is displayed at the top left of the chart and is used to show the actual value. It is bound to the `measures` property of the UI.Chart annotation term. It is always formatted as a numeric value, that is, the value 20,000 is formatted to “20K”. The last data point is treated the same way: the top right label corresponds to the last data entry’s `measures` property and the bottom right label corresponds to its `dimensions` property.

Formatting

The top labels are always formatted by the `sap.ui.core.format.NumberFormat`.

The bottom labels are formatted by the number or date formatter depending on the `dimensions` property's type and its annotations:

Table 120: Formatted bottom labels in different annotation cases

Type of `dimension` property	Annotations	Raw value	X-coordinate value	Formatted value for bottom label
Any type	`sap:text="DateLabel"` (or has annotation `Common.Text` whose Path="DateLabel")	"any text" (raw value of `DateLabel` property)	Depends on property type (use index value if neither date nor number type)	'any text'
Edm.String	`sap:semantics="year"` (or has annotation `IsCalendarYear`)	2016	Timestamp	1/1/16
Edm.String	`sap:semantics="yearmonth"` (or has annotation `IsCalendarYearMonth`)	201612	Timestamp	12/1/16
Edm.String	`sap:semantics="yearmonthday"` (or has annotation `IsCalendarDate`)	20161225	Timestamp	12/25/16
	---	'any text'	Indices	
Edm.DateTime		/Date(1472629368000)/	Timestamp	8/31/16
Edm.Int32 or other number types		20000	20000	20K

- If the `dimensions` property has a V2 annotation `sap:text` (or V4 annotation `Common.Text`) pointing to another property, that property's value will be used to display the bottom label, no matter what the primary property's data type is. The x-coordinate's value depends on the primary property's data type. If its type is `DateTime`, the date is converted to a timestamp; if it has a numeric type, the value is used directly. In other cases, each point's index from within the data list is used, causing an evenly distribution of points on the x-axis.
- If the type of the `dimensions` property is 'string' and it has a V2 annotation `sap:semantics` (or the V4 annotation `IsCalendarYear`, or similar), the raw value is formatted to a shortened date string based on the pattern provided by `sap:semantics` (or a corresponding pattern of `IsCalendarYear`) and the value

of the x-coordinate is set to the formatted date's timestamp. The bottom label displays the formatted short date.

- If the type of the `dimensions` property is `DateTime` or another numeric type without any of the annotations mentioned above, the raw value is formatted by the number formatter or date formatter depending on the data type. Each point's x-coordinate value is set to the date's timestamp representation if its type is `DateTime`, and to a numeric value if its type is numeric.

Label Colors of the Chart

You use the Criticality Calculation feature to control the color of the labels at the top.

You cannot set the label color directly. The color of the labels at the bottom is unchangeable and will always be displayed with `ValueColor.Neutral`. The color of the labels at the top is calculated using the thresholds that are common for every Criticality Calculation across the Smart Micro Charts. For example:

Table 121: Thresholds

Property	Type	Value
DeviationLowValue	Negative	10
ToleranceLowValue	Critical	45
ToleranceHighValue	Critical	55
DeviationHighValue	Negative	80

Table 122: Resulting colors of different sample values in different Improvement Directions

Value	Maximize direction	Minimize direction	Target direction
5	Error	Good	Error
15	Critical	Good	Critical
50	Good	Good	Good
70	Good	Critical	Critical
90	Good	Error	Error

Smart Radial Micro Chart

The `sap.ui.comp.smartmicrochart.SmartRadialMicroChart` control creates an `sap.suite.ui.microchart.RadialMicroChart` control based on OData metadata.

For more information about this control, see the API Reference and the sample in the Demo Kit.

By using the `chartType` property *Donut* of the *UI.Chart* annotation the corresponding `SmartRadialMicroChart` control is rendered. The `entitySet` attribute needs to be specified to use the

control. The attribute is used to fetch metadata and annotation information from the given default OData model. Based on this, the RadialMicroChart UI is created. As the other SmartMicroCharts, the `sap.ui.comp.smartmicrochart.SmartRadialMicroChart` control also uses the OData metadata annotations to determine the binding paths and values inside the chart.

> **i** Note
>
> The control can also be used directly (without creating a SmartMicroChart).

Associated Labels

For information on how the control provides *Title*, *Description* and *UnitOfMeasure* values retrieved from the annotations, see the *Title*, *Description* and *UnitOfMeasure* values in the Smart Area Micro Chart [page 2432] documentation.

In addition, the Smart Radial Micro Chart supports a *FreeText* value. This can be annotated by the term Label. For more information, see the API Reference.

Binding

The `SmartRadialMicroChart` control should be bound to **one** entity and not to a collection of entities (entitySet). It supports both `enableAutoBinding` = **`false`** (no binding is done inside the control) and `enableAutoBinding` = **`true`** :

- If the `chartBindingPath` is provided, the control will be bound relatively to it. For example, the `chartBindingPath` can be a navigation property.
- If `chartBindingPath` is not provided, the control will be bound absolutely to the entitySet.

> **i** Note
>
> If `enableAutoBinding` = **`true`** and `chartBindingPath` is set then the `bindingContext` for the *Title*, *Description*, *UnitOfMeasure* and *FreeText* is set to the value as well.

Criticality

The color of the chart can be controlled by the *Criticality* property either directly or by criticality calculation. For the `sap.ui.comp.smartmicrochart.SmartRadialMicroChart` control, there are two options for setting the color of the RadialMicroChart by using its `valueColor` property:

- by setting it directly via the *Criticality* property of the *UI.DataPoint* annotation
- with the criticality calculation using the *CriticalityCalculation* property of the *UI.DataPoint* annotation

Using the first option, the user can bind the property to a path in the application's model. The criticality is then mapped to a valueColor:

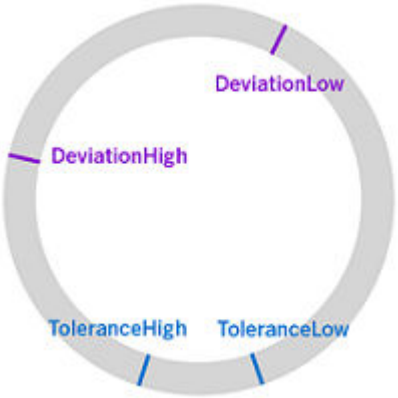
```
<PropertyValue Property="Criticality" Path="Criticality"/>
```

The mapping is done as follows:

Table 123: Criticality mapped to valueColor

Criticality	ValueColor
Neutral	Neutral
Positive	Good
Critical	Critical
Negative	Error

In the second option, the criticality can be calculated using customer-defined thresholds.



> **Note**
>
> The thresholds are not renderd in SmartRadialMicroChart.

Table 124: Thresholds

Property	Type	Sample Values
DeviationLowValue	Negative	10
ToleranceLowValue	Critical	45
ToleranceHighValue	Critical	55
DeviationHighValue	Negative	80

With the `ImprovementDirection` property the thresholds can determine the `valueColor`. For `sap.ui.comp.smartmicrochart.SmartRadialMicroChart` the *Maximize* and *Minimize* directions are supported:

- **ImprovementDirection: Maximize**
  With the Maximize direction, it is calculated the higher the value the more the circle color trends to green or the more positive is its status. Depending on the relevant thresholds (`DeviationLowValue` and `ToleranceLowValue`), there are appropriate certain points where the color changes. Reflecting the sample data above, values lower than 10 will be shown in red color, values lower than 45 but higher than 10 are displayed in orange color and all values bigger than 45 are shown in green color.

- **ImprovementDirection: Minimize**
  With theMinimize direction, it is calculated the lower the value the more the circle color trends to green or the higher the value the more negative is its status. This direction uses the `ToleranceHighValue` and `DeviationHighValue` thresholds. Reflecting the sample data above, values higher than 80 will be shown in red color, values lower than 80 but higher than 55 are displayed in orange color and all values lower than 55 are shown in green color.

Smart Multi Edit

`SmartMultiEdit` enables you to perform mass editing operations on objects that have the same structure.

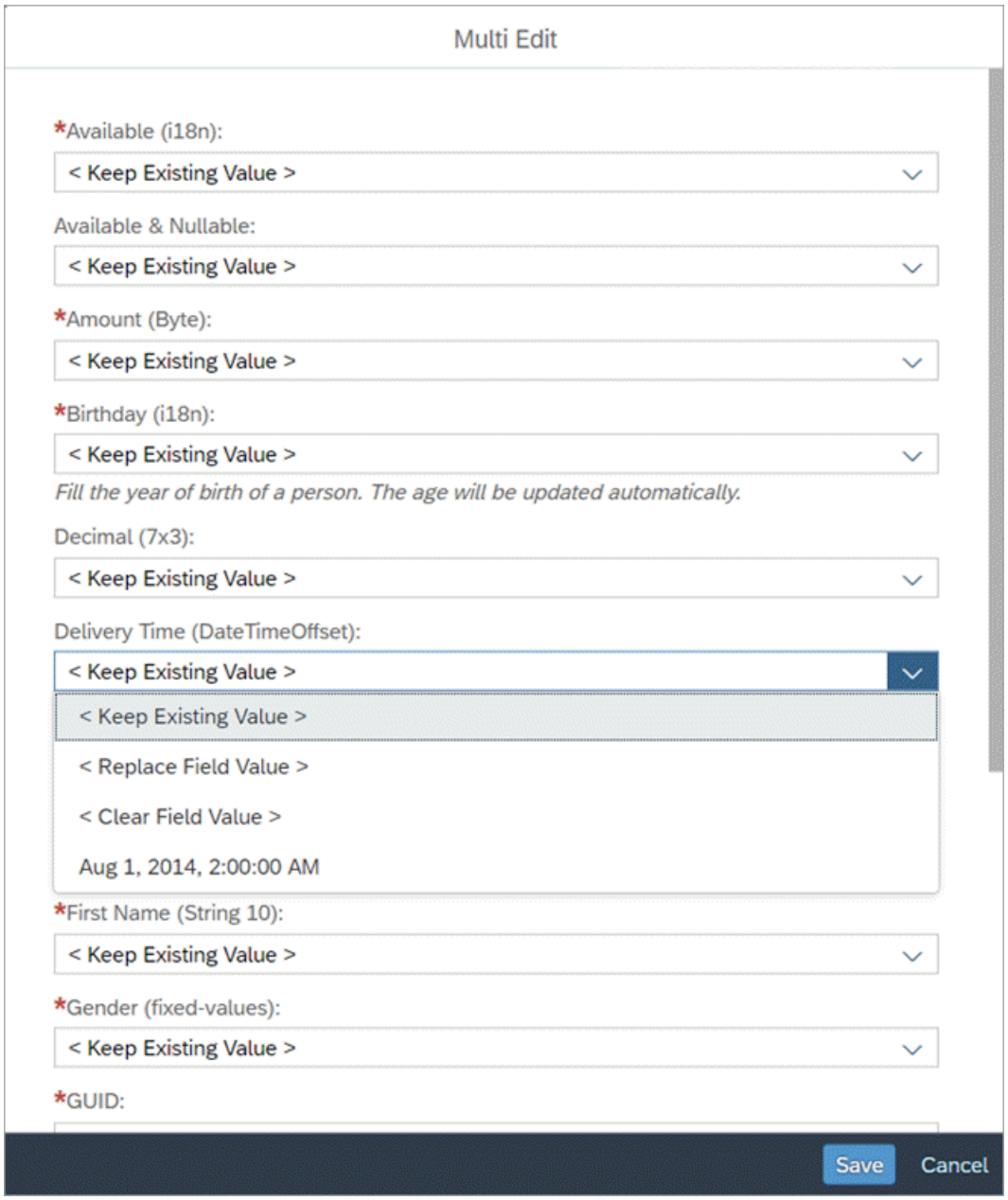
For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

With `SmartMultiEdit`, you can edit multiple homogeneous objects simultaneously. It allows you to select a field value from a combo box for all objects being edited. `SmartMultiEdit` can also handle metadata for a specific OData property when you need to enable mass editing for multiple contexts.

`SmartMultiEdit` consists of two controls:

- `SmartMultiEdit.Field` – A field that allows you to select a new value and apply it to the selected objects that include this field in their structure.
- `SmartMultiEdit.Container` – A container that provides the layout and context bindings for the `SmartMultiEdit` fields it includes.



Details

Implementation and Layout

- You can implement `SmartMultiEdit` along with a smart table [page 2444], adding a button for it to the smart table's toolbar. When you click the button, a dialog appears that contains a smart form [page 2420] with editable `SmartMultiEdit` fields.
- To specify the layout of the smart form dialog that contains the `SmartMultiEdit` fields, use the `layout` aggregation in the `SmartMultiEdit.Container` control.

Data Binding

- You can define the context binding [page 922] and the OData entity set for your multi edit fields using the `contexts` and `entitySet` properties in the `SmartMultiEdit.Container` control.
- To specify the OData property related to a specific `SmartMultiEdit` field, use the `propertyName` property of the `SmartMultiEdit.Field` control.

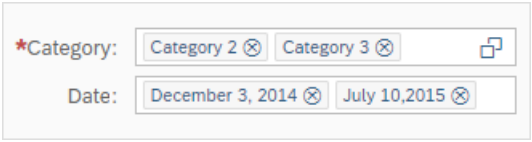
Smart Multi Input

The `SmartMultiInput` control can be used to create a multi-input field or a multi-input combobox.

For more information about this control, see the API Reference and the samples in the Demo Kit.

Overview

The `SmartMultiInput` control acts as a wrapper control for the `sap.ui.comp.smartfield.SmartField` control and supports the same settings and annotations. It interprets OData metadata to create a multi-input field or a multi-input combobox, similar to `sap.m.MultiInput` and `sap.m.MultiComboBox` but with added annotation capabilities. Thanks to the annotations support, smart multi input fields and comboboxes can be embedded into other smart controls, such as `sap.ui.comp.smartform.SmartForm`.



Details

Implementation

- You can create a `SmartMultiInput` control inside a smart field [page 2410] or a smart form [page 2420] control. For an example of smart multi input fields inside a smart form, see the samples in the Demo Kit.

Data Binding

- Smart multi input fields and comboboxes support three modes of data binding: two-way binding, one-way binding, and one-time binding. To learn more about binding modes, see Data Binding [page 815].
- The OData entity property that is displayed in the smart multi input field or combobox can be defined in the parent control's `value` property. When any of the data binding modes is used, the `value` property should be bound to a navigation property. When no data binding is used, the `value` property can be bound to an arbitrary property of the entity set specified in the parent control's `entitySet` property.

Smart Table

The `sap.ui.comp.smarttable.SmartTable` control is used to create different types of tables based on OData metadata. The control allows the user to define personalized table settings.

The frequently asked questions section below aims at answering some basic questions that you might have when using this control.

> **i Note**
>
> The code samples in this section reflect examples of possible use cases and might not always be suitable for your purposes. Therefore, we recommend that you do not copy and use them directly.

For more information about this control, see the API Reference and the samples.

For more information about annotations for this control, see the API Reference.

Overview

The `SmartTable` control is a wrapper control around any SAPUI5 table. The control analyzes the `$metadata` document of an OData service and renders a table for a specific `entitySet`.

The control allows the consuming application to build list patterns based on OData services in an efficient and consistent way and thus makes it easy for the user to create tables without much effort. For example, the control enables the automatic creation of columns.

The consuming application can overwrite the OData default information. The `SmartTable` control offers you additional built-in features, such as a row count and an export to a spreadsheet application.

> **i Note**
>
> Once the `SmartTable` control has been initialized, most of the property and aggregation changes (for example, for `entitySet` or `persistencyKey`) won't have any effect. Also, any changes of the inner table are not recognized, for example, by the personalization settings, and therefore must not be made.

Details

When using `SmartTable` with an internal responsive table, you can set the `demandPopin` property to `true`. This property renders columns that exceed the space available on the screen by displaying popins.

`SmartTable` checks the custom data section for the columns and reads the `columnIndex` attribute to determine when the columns that are defined in the XML view are rendered.

If you want to show and follow `navigationProperty` fields for `EntityType`, the `SmartTable` control automatically performs a `$expand` operation for these fields.

> **i** Note
>
> If you perform `$expand` operations while doing an export to a spreadsheet, the `$expand` parameters will automatically be removed (only relevant for the Gateway export type).

Integration with Other Controls

The `SmartTable` control is closely linked to the following other controls:

- `VariantManagement`
- `SmartFilterBar`
- `P13nDialog`

The control also supports the popover of the `SmartLink` control.

For more information about the various smart controls, see sap.ui.comp [page 2401].

FAQ

1- How can I enable personalization for custom columns, and how do they differ from the regular ones, especially when used with personalization settings or the spreadsheet export?

You can specify custom data for the personalization of the columns in your table as shown in the examples.

Example 1 for a normal aggregation:

```
<table:Column id="Ledger" minScreenWidth="Tablet" demandPopin="true">
    <table:customData>
        <core:CustomData key="p13nData"
           value='\{"columnKey": "Ledger", "leadingProperty":"Ledger",
            "additionalProperty":"LedgerName", "sortProperty": "Ledger",
            "filterProperty": "Ledger", "type":"numeric"}' />
    </table:customData>
    <Label text="Ledger" />
    <table:template>
        <Text text="{Ledger}" />
    </table:template>
</table:Column>
```

To use the SAPUI5 shortcut notation, add the following namespace part in the XML view:

```
xmlns:customData="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"
```

Example 2 for use of the shortcut notation:

```
<table:Column id="CompanyCode" minScreenWidth="Tablet" demandPopin="true"
     customData:p13nData='\{"columnKey": "CompanyCode",
"leadingProperty":"CompanyCode",
"additionalProperty":"CompanyName","sortProperty": "CompanyCode",
"filterProperty": "CompanyCode", "type":"numeric", "maxLength":"4"}'>
    <Label text="Company Code" />
    <table:template>
        <Text text="{CompanyCode}" />
    </table:template>
</table:Column>
```

In the `p13nData` object you can specify the following properties:

- `columnKey`
  A unique key used to save, retrieve, or apply personalization for a column.
- `leadingProperty`
  Retrieves data for the OData property specified here from the backend system when the column is made visible.
  OData model property name must be used.
- `additionalProperty`
  Property has to be requested if a column is visible.
  OData model property name must be used.
  Multiple property names can be specified here as comma-separated values (CSV).
- `sortProperty`
  Sorts the table based on the column specified.
  OData model property name must be used.
  This property is similar to `sortProperty` of `sap.ui.table.Column` of the grid table and should only be used if the latter does not support this feature.
- `filterProperty`
  Filters the table with the condition that has been defined.
  OData model property name must be used.
  This property is similar to `filterProperty` of `sap.ui.table.Column` of the grid table and should only be used if the latter does not support this feature.
- `isGroupable`
  Shows a field in the *Group* panel of the *View Settings* automatically; otherwise, a field might become visible only once the table (rows) are bound.
  This property is only required for the type `AnalyticalTable`. `SmartTable` automatically sets this property to `true` if a field is sortable, filterable, and a dimension.
- `type`
  Determines the type of a control; its value can be `date`, `time`, `boolean`, `numeric`, `stringdate`, `string`, or `undefined`. The control will be adapted according to the type.
  `stringdate` is used to export fields with the `IsCalendarDate` annotation.
- `maxLength`
  Numeric value to restrict number of entries in input fields
- `precision`
  Numeric value for precision

- `scale`
  Numeric value for scale
- `nullable`
  Defines whether a field can have no value (and is then relevant for filtering with the *Empty* value). Consumers of the control can use the string value `false` to indicate that the field is not nullable. The default is `nullable`.

> **i Note**
>
> Some properties that also exist in a table, for example, in `sortProperty`, will take precedence if specified in both places.

The following additional properties are required in `P13ndata` for proper formatting of custom columns for an SAPUI5-client-side export to the spreadsheet:

Table 125: P13n Properties for Custom Columns

Property	Explanation
`unit`	Name of the unit property to be used for unit of measure and currency formatting
`isCurrency`	If the column is of type `currency`, the amount with the currency is shown in the exported spreadsheet.
`align`	Configures the alignment of the column, for which you can use the same value as for the `hAlign`property of the column.
`edmType`	Actual `Edm.Type` of the OData property, which might be needed for proper formatting of columns in the spreadsheet.
`description`	Field that points to the description (`UI.Text` annotation) of this column, or, if custom-formatted columns are used, you can use the description that is used in the `formatter` function.
`displayBehaviour`	Various combinations of the description that are displayed on the UI in the following way: • `descriptionOnly`: Shows a description only • `descriptionAndId`: Shows the description followed by the ID • `idAndDescription`: Shows the ID followed by the description • `idOnly`: Shows the ID only
`width`	Width of the column, for which you should use the same value as for the column itself

Property	Explanation
`isDigitSequence`	Can be used for `Edm.String` columns that represent a numeric field with leading zeros. If set to `true`, the leading zeros are removed from this string field as the `number` format will be used in the spreadsheet.

2- Why do I not see data for my manually added column in the XML view? And why do I get an error "Select at least one column to perform the search" even though I have added several columns manually to the `sap.m.Table` inside the control?

You need to specify custom data with at least one `leadingProperty` for the table columns without a `leadingProperty` available in the control itself so the `SmartTable` control can fetch the data correctly.

```
<Column>
    <customData>
          <core:CustomData key="p13nData" value='\
{"columnKey":"Id","leadingProperty": "Id","sortProperty": "Id","filterProperty":
"Id"}'/>
     </customData>
    <Text text="Sales Order" />
</Column>
```

Without this, the `SmartTable` control cannot recognize the column.

3- How can I define columns with my own style, for example, using the formatting or the controls I require, in a responsive `SmartTable` control (`tableType="ResponsiveTable"`)?

For the `sap.m.Table/ResponsiveTable`, you need to provide a corresponding `ColumnListItem` in the `items` aggregation in addition to the columns, as you would when using this SAPUI5 table on its own.

```
<smartTable:SmartTable entitySet="Items"..>
    <Table>
          <!-- Columns must have unique IDs if table personalization service is
used -->
          <columns>
         <Column id="Name" width="auto" minScreenWidth="Tablet" visible="false"
             customData:p13nData='\{"leadingProperty":"Name",
"columnKey":"Name", "sortProperty":"Name", "type":"numeric"\}'>
             <header>
            <Label text="{/#Item/Name/@sap:label}"/>
             </header>
         </Column>
         </columns>
         <ColumnListItem id="columnListItem" vAlign="Middle" type="Navigation"
press="onItemPressed">
        <cells>
                  <Text text="{path:'Name',
formatter:'my.own.formatter.functionName'}" maxLines="2"... />
            </cells>
         </ColumnListItem>
      </Table>
 </smartTable:SmartTable>
```

The `sap.m.Table` uses the `columns` aggregation for the header and the `items` aggregation containing `ColumnListItem` with cells for the template control that is cloned for each row in the table.

> **i Note**
>
> For any supported SAPUI5 table, you can add custom columns in the XML view along with the required `customData` for the column.

4- How do I use the `SmartField` control in combination with the `SmartTable` control? And how can I make use of metadata and field controls to manage the state of editable fields in my table?

For editing scenarios with backend metadata and field controls, it is recommended to use the `SmartField` control along with the `SmartTable` control. You can have the `SmartTable` automatically create `SmartField` controls using the following code:

```
<smartTable:SmartTable id="ItemsST" entitySet="Items"
customData:useSmartField="true"...>
```

To make this work, the `customData` namespace has to be defined correctly in the XML view to enable SAPUI5 shortcut notation for custom data aggregations: `xmlns:customData="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"`.

5- How do I format the group headers shown in `SmartTable` of type `ResponsiveTable` (`sap.m.Table`)? And why is a technical key shown instead of a description when grouping is done in `SmartTable` of type `ResponsiveTable`?

Grouping in `ResponsiveTable` is done by sorting table entries. You can define your own formatting for the group title in `ResponsiveTable` by specifying a group function for the first sorter.

For more information on how this can be done, see the code sample.

When using the `SmartTable` control, you can use the `beforeRebindTable` event and get available sorters using the `bindingParams` (event parameter). Check if the first sorter there has a group.

```
onBeforeRebindTable: function(oEvent) {
    var mBindingParams = oEvent.getParameter("bindingParams");
    var oSorter = mBindingParams.sorter[0];
 //Check if sorter is for Grouping
    if(oSorter.vGroup){
```

There are two options. The first option looks like this:

```
//Replace the Group function
        oSorter.fnGroup = this.mGroupFunctions[oSorter.sPath];
```

You can also do the following:

```
//Replace the Grouping sorter itself
        mBindingParams.sorter[0] = new Sorter(oSorter.sPath, bDescending,
this.mGroupFunctions[oSorter.sPath]);
 }
```

For more information on how `this.mGroupFunction` has to be implemented, see the code sample in the Demo Kit.

You can replace group functions for the sorter or the sorter itself with the ones you have defined (with own formatter for grouping based on the property) if the sorter is used for grouping.

6- How do I create and pass a search query (`$search="foo"`) when using the `SmartTable` control?

Use the `beforeRebindTable` event and implement this manually.

```
onBeforeRebindTable: function(oEvent) {         var
        mBindingParams = oEvent.getParameter( "bindingParams" );
        mBindingParams.parameters[ "custom" ] =
        {                          "search-focus" :
        sBasicSearchFieldName, //  the name of the search
        field                        "search" :
        sBasicSearchText //  the search text
        itself!                 };  }
```

This will then be used internally when creating the table binding.

> **i** Note
>
> In the same way, you can also add any custom URL parameters or use this event to add OData `$expand` parameters: Instead of `"custom"`, use `"expand"` as shown in the example above.

7- How do I get data for custom columns (icons, formatters etc.) that are not present in the columns/binding (`select` = 'ColA,ColB,foo,bar') of the `SmartTable` control?

Use the `beforeRebindTable` event and implement this manually.

```
onBeforeRebindTable:  function  (oEvent) {
        var   oBindingParams = oEvent.getParameter( "bindingParams" );
         if  (oBindingParams.parameters.select.search( "SomeIconCode1" ) < 0) {
                          oBindingParams.parameters.select +=
",SomeIconCode1" ;
         }
      if (oBindingParams.parameters.select.search( "SomeIconCode2" ) < 0) {
                          oBindingParams.parameters.select +=
",SomeIconCode2" ;
         } . . .
```

This will then be used internally when creating the table binding.

> **i** Note
>
> If only one property is needed for a given column, you can also use `additionalProperty` in `customData` as already mentioned instead of the event-based approach as described here.

> **i** Note
>
> For an `AnalyticalTable` control or `AnalyticalBinding`, you have to use a dummy column (`visible="false"`) with the `leadingProperty` you require and the set attribute `inResult="true"` instead.

8- I would like to pass my own custom sorters, filters, and binding parameters when binding the table data in the `SmartTable` control. How do I do this? And how can I have my own binding implementation for the `SmartTable` control?

You can modify the array of filters before binding is triggered in the `SmartTable` control by listening to the `beforeRebindTable` event.

To enable this, your code in the XML view should look like this:

```
<smartTable:SmartTable entitySet="LineItemsSet"
beforeRebindTable="onBeforeRebindTable"…
```

You can now get the list of filters and sorters to be used in table binding using the following:

```
...
onBeforeRebindTable: function(oEvent) {
              var mBindingParams = oEvent.getParameter("bindingParams");
              var aFilters = mBindingParams.filters
```

With this, you need to set back the value to `mBindingParams.filters`, and you can pass a new filter array as well.

> **i Note**
>
> In some exceptional cases, you can set `mBindingParams.preventTableBind="true"` to prevent table binding from taking place (optional) and do the binding at a later point in time. This is shown in the following method:
>
> ```
> someMethod: function() {
>           //get the smartTable and call the method rebindTable()
>           oSmartTable.rebindTable();
>       }
> ...
> ```

> **i Note**
>
> If you would like to trigger the binding manually, use the `rebindTable` method of the `SmartTable` control and do not use `bindRows` in the underlying table.

> **i Note**
>
> For a **custom multi-filter scenario**: If you want to pass multi-filters (filters combined with AND/OR explicitly) in your custom implementation, the SAPUI5 default logic in the core classes combines multiple multi-filters with an OR by default. If you would like to use AND in combination with the multi-filters returned in the `beforeRebindTable` event of the `SmartTable` control (if a filter is set in `SmartFilter`) and your own `MultiFilter`, you have to replace the filters in the `beforeRebindTable` event with an explicit AND `MultiFilter`. There is currently no way to combine multiple multi-filters in the `SmartTable` control itself. You as the consumer of the control have to make sure you combine these multi-filters yourself by checking first if any internal multi-filters exist. You also have to ensure that the internal multi-filter is added first in the array of filters present in the `beforeRebindTable` event.

```
onBeforeRebindTable: function(e) {
                   var b = e.getParameter("bindingParams");
                    var aDateFilters = [new
sap.ui.model.Filter("BindingPeriodValidityEndDate",sap.ui.model.FilterOperator.LE
,d), new
sap.ui.model.Filter("BindingPeriodValidityEndDate",sap.ui.model.FilterOperator.GT
, null).....];
                    var oOwnMultiFilter = new sap.ui.model.Filter(aDateFilters,
true);
                    //Special handling for multiple multi-filters
                    if (b.filters[0] && b.filters[0].aFilters) {
                           var oSmartTableMultiFilter = b.filters[0];
```

```
                            // if an internal multi-filter exists then combine
custom multi-filters and internal multi-filters with an AND
                            b.filters[0] = new
sap.ui.model.Filter([oSmartTableMultiFilter, oOwnMultiFilter], true);
                        } else {
                            b.filters.push(oOwnMultiFilter);
                        }
}
```

9- How do I add custom toolbar buttons in the `SmartTable` control?

You can do this using the `customToolbar` aggregation, as shown below:

```
<smartTable:SmartTable id="ItemsST" entitySet="Items" ...
        <smartTable:customToolbar>
                <OverflowToolbar design="Transparent">
                        <ToolbarSpacer/>
                        <Button text="Test"/>
                        <Button text="Click me!"/>
                </OverflowToolbar>
       </smartTable:customToolbar>
```

> **i Note**
>
> We recommend to use `OverflowToolbar` instead of `Toolbar`, as shown above, to make the toolbar responsive.

10- How do I fetch data from `navigationProperty` (or association entities) using the `SmartTable` control?

`SmartTable` provides a `tableBindingPath` property in addition to `entitySet`, which can be used to specify a navigation property path, for example, `SalesOrder(123)/toItems`.

> **i Note**
>
> For an `AnalyticalTable` control or `AnalyticalBinding`, you might have to pass `entitySet` in `bindingParameters` using the `beforeRebindTable` event. This is necessary if the `entitySet` path does not conform to the one that is checked internally by `AnalyticalBinding`, for example, in the `beforeRebindTable` event, as shown here:
>
> ```
> var mBindingParams = oEvent.getParameter("bindingParams");
> mBindingParams.parameters.entitySet = "NameOfEntitySet";
> ```

11- Why do I not see any columns in my `SmartTable` control?

`SmartTable` creates the initially visible column based on the `LineItem` or the `PresentationVariant` annotations. You can either specify the initial fields there or create it manually in the XML view by adding columns to the underlying table.

For more information, see the sample.

12- How can I use `applyVariant()` in combination with the `SmartTable` control?

The `applyVariant()` is an interface function for the `SmartVariantManagement` control that sets the current view for the `SmartTable` control.

Applications can also create their own application-specific views as the default, which are **not** standard views defined by the `SmartTable` control. These application views are only called once during the initialization of the `SmartTable` control.

If an application default view has been defined, then all other views are based on this application default view. Any change made to the `SmartTable` control and saved as a view is merged with the application default view by the `SmartTable` control. This data is then stored as a **new** view, a combination of the change made and the application default view.

For example, an application default view contains two groups that have been defined for the `SmartTable` control. When a new group is added and saved as a view, the new view will comprise the newly added group and the application default view. Thus, the end result are three groups for the `SmartTable` control.

> **i** Note
>
> When the user selects *Ungroup All* in the *Group Header Menu* on the UI of `AnalyticalTable` in the `SmartTable` control, and this is then stored as a view, the change is saved as a combination of the ungrouping change and the application default view that contains the grouping information. Thus, the end result will be the application default view.

13- Why do I see an incorrect count in the header of my `SmartTable` control?

If the backend does not support count, if the count support has been disabled in the model, or in case of a tree scenario, you have to set `showRowCount="false"` in the `SmartTable` control.

14- Can I use annotations with qualifiers in the `SmartTable` control? And, in particular, how can I use the `LineItem` and `PresentationVariant` annotations with a qualifier within the `SmartTable` control?

As a general rule, the `SmartTable` control looks for annotations **without** a qualifier, the primary annotations. However, you can also use the `PresentationVariant` and `LineItem` annotations with qualifiers as mentioned below.

We first look for `PresentationVariant` and try to get the `LineItem` annotation from there. If no such annotation exists, we look for the `LineItem` annotation directly on the entity.

You can set the qualifier that you want to use for the `LineItem` annotation in the `SmartTable` control using `lineItemQualifier` `customData`:

```
<SmartTable customData:lineItemQualifier=”Customer360” …>
```

Or, if you want us to use a specific `PresentationVariant`, `presentationVariantQualifier` is also supported, which can be used as shown below:

```
<SmartTable customData:presentationVariantQualifier =”Customer360” …>
```

If **no qualifier** has been defined for the presentation variant, you can use the fallback option and check if there is a `LineItem` annotation with or without a qualifier, as specified by the application developer.

> **i** Note
>
> `customData` is the shortcut notation for specifying custom data for the control, provided you have added the following to the XML view: `customData="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"`.

For more information on how to use custom data in XML views, see Custom Data - Attaching Data Objects to Controls [page 1042].

15- How can I dynamically switch between read-only and editable controls in my table using the `editable` property?

For the switch you have to enable the `SmartToggle` feature in the `SmartTable` controls as follows:

```
<smartTable:SmartTable id="ItemsST" entitySet="Items"
customData:useSmartToggle="true"...>
```

In order for this to work, the `customData` namespace in the XML view must be declared correctly to enable shortcut notation for custom data aggregations:`customData="http://schemas.sap.com/sapui5/extension/sap.ui.core.CustomData/1"`.

> **i Note**
>
> `SmartToggle` is a feature provided by the `SmartTable` control that allows you to toggle between display and edit mode for all the relevant controls in the cells of the `SmartTable` control.

16- I did an export using the *Export to Spreadsheet* button in the `SmartTable` control. Why is the data in the spreadsheet different from the data I see on the UI?

The formatting in an SAPUI5 client-side generated spreadsheet can differ slightly from the UI. However, if you notice large differences, the issue might be due to custom columns that do not have sufficient `P13ndata`, as required for a client-side export.

The following additional properties are required in `P13ndata` for proper formatting of custom columns for an SAPUI5-client-side export to the spreadsheet:

Table 126: P13n Properties for Custom Columns

Property	Explanation
`unit`	Name of the unit property to be used for unit of measure and currency formatting
`isCurrency`	If the column is of type `currency`, the amount with the currency is shown in the exported spreadsheet.
`align`	Configures the alignment of the column, for which you can use the same value as for the `hAlign`property of the column.
`edmType`	Actual `Edm.Type` of the OData property, which might be needed for proper formatting of columns in the spreadsheet.
`description`	Field that points to the description (`UI.Text` annotation) of this column, or, if custom-formatted columns are used, you can use the description that is used in the `formatter` function.

Property	Explanation
`displayBehaviour`	Various combinations of the description that are displayed on the UI in the following way: • `descriptionOnly`: Shows a description only • `descriptionAndId`: Shows the description followed by the ID • `idAndDescription`: Shows the ID followed by the description • `idOnly`: Shows the ID only
`width`	Width of the column, for which you should use the same value as for the column itself
`isDigitSequence`	Can be used for `Edm.String` columns that represent a numeric field with leading zeros. If set to `true`, the leading zeros are removed from this string field as the `number` format will be used in the spreadsheet.

Apart from the configuration, you can use the `beforeExport` event to change the formatting and configuration of an SAPUI5 client-side export to a spreadsheet as shown in the samples.

For more information, see the sample for `SmartTable` and the samples for `sap.ui.export.Spreadsheet` .

17- Why is the `SmartTable` control (of type `AnalyticalTable`) showing the columns with the `Text` annotation incorrectly in my app? Why is the `SmartTable` control not using `TextArrangement` annotations as I expected?

The `SmartTable` control supports the `Text` and `TextArrangement` annotations which allow you to show descriptions and ID fields together as annotated by the `TextArrangement` annotation (or configured using `displayBehaviour` custom data). The use of this annotation has some limitations. For more information, see the API Reference: `Text` annotation.

In addition, `AnalyticalBinding` ignores the entire `$select` if the dimensions and measures do not match what has been calculated internally. Using `Text` or `TextArrangement` automatically only works if `$select` contains the correct fields. If you add some dimensions to `requestAtLeast`, `Criticality`, or other fields, `$select` is ignored, and fetching the descriptions and the criticality, for example, does not work.

During automatic dimension selection `AnalyticalBinding` ignores the specified `$select` if a text or description column is added to the table content **without** the corresponding dimension. To work around this binding limitation, the `SmartTable` control calculates and adds the dimension of the property of the `Text` annotation to the `additionalProperty` custom data (used to calculate `$select`) in the `SmartTable` control wherever possible.

There are a few other things to keep in mind when using this feature:

- Multiple dimensions must not point to the same property of the `Text` annotation (only **one** will be used).
- `Text` annotation itself is marked as a dimension, the actual dimension field is not automatically selected.
- The `Text` annotation for ignored or hidden dimension fields does not automatically contain the dimension name in `additionalProperty`, as metadata analysis for such fields is ignored.

18- How come the column headers in the `SmartTable` control of type `ResponsiveTable`If the property of the are clickable?

`SmartTable` internally activates actions for the column headers of a responsive table. Users can select a column and sort and filter it using the buttons that are displayed.

Keep in mind that the following applies:

- The active column headers are only available if `useTablePersonalisation="true"`.
- If a column has already been filtered, clicking the filter button will not show the selected column directly (this is the same as in a grid table, since there is a restriction for `p13nFilterPanel`).
- Pressing the `Ctrl` key and clicking on a column to add multiple sorters is currently not supported.

For more information, see the sample.

19- How come the *Export to Spreadsheet* button of the `SmartTable` control has a menu?

If `exportType="UI5Client"` for `SmartTable`, the *Export to Spreadsheet* button will be rendered as `sap.m.MenuButton` with the menu options *Export* and *Export As*. *Export As* provides a number of export settings that can be configured by the user.

For more information about the spreadsheet export and the options on the export UI, see Spreadsheet Export [page 1286].

20- Why does the `initialise` event of `SmartTable` not get fired in my scenario?

The `SmartTable` control fires the `initialise` event just **once** after it has completed analyzing the metadata and has initialised its inner state for the first time. Therefore, using `attachInitialise` does not help. However, the `isInitialised` method can be used in such scenarios.

You can also use the following code sample to handle scenarios where you need to trigger some function after this control has been initialized. It should work in scenarios where the event has already been fired:

```
if (oSmartControl.isInitialised()){
  runSomeCodeAfterInit();
} else {
  oSmartControl.attachInitialise(runSomeCodeAfterInit);
}
```

21- How is sorting done in amount fields with multiple currencies if `ApplyMultiUnitBehaviorForSortingAndFiltering` is set?

The `SmartTable` control automatically adds the unit, in this case the currency code, as an additional sorter before the amount unless the unit is added as a sorter explicitly. This happens if the `ApplyMultiUnitBehaviorForSortingAndFiltering` annotation is applied. This behavior is applicable for all columns in the `SmartTable` control.

For more information, see the API Reference.

For custom columns, the required `p13nData` information has to be provided by the consumers (`isCurrency` and `unit`).

22- Which key combinations can be used for the `SmartTable` control?

The following key combinations are supported:

- `CTRL` + `SHIFT` + `E` or `⌘ / Command` + `SHIFT` + `E`: Opens the *Export As* dialog for the spreadsheet export

- `CTRL` + `,` or `⌘ / Command` + `,`: Opens the personalization settings

23- Why does the first column not appear in the personalization settings when using `treeTable` as the table type?

The `SmartTable` control disables the personalization for the first column if `tableType` is set to `sap.ui.comp.smarttable.TableType.TreeTable`. The `TreeTable` control provides a comprehensive set of features for displaying hierarchical data where a table is displayed as a hierarchical tree. `TreeTable` is designed to retain this structure also for the first column of the table, which is why `SmartTable` excludes it from personalization.

24- Why is the unit of measure (UoM) or currency not right aligned with the column header?

The currency and UoM columns in the `SmartTable` control are built to be aligned at decimal point. Although the overall content is right aligned, the header and cell might not be aligned as this is a generic control that is not built to handle a single currency or UoM (for example, %, h).

The UoM part always uses ~3em to prevent the content from being cut off. That is why there might be a space behind the unit in the cell in contrast to the header, depending on the UoM or currency used.

Also, if either the *Product Depth* or *Product Width* column is added to a `SmartTable` control, the UoM, such as m or cm, might not be aligned, but the aim is to align the values at decimal point and not at header and content level.

The application can use custom columns if an exact alignment of content and header is required.

For more information, see the Sample.

Smart Variant Management

The `sap.ui.comp.smartvariants.SmartVariantManagement` control provides an interface to enable a simple integration of the `sap.ui.comp.variants.VariantManagement` control and access to the layered repository of SAPUI5 flexibility for easy communication.

The frequently asked questions section below aims at answering some basic questions that you might have when using this control.

> **i Note**
>
> The code samples in this section reflect examples of possible use cases and might not always be suitable for your purposes. Therefore, we recommend that you do not copy and use them directly.

For more information about this control, see the API Reference and the sample.

Overview

The `SmartVariantManagement` control is a specialization of the `VariantManagement` control. This basic control handles the visual representation of variants, or views, on the user interface.

> **i Note**
>
> You can define views for specific selections of data on the user interface, for example, based on filter settings. Views are also called variants, usually in a more technical context, for example, in the API names and texts of the control.

The `SmartVariantManagement` control communicates with the layered repository. The layered repository provides a way to store and retrieve flexibility information, such as personalization data and views for other controls.

End users can create, change, and save views that will then be stored in the `USER` or `CUSTOMER` layer of the layered repository depending on the relevant use case.

For more information about SAPUI5 flexibility and the layering concept, see SAPUI5 Flexibility: Adapting UIs Made Easy [page 1152].

> **i Note**
>
> We recommend to use the `SmartVariantManagement` control rather than the `VariantManagement` control, because it enables the communication with the layered repository.

The `SmartVariantManagement` control can be used in combination with the following smart controls:

- `SmartFilterBar`
- `SmartTable`
- `SmartChart`

Prerequisites

To use the `SmartVariantManagement` control, consuming applications have to provide the following information and comply with the interface standard:

- The control using the personalization data
- A type
- The name of the property describing the key
- Optional information about the data source

This information has to be transferred to the `SmartVariantManagement` control during creation using the `personalizableControls` association. To transfer the data, the `sap.ui.comp.smartvariants.PersonalizableInfo` class must be used.

The control using the personalization data must also be attached to the `Initialise` event of the control and call the `initialise` method of the control, as shown in the following example:

```
sap.ui.require([
    'sap/ui/comp/smartvariants/SmartVariantManagement',
    'sap/ui/comp/smartvariants/PersonalizableInfo'
], function (SmartVariantManagement, PersonalizableInfo) {
    var oSmartVariantManagement = new SmartVariantManagement();
    var oPersInfo = new PersonalizableInfo({
        type: "filterBar",
        keyName: "persistencyKey",
        dataSource: this.getEntityType()
```

```
    });
    oPersInfo.addControl(this);
    oSmartVariantManagement.addPersonalizableControl(oPersInfo);
    this.fnInitialiseVariants = jQuery.proxy(this._initialiseVariants, this);
    oSmartVariantManagement.attachInitialise(this.fnInitialiseVariants);
    oSmartVariantManagement.initialise();
})
```

Once the `SmartVariantManagement` control has initialized the layered repository and retrieved the relevant changes, it informs the control using the personalization data about the end of the initialization phase with the `Initialise` event.

Details

To exchange data with the layered repository, the control using the personalization data has to provide the following methods that can return and retrieve variants:

- `fetchVariant`
- `applyVariant(oVariant)`

The `fetchVariant` method is called by the `SmartVariantManagement` control every time an interaction takes place with the `VariantManagement` control and when executing a `Save`. In the latter case, the control using the personalization data has to return a JSON-compliant object. The layered repository treats this information as a black box. It does not manipulate this object in any way.

The `applyVariant` method is called by the `SmartVariantManagement` control every time the user selects a new entry in the view list. The previously stored JSON-compliant object will be transferred to the `applyVariant` method, and the control using the personalization data can now respond to the information stored in this object, as shown in the example:

```
sap.ui.comp.smartfilterbar.SmartFilterBar.prototype.fetchVariant = function() {
    var aFiltersInfo;
    var oVariant = {};
    aFiltersInfo = this._determineVariantFiltersInfo();
    oVariant.filterbar = (!aFiltersInfo) ? [] : aFiltersInfo;
    oVariant.filterBarVariant = this._fetchVariantFiltersData();
    return oVariant;
};
sap.ui.comp.smartfilterbar.SmartFilterBar.prototype.applyVariant =
function(oVariant) {
    this._applyVariant(oVariant);
};
```

> **i** Note
>
> The `SmartVariantManagement` control triggers the `fetchVariant` method without any user interaction right after it fires the `Initialise` event. This enables the `SmartVariantManagement` control to handle the standard view. This view represents the state of the user interface that is delivered by default. The control can revert the data to this view every time the user selects the standard view at a later point in time.

Page Variants

A page variant is a single UI instance of the `SmartVariantManagement` control that can personalize **multiple** smart controls instead of only one.

To use this enhanced function of the `SmartVariantManagement` control, take the following into consideration:

- The `SmartVariantManagement` control has a `persistencyKey` property of its own.
  This is the key for storing the personalization data of the smart controls that you want to personalize.
- For each smart control, a persistency key has to be provided.
  Within the personalization data, this key will identify the specific data for each individual smart control.
- The smart controls support the `smartVariant` association which has to be assigned along with the page variant reference.

> **i Note**
>
> If the page variant is used by the `SmartFilterBar` control, the persistency key of the page variant has to be assigned using the `pageVariantPersistencyKey` custom data of the `SmartFilterBar` control. The `SmartFilterBar` control internally adapts the related `SmartChart` or `SmartTable` controls, and therefore, the `smartVariant` association doesn't have to be assigned.

For more information about page variants, see the sample.

Favorites

If you want to use favorites to manage your views, you have to set the `useFavorites` property in the `VariantManagement` control to `true` (default is `false`).

In the `VariantManagement` control, each `VariantItem` has a `favorite` property that determines if the `VariantItem` in question is treated as a favorite.

The `SmartVariantManagement` control automatically starts in a mode where favorites are activated.

You can define favorites in the *Manage Views* dialog. Favorites selected in the dialog are stored as changes in the layered repository that are applied each time the `SmartVariantManagement` control is initiated.

FAQ

How can I share my views?

The `SmartVariantManagement` control allows you as the end user to share your views with other users. This can be done by making them public by selecting *Public* in the *Save View* dialog when saving a new view.

When you want to save the view, you have to select a transport request. The selection you get depends on the setup of your system and how you use SAPUI5 flexibility. The transport function for the layered repository has to be enabled.For more information about sharing views, see the API Reference and the sample.

How can I make sure that only key users can make views public?

You can use a setting in SAPUI5 flexibility that determines whether views can be shared or made public by all users (default) or key users only. For more information on how to activate the related key user check, see 2658662. For more information about making views public, see Step 7: View Management [page 589].

sap.ui.core

This library contains the jQuery plugins (`jQuery.sap.*`), the core and all its components, base classes for controls, components and the Model-View-Controller (MVC) classes.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]
API Reference: `sap.ui.core`

Icon and Icon Pool

The `sap-icon://` protocol supports the use of icons in your application based on the icon font concept, which uses an embedded font instead of a pixel image.

Compared to image-based icons, icon font is easily scalable and you can change the color and apply various effects via CSS. SAPUI5 provides the `Icon` control in the `sap/ui/core/Icon` module and a set of predefined icons available in `IconPool` in the `sap/ui/core/IconPool` module.

> **i** Note
>
> The icon font will not work if Web fonts are blocked for the user's operating system, for example, by the *Blocking Untrusted Fonts* feature in Microsoft Windows (see Block untrusted fonts in an enterprise in the Microsoft Windows IT Center).

> **!** Restriction
>
> In SAP Fiori app, you should not use icons with active state to trigger actions, use icon-only buttons (`sap.m.button`) instead.

Using Custom Icons

To display your custom icons in all browsers that SAPUI5 supports, you need both, the `woff` and the `woff2` version of your icon file. To use your own icon font files in the `Icon` control, the font file and the metadata for

the icons in the font file need to be registered in the `IconPool`. You can register both of them by calling the `IconPool.registerFont` with a config object which contains the following options:

- `fontFamily`: Name of the font file without the font extension
- `fontURI`: URI of the folder where the `woff` and `woff2` files are included. You can use the `sap.ui.require.toUrl` function to resolve a folder path based on the resource path setting.
- `collectionName` (optional): Collection name which can be used in the `sap-icon` URI to reference the icons. If this is not provided, the `fontFamily` is used as `collectionName`.
- `metadata` (optional): Object that contains the mapping of the icon name to the icon's hex code, for example `{ "code1": "e011", "code2": "e012", "spike-arrest": "e013", "verify-api": "e014" }`.
- `metadataURI` (optional): URI of a JSON file that contains the mapping of the icon name to the icon's hex code for every icon in the icon file
- `lazy` (optional): Metadata for the icons is not loaded until the first icon from the icon set is used

Note

If neither `metadata` nor `metadataURI` is provided, a request is sent to `fontURI/fontFamily`.json to load the metadata.

Example

The `sap.tnt` library provides an extra icon set. The `sap/tnt/themes/base/fonts` folder contains `SAP-icons-TNT.woff` and `SAP-icons-TNT.woff2` as well as the `SAP-icons-TNT.json` JSON file, which contains the mapping of the icon name and the icon's hex code:

```
{
  "technicalsystem": "e000",
  "systemjava": "e001",
  "systemabap": "e002",
  "systemrecommendations": "e003",
  "system": "e004",
  "systemtrex": "e005",
  "systemtracks": "e006",
  "technicalinstance": "e008",
  "technicalscenario": "e007",
  "throughput-backlog": "e009",
  ...
}
```

The JSON file has the same name as the `woff` and `woff2` files, so it is not necessary to set `metadataURI`. To register the icon in the `IconPool`, use the following code. Note that in the example the metadata is not loaded until one icon from this icon set is used because `lazy` is set to true.

```
// "IconPool" required from module "sap/ui/core/IconPool"
IconPool.registerFont({
    collectionName: "tnt",
    fontFamily: "SAP-icons-TNT",
    fontURI: sap.ui.require.toUrl("sap/tnt/themes/base/fonts"),
    lazy: true
});
```

Referencing Icons

To reference icons, you assign the icon URI to a control by setting `sURI` for the control's corresponding property. To get the icon URI, the following two options exist:

- Call `IconPool.getIconURI` with the `iconName` property:

```
// "IconPool" required from module "sap/ui/core/IconPool"
var sURI = IconPool.getIconURI("accidental-leave"); //please change the
parameter to the name of your desired icon
```

- If you know the collection name and the icon name, write the icon URI directly in the following format:

```
sap-icon://[collection-name]/[icon-name]
```

> **i** Note
>
> You need the collection name only for custom icons. The URI for predefined icons does **not** need the collection name.

Using Icons in Controls

The following code snippet shows how the sap.m.Dialog control that already supported image URI has been adapted to also support icon URI. `IconPool.createControlByURI` returns an instance of `Icon` if `sURI` is an icon URI. Otherwise, the second parameter is called as a constructor method to create an instance. The `sURI` is set for the `src` property of the instance.

```
    // "IconPool" required from module "sap/ui/core/IconPool"
    // "Image" required from module "sap/m/Image"
    // "Device" required from module "sap/ui/Device"
    Dialog.prototype.setIcon = function(sURI){
        this.setProperty("icon", sURI, true);
        if (!Device.os.ios){
           //icon is only shown in non iOS platform
           if (this._iconImage) {
               this._iconImage.setSrc(sURI);
           } else {
               this._iconImage = IconPool.createControlByURI({
                   src: sURI //src is mandatory
                   /* other properties can be put here, such as id, ...*/
               }, Image);
           }
       }
       return this;
    };
```

If the `img` tag is rendered directly in the control, and not by creating an image control, use the `writeIcon` method on `sap/ui/core/RenderManager`. The `writeIcon` method accepts a URI as the first parameter. Depending on this parameter, it renders either an `img` or a `span` tag. The classes and attributes defined in the second and third parameter are also added to the rendered tag.

Font face is inserted into the style sheet dynamically when `Icon` or `writeIcon` are used for the first time. If the special character needs to be written into the CSS to show the icon in a control, call the

`IconPool.insertFontFaceStyle` function to insert the built in font face in your CSS. This is shown in the following code snippet:

```
    // "IconPool" required from module "sap/ui/core/IconPool"
    IconPool.insertFontFaceStyle();
});
```

Styling the Icon Control

If you render the icon span directly in your control, or use icon font in your CSS, you have the maximal freedom to style the Icon control.

If you use the icon by creating an instance of `Icon` within your control, however, use the CSS class `sapUiIcon` to add a new style to the icon. To avoid influencing the style of icons used elsewhere, wrap the icon CSS class with your control's root DOM class.

sap.ui.richtexteditor

The sap.ui.richtexteditor offers functionality for text editing - like, for example, bullets, indentation, fonts, and coloring.

Overview

The `RichTextEditor` uses an open-source library called TinyMCE. Beside the native toolbar, it can also use a toolbar built with sap.m controls.

Preamble

The `RichTextEditor` uses a third-party component and therefore some additional limitations apply for its proper usage and support:

- If you use API calls to the native API of TinyMCE, we **cannot** guarantee backwards compatibility after an upgrade of the TinyMCE library.
- As of version 1.60, the default editor type is TinyMCE4. Keep in mind that TinyMCE3 is no longer supported and cannot be used.
- Accessibility features that the wrapper control provides, like high-contrast themes and keyboard handling, are not fully available for the native toolbar.
- The third-party component TinyMCE does not fully support the High Contrast themes. The control, which internally uses TinyMCE, is thus also not compliant to this product standard. Applications, which embed the `RichTextEditor` control and use the high-contrast theme, will not have a full High Contrast support. Certain buttons, menus etc. are available in the correct theme, but many elements are still showing up with a normal, non-contrast style.
- The `RichTextEditor` uses a third-party component, which might in some cases not be completely compatible with the way UI5's (re-)rendering mechanism works. **If you keep hidden instances of the**

control (instances which are not visible in the DOM), you might run into problems with some browser versions. In this case, please make sure that you destroy the `RichTextEditor` instance instead of hiding it, and create a new one when you show it again.

- Known cases that might cause Content Security Policy relevant issues:
  - If you are using one of the following plugins: `compat3x`, `linkchecker`, `preview`.
  - If you are using the `tinymce.ui.Iframe` widget.
  - If you are using the Internet Explorer 11, or below versions, and modify the `document.domain`.
- If you want to use the custom toolbar, you need to instantiate the `RichTextEditor` in the application's controller. This way the controller can check and wait for the `sap.m` library to be loaded and then init the controls. RichTextEditor can be embedded in an XML view, but as the XML view adds an additional layer, this may lead to problems while loading the custom toolbar. The sap.m library cannot be required from TinyMCE. This means that `sap.m` may not be available in time for the rendering of the custom toolbar.
  See Supported Library Combinations [page 26].
  Using the native toolbar in an XML view is still possible.

Guidelines

- Do not instantiate the `RichTextEditor` from a hidden container (for example a `div` with `visibility="hidden">`).
- Make sure that you destroy the `RichTextEditor` instance instead of hiding it, and create a new one when you show it again.
- The `RichTextEditor` has to be used only for desktop scenarios.

Custom sap.m. Toolbar

To replace the native toolbar, set the following properties:

- `customToolbar=true`
- `editorType = tinyMCE4`
- If you want to use the custom toolbar, you need to instantiate the `RichTextEditor` in the application's controller. It cannot be embedded in an XML view.

> ⚠ Caution
>
> In order to render the custom toolbar, make sure that your application has loaded the sap.m library.

Custom Buttons

With version 1.48 you can add your own buttons to the custom toolbar. The buttons are stored in the `customButtons` aggregation. Make sure that you provide dedicated click events for your buttons.

Tables

As of version 1.50, the table functionality is available just by adding the table button group for both toolbars (TinyMCE and custom).

> ℹ Note
>
> This changes won't affect applications already using a `sap.ui.richtexteditor.RichTextEditor` with an added table plugin.
>
> When the table button group is added and the `customToolbar` property is set to `true`, a button with a table icon will be visible in the custom toolbar. This button opens a `sap.m.Dialog` for inserting tables. You

can choose how many rows and columns your table will have and set the height and width properties. After creating the table, the native TinyMCE context menu for modifying the table properties, triggered from selecting the table in the edit area, will be available.

Headings

As of version 1.52, the heading functionality is available by adding the `styleselect` or `formatselect` button group to both toolbars (TinyMCE and custom). The available formatting options are heading 1 to heading 6 and paragraph.

> **i Note**
>
> These changes won't affect applications already using a `sap.ui.richtexteditor.RichTextEditor` with an added `styleselect` or `formatselect` option.

For more information, see the sample.

Custom Button Order

As of version 1.54, you can customize the position of the button groups in the custom toolbar. This can be done by supplying a value for the new `customToolbarPriority` property of the button group. The groups in the toolbar are placed in ascending order respective to their `customToolbarPriority` values.

> **Example**

```
RichTextEditor.setButtonGroups([{
                        name: "font-style",
                        visible: true,
                        row: 0,
                        priority: 10,
                        customToolbarPriority: 30,
                        buttons: [
                                "bold", "italic", "underline", "strikethrough"
                        ]
                }, {
                        name: "text-align",
                        visible: true,
                        row: 0,
                        priority: 20,
                        customToolbarPriority: 20,
                        buttons: [
                                "justifyleft", "justifycenter", "justifyright",
"justifyfull"
                        ]
                }, {
                        name: "clipboard",
                        visible: true,
                        row: 1,
                        priority: 10,
                        customToolbarPriority: 10,
                        buttons: [
                                "cut", "copy", "paste"
                        ]
                }]).
```

Related Information

Supported Library Combinations [page 26]

sap.ui.table

Table-like controls, mainly for desktop scenarios.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Row Virtualization

To improve rendering and memory performance, only the number of rows that are visible on the user interface are created internally for `sap.ui.table.Table` (this is called "row virtualization"). For example, if the table has enough space to render 20 rows, exactly 20 rows are created internally.

> **Example**
>
> Imagine an OData service with 10 million entries. Keeping 10 million row controls, either inside or outside the DOM, is simply not technically feasible for most client devices. Also, a single table row usually contains additional controls inside each cell, such as `sap.m.Labels` or `sap.ui.unified.Currency` controls. The number of used control instances would then be multiplied by the number of columns for every row. Instead of creating all of these SAPUI5 controls for every data entry, the table virtualizes the rows, and in this way only a limited set of control instances are created for each table control instance.

OData Model

OData as a RESTful protocol provides a specified and a generalized way to access back-end services via HTTP requests. The SAPUI5 OData model provides a stable module for querying OData services via the network. All `sap.ui.table.*` controls fully support data bindings over OData V2. Since OData services can hold millions of entries, and these entries have to be loaded somehow to the client and rendered, the `sap.ui.table.*` controls implement advanced paging mechanisms based on the underlying `ODataListBinding` and `ODataTreeBinding`. The SAPUI5 OData bindings take care of all necessary back-end requests to retrieve the currently-needed data entries. This is done as efficiently as possible with the minimum amount of back-end requests.

Related Information

OData V2 Model [page 883]
API Reference: `sap.ui.table`
Tables: Which One Should I Choose? [page 2286]
Supported Library Combinations [page 26]
Browser and Platform Support [page 20]

sap.ui.vk

The `sap.ui.vk` library provides controls for the visualization and manipulation of 2D and 3D models in your application.

> **i** Note
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Native Viewport

The Native Viewport control (`sap.ui.vk.NativeViewport`) provides a rendering canvas for 2D images loaded into the Viewer application.

API Reference / Sample

- sap.ui.vk.NativeViewport in the *API Reference* in the Demo Kit

Overview

The Native Viewport control (`sap.ui.vk.NativeViewport`) loads files supported natively by the browser into a viewing area using standard HTML and CSS. Viewing of the loaded images is enhanced with standard VIT pan and zoom gesture support.

The `sap.ui.vk.NativeViewport` control can occupy all or part of the user interface.

Details

Loading Images into the Native Viewport

The `NativeViewport` control supports the loading of images that are natively supported by the browser.

Currently, the `NativeViewport` control supports the loading of the following file formats:

- JPG / JPEG
- PNG
- GIF
- TIFF
- BMP
- SVG

You must specify the extension of the file you want to load. Otherwise, the file will not load.

If it can be detected, a console message will be displayed when a problem occurs during file loading.

Gesture Handling in the Native Viewport

Gestures such as pan and zoom are captured and processed by the sap.ui.vk.Loco library. A new NativeViewport instance is initialized with an instance of the sap.ui.vk.Loco library attached, so that gestures can be captured and processed.

Keyboard Shortcuts in the Native Viewport

In addition to mouse and touch gestures, keyboard shortcuts are available for navigating around the NativeViewport instance:

Table 127: Native Viewport keyboard shortcuts

Keyboard shortcut	Function
`DIRECTIONAL ARROW`	Pan the image.
+	Zoom into the image.
-	Zoom out of the image.

Constraints

- The `NativeViewport` control only loads 2D images. For loading 3D models, use the `sap.ui.vk.Viewport` control.
- Currently, the `NativeViewport` control does not support interactive SVG files; that is, any links in an SVG file will not work in the Native Viewport.

Related Information

- Viewport [page 2480]

Scene Tree

The `sap.ui.vk.SceneTree` control presents a hierarchical view of the nodes in a given scene.

API Reference / Sample

- sap.ui.vk.SceneTree in the *API Reference* in the Demo Kit
- Step 4 - Viewport with Scene Tree in the Demo Kit

Overview

The Scene Tree control (`sap.ui.vk.SceneTree`) presents a hierarchical view of all the nodes in a given scene.

Prerequisites

Before a scene's nodes can appear in the Scene Tree, the Scene Tree control needs to be connected to:

- a `ViewStateManager` object, which handles the visibility and selection states of a scene
- a `Scene` object, so that the Scene Tree knows the nodes to display in the hierarchy

Details

The Scene Tree displays the collection of nodes in a scene. You can add a Scene Tree to your application by using the `sap.ui.vk.SceneTree` control. The Scene Tree is also avaliable in the composite `sap.ui.vk.Viewer` control.

Before a scene's nodes can appear in the Scene Tree, you will need to set up two-way data binding between the Scene Tree and the Viewport in your application. Note that you can only bind to one Viewport instance at a time.

The following sections outline the Scene Tree's selection and visibility behavior in more detail.

Hiding or Viewing the Scene Tree

When using the `sap.ui.vk.Viewer` control, the Scene Tree is enabled by default, which means that a `SceneTree` instance is created. You can toggle the visibility of the Scene Tree using the Scene Tree button in the Toolbar.

The following table outlines what the Scene Tree button looks like in these different states:

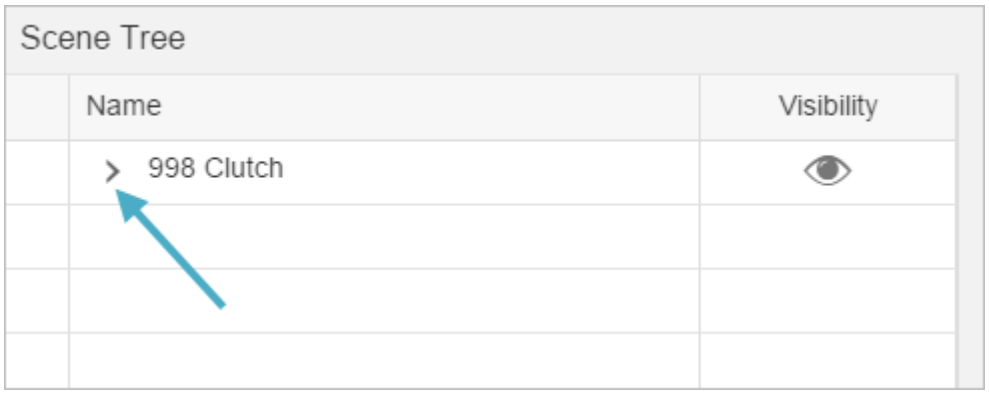
Table 128: Scene Tree button states

Scene Tree button state	Description
	Scene Tree is shown (this is the default state).
	Scene Tree is hidden.
	Scene Tree is disabled, or, if a 2D file is loaded into the Viewer application.

Selecting Nodes in the Scene Tree

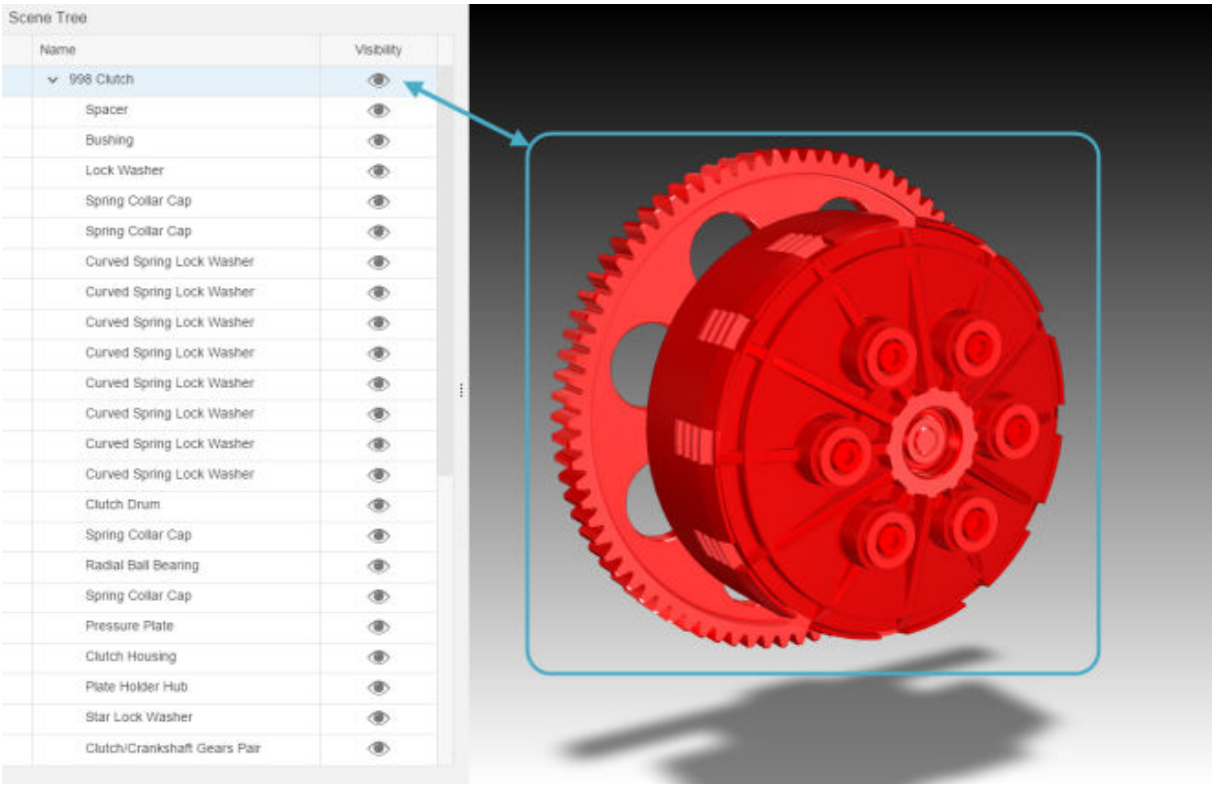
By default, the Scene Tree is collapsed so that only the top-level nodes in the Scene are displayed.

You can expand the Scene Tree by clicking on the > icon next to a node to display that node's child nodes.



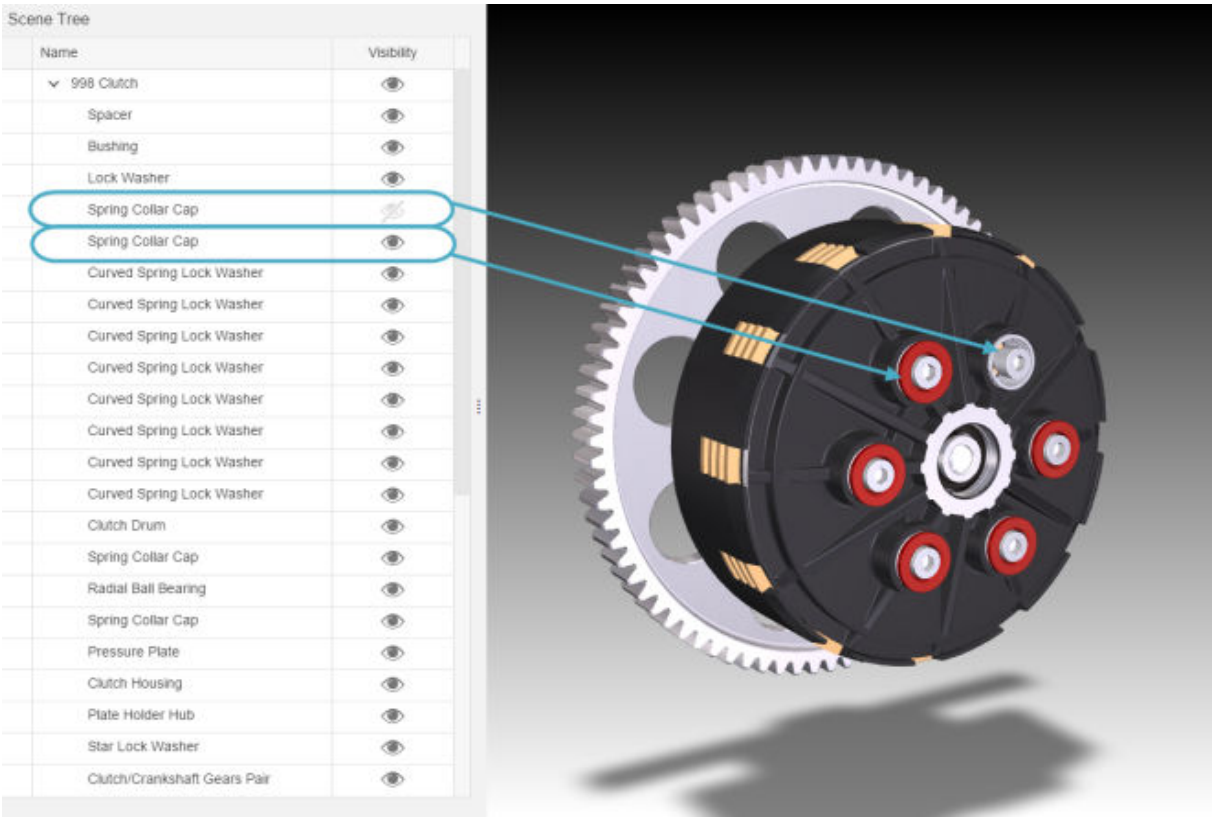
Selecting a node in the Scene Tree will highlight that node in the Viewport, and vice versa. If you select a node in the Viewport that is currently hidden in the Scene Tree, the Scene Tree will expand automatically to display the selected node in the Scene Tree's hierarchy.

Selecting a node in the Scene Tree that has child nodes will result in the node and its child nodes being selected in the Viewport.

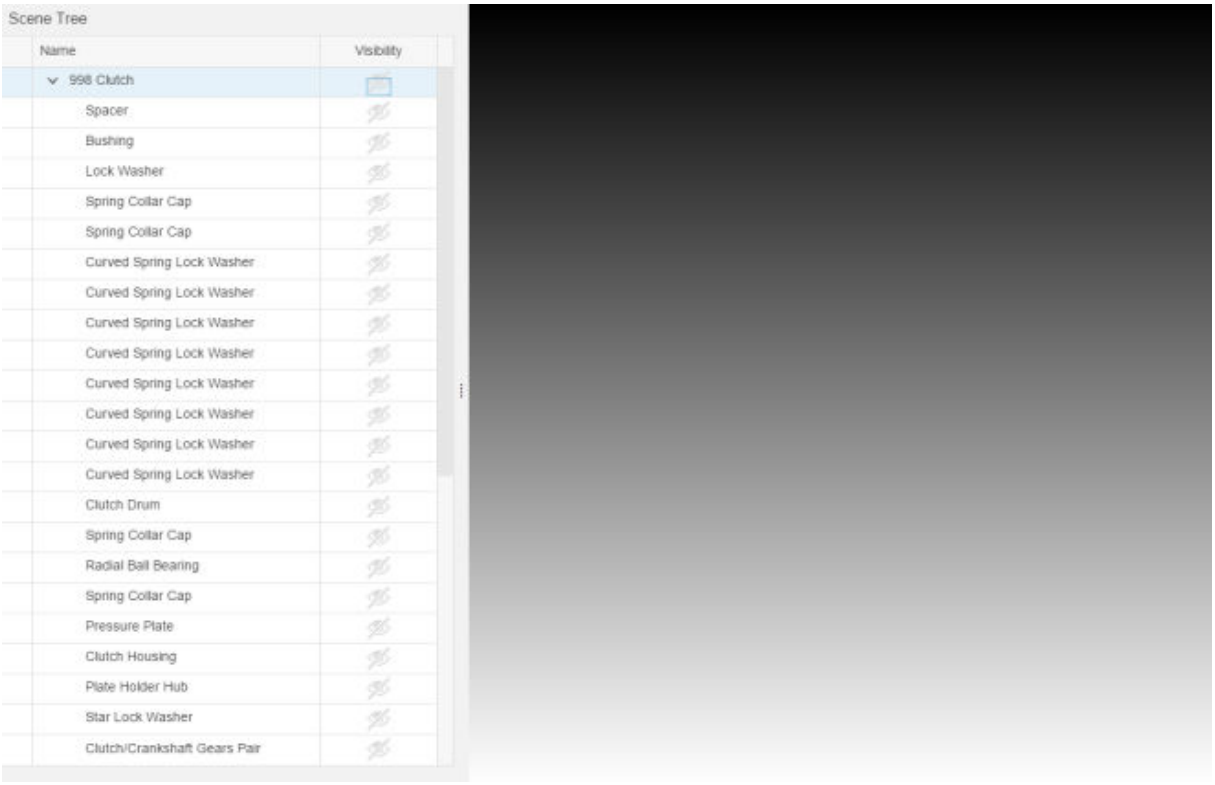


Toggling Node Visibility in the Scene Tree

You can toggle between displaying or hiding a node from view in the Viewport by clicking the 'eye' icon next to a node in the Scene Tree. The following example shows a comparison of two similar nodes, but with one node being hidden from view.



If you toggle the visibility for a node that has child nodes, the visibility change will apply to the child nodes as well. The following example shows the root node being set to hidden, resulting in all of its child nodes being hidden as well.



Related Information

- Viewport [page 2480]

Step Navigation

The `sap.ui.vk.StepNavigation` control enables navigation and activation of procedures and steps contained in a single 3D scene.

API Reference / Sample

- sap.ui.vk.StepNavigation in the *API Reference* in the Demo Kit
- Step 5 - Viewport with Scene Tree and Step Navigation in the Demo Kit

Overview

The VDS file format supports pre-authored animations, also known as steps. Steps are grouped into procedures.

The Step Navigation (`sap.ui.vk.StepNavigation`) control allows you to see which animations are available in a 3D scene. You can play an animation for an individual step, or play all the steps for a single procedure from start to finish.

Details

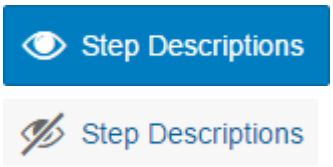
Step Navigation User Interface

The Step Navigation control can be divided into two parts: the top part of the control contains buttons and drop down menus that allow you to choose a procedure, set the play mode, and play, pause, or skip a step. Below these buttons, thumbnails of each step are displayed. You can hover over a thumbnail to show the name of the step, and click on a thumbnail to play the step.

The following table describes the functionality of each of the buttons or dropdown menus in the Step Navigation control:

Table 129: Step Navigation buttons and dropdown menus

Button / Menu name	Icon	Description
Previous		Plays the previous step in the procedure.
Play		Plays the currently selected step in the procedure.
Pause		Pauses the animation at the current step.

Button / Menu name	Icon	Description
Step Descriptions		Shows or hides descriptions associated with the steps. By default, Step Descriptions are hidden. When step descriptions are shown, a dialog appears for the step that is being played. The dialog contains the step's name, and below it, any description associated with the step.
Next		Plays the next step in the procedure.
Play Options		By default, there are three options: • *Play Step* - play the currently selected step only. • *Play All* - play all the steps in the procedure. • *Play Remaining* - play the currently selected step, and all the subsequent steps.
Procedures	(the dropdown menu located to the right of the Play Options dropdown menu)	Select which procedure in the scene to play. Depending on the loaded model, you may have one or more procedures to select from.

> **i Note**
>
> Individual buttons and menus cannot be disabled.

In addition to using these buttons and drop down menus to select which step or steps to play, you can also click on the thumbnail for a step, and then click the *Play* button to play the animation associated with that step.

While a step is playing, you can still rotate, pan, or zoom in the scene.

Hiding or Displaying the Step Navigation Control

You can hide or view the Step Navigation control using the *Step Navigation* button in the toolbar (if `sap.ui.vk.Toolbar` is used in the application).

By default, Step Navigation is hidden from view. You can change this so that when a scene with animation is loaded into your application, then the Step Navigation control is displayed.

Table 130: Step Navigation button states

Step Navigation button state	Description
	Step Navigation is shown

Step Navigation button state	Description
	Step Navigation is hidden

Viewer

This control is intended to help application developers include simple 3D visualisation capability in their application by connecting, configuring and presenting the essential Visualisation Toolkit controls a single composite control.

API Reference / Sample

- sap.ui.vk.Viewer in the *API Reference* in the Demo Kit
- Step 1 - 3D Viewer With Single File Loading in the Demo Kit
- Step 2 - 3D Viewer With Multiple File Loading in the Demo Kit

Overview

The Viewer (`sap.ui.vk.Viewer`) control is intended to help application developers include simple 3D visualisation capability in their application by connecting, configuring and presenting the essential Visual Interaction toolkit (`sap.ui.vk`) controls into a single, composite control.

Most applications require the simplest possible visualisation capability, which includes the loading of a single file into the application, and the initialisation of a 3D Viewport. Consumers of such an application also expect to be able to pan, zoom, or rotate (if applicable) the scene, as well as receive visual cues when they select an object in the 3D Viewport. The Viewer control aims to make it as easy as possible for an application developer to include 3D visualisation capability by connecting, configuring and presenting some of the core Visual Interaction toolkit controls into a single, composite control.

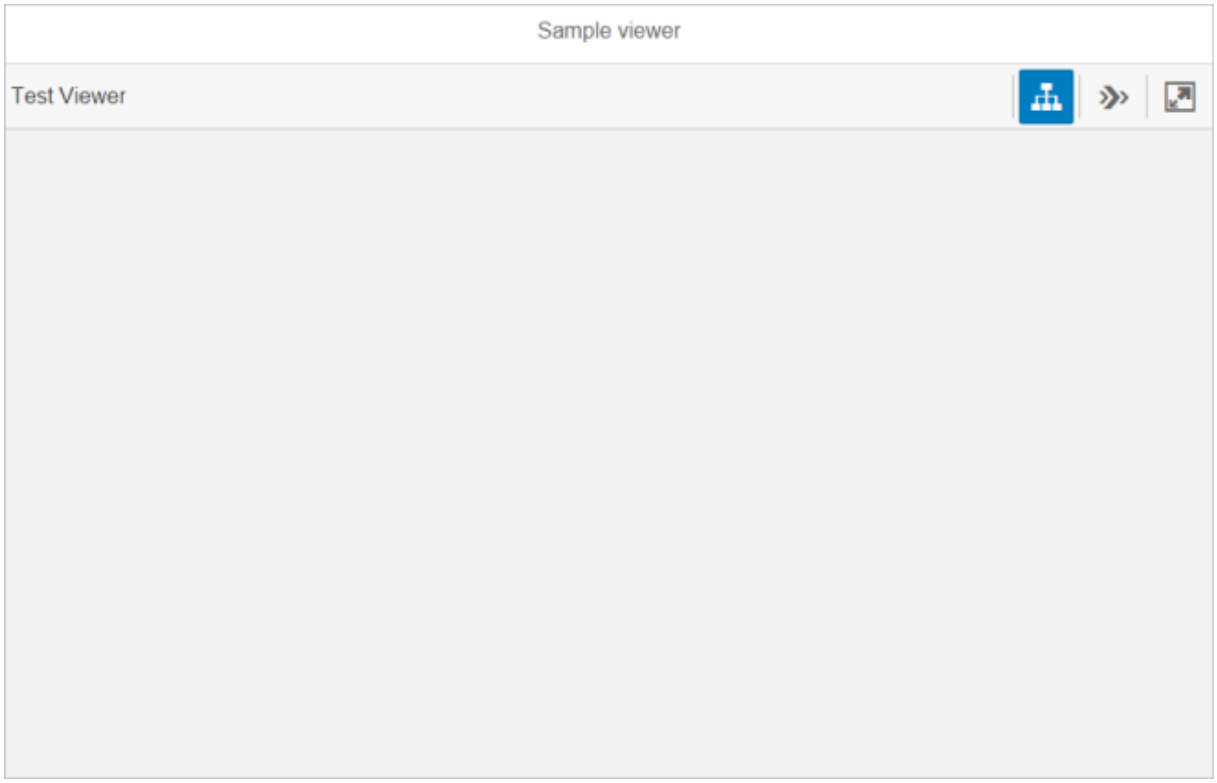
Prerequisites

- You must be able to create or obtain files in the SAP VDS (`.vds`) format to be able to display them in the Viewer.

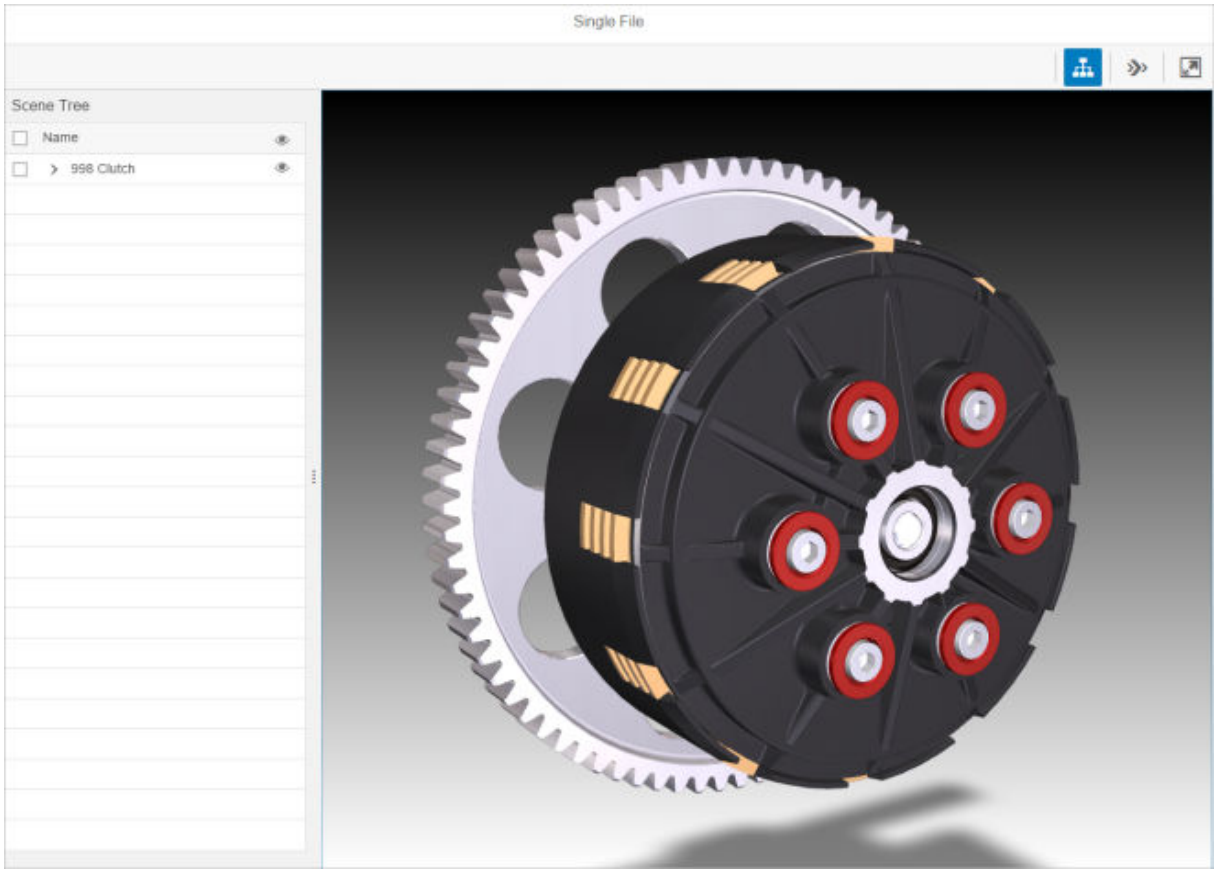
Details / Features

Default Layout

The following figure shows the default Viewer layout after it is initialized:



The following figure shows the Viewer layout with a file loaded and all available controls enabled and visible:



Enabling or Disabling Features

By default, instances of the Toolbar, Scene Tree, and Step Navigation controls are created after a Viewer is initialized. You can change the following Viewer properties so that instances of these features are not created/ dislayed:

- `enableSceneTree`
- `enableStepNavigation`
- `enableToolbar`

'Disabling' the Toolbar will result in no Toolbar being shown in the application.

If the Toolbar feature is enabled, 'disabling' the Scene Tree or Step Navigation feature will result in their respective toolbar buttons being grayed out.

Constraints

- The Viewer control is only capable of loading the SAP VDS file (`.vds`) format.
- The Viewer control is designed for simple consumption scenarios. If an application demands more control of the elements in the Viewer or extended functionality, then the developer may need to compose their own 'Viewer' from the various Visual Interaction toolkit controls.

Related Information

- Step 1: 3D Viewer With Single File Loading [page 609]
- Step 2: 3D Viewer With Multiple File Loading [page 617]

Viewport

The `sap.ui.vk.Viewport` control provides a rendering canvas for the 3D elements of a loaded scene/loaded scenes.

API Reference / Sample

- sap.ui.vk.Viewport in the *API Reference* in the Demo Kit
- Step 3 - Standalone Viewport in the Demo Kit

Overview

The Viewport control's primary function is to provide a rendering surface for all or part of a loaded scene. The Viewport can occupy all or part of the user interface.

Prerequisites

When initializing a Viewport instance, you must:

- attach it to a `GraphicsCore` instance, which handles the rendering of loaded scenes
- attach it to a `Scene` object so that the Viewport knows what scene to render

Details

Loading Scenes Into the Viewport

The Viewport supports the loading of the VDS file format.

File formats that are natively supported by browsers are loaded by the Native Viewport control.

Node Visibility in the Viewport

The Viewport can be connected to a `ViewStateManager` object to handle the selection and visibility states of nodes in the scene. This means is that when selecting a node in the scene, that node will be highlighted. In

addition, if the Viewport is connected to a Scene Tree control, the selection of a node in the Viewport will highlight the associated item in the Scene Tree.

The visibility state of a node can only be affected when the Viewport is connected to a Scene Tree. Nodes can be hidden or displayed in the Viewport using the Scene Tree functionality.

Gesture Handling in the Viewport

Gestures such as pan, zoom, and rotate are captured and processed by the `sap.ui.vk.Loco` library. The Viewport should be initialized with an instance of the `sap.ui.vk.Loco` library attached, in order for gestures to be captured and processed.

Keyboard Shortcuts in the Viewport

In addition to using mouse and touch gestures to navigate the scene in a Viewport, keyboard shortcuts are available.

Table 131: Viewport keyboard shortcuts

Keyboard Shortcut	Function
`DIRECTIONAL ARROW`	Rotate the scene.
`SHIFT` + `DIRECTIONAL ARROW`	Pan the scene.
`+`	Zoom into the scene.
`-`	Zoom out of the scene.

Constraints

- The Viewport control loads 3D models in the SAP VDS (`.vds`) format only. Some 2D image formats can be loaded by the `sap.ui.vk.NativeViewport` control. See Native Viewport.

Related Information

- Scene Tree [page 2470]
- Native Viewport [page 2468]

sap.uxap

This library includes controls associated with the `ObjectPage`.

> **i Note**
>
> The following sections only provide additional information for some of the controls. For a complete list of all controls and their documentation, see the API Reference and the Samples.

Related Information

Supported Library Combinations [page 26]
API Reference: `sap.uxap`

Object Page Layout

The `ObjectPageLayout` control provides a layout that allows apps to easily display information related to a business object.

Overview

The `ObjectPageLayout` layout is composed of a header (title and content), an optional anchor bar and block content wrapped in sections and subsections that structure the information.

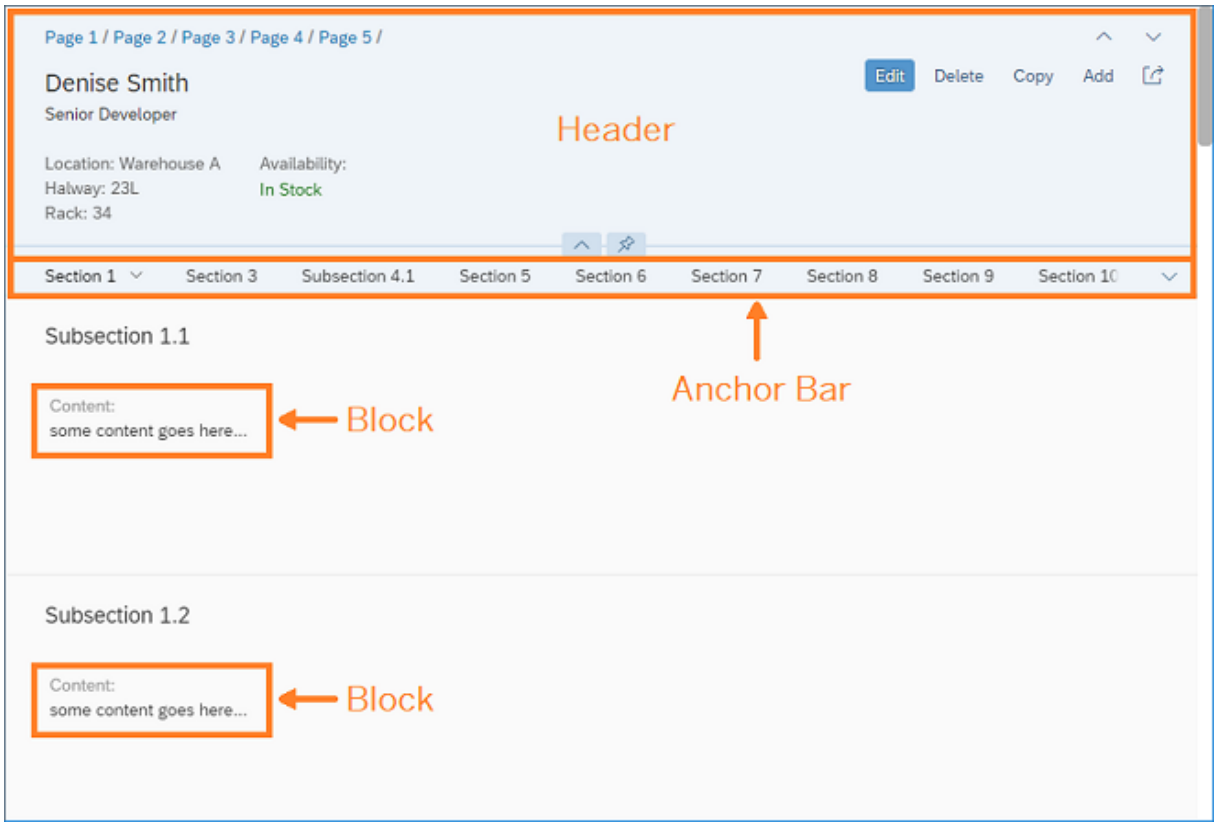


Figure 355: `ObjectPageLayout` main structure

Header (Title and Content)

The `ObjectPageLayout`'s header consists of two parts: header title and header content.

The header title is the topmost part of the `ObjectPageLayout` that is always visible. Its main purpose is to display the name of the represented business object along with actions that the user can perform.

The header content scrolls along with the content of the page until it disappears (collapsed header). When scrolled back to the top it becomes visible again (expanded header). It contains all the additional information of the object.

Here is how the header title and header content are defined in both views:

XML view:

```
<ObjectPageLayout id="ObjectPageLayout">
     <headerTitle>
        <ObjectPageHeader objectTitle="John Smith">
            <actions>
                <ObjectPageHeaderActionButton icon="sap-icon://edit"
text="Edit" />
                <ObjectPageHeaderActionButton icon="sap-icon://save"
text="Save" />
            </actions>
        </ObjectPageHeader>
    </headerTitle>
    <headerContent>
        <m:Label text="Personal description"/
>
        <m:Text value="some KPI info"/>
    </headerContent>
</ObjectPageLayout>
```

JavaScript view:

```
// Create a header title, set the objectTitle property and add some action
buttons
var oHeaderTitle = new sap.uxap.ObjectPageHeader();
oHeaderTitle.setObjectTitle("John Smith");
oHeaderTitle.addAction(new sap.uxap.ObjectPageHeaderActionButton({icon: "sap-
icon://edit", text: "Edit"}));
oHeaderTitle.addAction(new sap.uxap.ObjectPageHeaderActionButton({icon: "sap-
icon://save", text: "Save"}));
oObjectPage.setHeaderTitle(oHeaderTitle);
// Add arbitrary header content
oObjectPage.addHeaderContent(new sap.m.Label({text:"Personal description"}));
oObjectPage.addHeaderContent(new sap.m.Text({value:"some KPI info"}));
```

Sections, Subsections, Blocks

The content of the page that appears below the header is composed of blocks structured into sections and subsections.

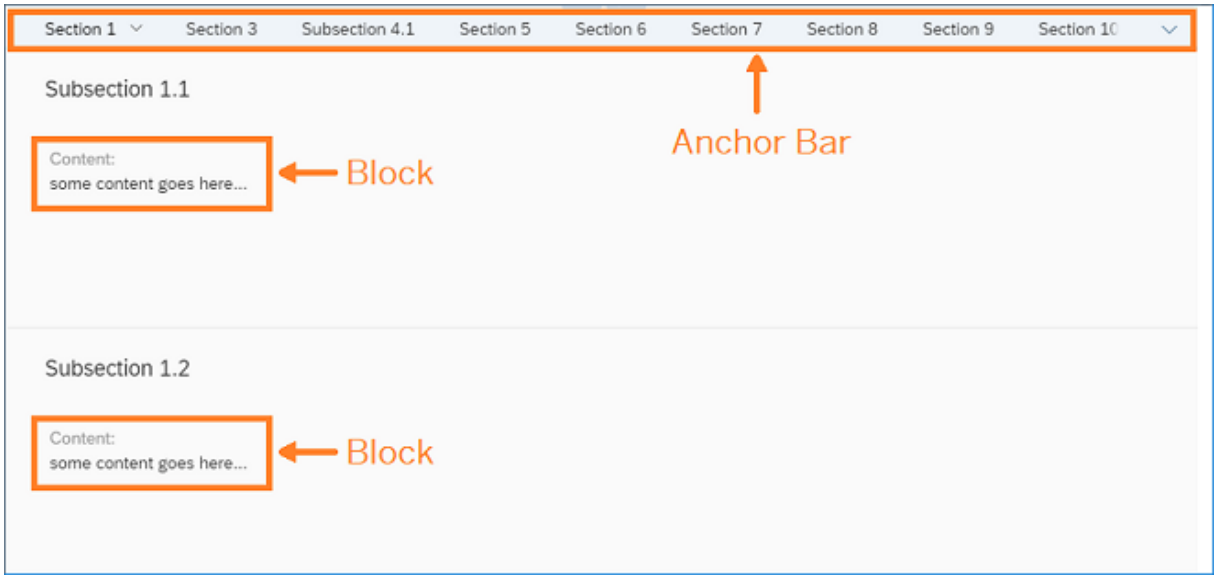


Figure 356: Blocks structured into sections and subsections

The blocks hold the actual app content, while the purpose of the sections and subsections is to define grouping.

A subsection groups together a set of blocks (under a common title), while a section groups together a set of subsections (under a common title).

The grouping enables the control to automatically create an internal menu (anchor bar) that shows the titles of the sections and subsections as separate anchors. The user can select them to scroll to the respective section or subsection content.

Here are some examples of how sections are initialized in both views:

XML view:

```
<ObjectPageLayout id="ObjectPageLayout" >
        <sections>
            <ObjectPageSection title="Payroll" >
                <subSections>
                    <ObjectPageSubSection title="sub payroll title">
                        <blocks>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>
        </sections>
    </ObjectPageLayout>
```

JavaScript view:

```
var oSubSection1 = new sap.uxap.ObjectPageSubSection({title:"sub payroll
title"});
    var oSection1 = new sap.uxap.ObjectPageSection({title:"Payroll"});
    oSection1.addSubSection(oSubSection1);
    oObjectPage.addSection(oSection1);
```

Layout Options

The `subSectionLayout` property provides information on how all the underlying subsections arrange the blocks within their internal grid. The default is set to `titleOnTop`, which arranges the blocks content in columns where the first column is below the section and subsection titles.

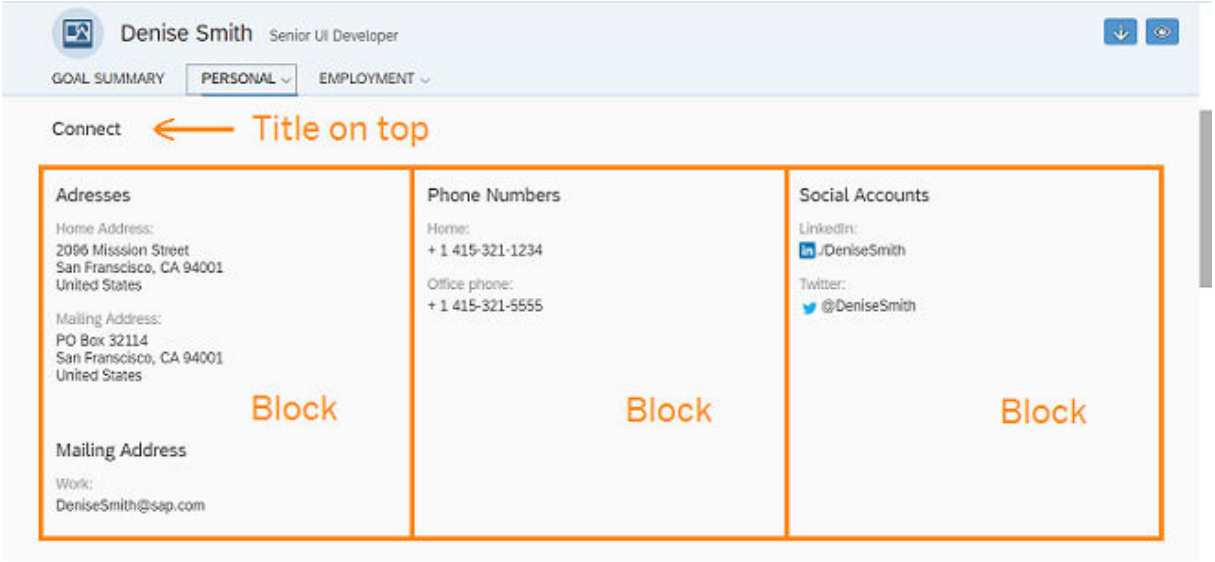


Figure 357: Blocks Content Arranged in Columns with Section and Subsection Titles Displayed on Top

Additionally, a second layout named `titleOnLeft` arranges the blocks content from the second column, leaving the first one for section and subsection titles only.

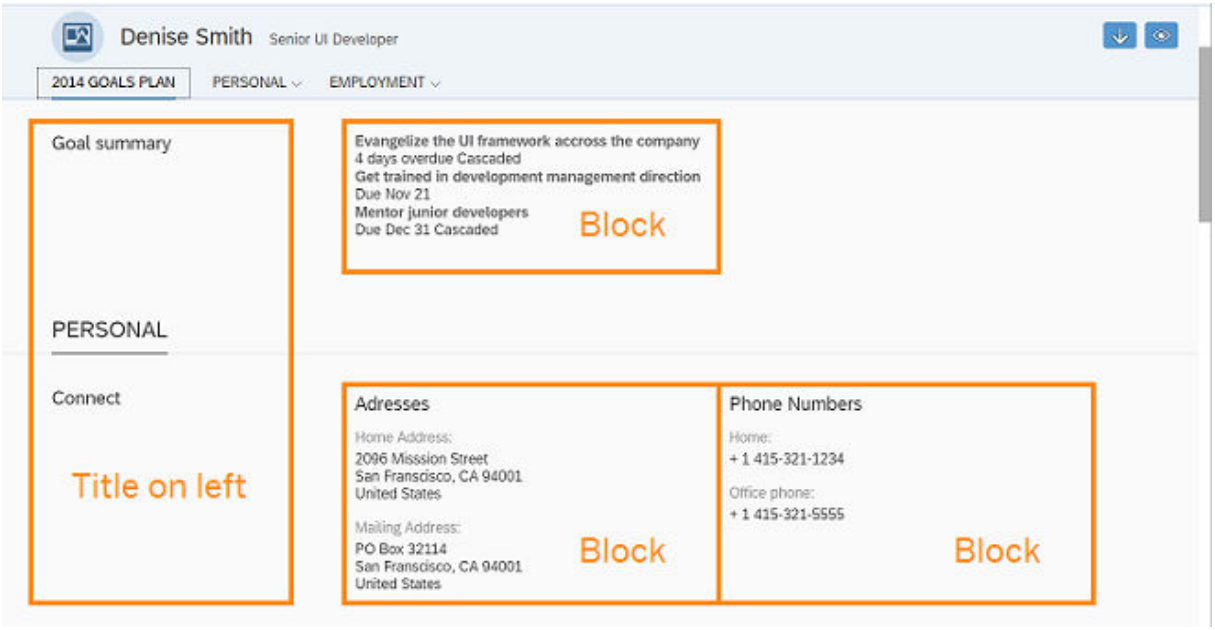


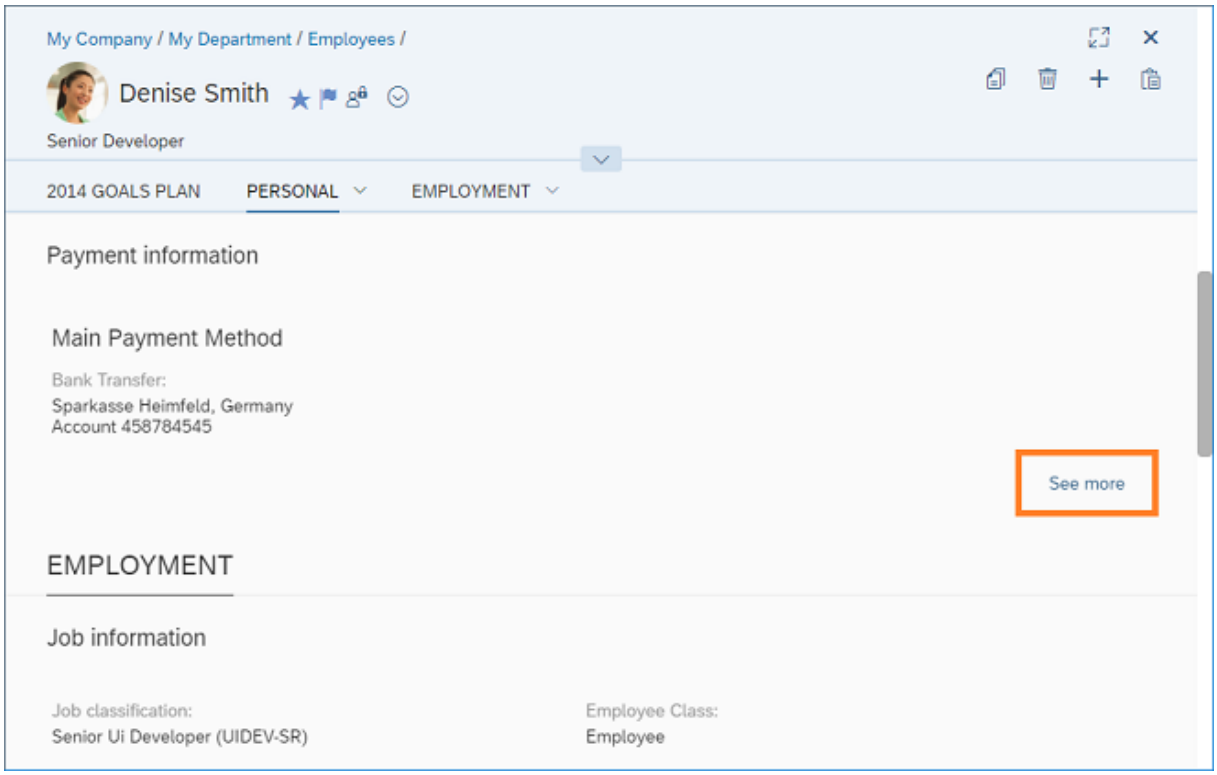
Figure 358: Blocks Content Arranged in Columns with Section and Subsection Titles Displayed on the Left

Here is how this property is set in the XML view:

```
<ObjectPageLayout id="ObjectPageLayout" subSectionLayout="titleOnTop">
        <sections>
```

```
            <ObjectPageSection title="Payroll" >
                <subSections>
                    <ObjectPageSubSection title="sub payroll title">
                        <blocks>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>
        </sections>
    </ObjectPageLayout>
```

The `moreBlocks` aggregation of `sap.uxap.ObjectPageSubSection` allows you to specify blocks to be displayed only after the user clicks the internally created *See more* button:



The *See more* button is only displayed for subsections that contain one of the following:

- Visible blocks in the `moreBlocks` aggregation
- Visible `BlockBase` block that has the `showSubSectionMore` property set to `true`

Additional Rules for Displaying Sections and Subsections

The following additional rules are internally applied to display the contents of the `ObjectPageLayout` correctly. Each rule is applied to the output of the preceding rule.

1. If the subsection content is empty (contains no blocks), it is not displayed (no anchor is displayed for that subsection in the anchor bar and no title is displayed in the page body).

2. If the section content is empty (contains no subsections), it is not displayed (no anchor is displayed for that section in the anchor bar and no title is displayed in the page body).
3. If a section without a title contains only one subsection with a title, the section gets the title of the subsection (`SectionTitle=SubsectionTitle` and `SubsectionTitle=NULL`).
4. If the `ObjectPageLayout` contains only one section, no anchor bar is displayed.
5. If there are more than one sections, the first one will not have a title.

Lazy Loading

The lazy loading mechanism allows you to load data only when the blocks are inside or near the visible area on the screen. This way, you avoid sending too many requests from the start of the page loading.

If you want to use lazy loading, all your blocks must be based on `BlockBase`, otherwise they are loaded as normal SAPUI5 components.

Lazy loading is disabled by default. To enable it, set the `enableLazyLoading` property to `true`:

```
<ObjectPageLayout id="ObjectPageLayout" enableLazyLoading="true">
```

The `ObjectPageLayout` control ensures that only the visible blocks and those adjacent to them have loaded their data, but not the entire page. As the user scrolls or navigates within the page, new data is requested as needed.

> **i Note**
>
> Setting `enableLazyLoading` to `true` after the `ObjectPageLayout` has been instantiated does not work, as all bindings will have been resolved by then.

Related Information

Object Page Headers [page 2488]
Anchor Bar [page 2502]
Object Page Blocks [page 2504]
Creating Blocks [page 2507]
Object Page Scrolling [page 2509]
API Reference: `sap.uxap.ObjectPageLayout`
API Reference: `sap.uxap.ObjectPageSection`
API Reference: `sap.uxap.ObjectPageSubSection`
API Reference: `sap.uxap.BlockBase`

Object Page Headers

The `sap.uxap.ObjectPageLayout` control has two types of header - classic header and dynamic header.

Overview

The `sap.uxap.ObjectPageLayout` control implements the snapping header concept. This means that the upper part of the header (Header Title) always stays visible, while the lower part (Header Content) can scroll out of view.

The common pattern is that the most important information describing the object, such as title, subtitle, and image is in the Header Content area when the header is expanded, and moves to the Header Title area when the header is collapsed (snapped - its lower part scrolled out of view).

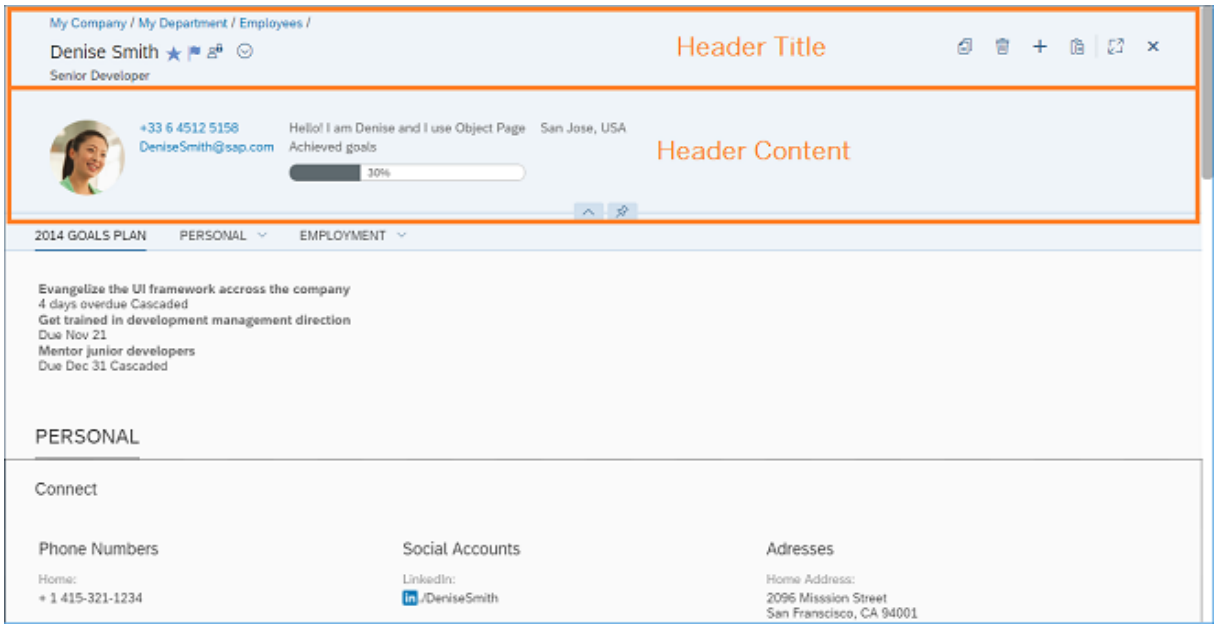


Figure 359: `sap.uxap.ObjectPageLayout` header in expanded state

The following image shows the collapsed (snapped) header is where the Header Content area is scrolled out and not visible, and the main information is visible in the Header Title area.

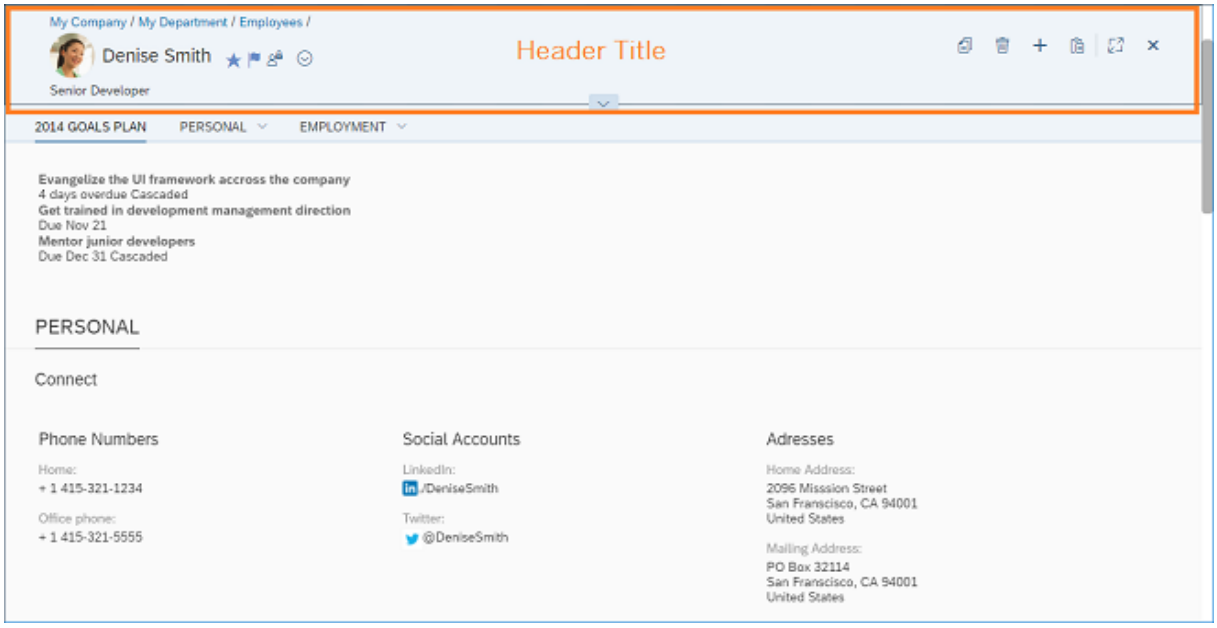


Figure 360: `sap.uxap.ObjectPageLayout` Header in Collapsed (snapped) State

The Classic Header

Up to version 1.52, only `sap.uxap.ObjectPageHeader` could have been used to build up the `sap.uxap.ObjectPageLayout` header.

Header area	`sap.uxap.ObjectPageLayout` aggregation	App must provide:
Header Title	`headerTitle` (0..1)	An instance of the `sap.uxap.ObjectPageHeader` control
Header Content	`headerContent` (0..n)	An array of arbitrary controls. **i Note** `sap.uxap.ObjectPageHeaderContent` control is used internally to display the controls.

The app provides an instance of `sap.uxap.ObjectPageHeader` as the value of the `headerTitle` aggregation, and arbitrary controls as the value of the `headerContent` aggregation (which are internally added to an instance of the `sap.uxap.ObjectPageHeaderContent` control).

The Dynamic Header (Since Version 1.52)

As of version 1.52, a new `sap.uxap.ObjectPageDynamicHeaderTitle` control can be used to build a dynamic header for `sap.uxap.ObjectPageLayout`.

Header Area	`sap.uxap.ObjectPageLayout` aggregation	App Must Provide:
Header Title	`headerTitle` (0..1)	An instance of the `sap.uxap.ObjectPageDynamicHeaderTitle` control
Header Content	`headerContent` (0..n)	An array of arbitrary controls. **i Note** `sap.uxap.ObjectPageDynamicHeaderContent` control is used internally to display the controls.

Again, the app provides an instance of `sap.uxap.ObjectPageDynamicHeaderTitle` as the value of the `headerTitle` aggregation and a list of controls for the `headerContent` aggregation (`sap.uxap.ObjectPageLayout` uses internally `sap.uxap.ObjectPageDynamicHeaderContent` to lay out the controls).

Related Information

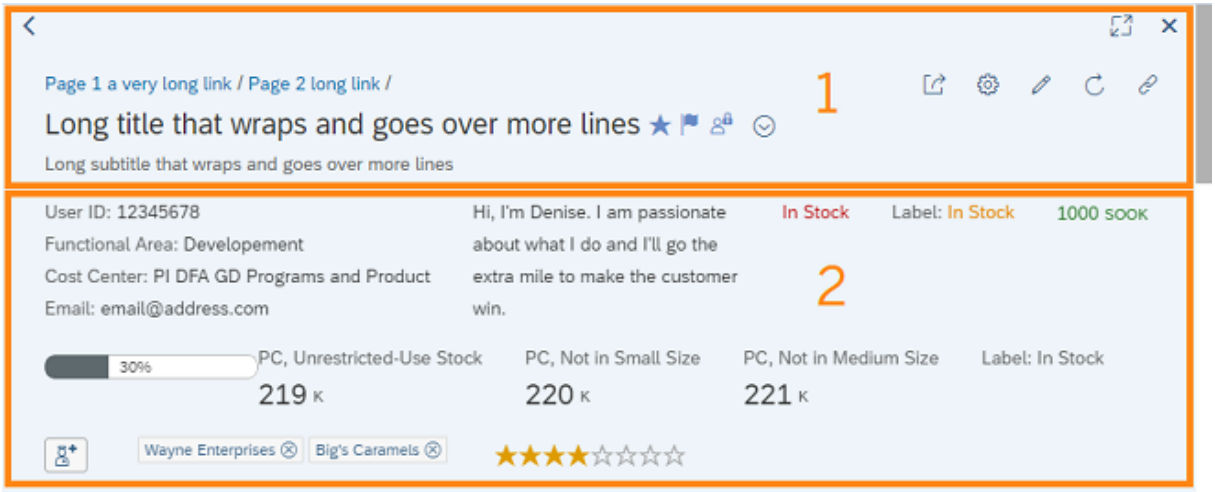
API Reference: `sap.uxap.ObjectPageLayout`
API Reference: `sap.uxap.ObjectPageHeader`
API Reference: `sap.uxap.ObjectPageDynamicHeaderTitle`

Object Page Classic Header

Overview of the structure and features for `sap.uxap.ObjectPageLayout`'s classic header.

Main Structure

The `ObjectPageHeader` control consists of two main parts - Header Title and Header Content.



1. Header Title (`headerTitle`) - Displayed at the top of the header and always remains visible above the scrollable content of the page. It contains the title and most prominent details of the object.
2. The Header Content (`headerContent`) - Scrolls along with the content of the page until it disappears (collapsed header). When scrolled back to the top it becomes visible again (expanded header). It contains all the additional information of the object.

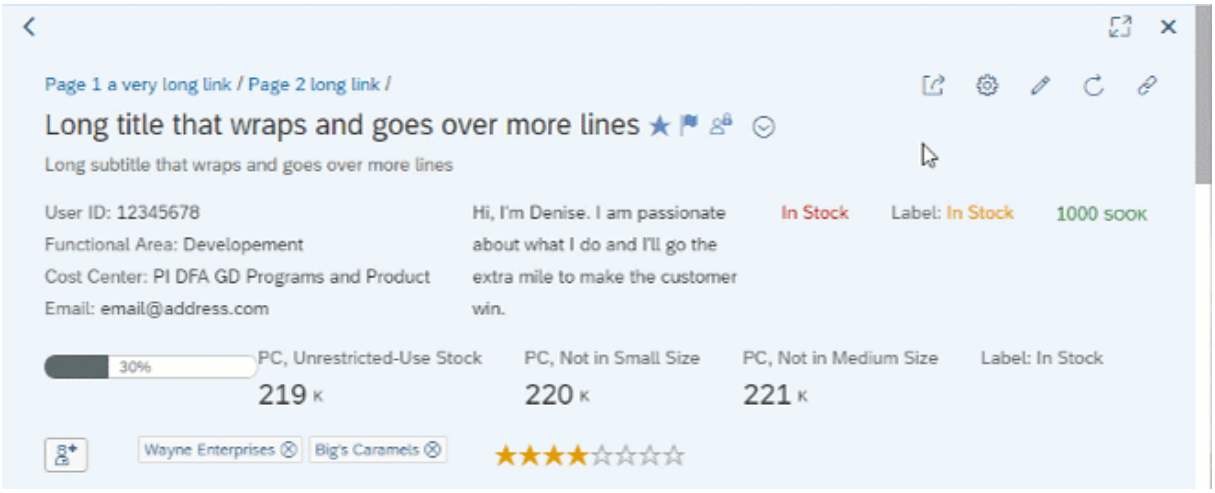


Figure 361: Collapsing and Expanding the Header

Header Title

This part of the header contains the basic information of the object.

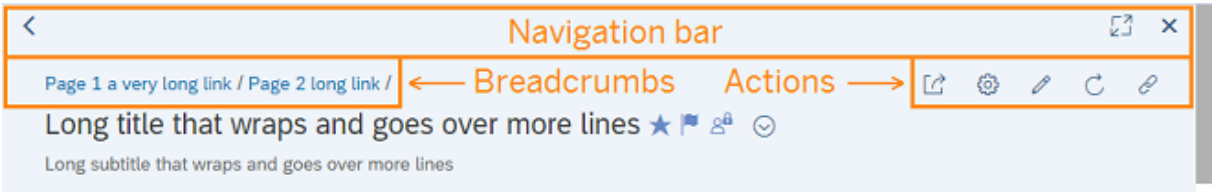


Figure 362: Navigation bar, Breadcrumbs and Actions in the Header Title

The top area in the Header Title is for the navigation bar (`navigationBar`). It contains the top-most element (`sap.m.Bar`) and provides the option to have a *Back* button for returning to the previous selection and navigation actions on the opposite side.

The area below the navigation bar is reserved for breadcrumbs navigation on one side (`breadcrumbs`) and actions on the other (`actions`). The actions are declared as `sap.uxap.ObjectPageHeaderActionButton` instances.

Figure 363: Title with Optional Indicators and Subtitle

You can set title (`objectTitle`) and subtitle (`objectSubtitle`). On larger screens the subtitle is displayed next to the title. After a certain breakpoint, the subtitle moves below the title.

You can display several optional indicators right after the title. They are considered part of the title and when there is not enough space they are wrapped and moved to more lines along with the title text.

Table 132: Optional indicators in the title

Optional Indicator	API Properties
Favorite	`markFavorite`
Flagged	`markFlagged`
Locked	`markLocked`
Unsaved changes	`markChanges`
Selector	`showTitleSelector`

> **i Note**
>
> Keep in mind that *Locked* and *Unsaved changes* are mutually exclusive. If both of them are set to be visible, only the *Locked* state is displayed.

You can show and hide both the markers (*Favorite* and *Flagged*) simultaneously with the `showMarkers` boolean property.

Figure 364: Object Image in Circle and Square Shapes

You can add an icon-sized image before the title by defining the image location in the `objectImageURI` property. You can set the text used for the `Alt` and `Tooltip` attributes of the image with the `objectImageAlt` property. To set the shape to `Circle` or `Square`, use the `objectImageShape` property.

You can control whether the image, title, subtitle, and actions are always visible or visible only when the header is collapsed (snapped).

> → Tip
>
> To build a custom `headerTitle`, you can extend the `ObjectPageHeader` class and then use any control in the `headerTitle` aggregation. The `ObjectPageLayout`, however, needs correct values for the `objectImageURI` / `objectImageShape` and `headerDesign`, as those properties are important for the `headerContent` in order to style it properly.

Header Content

The second part of the header is the Header Content. This is an aggregation of controls that are displayed in a float layout underneath the Header Title. The controls that can be used in the `headerContent` aggregation are the standard SAPUI5 controls and they are automatically styled to fit the current header style.

With the use of the `sap.uxap.ObjectPageHeaderLayoutData` class, you can specify for each control used in the `headerContent` aggregation, whether it's visible on small, medium or large-sized layouts, what width it takes and whether it has a visual separator displayed before and/or after itself.

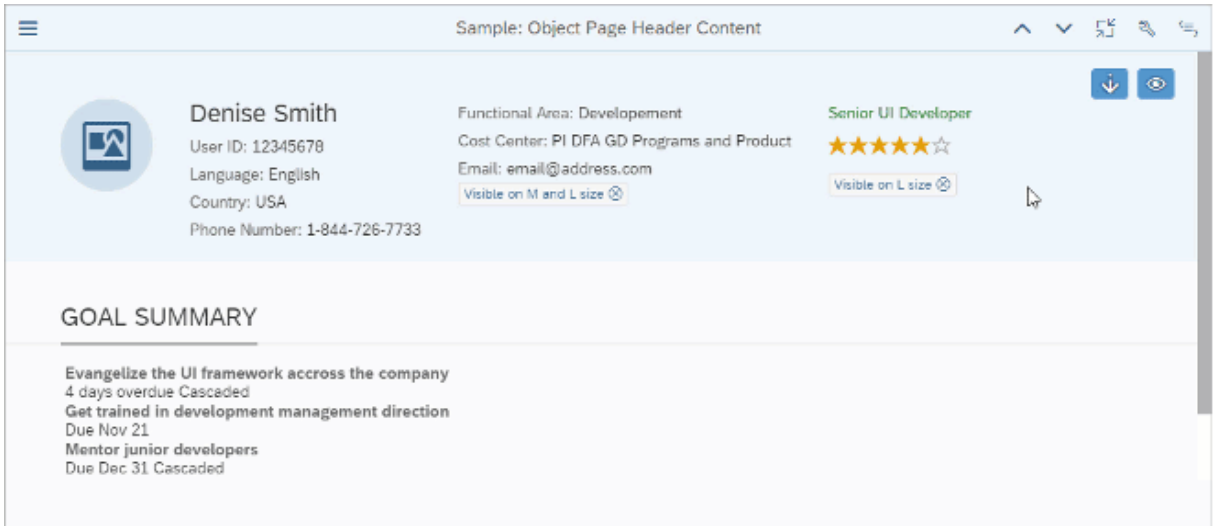


Figure 365: Header Content with `sap.uxap.ObjectPageHeaderLayoutData` - large, middle and small-sized layout

Related Information

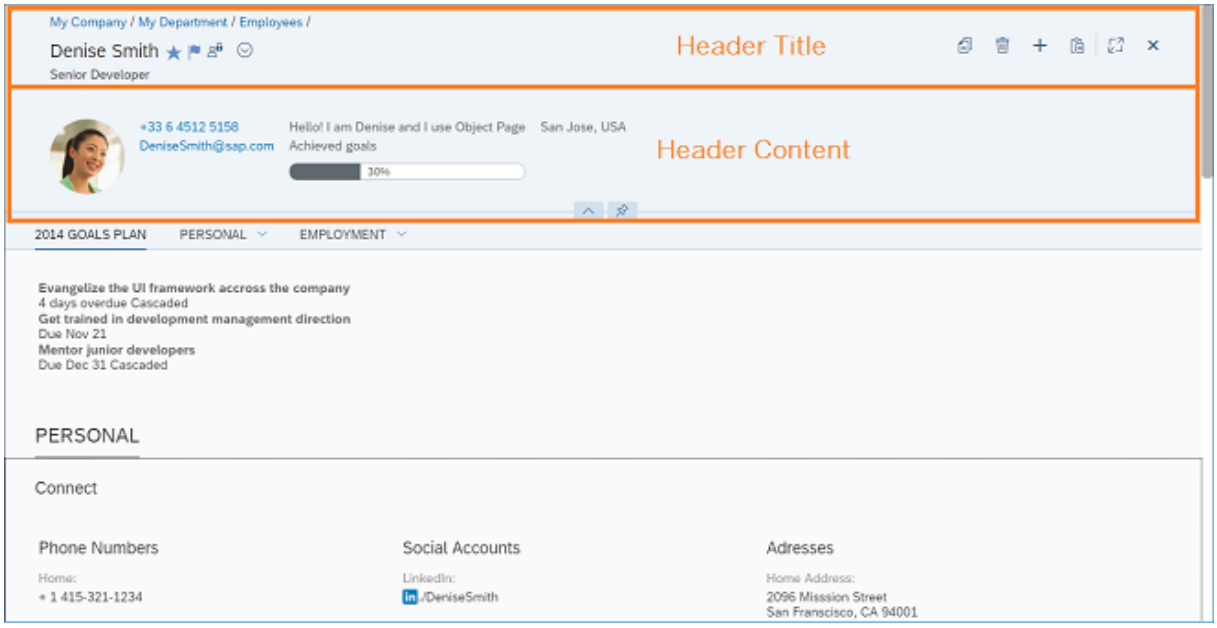
API Reference: `sap.uxap.ObjectPageHeader`
API Reference: `sap.uxap.ObjectPageHeaderLayoutData`
API Reference: `sap.uxap.ObjectPageLayout`
Object Page Headers [page 2488]
Object Page Headers Comparison [page 2497]

Object Page Dynamic Header

Overview of the structure and features for `sap.uxap.ObjectPageLayout`'s dynamic header.

The `sap.uxap.ObjectPage`'s dynamic header is flexible and provides general-purpose aggregations that allow you to build a custom header layout.

It consists of two parts - Header Title and Header Content.



The upper part of the Header Title is reserved for `breadcrumbs` navigation. The opposite side of this upper area is occupied by the `navigationActions` after a certain breakpoint.

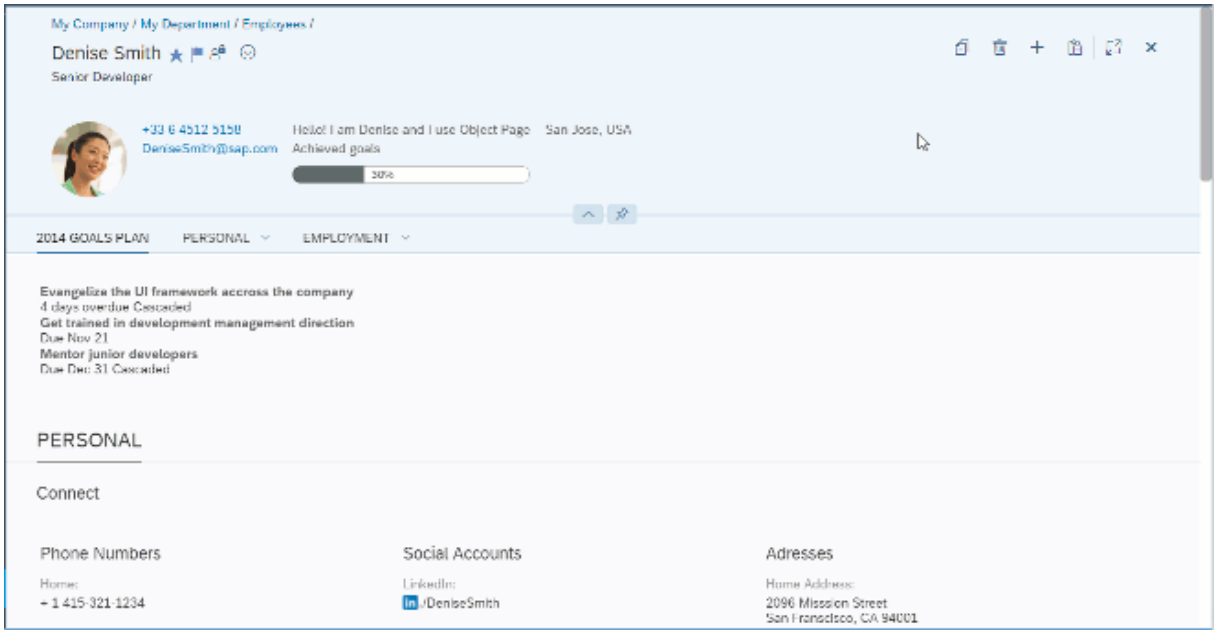


Figure 366: Breadcrumbs and Navigation Actions in the Header Title

The Header Title area can be clicked/tapped to expand/collapse the dynamic header. Whenever the feature is enabled (`toggleHeaderOnTitleClick` is set to `true`), an arrow button is positioned either below the Header Content (when header is expanded) or below the Header Title (when header is collapsed). The expand/collapsed state of the header can be toggled by either clicking on the Header Title area, or the arrow button.

When hovering over the arrow button or the Header Title area, both areas are highlighted indicating to the user that an action can be taken.

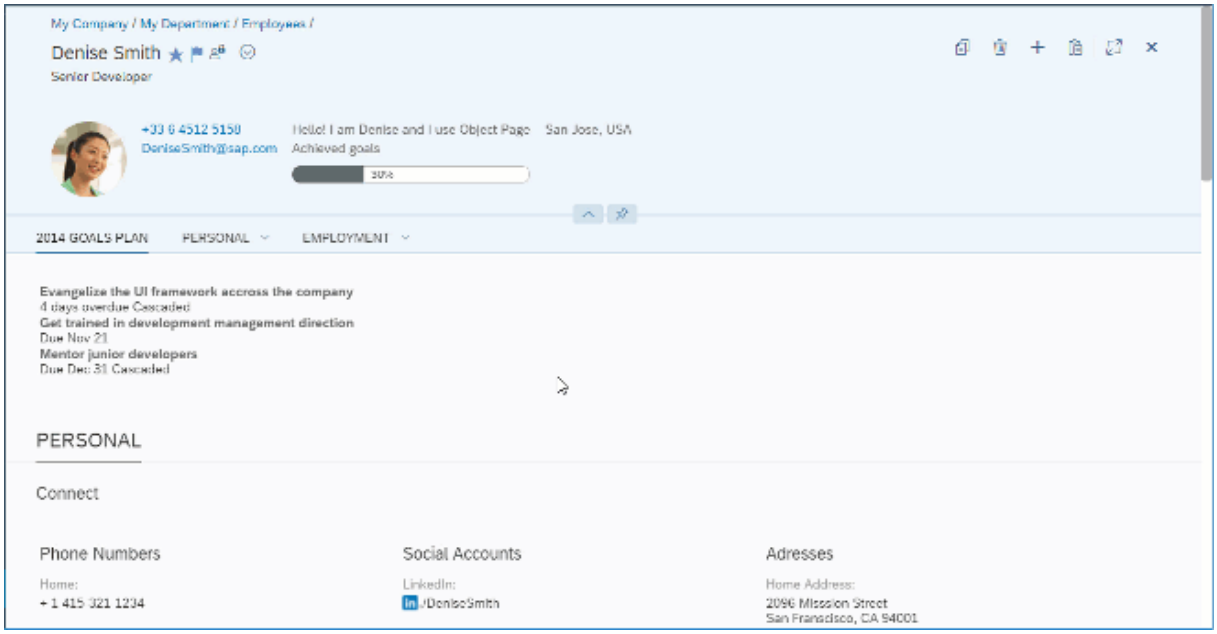


Figure 367: Expanding/Collapsing the Header Using Arrow Button and Title Click

The Header Content can be pinnable (`headerContentPinnable` is set to `true`). When the feature is enabled, a pin toggle button is available allowing the header content to remain expanded when scrolling the page.

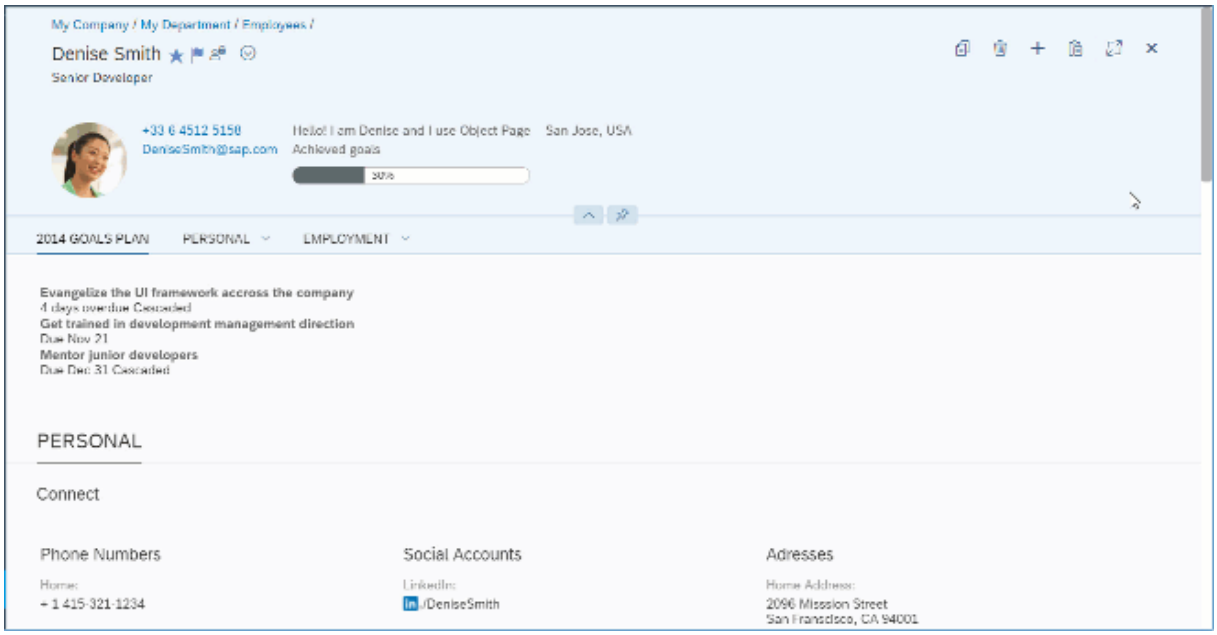


Figure 368: Pinning the Header to Remain Expanded when Scrolling the Page

Header Title

To implement the dynamic header, the app developer needs to provide an instance of the `sap.uxap.ObjectPageDynamicHeaderTitle` control for the `headerTitle` aggregation of the `sap.uxap.ObjectPageLayout` control.

The `sap.uxap.ObjectPageDynamicHeaderTitle` extends `sap.f.DynamicPageTitle`. It can hold any control and displays the most important information regarding the object that will always remain visible when scrolling.

Header Content

To populate the header content area, provide an array of desired controls to the `headerContent` aggregation of the `sap.uxap.ObjectPageLayout` control. `sap.uxap.ObjectPageLayout` uses internally `sap.uxap.ObjectPageDynamicHeaderContent` to layout the controls.

Related Information

API Reference: `sap.uxap.ObjectPageDynamicHeaderTitle`
API Reference: `sap.uxap.ObjectPageLayout`
Object Page Headers [page 2488]
Object Page Headers Comparison [page 2497]

Object Page Headers Comparison

This section explains the differences and similarities between the two types of header of the `sap.uxap.ObjectPageLayout` control.

Table 133: Summary of the controls used in the classic and the dynamic header

Header Area	Classic Header	Dynamic Header
Title	`sap.uxap.ObjectPageHeader`	`sap.uxap.ObjectPageDynamicHeaderTitle`
Content (controls are used internally)	`sap.uxap.ObjectPageHeaderContent`	`sap.uxap.ObjectPageDynamicHeaderContent`

Differences between the classic and the dynamic header

The classic header title is largely semantic, meaning that it has properties, such as `objectTitle`, `objectSubtitle` and `objectImageURI`. It has a very specific layout based on these properties.

On the contrary, the dynamic header title is general-purpose. It doesn't have any properties describing the represented object, but rather several aggregations, such as `heading` and `content`, which the app can use to display any information and build any layout. This requires more work by the app developer, but also more

flexibility. The new header uses internally `sap.m.OverflowToolbar` for the implementation of the `actions` aggregation, which allows actions to have priority, grouping, and other `sap.m.OverflowToolbar` features.

The main difference between the classic and dynamic header content is that the dynamic header has the Pin functionality, allowing the user to prevent it from scrolling out of view.

> **i Note**
>
> The controls, comprising the dynamic header title and header content, extend the `sap.f.DynamicPage` title and header controls. They are adapted for the `ObjectPageLayout` use case, but essentially they provide the same functionality.

Table 134: Relation between the `sap.uxap.ObjectPageLayout` dynamic header controls and the `sap.f.DynamicPage` controls:

Layout Control	Header Title	Header Content
`sap.uxap.ObjectPageLayout`	`sap.uxap.ObjectPageDynamicHeaderTitle`	`sap.uxap.ObjectPageDynamicHeaderContent`
`sap.f.DynamicPage`	`sap.f.DynamicPageTitle`	`sap.f.DynamicPageHeader`

Similarities between the classic and the dynamic header

Both header title controls have the `actions` aggregation, intended for buttons that perform actions on the represented object.

Both header content controls have the `content` aggregation.

Features exclusive to the classic or the dynamic header

Some `ObjectPageLayout` features associated with the behavior of the classic header are considered legacy (although technically not deprecated), and have more robust counterparts for the dynamic header.

Similarly, the dynamic header comes with a set of features (apart from its general structure) that are exclusive to itself, and are not taken into account in the use case of the classic header.

Table 135: Overview of features exclusive to the classic header (all being `ObjectPageLayout` properties with the exception of the `sap.uxap.ObjectPageHeaderLayoutData` class):

Features Exclusive to the Classic Header	Description
`showTitleInHeaderContent`	Determines whether the title, image, markers and `selectTitleArrow` are displayed in the Header Content area.

Features Exclusive to the Classic Header	Description
`isChildPage`	Determines whether the page is a child page and renders it with a different design. Child pages have an additional (darker/lighter) stripe on the left side of their header content area.
`alwaysShowContentHeader`	Determines whether Header Content will always be expanded on desktop.
`showEditHeaderButton`	Determines whether an *Edit* button will be displayed in the Header Content.
`sap.uxap.ObjectPageHeaderLayoutData`	The `sap.uxap.ObjectPageHeaderLayoutData` can only be set on `headerContent` items for the classic header use case.

Table 136: Overview of features exclusive to the dynamic header (all being `ObjectPageLayout` properties):

Features Exclusive to the Dynamic Header	Description
`headerContentPinnable`	Determines whether the Header Content area can be pinned. When set to true, a pin button is displayed within the Header Content area. The pin button allows the user to make the Header Content always visible at the top of the page above any scrollable content.
`toggleHeaderOnTitleClick`	Determines whether the user can switch between the expanded/collapsed states of the dynamic header by clicking/tapping on the Header Title. If set to `false`, the Header Title is not clickable and the app must provide other means for expanding/collapsing the dynamic header, if necessary.
`preserveHeaderStateOnScroll`	Preserves the current header state when scrolling. For example, if the user expands the header by clicking on the title and then scrolls down the page, the header will remain expanded.
`toggleHeaderOnTitleClick`	When the feature is enabled, arrow buttons below the Header Content appear, the Header Title and the arrow buttons can be clicked/tapped for collapsing/expanding the header and there is additional visual indication while hovering over the Header Title area or the arrow buttons.

If a legacy property, for example `showTitleInHeaderContent` is set, but an instance of `sap.uxap.ObjectPageDynamicHeaderTitle` is used for the `headerTitle` aggregation (which will be paired internally with an instance of `sap.uxap.ObjectPageDynamicHeaderContent` for the header content), this property will be ignored.

Similarly, if `toggleHeaderOnTitleClick` is set, but the classic title is used (`sap.uxap.ObjectPageHeader` passed as the value of the `headerTitle` aggregation), the property will be ignored as this feature is not supported by the classic header title/header content pair.

Which header should I use in my app?

The dynamic header is recommended as it supports advanced features, such as pinning and collapse/expand visual indication.

Here is a sample usage of the dynamic header - the value of the `headerTitle` aggregation in an XML view:

```
<headerTitle>
    <ObjectPageDynamicHeaderTitle primaryArea="Left">
        <breadcrumbs>
            <m:Breadcrumbs currentLocationText="My Profile">
                <m:Link text='My Company' />
                <m:Link text='My Department' />
                <m:Link text='Employees' />
            </m:Breadcrumbs>
        </breadcrumbs>
        <expandedHeading>
            <m:FlexBox wrap="Wrap" fitContainer="true" alignItems="Center">
                <m:Title text="Denise Smith" wrapping="true"
class="sapUiTinyMarginEnd"/>
                <m:FlexBox wrap="NoWrap" fitContainer="true" alignItems="Center"
class="sapUiTinyMarginEnd">
                    <m:ObjectMarker type="Favorite" class="sapUiTinyMarginEnd"/>
                    <m:ObjectMarker type="Flagged"/>
                    <m:Button icon="sap-icon://private" type="Transparent"/>
                    <m:Button icon="sap-icon://arrow-down" type="Transparent"/>
                </m:FlexBox>
            </m:FlexBox>
        </expandedHeading>
        <snappedHeading>
            <m:FlexBox wrap="Wrap" fitContainer="true" alignItems="Center">
                <m:FlexBox wrap="NoWrap" fitContainer="true" alignItems="Center"
class="sapUiTinyMarginEnd">
                    <f:Avatar src="../../sap/f/images/Woman_avatar_02.png"
displaySize="S" class="sapUiTinyMarginEnd"/>
                    <m:Title text="Denise Smith" wrapping="true"
class="sapUiTinyMarginEnd"/>
                </m:FlexBox>
                <m:FlexBox wrap="NoWrap" fitContainer="true" alignItems="Center"
class="sapUiTinyMarginEnd">
                    <m:ObjectMarker type="Favorite" class="sapUiTinyMarginEnd"/>
                    <m:ObjectMarker type="Flagged"/>
                    <m:Button icon="sap-icon://private" type="Transparent"/>
                    <m:Button icon="sap-icon://arrow-down" type="Transparent"/>
                </m:FlexBox>
            </m:FlexBox>
        </snappedHeading>
        <expandedContent>
            <m:Text text="Senior Developer" />
        </expandedContent>
        <snappedContent>
           <m:Text text="Senior Developer" />
        </snappedContent>
        <content>
            <m:OverflowToolbar>
                <m:Button text="KPI 1" class="sapUiTinyMargin"/>
                <m:Button text="KPI 2" class="sapUiTinyMargin"/>
                <m:Button text="KPI 3" class="sapUiTinyMargin"/>
```

```
                <m:Button text="KPI 4" class="sapUiTinyMargin"/>
                <m:Button text="KPI 5" class="sapUiTinyMargin"/>
                <m:Button text="KPI 6" class="sapUiTinyMargin"/>
            </m:OverflowToolbar>
        </content>
        <actions>
            <m:OverflowToolbarButton type="Transparent" icon="sap-icon://copy"/>
            <m:OverflowToolbarButton type="Transparent" icon="sap-icon://
delete"/>
            <m:OverflowToolbarButton type="Transparent" icon="sap-icon://add"/>
            <m:OverflowToolbarButton type="Transparent" icon="sap-icon://paste"/>
        </actions>
        <navigationActions>
            <m:OverflowToolbarButton type="Transparent" icon="sap-icon://full-
screen" tooltip="Enter Full Screen Mode"/>
            <m:OverflowToolbarButton type="Transparent" icon="sap-icon://
decline" tooltip="Close column"/>
        </navigationActions>
    </ObjectPageDynamicHeaderTitle>
</headerTitle>
```

→ Tip

When `sap.uxap.ObjectPageLayout` is given the `sap.uxap.ObjectPageDynamicHeaderTitle`, it loads the `sap.f` library on demand as a lazy dependency. To speed up your app, you should preload the `sap.f` library directly in the SAPUI5 bootstrap along with the other libraries needed for your app.

Example:

```
data-sap-ui-libs="sap.m,sap.uxap,sap.ui.layout,sap.f"
```

This way, the `ObjectPageLayout` will already have `sap.f` loaded and it will not need to fetch it.

Related Information

API Reference: `sap.uxap.ObjectPageHeader`

API Reference: `sap.uxap.ObjectPageDynamicHeaderTitle`

API Reference: `sap.f.DynamicPageTitle`

API Reference: `sap.f.DynamicPageHeader`

Anchor Bar

Displays the titles of the sections and subsections in the `ObjectPageLayout` and allows the user to scroll to the respective content.

Overview

The anchor bar is an automatically generated internal menu that shows the titles of the sections and subsections and allows the user to scroll to the respective section and subsection content.

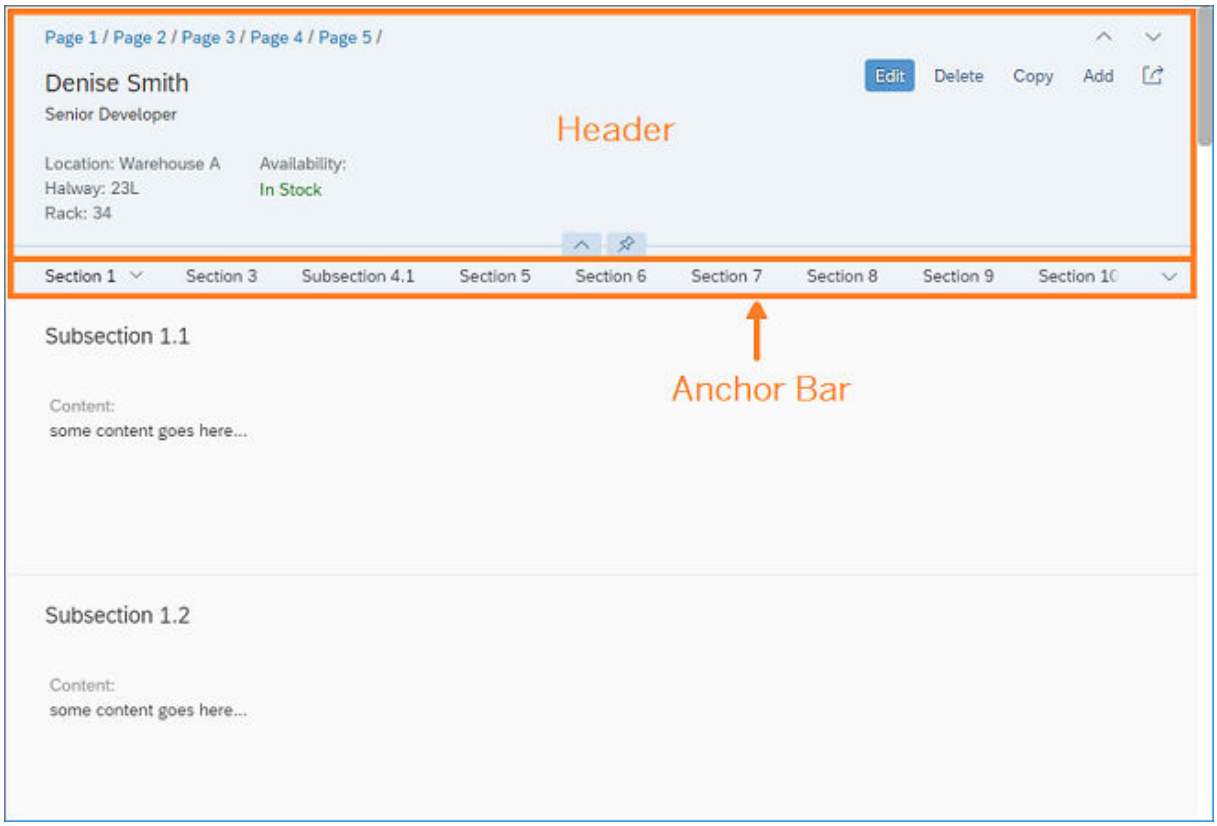


Figure 369: Anchor Bar with Sections and Subsections

When the user scrolls the page content, the anchor bar remains at the top of the screen.

Usage

The anchor bar is displayed by default. You can hide it by using the `showAnchorBar` property:

```
<ObjectPageLayout id="ObjectPageLayout" showAnchorBar="false">
</ObjectPageLayout>
```

In some apps or in some rendering modes of apps, it may not be desirable or necessary to display the anchor bar. In these cases, you can hide it using the `setShowAnchorBar` function.

```
oObjectPage.setShowAnchorBar(false);
```

The `toggleAnchorBar` event is fired by the `ObjectPageLayout` control when the anchor bar is switched from moving to fixed.

An additional option for displaying the anchor bar is to use a `sap.m.IconTabBar` control instead of the default `sap.uxap.AnchorBar`. This is done with the `useIconTabBar` boolean property. If set to `true`, it will also set `showAnchorBar` to `false` in order to avoid showing two navigation bars. This is how it looks in the two views:

XML view:

```
<ObjectPageLayout id="ObjectPageLayout" useIconTabBar="true">
</ObjectPageLayout>
```

JavaScript:

```
oObjectPage.setUseIconTabBar(true);
```

Custom Anchor Bar Buttons

By default, you don't need to specify anything for a `sap.uxap.ObjectPageSectionBase` to have its button included in the anchor bar. At runtime, the `ObjectPageLayout` control creates a button that has the same text as the corresponding section title. However, you may want to use your own control for rendering the anchor bar button instead of the default `sap.m.Button`. You can specify the custom control at `sap.uxap.ObjectPageSectionBase` level, as shown here:

```
<ObjectPageSection>
    <customAnchorBarButton>
        <m:Button text="Employee Info"/>
    </customAnchorBarButton>
</ObjectPageSection>
```

Scrolling is handled automatically, so you don't need to add anything to enable this feature. However, if you want to handle the `press` event differently, then you can add an event handler to the button and can also optionally customize the button.

```
<ObjectPageSection>
    <customAnchorBarButton>
        <m:Button text="Employee Info" press="handleAnchorBarPress"
type="Transparent"/>
    </customAnchorBarButton>
</ObjectPageSection>
```

Here is an example showing the usage of custom controls for the anchor bar buttons:

```
<ObjectPageLayout id="ObjectPageLayout">
    <headerTitle>
        <ObjectPageHeader id="headerExpandedGrid" />
    </headerTitle>
    <sections>
        <ObjectPageSection id="section1" title="Employee Info" >
```

```
            <customAnchorBarButton>
                <!-- this sap.m.ToggleButton will be used in the anchor bar for
navigating to that section -->
                <m:ToggleButton text="Employee Info" />
            </customAnchorBarButton>
        </ObjectPageSection>
        <ObjectPageSection  id="section2" title="Personal Info">
           <customAnchorBarButton>
                <!-- this sap.m.Button will be used in the anchor bar for
navigating to that section -->
               <m:Button type="Accept" text="Personal Info" />
           </customAnchorBarButton>
        </ObjectPageSection>
    </sections>
</ObjectPageLayout>
```

Related Information

API Reference: `sap.uxap.AnchorBar`

Object Page Blocks

The contents of the subsections in the `ObjectPageLayout` control are organized into blocks.

The blocks are used to group the app content that is displayed in the sections and subsections of the `ObjectPageLayout`.

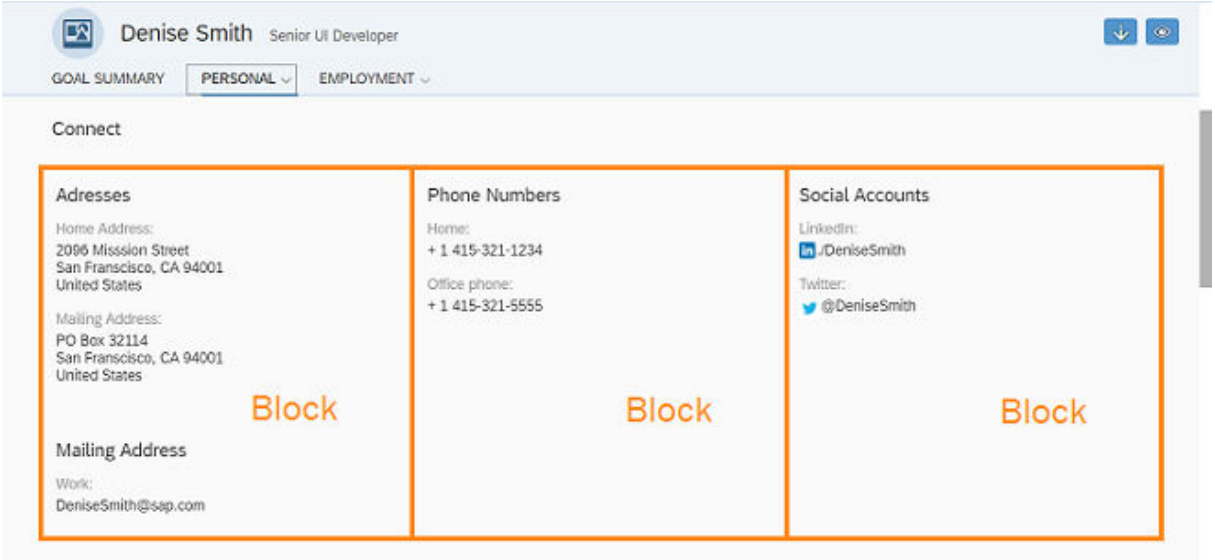


Figure 370: Object Page content grouped in Blocks

To add blocks, use the `blocks` aggregation of `sap.uxap.ObjectPageSubSection`:

```
<ObjectPageLayout id="ObjectPageLayout" subSectionLayout="titleOnTop">
        <sections>
            <ObjectPageSection title="Payroll" >
```

```
                <subSections>
                    <ObjectPageSubSection title="sub payroll title">
                        <blocks>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>
        </sections>
    </ObjectPageLayout>
```

Any control can be a block. However, the blocks that extend `sap.uxap.BlockBase` provide additional features:

- Lazy loading: Only the blocks that are currently displayed and those in their direct proximity are instantiated
- Column layout: Blocks provide information to the subsection only on the width they should be using for an optimal experience

Blocks API & Guidelines

Blocks that are used in an `ObjectPageLayout` have to comply with the following rules regarding their API. They must:

- Extend `sap.uxap.BlockBase`

```
sap.uxap.BlockBase.extend("<BlockName>", {
        metadata: {
         }
     });
```

- Support the modes described in `sap.uxap.ObjectPageSubSectionMode.type` - `Collapsed` and `Expanded`. For each mode, declare its associated view. It is recommended you use the XML view if no templating is needed:

```
sap.uxap.BlockBase.extend("<BlockName>", {
        metadata: {
            views: {
                Collapsed: {
                    viewName: "<collapsedViewName>",
                    type: "XML"
                },
                Expanded: {
                    viewName: "<expendedViewName>",
                    type: "XML"
                }
            }
        }
     });
```

- Come with their own controller (if needed). This controller should just react to the internal events of the block, as the `ObjectPageLayout`'s own controller should only manage the page and its sections and subsections.
- Follow the SAPUI5 naming guidelines: see *Related Information*
- Use the `modelMapping` mechanism to declare distinct model per logical entity.

Example

Let's consider an *Employee Goals* block that displays an employee together with his or her goals.

One *Employee* model for the employee entity:

```
<Text text="{Employee>FirstName}"></Text>
```

One *Goals* model for the goal collections:

```
<List items="{Goals>}">
```

In one backend service, goals may be a navigation property of employees, but in another this may not be the case. For this reason, when implementing the *Employee Goals* block, you should use two distinct models in the block views.

Example

An app wants to use the *Employee Goals* blocks described above. These are therefore embedded into a page that has a model named `ApplicationModel`, in which `Goals` are a navigation property of employees:

```
<EmployeeGoals>
    <mappings>
        <uxap:ModelMapping externalModelName="ApplicationModel"
externalPath="/Employee('121')" internalModelName="Employee" />
        <uxap:ModelMapping externalModelName="ApplicationModel"
externalPath="/Employee('121')/Goals" internalModelName="Goals" />
    </mappings>
</EmployeeGoals>
```

A second app uses the same blocks, but in its service, `Goals` and `Employees` are unrelated entities:

```
<EmployeeGoals>
    <mappings>
        <uxap:ModelMapping externalModelName="ApplicationModel2"
externalPath="/Employee('121')" internalModelName="Employee" />
        <uxap:ModelMapping externalModelName="ApplicationModel2"
externalPath="/Goals" internalModelName="Goals" />
    </mappings>
</EmployeeGoals>
```

`BlockBase` interprets this in the following order:

1. Looks for a model with the name specified in the `externalModelName`.
2. Sets this model on itself with the name specified in the `internalModelName`.
3. Creates a context corresponding to the path.

Note

This model mapping is not mandatory as models used in a view can also be provided by standard SAPUI5 techniques (model inheritance, setModel).

Standard Block Implementation

The standard block implementation is to extend the `sap.uxap.BlockBase` control and inherit the default implementation of `setMode` and rendering. `setMode` in `BlockBase` supports two different ways of building blocks:

- Single view blocks: A single XML view is used for all layout modes. This XML view should be named `<name>.view.xml`.
- Multiple view blocks: Different views are provided for the different layout modes.
  - These views should be added in the `views` section of the block metadata (this section is added by the `BlockBase` class).
  - For each mode, the `BlockBase` class must declare a view name and type:

```
sap.uxap.BlockBase.extend("<BlockName>", {
        metadata: {
            views: {
                Collapsed: {
                    viewName: "<collapsedViewName>",
                    type: "XML"
                },
                Expanded: {
                    viewName: "<expendedViewName>",
                    type: "XML"
                }
            }
        }
    });
```

Related Information

Creating Blocks [page 2507]
API Reference: `sap.uxap.BlockBase`

Creating Blocks

Important points when creating blocks for the `sap.uxap.ObjectPageLayout`

Decide which kind of block to use:

- Single view block if `Collapsed`, `Expanded`, and `Compact` modes are similar and easy to develop with a single view.
- Multiple view blocks if it's easier to provide different views for the different modes.
- Free form if none of the above suit your needs.

Single View Block Creation

- Create the block folder in the sources of the app.
- Add a `BlockName.js` file, which extends `sap.uxap.BlockBase`.

> **i Note**
>
> Naming guideline: The block name should end with the word *Block*.

- Add a `BlockName.view.xml` XML view.
- If needed, add the associated controller: `BlockNameController.controller.js`.

> **i Note**
>
> It's not mandatory to put the `BlockName.js` file and the related XML view in the same folder since you are able to provide the view file path by using the `sap.uxap.BlockBase's` API. However, if no path is provided, the `sap.uxap.BlockBase` will look for an XML view file with a matching name in the same folder where the `BlockName.js` is located.
>
> For example, `sap.uxap.BlockBase` would match `AddressesBlock.js` with `AddressesBlock.view.xml`.

Multiple View Block Creation

- Create the block folder in the sources of the app.
- Add a `BlockName.js` file, which extends `sap.uxap.BlockBase`.
- Declare the views to be used for the different modes in the `views` section of the metadata.
- Add the `Expanded` and `Collapsed` XML views.

> **i Note**
>
> Naming guideline: Name these files as `BlockNameCollapsed.view.xml` and `BlockNameExpanded.view.xml`

- If needed, add the associated controllers. Whether you use the same controller for all views or one controller per view is your decision.

Free Form Block Creation

- Create the block folder in the sources of the app.
- Add a `BlockName.js` file, which extends `sap.uxap.BlockBase`.
- Override the `setMode` method. From here on, all changes are up to you.

Related Information

API Reference: `sap.uxap.BlockBase`

Object Page Scrolling

The object page offers different ways to handle specific scrolling scenarios.

General Scrolling Behavior

By default, the `headerContent` of the `ObjectPage` snaps to the `headerTitle` when scrolling a longer section. Some of the contents of the `headerContent` move to the `headerTitle` and thus always remain visible. You can see an example of this behavior in the screenshot below.

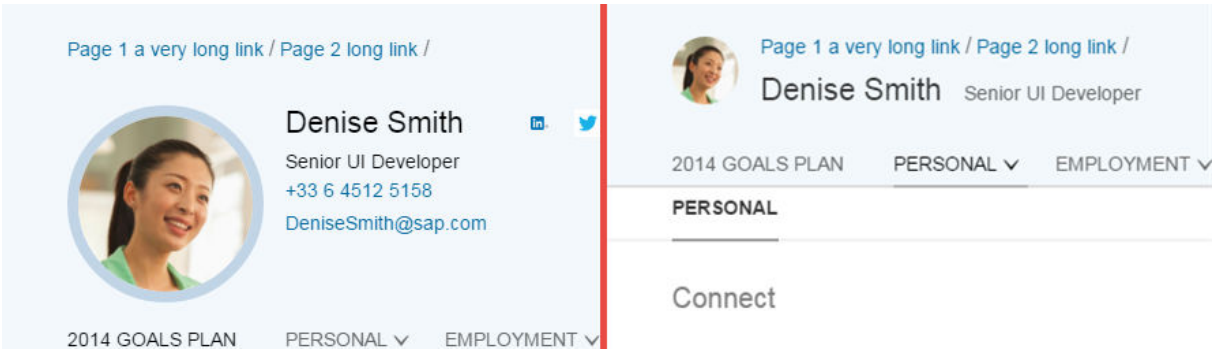


Figure 371: Object Page Header: Expanded (left) / Snapped (right)

This behavior can be altered. Setting the property `alwaysShowContentHeader` to `true` will prevent the `headerContent` from snapping.

> **i Note**
>
> This property only affects desktop environments.

Selected Section

As the user scrolls through the sections, the currently scrolled section is internally set to the `selectedSection` association of `ObjectPageLayout`. The app can also modify its value:

- The app can set which section the page should scroll to upon initial display:

```
<ObjectPageLayout id="ObjectPageLayout" selectedSection="mySectionId">
        <sections>
            <ObjectPageSection title="Payroll" id="mySectionId">
                <subSections>
```

```
                    <ObjectPageSubSection title="sub payroll title">
                        <blocks>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                            <myNameSpace:myBlock/>
                        </blocks>
                    </ObjectPageSubSection>
                </subSections>
            </ObjectPageSection>
        </sections>
    </ObjectPageLayout>
```

- The app can also change the currently scrolled section at runtime:

```
//navigate to a specific section on open
this.oObjectPageLayout = this.getView().byId("ObjectPageLayout");
this.oTargetSection = this.getView().byId("empl");
this.oObjectPageLayout.setSelectedSection(this.oTargetSection)
```

Related Information

Sample: `sap.uxap.sample.ObjectPageState`

Glossary

List of terms used in SAPUI5.

Term	Meaning	Source/ Comments	Link
SAPUI5 ABAP repository	Used to store SAPUI5 apps, components, and libraries; based on the Business Server Page (BSP) repository of the ABAP server. A SAPUI5 application stored in the ABAP repository can be deployed and executed in a browser directly. It is connected to the ABAP transport system.	SAPUI5 only	The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure [page 1507]

Term	Meaning	Source/ Comments	Link
aggregation	An **aggregation** is a special relation between two UI element types. It is used to define the parent-child relationship within the tree structure. The parent end of the aggregation has cardinality 0..1, while the child end may have 0..1 or 0..*. The element's API offers convenient and consistent methods to deal with aggregations (e.g. to get, set, or remove target elements). Examples are table rows and cells, or the content of a table cell.	SAPUI5/ OpenUI5	Essentials [page 691]
association	An **association** is a type of relation between two UI element types which is independent of the parent-child relationship within the tree structure. Directed outgoing associations to a target of cardinality 0..1 are supported. They represent a loose coupling only and are thus implemented by storing the target element instance's ID. The most prominent example is the association between a label and its field.	SAPUI5/ OpenUI5	Essentials [page 691]
ARIA	*WAI-ARIA, the Accessible Rich Internet Applications Suite, defines a way to make Web content and Web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with Ajax, HTML, JavaScript, and related technologies.* (Quote from w3c.org)	`w3c.org`	W3C ARIA
asynchronous (async) processing	In contrast to synchronous processing this processing mode does not keep the browser thread busy but does the processing in the background and continues with the next task. Code can be executed asynchronously and a callback function is triggered when a certain condition is met. Similarly, a file can be loaded asynchronously. Asynchronous processing is highly recommended for performance reasons and to not freeze the UI.	SAPUI5/ OpenUI5	
asynchronous module definition (AMD)	A mechanism for defining a module in a way that modules and their dependencies can be loaded asynchronously.	`requirejs .org`	

Term	Meaning	Source/ Comments	Link
bootstrap	To use the SAPUI5 features in your web page, you have to load and initialize – or "bootstrap" – the SAPUI5 runtime in your HTML page.	SAPUI5/ OpenUI5	Bootstrapping: Loading and Initializing [page 692]
BPMN	*A standard Business Process Model and Notation (BPMN) will provide businesses with the capability of understanding their internal business procedures in a graphical notation and will give organizations the ability to communicate these procedures in a standard manner. Furthermore, the graphical notation will facilitate the understanding of the performance collaborations and business transactions between the organizations.*	`bpmn.org`	BPMN.org
(application) cache buster	A cache buster allows the application to notify the browser to refresh the resources only when the application resources have been changed. Otherwise the resources can always be fetched from the browser's cache. The application cache buster is a special mechanism to extend this function to application resources	SAPUI5/ OpenUI5	Cache Buster for SAPUI5 [page 1132] Application Cache Buster [page 1134]
cache busting: single application resources	Cache busting on the level of a single app, component, or library in the SAPUI5 ABAP repository	SAPUI5 only	The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure [page 1507]
cache busting; multiple application resources	Cache busting on the level of multiple apps, components, or libraries in the SAPUI5 ABAP repository	SAPUI5 only	The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure [page 1507]
clickjacking	Clickjacking, or UI redressing, tricks the user into triggering actions within an application by redirecting clicks. This is done, for example, by using an invisible iFrame which is positioned above a fake UI. When the user clicks on something on the fake UI, the content of the invisible iFrame handles the click.	SAPUI5/ OpenUI5	Browser Security [page 1470]
composite control	Composite controls are intended for reuse within control development and allow you to include existing controls in a complex control.	SAPUI5/ OpenUI5	Standard Composite Controls [page 2226]

Term	Meaning	Source/ Comments	Link
content density	The devices used to run apps that are developed with SAPUI5 run on various different operating systems and have very different screen sizes. SAPUI5 contains different content densities for certain controls that allow your app to adapt to the device in question, allowing you to display larger controls for touch-enabled devices and a smaller, more compact design for devices that are operated by mouse.	SAPUI5/ OpenUI5	Content Densities [page 1142]
control	UI elements that can be used independently. From a developer's point of view, a **control** (e.g. `Button`, `Label`, `TextField`, or `Table`) is the most important artifact. It is an object which controls the appearance and user interaction of a rectangular screen region. It is a special kind of user interface element which can be used as the root of such a tree structure. In this way, it serves as an entry point, especially for rendering. Besides controls, there are also other **non-control elements**, which cannot be used as the root of such a tree structure, but only as a dependent part within it (e.g. `TableRow`, `TableCell`).	SAPUI5/ OpenUI5	More About Controls [page 2252] Samples API Reference Essentials [page 691]
controller	An application unit containing the active part of the application. It is the logical interface between a model and a view, and corresponds to the model view controller (MVC) concept.	SAPUI5/ OpenUI5	Controller [page 807]
Cross-Site Scripting (XSS)	Cross-site scripting is about injecting script code into a web page, which is then executed in the context of the page. Therefore it can access any information which is currently displayed on the screen. Additionally, XSS attacks can access session information contained in cookies, or send new requests to the server within the current session, or even try to exploit browser vulnerabilities to get full access to the machine that the browser is running on.	SAPUI5/ OpenUI5	Cross-Site Scripting [page 1475]

Term	Meaning	Source/ Comments	Link
data binding	A technique that binds two data sources together in order to keep them in sync. All changes in one data source are automatically reflected in the other; the involved layers are the view and the model.	SAPUI5/ OpenUI5	Data Binding [page 815]
data type	**Data types** are first-class entities in the meta model. This allows reuse of types across libraries and extensibility of the type system. The core library (technically, this is the `sap.ui.core` library) already defines a core set of types that can be used in other libraries.	SAPUI5/ OpenUI5	Essentials [page 691]
Demo Kit	The Demo Kit is the SAPUI5 software development kit (SDK). The Demo Kit is your one-stop shop for all information about SAPUI5: documentation, API reference, samples, demo apps.	SAPUI5/ OpenUI5	
diagnostics	A diagnostics window is available in SAPUI5 applications. To open it, use the following shortcut: `CTRL` + `SHIFT` + `ALT` + `S`.	SAPUI5/ OpenUI5	Diagnostics [page 1326]
distribution layer	Contains the control libraries and theme libraries; the SAPUI5 distribution layer is delivered to customers via the MIME repository.	SAPUI5 only	
Document Object Model (DOM)	*The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.* (Quote from w3c.org)	`w3c.org`	W3C DOM
element	A (UI) **element** is the basic building block of our user interfaces; it is a reusable entity with properties, events, methods, and relations. The most important relations are aggregations to other UI elements, and in this way a tree structure of elements can be created.	SAPUI5/ OpenUI5	Essentials [page 691]

Term	Meaning	Source/ Comments	Link
SAP Fiori	SAP Fiori is the user experience (UX) for SAP software that applies modern design principles. SAP solutions are using the SAP Fiori UX to provide a personalized, responsive, and simple user experience.	SAP Fiori	http://www.sap.com/fiori http://help.sap.com/fiori https://experience.sap.com/fiori/
SAP Fiori launchpad	SAP Fiori launchpad is a shell that hosts SAP Fiori apps, and provides the apps with services such as navigation, personalization, embedded support, and application configuration.	SAP Fiori	http://help.sap.com/fiori
event	An **event** has a name as well as any number of parameters. The element's API offers support to manage event subscriptions.	SAPUI5/ OpenUI5	Essentials [page 691]
JAWS Screen Reader	*JAWS, Job Access With Speech, is the world's most popular screen reader, developed for computer users whose vision loss prevents them from seeing screen content or navigating with a mouse. JAWS provides speech and Braille output for the most popular computer applications on your PC.*	Freedom Scientific	Freedom Scientific JAWS
jQuery	JavaScript library that is packaged with SAPUI5. *jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.* (Quote from jquery.com)	`jquery.com`	jQuery Home Page
library	The top-level structural unit is called a **library**. Libraries are the master artifacts in the extensibility concept. They bundle a set of controls and related types and make them consumable by Web applications. There are predefined and standard libraries, like `sap.m`, with many commonly used controls. At the same time, it treats custom UI libraries as first-class citizens, making it easy for you to write and use your own controls alongside the predefined ones.	SAPUI5/ OpenUI5	Essentials [page 691]

Term	Meaning	Source/ Comments	Link
mock server	The mock server is a mocking framework for HTTP and HTTPS that is used to simplify integration testing and to decouple development teams by allowing them to develop against a service that is incomplete or unstable.	SAPUI5/ OpenUI5	Mock Server [page 1222]
model	Data provider for the application where the model instance is assigned to the UI and the controls are bound to the model. Various model types are available; the model type used depends on the data format available on the server side.	SAPUI5/ OpenUI5	Models [page 882]
MVC concept	A UI programming model that separates the layout (view) from the content (model) and the behavior (controller). The MVC concept is used by the framework to model the architecture of the applications.	SAPUI5/ OpenUI5	Model View Controller (MVC) [page 784]
notepad control	A control that is defined on the fly without a library definition or running generation steps.	SAPUI5/ OpenUI5	Developing Controls [page 2158]
OData model	A model implementation for the Open Data (OData) Web Protocol format.	SAPUI5/ OpenUI5	OData V2 Model [page 883] OData V4 Model [page 918]
property	A **property** has a name and an associated data type. It has a well-defined default value expressed as a literal of that data type. Properties are accessible to application code via the element's API as getters and setters, but are also used by a control's renderer in a read-only way.	SAPUI5/ OpenUI5	Essentials [page 691]
SAPUI5 repository upload and download reports	Alternative for the team repository provider, with similar functionality.	SAPUI5 only	The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure [page 1507]
right-to-left (RTL) text directionality	*The `dir` attribute is used to set the base direction of text for display. It is essential for enabling HTML in right-to-left scripts such as Arabic, Hebrew, Syriac, and Thaana. Numerous different languages are written with these scripts, including Arabic, Hebrew, Pashto, Persian, Sindhi, Syriac, Dhivehi, Urdu, Yiddish, etc.* (Quote from w3c.org)	`w3c.org`	HTML Text Directionality

Term	Meaning	Source/ Comments	Link
resource model	Used to bind texts of a control to language-dependent resource bundle properties.	SAPUI5/ OpenUI5	Resource Model [page 995]
single SAPUI5 repository	The SAPUI5 ABAP repository consists of *n* single SAPUI5 repositories, each represented by an individual BSP application (with specific characteristics) in the BSP repository.	SAPUI5 only	The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure [page 1507]
SAP Fiori elements	App developers can use SAP Fiori elements to create SAP Fiori applications based on OData services and annotations requiring no JavaScript UI coding. An app based on SAP Fiori elements uses predefined template views and controllers that are provided centrally, so no application-specific view instances are required. The SAPUI5 runtime interprets metadata and annotations of the underlying OData service and creates the corresponding views for the SAP Fiori app at startup. The predefined view templates and controllers ensure UI design consistency across similar apps. Additionally, the metadata-driven development model significantly reduces the amount of frontend code per app, so the developer can focus on the business logic. SAP Fiori elements comprise templates for "List Report", "Object Page", and "Overview Page".	SAPUI5 only	Developing Apps with SAP Fiori Elements [page 1535]
scalable vector graphics (SVG)	*SVG is a markup language for describing two-dimensional graphics applications and images, and a set of related graphics script interfaces* (Quote from w3c.org)	`w3c.org`	W3C SVG
synchronous (sync) processing	Synchronous processing will keep the current browser thread until the task is finished. The UI is not updated and no other tasks can be done in parallel. Consider using asynchronous processing for loading files and executing long-running code.	SAPUI5/ OpenUI5	
SAPUI5 text repository	Part of the SAPUI5 ABAP repository; only to be used as a fallback mechanism if translation using properties files is not possible	SAPUI5 only	The SAPUI5 ABAP Repository and the ABAP Back-End Infrastructure [page 1507]

Term	Meaning	Source/ Comments	Link
view	An application unit containing the control definitions for the user interface layer in the application, or in other words: defines how the user interface looks like.	SAPUI5/ OpenUI5	Views [page 787]

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

www.sap.com/contactsap

THE BEST RUN SAP