UX400

UI Development with SAPUI5

PARTICIPANT HANDBOOK
INSTRUCTOR-LED TRAINING

Course Version: 22
Course Duration: 5 Day(s)
Material Number: 50158630

SAP Copyrights, Trademarks and Disclaimers

© 2022 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. Please see https://www.sap.com/corporate/en/legal/copyright.html for additional trademark information and notices.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors.

National product specifications may vary.

These materials may have been machine translated and may contain grammatical errors or inaccuracies.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP SE or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP SE or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platform directions and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, which speak only as of their dates, and they should not be relied upon in making purchasing decisions.

Typographic Conventions

American English is the standard used in this handbook.

The following typographic conventions are also used.

This information is displayed in the instructor's presentation	
Demonstration	
Procedure	
Warning or Caution	
Hint	
Related or Additional Information	
Facilitated Discussion	
User interface control	*Example text*
Window title	*Example text*

© Copyright. All rights reserved.

Contents

vii	Course Overview	
1	**Unit 1:**	**SAP's UX Strategy**
3		Lesson: Understanding SAP's UX Strategy
15	**Unit 2:**	**SAPUI5 Overview**
17		Lesson: Understanding SAPUI5
29	**Unit 3:**	**SAP Business Application Studio Overview**
31		Lesson: Exploring the SAP Business Application Studio
53	**Unit 4:**	**Load and Initialize SAPUI5**
55		Lesson: Loading and Initializing SAPUI5
61	**Unit 5:**	**Dependency Management**
63		Lesson: Managing Dependencies
73	**Unit 6:**	**Views and Controllers**
75		Lesson: Using Views and Controllers
87	**Unit 7:**	**Layout Controls**
89		Lesson: Using Layout Controls
109	**Unit 8:**	**Fragments**
111		Lesson: Using Fragments
123	**Unit 9:**	**Models and Data Binding**
125		Lesson: Exploring Models and Data Binding
155	**Unit 10:**	**Localization**
157		Lesson: Implementing Localization
167	**Unit 11:**	**OData Services**
169		Lesson: Exploring OData Services
191	**Unit 12:**	**OData Model**
193		Lesson: Consuming OData Services with SAPUI5

© Copyright. All rights reserved.

207 Unit 13: Component Implementation

209 Lesson: Implementing Components

223 Unit 14: Worklist App Development

225 Lesson: Developing Worklist Apps

243 Unit 15: Appendix - OData V4 Model

245 Lesson: Instantiating a Model
249 Lesson: Using Data Binding
261 Lesson: Filtering and Sorting Model Data
267 Lesson: Consuming OData Operations
273 Lesson: Implementing Batch Control
277 Lesson: Creating and Deleting Entities

© Copyright. All rights reserved.

Course Overview

TARGET AUDIENCE

This course is intended for the following audiences:

- Development Consultant
- Business Process Owner/Team Lead/Power User
- Developer
- Solution Architect

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 1 SAP's UX Strategy

Lesson 1

Understanding SAP's UX Strategy 3

UNIT OBJECTIVES

- Understand SAP's UX Strategy

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Understanding SAP's UX Strategy

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand SAP's UX Strategy

SAP's UX Direction – SAP Fiori

SAP Fiori is the design language that brings great user experiences to enterprise applications based on SAP User Experience.

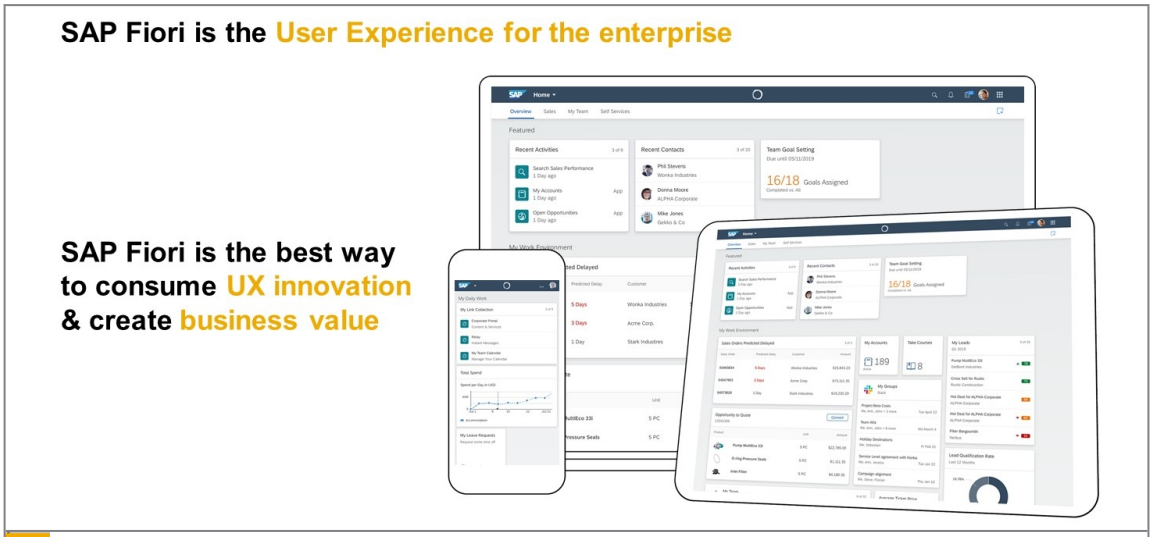


Figure 1: SAP's UX Direction – SAP Fiori (1 / 2)

With the release of SAPPHIRE in 2013, the first 25 apps for managers and employees with request and approval functions became available. Since then the number of apps has increased greatly. SAP Fiori 2.0 was introduced with SAP S/4HANA 1610, taking SAP Fiori to the next level. Today SAP Fiori 3 is the current target design, which advances the SAP Fiori design language for all SAP products to fully support the Intelligent Suite.

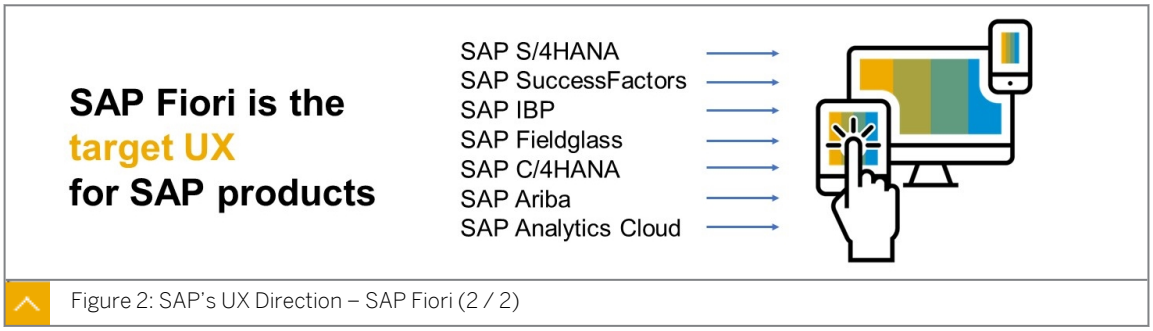


Figure 2: SAP's UX Direction – SAP Fiori (2 / 2)

© Copyright. All rights reserved.

All details of the SAP Fiori design have been made available as guidelines for general use. You can find all aspects of the SAP Fiori design starting with the five core design principles up to the floorplans of pages and details for UI elements. There are also several resources available for download, such as stencils and the SAP icon font, to help customers to design their own apps.

Since May 2016, SAP Fiori is also available for Apple iOS. Close cooperation between Apple and SAP has developed not only a merged design but also guidance and tools for developers to develop native apps for iOS. In addition, more and more apps are being developed by this partnership that leverages the features of Apple mobile devices.

Since June 2018, SAP Fiori is also available for Google Android. It provides a merged design and guidance for developers to develop native apps for Android. However, in contrast to SAP Fiori for iOS, no ready-to-use apps are provided to end users.

Since February 2019, design principles for conversational user experience (CUX) in SAP Fiori are available. It was already possible in SAP Fiori 2.0 to use the SAP Conversational AI, and it became a central part of the SAP Fiori 3 design.

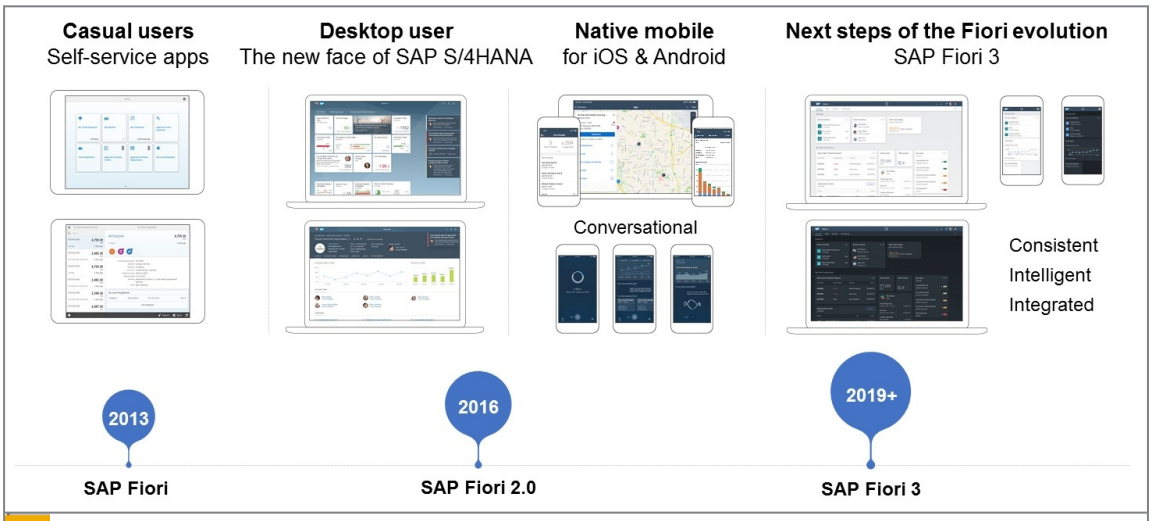


Figure 3: SAP Fiori 3 - The Latest Evolution

Beginning in 2013, SAP delivered the first 25 apps, shortly followed by several more. These apps were delivered in “waves” extending the number of functions available to end users step by step. It was already possible to use analytical apps if an SAP HANA was available in the system.

In October 2016, SAP shipped the SAP S/4HANA 1610 including the completely reworked SAP Fiori 2.0. All former apps could use the new design and features out-of-the-box. Although it was already possible via customizing, now SAP included thousands of classic applications automatically in SAP Fiori. With SAP Fiori Elements and SAP Fiori for iOS, new ways to develop SAP Fiori applications arose and were fine-tuned in the following years.

In October 2018, the new design, SAP Fiori 3 was introduced. However, in contrast to the previous versions, SAP did not deliver everything at once. SAP chose to deliver SAP Fiori 3 innovations in a step-by-step approach.

SAP defined SAP Fiori 3 jointly with all the various product units, taking SAP Fiori 2.0 as a starting point and looking to see which aspects must be added to best support them all.

One of the most requested new aspects is to provide more information on home pages, which the design supports via cards. In addition, SAP re-imagined how aspects of intelligence such as natural language interaction and machine intelligence can become a key part of the

© Copyright. All rights reserved.

experience. It deeply integrates them into the foundation of the design, providing the following features:

- A new theme for a fresh, modern look and feel.
- Intelligent home pages showing everything the user must focus on.
- Digital assistant with fully-integrated conversational user experience.
- Active business situation handling, with explanations, insights, and proposed actions.
- Dynamic content to support intelligent apps with embedded artificial intelligence (AI).
- Improved search previews, results pages, and result visualizations.
- A central inbox, also integrated with the digital assistant.

The goal of SAP Fiori 3 is to provide the perfect user interface for the intelligent suite that fully supports experience management.

SAP Key UI Technologies and Tools to Deliver SAP Fiori User Experience

The majority of SAP Fiori apps are web apps built using SAPUI5 as UI technology. SAPUI5 is based on HTML5 and can be consumed on every device using a browser. The recommended development environment for SAPUI5 is the SAP Business Application Studio.

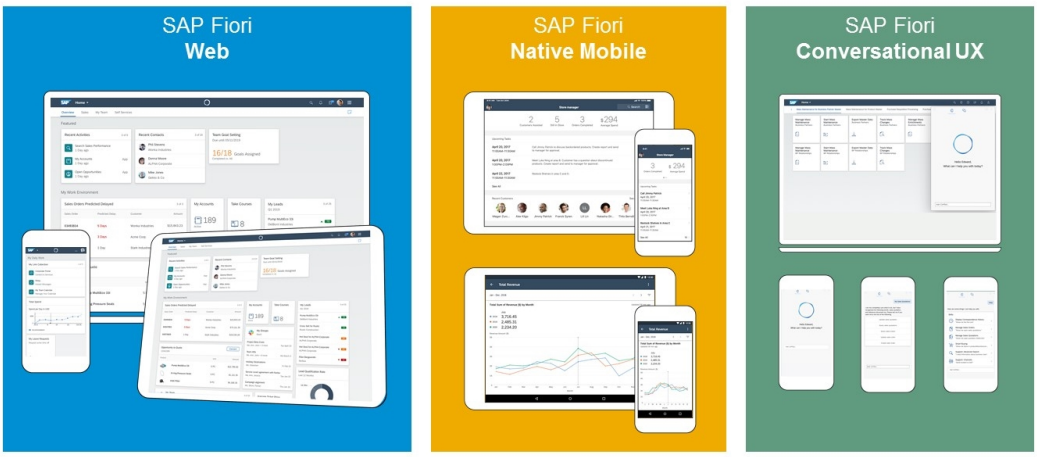


Figure 4: SAP Fiori Design Languages

SAP Fiori also supports native apps. These apps are developed in the native programming language used on a device, allowing a better integration. Apple and SAP are cooperating to develop native apps for iOS using Apple Xcode as the development environment. The open source language Swift, which was originally created by Apple, serves as the programming language (see https://swift.org). You can also develop native apps for Android using Java as the programming language in the Android Studio. In contrast to the cooperation with Apple, apps are neither shipped by Google nor SAP.

The term Conversational User Experience (CUX) describes digital assistants and chatbots for communicating with enterprise systems. The SAP Conversational Artificial Intelligence (CAI) is a combination of SAP CoPilot and the startup formerly known as Recast.AI. CAI enables users to smoothly interact with SAP software through one unique conversational interface. The skills of conversational agents are built, trained, generated, and connected using the Bot Builder.

© Copyright. All rights reserved.

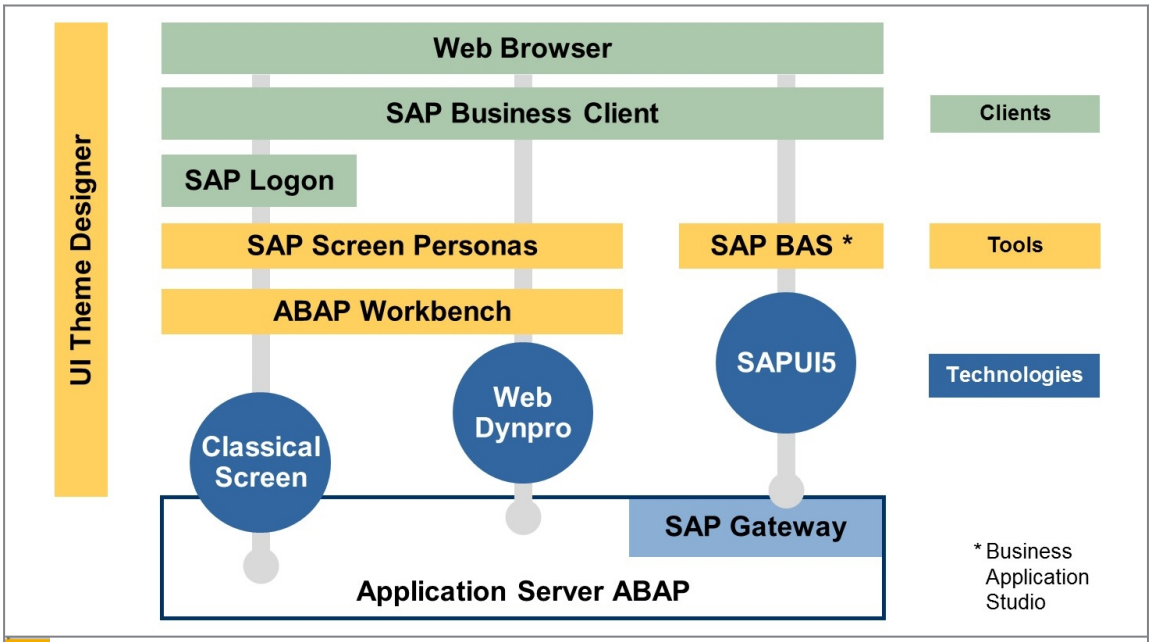


Figure 5: SAP Key UI Technologies and Tools to Deliver SAP Fiori User Experience

There are three main UI technologies used in current SAP applications. Classic screens are developed and run in ABAP. Using the SAP GUI for HTML, you can also run classic screens in a browser environment. Web Dynpro ABAP is developed in ABAP but uses JavaScript in the browser to communicate with the ABAP system. SAPUI5 is pure JavaScript and communicates with ABAP using SAP Gateway.

There are tools available to adapt the applications to the needs of current end users:

- SAP Screen Personas
  - Classic screens (including Control Framework (CFW))
  - Web Dynpro (including Floorplan Manager (FPM))
- SAP Business Application Studio
  - SAPUI5 (including SAP Fiori)

All UI technologies can be consumed using a web browser or the SAP Business Client. Classic screens are displayed using SAP GUI for Windows in the Business Client as well as SAP Logon. When it comes to the design of user interface, one tool handles all technologies: the UI Theme Designer.

SAP Fiori Elements

SAP Fiori Elements provide designs for UI patterns and predefined templates for commonly-used application patterns.

© Copyright. All rights reserved.

SAP Fiori Elements increase developer efficiency, ensure UX consistency, and keep apps up-to-date with evolving SAP Fiori design guidelines by applying a metadata-driven approach to application development.

Key capabilities

- Provides SAP Fiori-compliant page templates
- Includes standard UI logic, e.g. filtering, search, navigation, CRUD operations, message handling
- Supports UI Flexibility, and automatic test tools
- Works with CDS annotations and OData annotations
- Uses SAPUI5 controls for rendering screen objects
- Supports extensibility using SAPUI5

Benefits

- Rapid SAP Fiori app development
- Standard design and interaction patterns for SAP Fiori UX
- Consistent UX across business processes
- Reduced cost of application development and maintenance

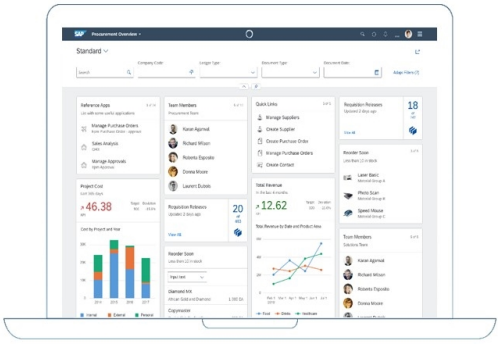


Figure 6: Developing Apps with SAP Fiori Elements

Applications based on SAP Fiori Elements use predefined 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 uses the corresponding views for the SAP Fiori app at startup. The predefined views and controllers ensure UI design consistency across similar apps. The metadata-driven development model also significantly reduces the amount of front-end code for each app, so the developer can focus on the business logic.

SAP Fiori Elements offer the following features:

- Reuse functionality that does not require specific programming
- A common look and feel and UI behavior for all apps
- Edit mode control, switching between display and edit, and submitting changes
- Message handling
- Control of the UI using OData annotations, which semantically enrich the OData metadata
- Handling of draft documents (draft saving is available)

SAP Fiori Elements are part of the SAPUI5 delivery.

© Copyright. All rights reserved.

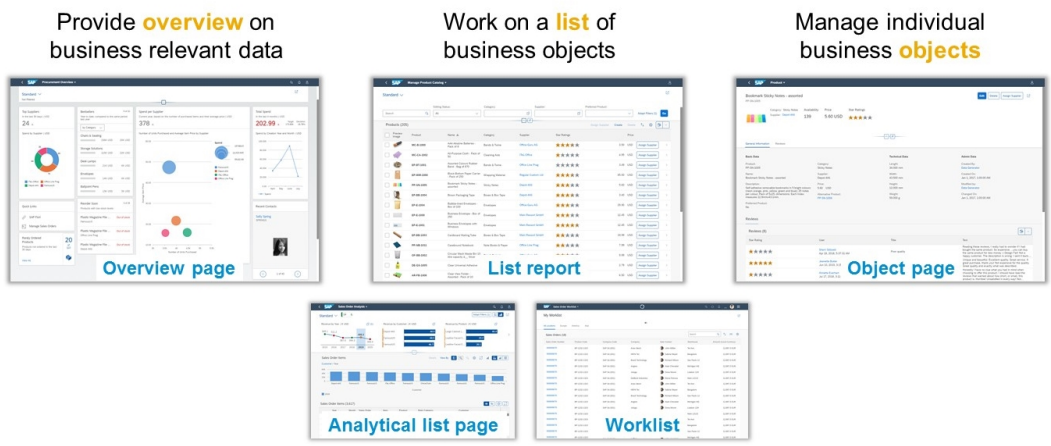


Figure 7: Currently Available Floorplans

With a List Report, users can view and work with a large set of items. This floorplan offers powerful features for finding and acting on relevant items. It is often used as an entry point for navigating to the item details, which are usually shown on an Object Page.

The Object Page allows users to display, edit, and create objects, and save drafts. It is suitable for both simple objects and more complex, multi-faceted objects. The Object Page view offers optimal support for multiple devices.

An Overview Page is a data-driven SAP Fiori application for organizing large amounts of information. Overview pages provide quick access to business information at a glance, in the form of visual, actionable cards. The application lets you create several cards for different types of content that helps in visualizing information in an attractive and efficient way. The displayed data is fully interactive, with clickable areas for easy navigation to relevant applications.

The Analytical List Page is an 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. All this can be done seamlessly within one page.

A Worklist displays a collection of items that are 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. 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.

© Copyright. All rights reserved.

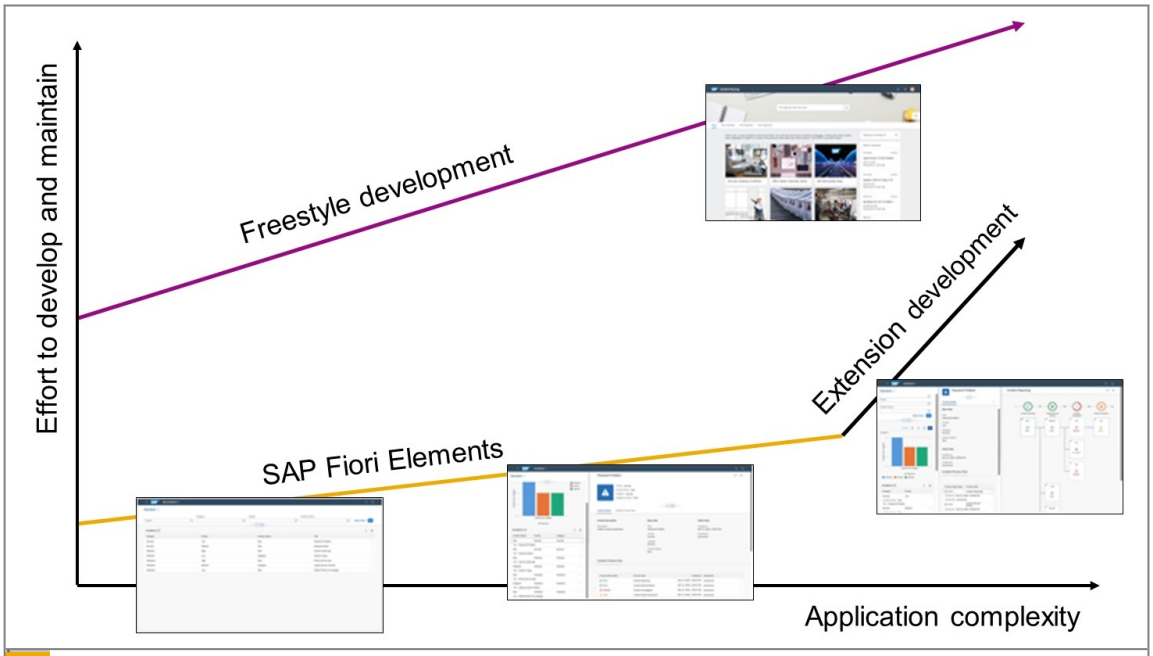


Figure 8: Choose the Right Development Approach for your SAPUI5 app

SAP Fiori Elements are particularly suitable for creating less complex, standardized applications. The effort involved is less than in SAPUI5 freestyle development. However, the amount of freedom with SAP Fiori Elements is limited. To develop more complex applications with special requirements, SAPUI5 freestyle development is recommended.

LESSON SUMMARY

You should now be able to:

- Understand SAP's UX Strategy

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which of the following statements are true when talking about SAP Fiori?

   Choose the correct answers.

   - ☐ A SAP Fiori is SAP's user experience that applies modern user-centric design principles
   - ☐ B SAP Fiori is SAP's product for implementing SAP Enterprise Portal components
   - ☐ C SAP Fiori is SAP's UI technology for implementing applications for mobile devices
   - ☐ D SAP Fiori can be personalized, is simple, and runs on any device
   - ☐ E SAP Fiori provides a consistent and role-specific experience across all tasks, for all lines of business

2. What is the vision of SAP when considering SAP's UX direction?

   Choose the correct answer.

   - ☐ A Deliver attractive applications that make people successful at work
   - ☐ B Provide the leading user experience for business applications
   - ☐ C Provide UX as service to gain productivity
   - ☐ D Deliver SAP modules in an SaaS manner

3. Which of the following SAP Key UI technologies are currently recommended by SAP?

   Choose the correct answers.

   - ☐ A Web Dynpro Java
   - ☐ B Web Dynpro ABAP with Screen Personas
   - ☐ C Classical Screen with Screen Personas
   - ☐ D Java Dynpro
   - ☐ E SAPUI5

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which of the following statements are true when talking about SAP Fiori?

   Choose the correct answers.

   [X] A SAP Fiori is SAP's user experience that applies modern user-centric design principles

   [ ] B SAP Fiori is SAP's product for implementing SAP Enterprise Portal components

   [ ] C SAP Fiori is SAP's UI technology for implementing applications for mobile devices

   [X] D SAP Fiori can be personalized, is simple, and runs on any device

   [X] E SAP Fiori provides a consistent and role-specific experience across all tasks, for all lines of business

   Correct. SAP Fiori is SAP's user experience that applies modern user-centric design principals. Applications can be personalized, are simple, and run on any device. Applications that follow SAP Fiori design principals have a consistent UI and role-specific experience across all tasks and for all lines of business.

2. What is the vision of SAP when considering SAP's UX direction?

   Choose the correct answer.

   [ ] A Deliver attractive applications that make people successful at work

   [X] B Provide the leading user experience for business applications

   [ ] C Provide UX as service to gain productivity

   [ ] D Deliver SAP modules in an SaaS manner

   Correct. The vision of SAP is to provide a leading user experience for business applications.

© Copyright. All rights reserved.

3. Which of the following SAP Key UI technologies are currently recommended by SAP?

   Choose the correct answers.

   - [ ] A Web Dynpro Java
   - [x] B Web Dynpro ABAP with Screen Personas
   - [x] C Classical Screen with Screen Personas
   - [ ] D Java Dynpro
   - [x] E SAPUI5

   Correct. Web Dynpro ABAP with Screen Personas, Classical Screen with Screen Personas, and SAPUI5 are currently the key UI technologies at SAP.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 2 SAPUI5 Overview

Lesson 1

Understanding SAPUI5 17

UNIT OBJECTIVES

- Understand SAPUI5

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Understanding SAPUI5

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand SAPUI5

SAP UI Development Toolkit for HTML5

The figure, SAP UI Development Toolkit for HTML5 General Information, provides general information on the SAP UI Development Toolkit for HTML5 (SAPUI5).

See also https://sapui5.hana.ondemand.com.

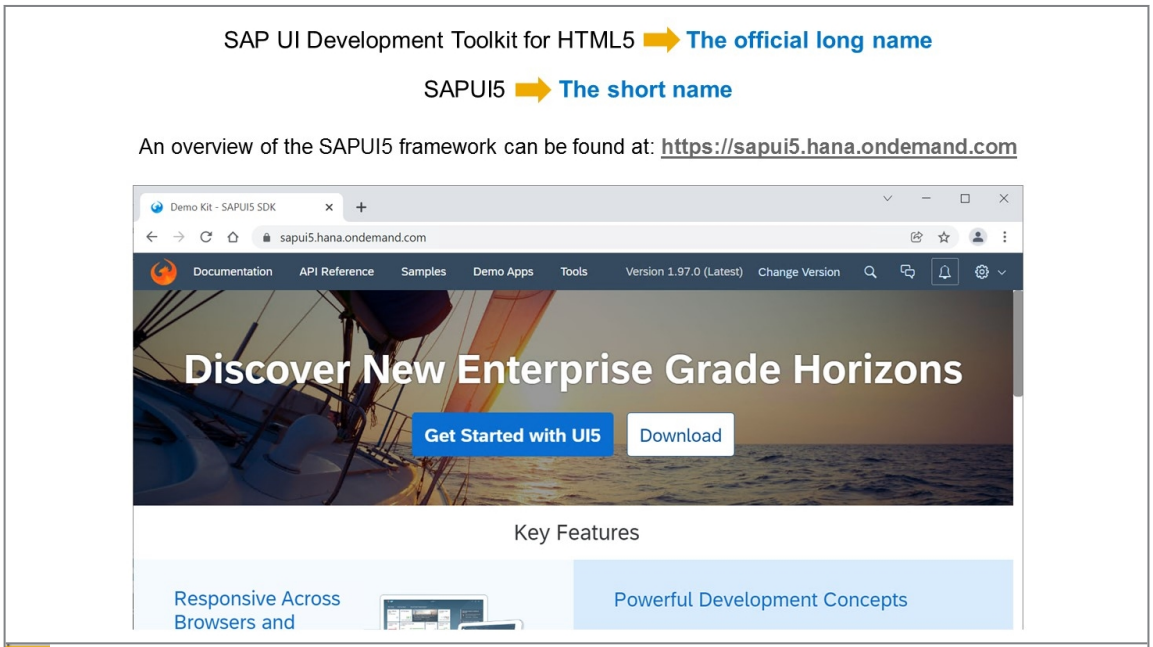


Figure 9: SAP UI Development Toolkit for HTML5 General Information

Main SAPUI5 Features

SAPUI5 is a JavaScript-based library for modern web business applications that provides the following features:

- One consistent user experience for your apps
- Responsive across browsers and devices, including smartphones, tablets, and desktops
- Feature-rich UI controls for handling complex UI patterns
- Keyboard interaction support and accessibility features
- Full translation support

© Copyright. All rights reserved.

- Based on open standards like JavaScript, CSS3, and HTML5
- Powerful theming support based on CSS
- Allows the inclusion and use of the popular jQuery library

SAPUI5 Architecture

When users access an SAPUI5 app, a request is sent to the respective server to load the application into the browser. This process is shown in the figure, SAPUI5 Architecture. The view accesses the relevant libraries. Usually, the model is also instantiated and business data is retrieved 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 on SAP Business Technology Platform, and business data can be accessed, for example, using the OData model through an SAP Gateway.

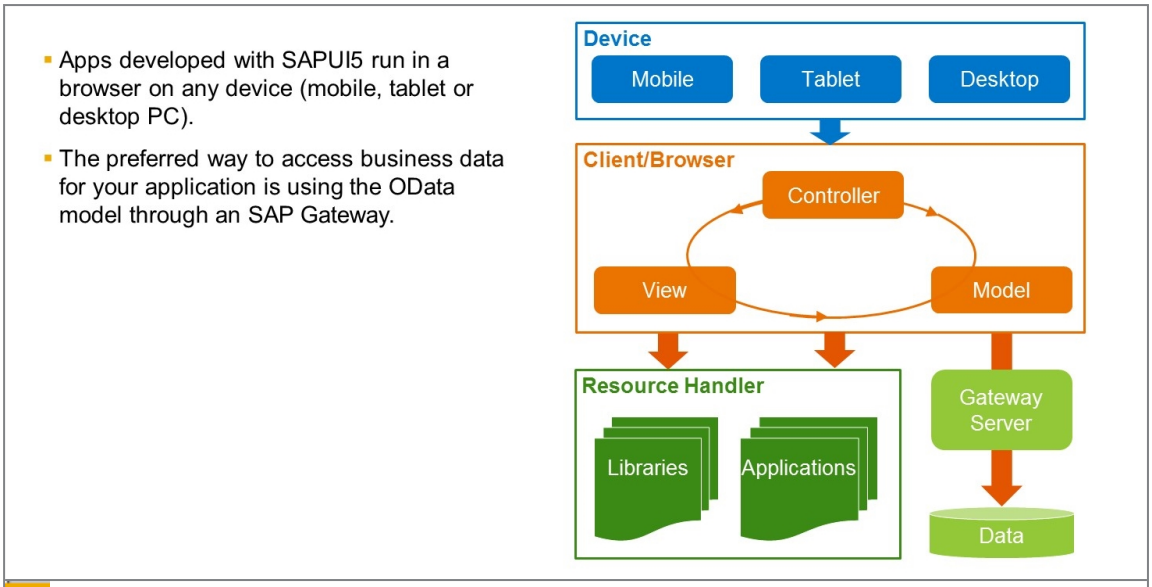


Figure 10: SAPUI5 Architecture

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.

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.
- You can change the view without touching the underlying business logic.
- You can 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 perspective, a view is an SAPUI5 control, and can have or inherit an SAPUI5 model.

© Copyright. All rights reserved.

Views and controllers represent reusable units, and distributed development is highly supported.

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.

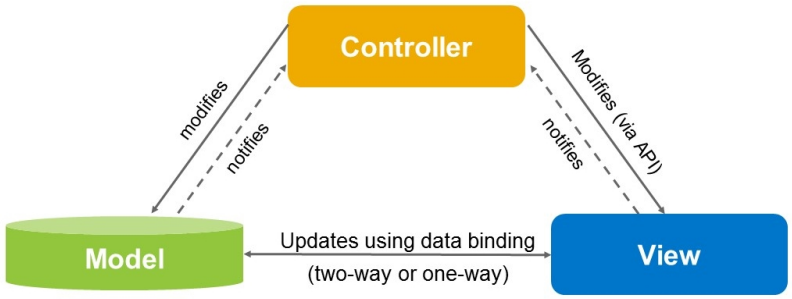


Figure 11: Model View Controller (MVC)

Browser and Platform Support

Specific libraries can have different browser support. The best source of information regarding supported browsers and platforms is the Product Availability Matrix (PAM), which you can find at https://support.sap.com/pam.

SAPUI5 is not a standalone product, so check the PAM for the product you are using SAPUI5 with.

SAP currently does not guarantee that newer ECMAScript standards, such as ES6/ES2015, work with SAPUI5.

- Because SAPUI5 is based on CSS3, HTML5, and the ECMAScript 5 (ES5) JavaScript API, only browsers with HTML5 capabilities are supported.
- In general, the major browser versions that are also supported by the respective platform are supported.
- Details regarding supported browsers, operating systems, reference devices and restrictions can be found on the Product Availability Matrix (PAM): https://support.sap.com/pam

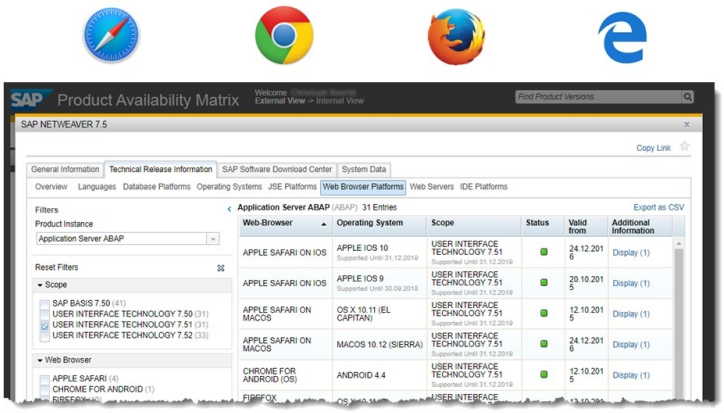


Figure 12: Browser and Platform Support

Versioning of SAPUI5

SAPUI5 uses a three-digit version identifier. For example, 1.84.20. These digits have the following meaning:

© Copyright. All rights reserved.

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

For an SAPUI5 version, SAP delivers patches that contain fixes, but no new features. New features are only delivered with the next (minor) version.

For further information regarding versioning, see the figure, Versioning of SAPUI5.

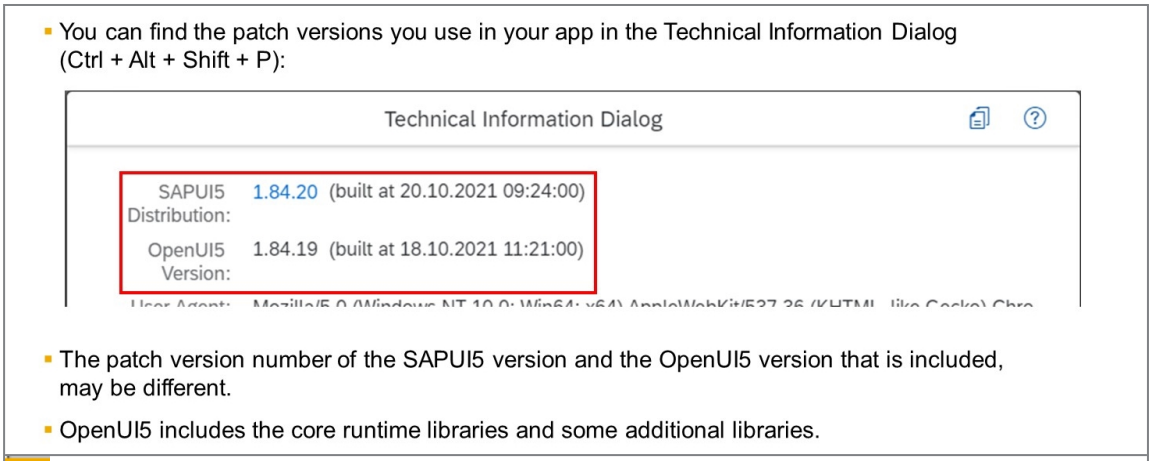


Figure 13: Versioning of SAPUI5

All SAPUI5 resources are available on the content delivery network Akamai. There, you can also find multiple SAPUI5 versions, and you can reference them in your application.

Maintenance Strategy

As long as an SAPUI5 version is still in maintenance, SAP provides patches with fixes.

Maintenance versions have an extended maintenance period, which means SAP still provides patches even though a higher version is already available. This is because the SAPUI5 version is included in a release of the SAP_UI component. For example, SAPUI5 version 1.84 has an extended maintenance, because SAPUI5 version 1.84 is included in SAP_UI 7.55.

In the version overview at https://sapui5.hana.ondemand.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.

To view the documentation for a specific version, check which versions are available at https://sapui5.hana.ondemand.com/versionoverview.html. You can view the version-specific Demo Kit by adding the version number to the URL, for example, https://sapui5.hana.ondemand.com/1.84.20/.

To get an overview of the new features of each version, see the “What's New in SAPUI5” section. To see the fixes contained in each patch, review the “Change Log”.

© Copyright. All rights reserved.

- Check the available **SAPUI5 versions** with respective **Maintenance Status** at: https://sapui5.hana.ondemand.com/versionoverview.html

SAPUI5 Versions Maintenance Status

The updated SAPUI5 maintenance strategy for SAP Business Technology Platform is available in this document.

Version	Maintenance Status	End of Maintenance	End of Cloud Provi...	SAP_UI version*	Front-end Server*
1.97 (2112)	Maintenance			Cloud only	Cloud only
1.96 (2111)	Maintenance	Long-term Maintenance, Q4/2024	Q4/2025	7.56	SAP FIORI FES 2021 FOR S/4HANA (SAP Note 3050199)
1.93 (2108)	Maintenance	Q1/2022	Q1/2023	Cloud only	Cloud only
1.90 (2105)	Maintenance	Q1/2022	Q1/2023	7.56	SAP FIORI FES 2021 FOR S/4HANA (SAP Note 3050199)
1.84 (2011)	Maintenance	Long-term Maintenance, Q4/2023	Q4/2024	7.55	SAP FIORI FES 2020 FOR S/4HANA (SAP Note 2919182)
[illegible]	[illegible]	Long-term	[illegible]	[illegible]	SAP Fiori FES 6.0 (SAP

- Even though SAP also provides older versions of SAPUI5, it is recommended to always upgrade to the latest version!
- Make sure that you always refer to a specific SAPUI5 version instead of using the generic reference to the latest released version. Use the generic reference for testing purposes only.

Figure 14: Maintenance Strategy

Available SAPUI5 Versions on the Application Server

To access information about the SAPUI5 versions installed on a server, use the URL highlighted in the figure, Available SAPUI5 Versions on the Application Server.

> Note:
> In the URL shown in the figure, Available SAPUI5 Versions On The Application Server, replace `<host>` with the host name and `<port>` with the http port number of your ABAP System.

- For SAP NetWeaver Application Server ABAP, use the following URL to get an overview of the SAPUI5 versions installed on the server:

http://<host>:<port>/sap/public/bc/ui5_ui5/index.html

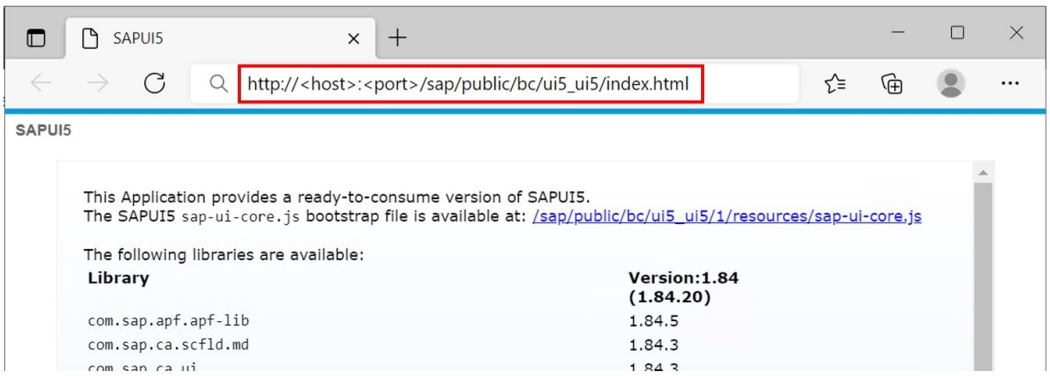


Figure 15: Available SAPUI5 Versions on the Application Server

SAPUI5 Libraries

A library bundles a set of controls and related types, to make them consumable by Web applications. There are predefined and standard libraries, like `sap.m`, with many commonly used controls. See the figure, SAPUI5 Libraries, for SAPUI5 common libraries.

© Copyright. All rights reserved.

Developers can create their own custom UI libraries, making it easy to use their own controls alongside predefined controls.

The core library (technically, this is the `sap.ui.core` library) defines a core set of types that can be used in other libraries.

SAPUI5 has a library concept, modularizing sets of controls for different usages, for example:

- **sap.m**
  The main SAPUI5 control library, with responsive controls that can be used in touch devices as well as desktop browsers.
- **sap.ui.comp**
  SAPUI5 library with smart controls
- **sap.ui.table**
  Table-like controls, mainly for desktop scenarios
- **sap.ui.layout**
  SAPUI5 library with layout controls
- **sap.ui.core**
  The SAPUI5 Core Runtime which contains the Core and all its components, base classes for Controls, Components, and the Model View Controller classes.

Figure 16: SAPUI5 Libraries

SAPUI5 and Extensibility

SAPUI5 extensibility options allow developers to do the following:

- Include their own JavaScript, HTML, and CSS in SAPUI5-based pages
- Include other JavaScript libraries where SAPUI5 lacks the controls or functionality
- Create composite controls from existing SAPUI5 controls
- Write new libraries and new controls

In this way, SAPUI5 development groups can avoid becoming a bottleneck for application groups in need of specific functionality.

OpenUI5

The figure, OpenUI5, provides general information about OpenUI5. More information is available at http://openui5.org/

- OpenUI5 includes all the main features of SAPUI5.
- OpenUI5 encompasses the complete SAPUI5 Core plus more than 200 frontend controls that are also included in SAPUI5.
- OpenUI5 does not contain, for example, sap.viz, sap.ui.comp.
- OpenUI5 is free to use, released under the Apache 2.0 license.

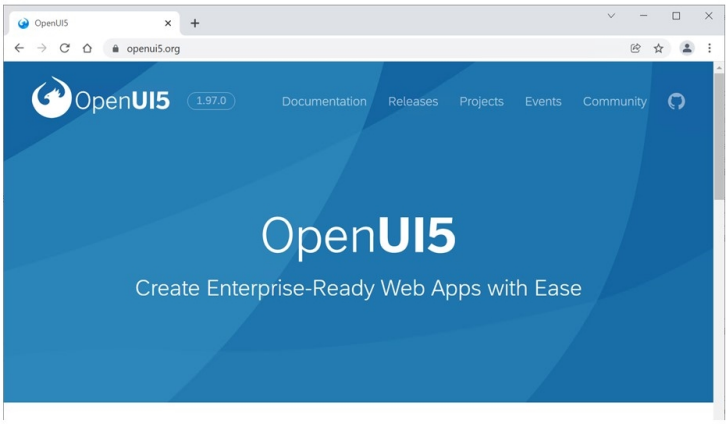


Figure 17: OpenUI5

© Copyright. All rights reserved.

SAPUI5 and OpenUI5: Two Deliveries of the UI Development Toolkit

With SAPUI5 and OpenUI5, SAP provides two deliveries of the UI development toolkit. The two are closely related, but have some differences.

The main difference between the two is the license. OpenUI5 is open source, free to use, and released under the Apache 2.0 license. SAPUI5 is not a separate SAP product with a separate license. It is integrated, for example, with the following products:

- SAP HANA
- SAP Business Technology Platform
- SAP NetWeaver 7.4 or higher (included in the UI technologies (SAP_UI) component)
- The user interface add-on for SAP NetWeaver Application Server 7.3x

The easiest way to learn which libraries are delivered is to review the API Reference of each Demo Kit. You will see that the list of libraries in SAPUI5 is much longer, but this does not mean that OpenUI5 provides a 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`, and `sap.ui.unified`). The additional libraries in SAPUI5 include more controls on top, such as charts. SAPUI5 also lets you use smart controls, which are controls that are automatically configured by OData annotations from the back-end system. The exact feature range of SAPUI5 also depends on the platform you are using.

OpenUI5 is open source, and is available on GitHub. If you find a bug, or have an idea for a new feature, just submit a GitHub issue or change request.

For the OpenUI5 version, visit http://openui5.org/, where you can download the runtime and the Demo Kit (SDK) at http://openui5.org/releases. For SAPUI5 resources, check your platform installation. Both resources are also available online through the content delivery network provider Akamai, at https://openui5.hana.ondemand.com and https://sapui5.hana.ondemand.com.

Technically, you can switch between OpenUI5 and SAPUI5 (providing you have the respective license), if, for example, you want to use the SAPUI5-specific features. First, check which SAPUI5 version you need, because the version numbers of OpenUI5 and SAPUI5 might differ at the patch level (last digit of the version number). You can find this information in the Technical Information Dialog (Ctrl + Alt + Shift + P). If you are 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 all other cases, replace the runtime. Because the technical names (of controls, libraries, and so on) and API names are the same in both OpenUI5 and SAPUI5, the code will still work and you can start enhancing it directly.

LESSON SUMMARY

You should now be able to:

- Understand SAPUI5

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which of the following are key features of SAPUI5?

   Choose the correct answers.

   - ☐ A Full translation support
   - ☐ B Web Dynpro extensibility
   - ☐ C Responsive across browsers and devices
   - ☐ D Less open for custom development
   - ☐ E Based on open standards such as, JavaScript, CSS3, and HTML5

2. What is the key concept of SAPUI5 that enables the separation of information representation from user interaction?

   Choose the correct answer.

   - ☐ A Model View Controller
   - ☐ B Drag-and-drop
   - ☐ C Slideshow
   - ☐ D A forgiving format

3. Which of the following statements are true for OpenUI5?

   Choose the correct answers.

   - ☐ A OpenUI5 includes all of the main features of SAPUI5.
   - ☐ B OpenUI5 contains `sap.viz` and `sap.ui.comp`.
   - ☐ C OpenUI5 is free for use.
   - ☐ D OpenUI5 implements its own Core-Component.
   - ☐ E OpenUI5 encompasses the complete SAPUI5 Core, plus many front-end controls that are also included in SAPUI5.

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which of the following are key features of SAPUI5?

   Choose the correct answers.

   - [X] A Full translation support
   - [ ] B Web Dynpro extensibility
   - [X] C Responsive across browsers and devices
   - [ ] D Less open for custom development
   - [X] E Based on open standards such as, JavaScript, CSS3, and HTML5

   Correct. SAPUI5 provides full translation support for building internationalized applications. SAPUI5 is also a framework that was built to run on any browser and any device, and is based on open Web standards such as, JavaScript, CSS3, and HTML5.

2. What is the key concept of SAPUI5 that enables the separation of information representation from user interaction?

   Choose the correct answer.

   - [X] A Model View Controller
   - [ ] B Drag-and-drop
   - [ ] C Slideshow
   - [ ] D A forgiving format

   Correct. SAPUI5 uses the Model View Controller paradigm to separate the representation of information from user interaction.

© Copyright. All rights reserved.

3. Which of the following statements are true for OpenUI5?

   Choose the correct answers.

   [X] A OpenUI5 includes all of the main features of SAPUI5.

   [ ] B OpenUI5 contains `sap.viz` and `sap.ui.comp`.

   [X] C OpenUI5 is free for use.

   [ ] D OpenUI5 implements its own Core-Component.

   [X] E OpenUI5 encompasses the complete SAPUI5 Core, plus many front-end controls that are also included in SAPUI5.

   Correct. OpenUI5 is free to use, but includes all main features of SAPUI5 such as the complete SAPUI5 Core and many front-end UI controls.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 3 SAP Business Application Studio Overview

Lesson 1

Exploring the SAP Business Application Studio 31

UNIT OBJECTIVES

- Explore the SAP Business Application Studio

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 3
Lesson 1

Exploring the SAP Business Application Studio

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explore the SAP Business Application Studio

SAPUI5 Development Tools

Since 2014, SAP have provided the SAP Web IDE based on Orion, an open source project by Eclipse. It is available in three flavors:

- SAP Web IDE Full-Stack:

  Development environment on SAP Business Technology Platform, Neo Environment supporting for example SAP Fiori, SAP S/4HANA Cloud, SAP HANA, SAP Mobile Services including programming languages such as Java and JavaScript.

- SAP Web IDE for SAP HANA:

  Native development environment for SAP HANA provided by SAP HANA Extended Services Advanced including programming languages such as Java and JavaScript.

- SAP Web IDE Personal Edition:

  Local development environment for SAP Fiori and SAPUI5, which can be downloaded on https://tools.hana.ondemand.com/.

© Copyright. All rights reserved.

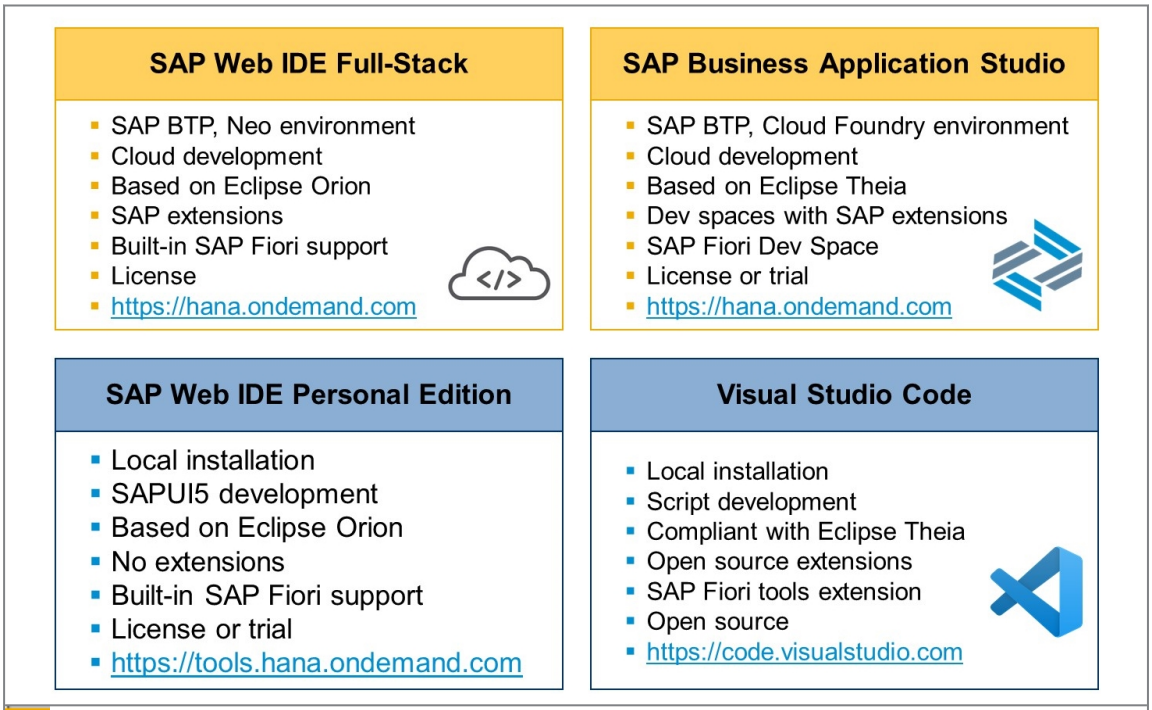


Figure 18: SAPUI5 Development Tools

SAP Web IDE for SAP HANA is the only IDE not supporting SAPUI5. This leaves one SAP Web IDE for developing SAPUI5 provided in the SAP Business Technology Platform and one provided for on-premise. One important difference is the missing support for extensions in the SAP Web IDE Personal Edition.

Since 2020, SAP have provided the SAP Business Application Studio based on Theia, an open source project by Eclipse. It is the next generation of SAP Web IDE provided in the SAP Business Technology Platform, Cloud Foundry environment. It provides pre-configured environments, so-called dev spaces, with pre-installed runtimes and tools tailored for key scenarios. These are based on extensions such as the SAP Fiori tools extension.

Another tool where the SAP Fiori tools extension can be included is Visual Studio Code. It is developed by Microsoft as an open source IDE following the same rules as Theia. Although not based on Theia, it is fully compatible. SAP offers the SAP Fiori tools beside others free of charge in the extensions marketplace of Visual Studio Code. Visual Studio Code can be downloaded under https://code.visualstudio.com/.

SAP Business Application Studio – Overview

SAP Business Application Studio is a SAP Business Technology Platform service that offers a modern development environment tailored for efficient development of business applications for the SAP Intelligent Enterprise.

© Copyright. All rights reserved.

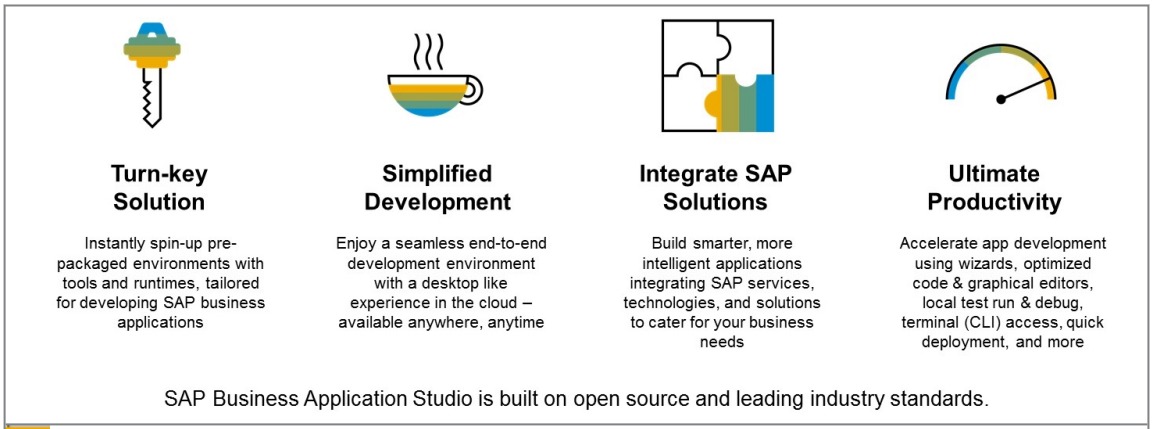


Figure 19: SAP Business Application Studio - Value Proposition

Available as a cloud service, SAP Business Application Studio provides a desktop-like experience similar to leading IDEs, with command line and optimized editors.

At the heart of SAP Business Application Studio are the dev spaces. The dev spaces are comparable to isolated virtual machines in the cloud containing tailored tools and preinstalled runtimes per business scenario, such as SAP Fiori, SAP S/4HANA extensions, Workflow, Mobile and more. This simplifies and saves time in setting up your development environment and allows you to efficiently develop, test, build, and run your solution locally or in the cloud.

Unlike other desktop IDEs, SAP Business Application Studio is equipped with various tools specially built for developing business applications in the SAP ecosystem. These tools cover the end-to-end development cycle:

- Clone an existing project with a Git client or create new project using templates.
- Use editors for SAP-specific technologies.
- Easily test your application on your dev space while consuming services from remote sources.
- Build and deploy the application as a multitarget application (MTA).

Launching the SAP Business Application Studio

To try out the SAP Business Application Studio, you can create a free trial account on the SAP Business Technology Platform.

© Copyright. All rights reserved.

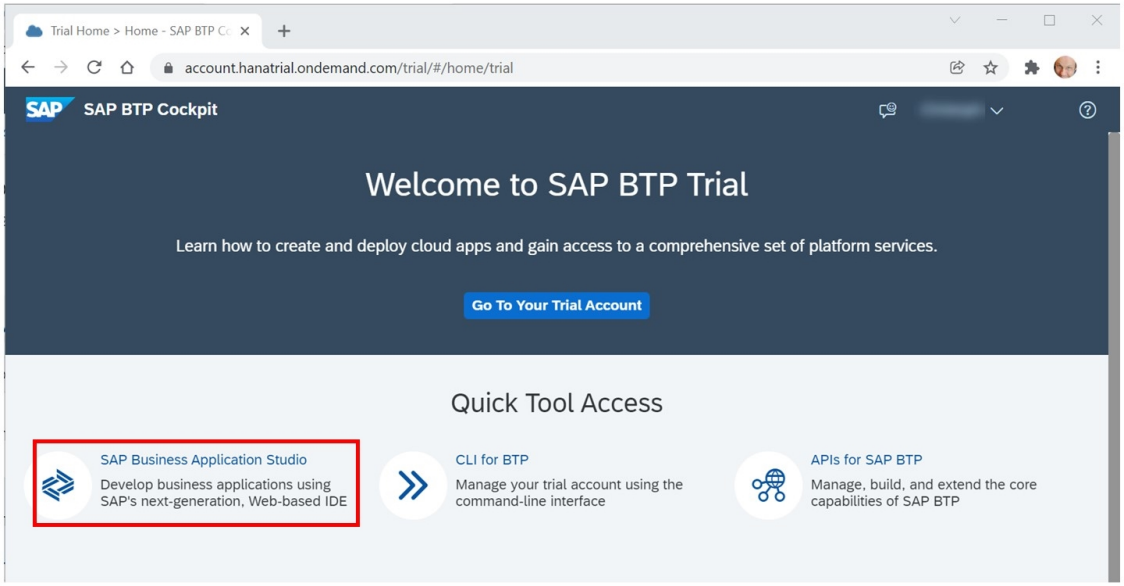


Figure 20: Starting the SAP Business Application Studio on SAP BTP Trial

To open the Business Application Studio, click the link *SAP Business Application Studio* in the *Quick Tool Access* area of the *Welcome to SAP BTP Trial* page.

Developing Applications with a Dev Space

Each type of application requires a different development environment. When you create a dev space, you are prompted to select the type of application that you want to create. Based on the type of application that you choose to build, you are provided with a different set of tools.

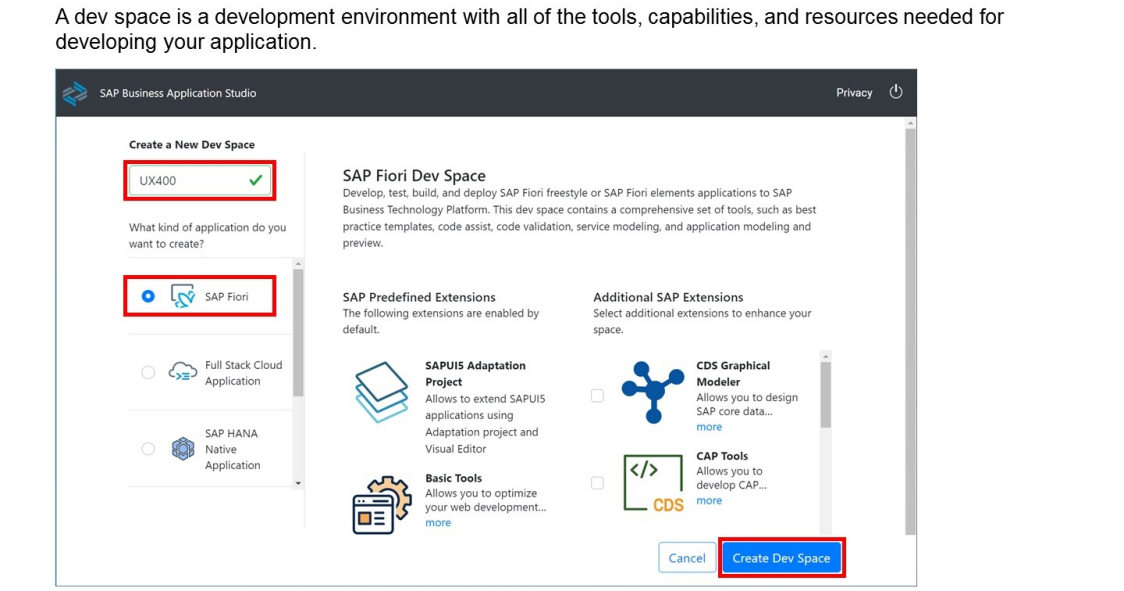


Figure 21: Creating a Dev Space

To create a dev space for SAPUI5 development, proceed as follows:

1. Open SAP Business Application Studio and log in with your credentials.
2. Click *Create Dev Space*.

© Copyright. All rights reserved.

3. Enter a name for the dev space.
4. Select the application type *SAP Fiori*.
5. (Optional) Select the relevant additional extensions to enhance your space.
6. Click *Create Dev Space*.

If you don't need to work with your dev space for a while, you can stop your dev space. When your dev space is running, it consumes memory, energy, and CPU. If you don't use your dev space and it sits idle for too long, the dev space will be stopped.

When you restart your stopped dev space, all content in your dev space, including files and settings, remain and will be available.

You can also delete a dev space.

Creating New Projects

There is a project wizard that can be used to create a project in SAP Business Application Studio. To start the wizard, you can choose *File* → *New Project from Template*.

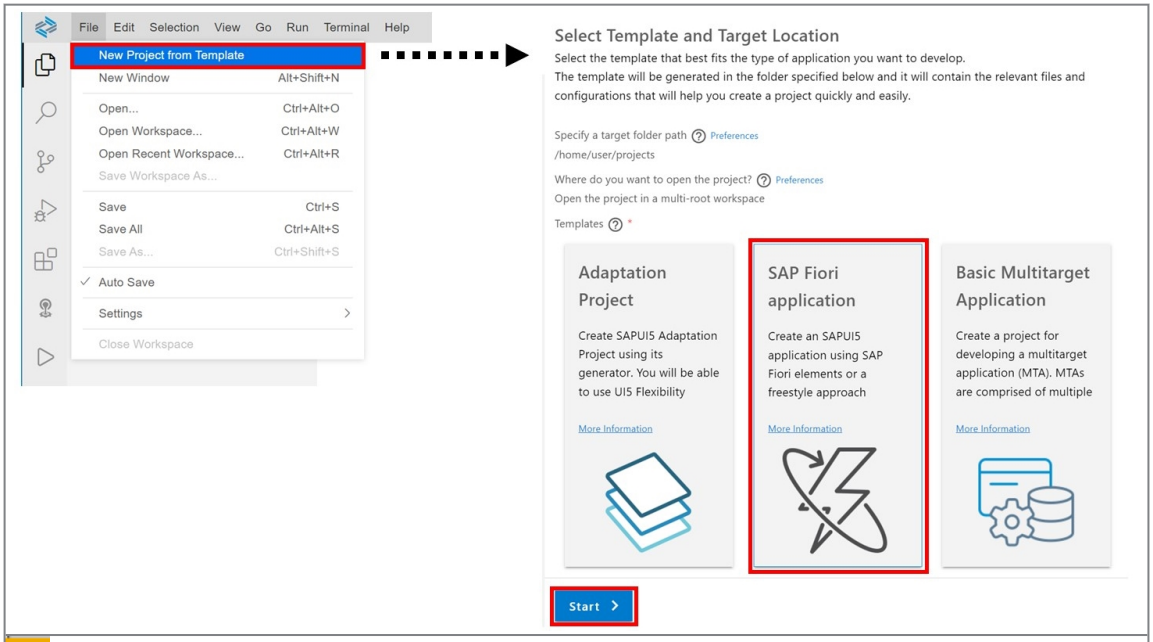


Figure 22: Creating a Fiori Application Project

Follow the wizard steps and provide the required information. You can move between steps using the *Previous* and *Next* buttons or clicking the steps from the wizard tree.

The wizard displays error notifications on the field level and on the step level.

The wizard generates the project including the relevant folder structure, artifacts, and resources.

To be able to work with a project, you have to open it in a workspace. You can use the menu path *File* → *Open Workspace...* for this. A workspace is an entity containing your project's settings, debug configurations, and task configurations.

You can choose to create a new workspace for each project, or you can set up a multi-root workspace.

© Copyright. All rights reserved.

Layout Editor

In the Layout Editor, you can display the content of an XML view to see it in a way that closely corresponds to how it will appear in the finished application.

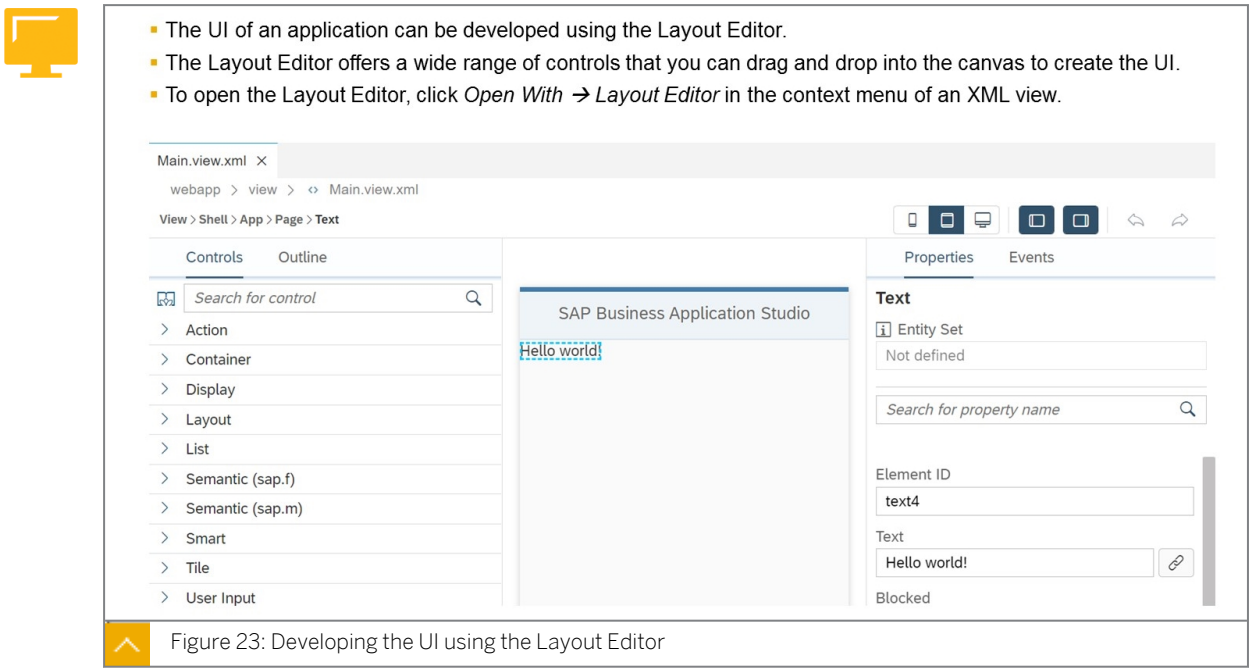


Figure 23: Developing the UI using the Layout Editor

The Layout Editor is composed of a canvas, a pane on the left side that includes the *Controls* and *Outline* tabs, and a pane on the right side that includes the *Events* and *Properties* panes.

For details on the Layout Editor, refer to the SAP Business Application Studio documentation.

Previewing an Application

After the application is successfully generated, several options to preview it are available in SAP Business Application Studio. You can start a preview via npm scripts from the terminal, or using the Run Control function in the *Run Configuration* pane. You can also create additional run configurations in the *Run Configuration* pane that define how your project is executed.

© Copyright. All rights reserved.

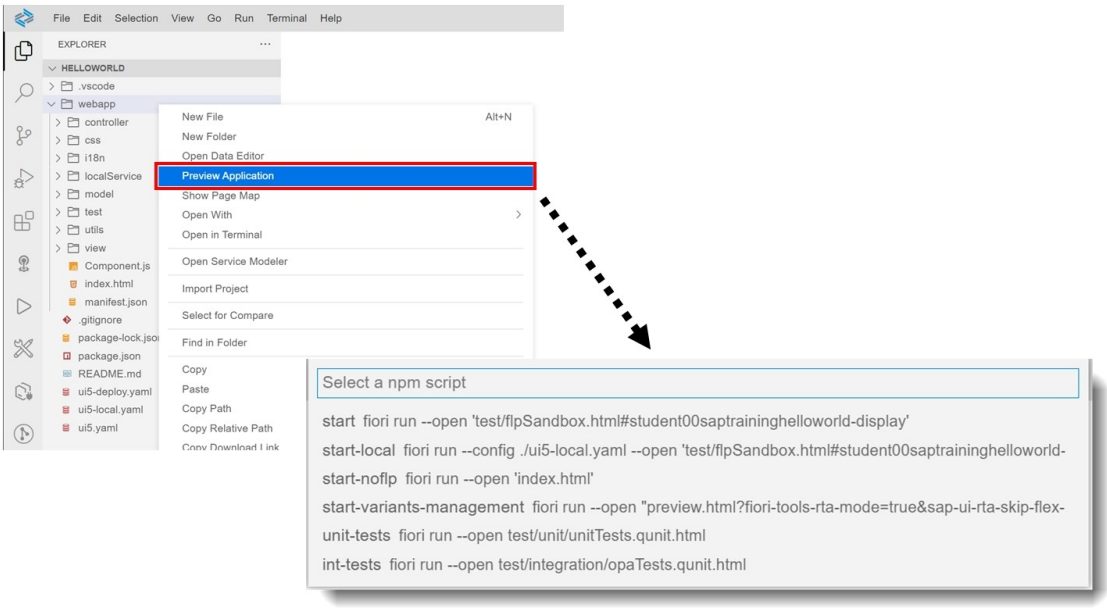


Figure 24: Starting a Preview from Context Menu

It is also possible to start a preview from the context menu. To it, right-click on any subfolder in your project and select *Preview Application* from the context menu that appears. You are then provided with the following options, among others:

- **`start`**: Starts the application using `npm start` and embeds it in an SAP Fiori launchpad sandbox. If the application uses an OData service, it is called using the live service to retrieve real data against the hosted version of SAPUI5 that was selected during project generation.
- **`start-noflp`**: Starts the application via the HTML page index.html using `npm start-noflp`. The application is not embedded in an SAP Fiori launchpad sandbox.

Connecting Cloud Applications to On-premise Systems

The Cloud Connector combines an easy setup with a clear configuration of the systems that are exposed to the SAP Business Technology Platform. In addition, you can control the resources available for the cloud applications in those systems. Thus, you can benefit from your existing assets without exposing the entire internal landscape.

The Cloud Connector runs as an on-premise agent in a secured network and acts as a reverse invoke proxy between the on-premise network and the SAP Business Technology Platform. Due to its reverse invoke support, you do not need to configure the on-premise firewall to allow external access from the cloud to internal systems.

The Cloud Connector provides fine-tuned control over the following:

- On-premise systems and resources that are accessible by cloud applications
- Cloud applications that make use of the Cloud Connector

The Cloud Connector allows the propagation of the identity of cloud users to on-premise systems in a secure way.

The Cloud Connector can be used to connect on-premise database, or BI tools to SAP S/4HANA databases in the cloud. That means it also supports the opposite connection direction (that is, from the on-premise system to the cloud).

© Copyright. All rights reserved.

You can use the Cloud Connector in business-critical enterprise scenarios. The tool automatically re-establishes broken connections, provides audit logging of the inbound traffic and configuration changes, and can be run in a high-availability setup.

The Cloud Connector supports protocols in addition to HTTP. For example, the RFC protocol supports native access to ABAP systems by invoking function modules.

The Cloud Connector must not be used with products other than SAP Business Technology Platform or SAP S/4HANA Cloud.

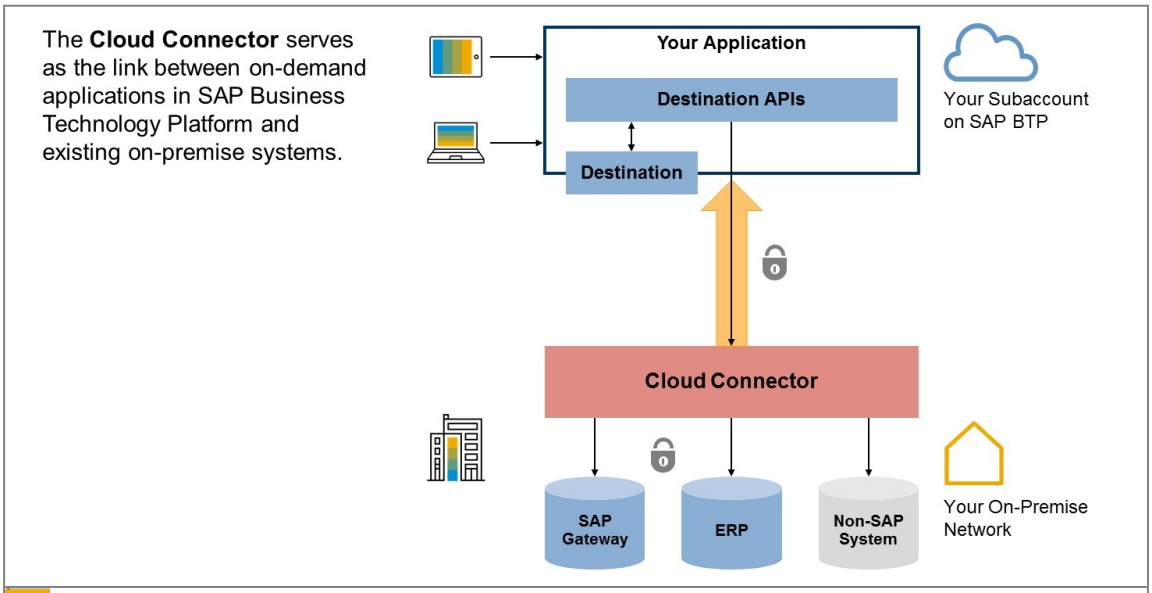


Figure 25: Connecting Cloud Applications to On-premise Systems

SAP Business Technology Platform Cockpit – Destinations

Connectivity *Destinations* are part of the SAP Business Technology Platform connectivity service.

The connectivity service resolves the *Destination* at runtime based on the symbolic name provided. The result is an object that contains customer-specific configuration details, such as the URL of the remote system or service, and the authentication type.

© Copyright. All rights reserved.

- *Connectivity Destinations* are used for the outbound communication of a cloud application to a remote system.
- *Destinations* contain the connection details for the remote communication of an application.
- *Destinations* are represented by symbolic names that are used by on-demand applications to refer to remote connections.

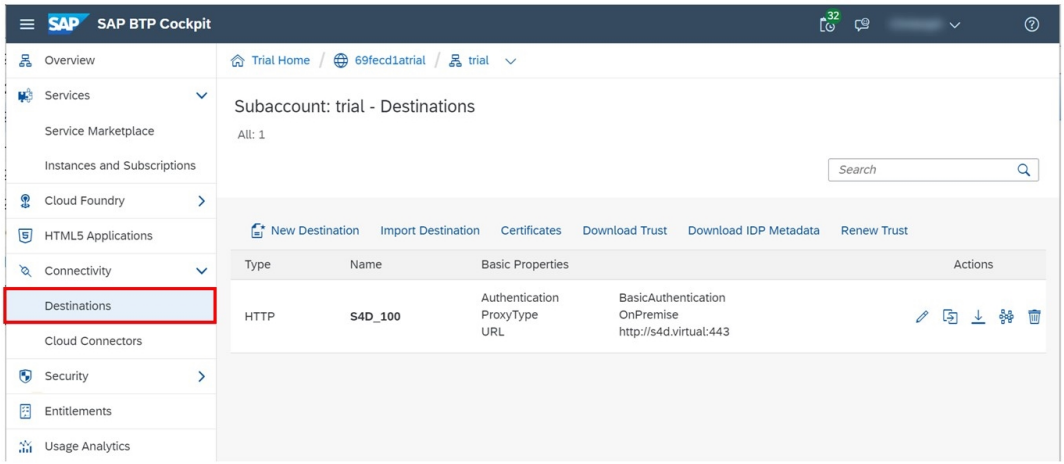


Figure 26: SAP Business Technology Platform Cockpit – Destinations

Destination Configuration for Establishing a Connection to an ABAP Back-end System

The currently supported *Destination* types are as follows:

- HTTP destination:

  The HTTP destination provides data communication using the HTTP protocol and is used for both Internet and on-premise connections.

- Mail destination:

  The Mail destination specifies an email provider for sending and retrieving emails using the SMTP, IMAP, and POP3 protocols.

- RFC destination:

  The RFC destination makes connections to ABAP on-premise systems using the RFC protocol with JCo as the API.

- LDAP destination:

  For your cloud applications, you can use LDAP-based user management if you are operating an LDAP server within your network. LDAP destinations carry connectivity details for accessing systems over Lightweight Directory Access Protocol (LDAP). In combination with the Cloud Connector they enable SAP BTP applications to access LDAP servers in an on-premise corporate network. LDAP destinations are intended to be used with the Java JNDI/LDAP Service Provider.

The Location ID provides an advanced option when using more than one Cloud Connector for a subaccount. Beginning with Cloud Connector 2.10.0, you can connect multiple Cloud Connectors to a subaccount if their Location ID is different.

© Copyright. All rights reserved.

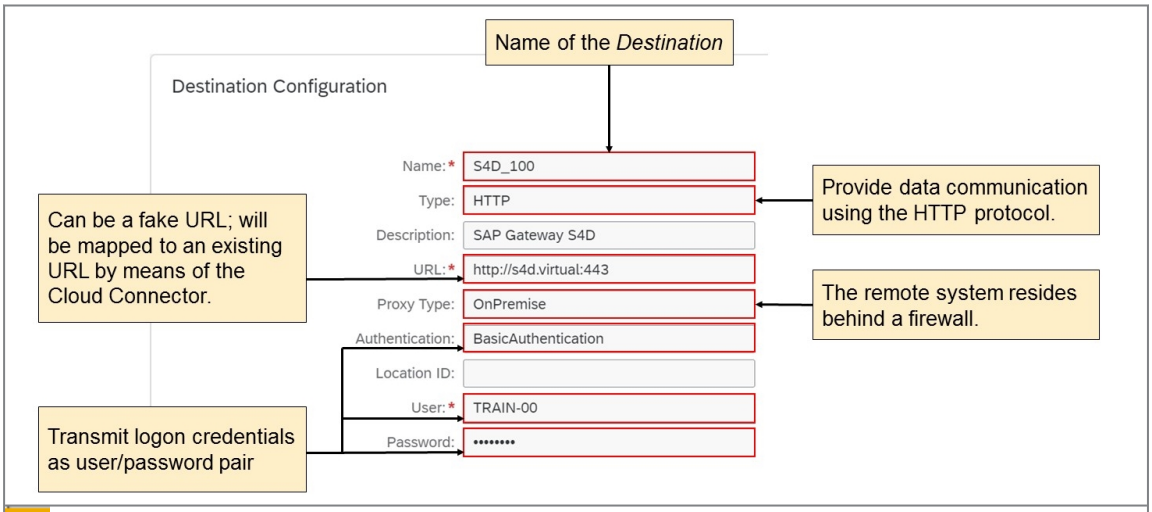


Figure 27: Destination Configuration for Establishing a Connection to an ABAP Back-end System (1)

If you want to connect to an ABAP system, you need to specify this in the *Destination* details. Let us assume that you want to connect to an ABAP system to consume some OData service for creating a new SAP Fiori app, or for extending some existing SAP Fiori apps. You need to pass this information to the SAP Business Application Studio. The way to do this is through the Destination's additional property *WebIDEUsage*. In this property, you need to specify the usages you want to have with this *Destination*, which are as follows:

- `odata_abap`:

  For the OData functionality of the SAP Gateway (corresponds to URL path: `/sap/opu/odata`).

- `dev_abap`:

  For extensibility scenarios and developing or deploying to an SAPUI5 ABAP Repository (corresponds to URL path: `/sap/bc/adt`).

When you enter multiple usages for a *Destination*, separate them using commas without spaces (for example, `dev_abap,odata_abap`).

Refer to the SAP Business Application Studio documentation for more information.

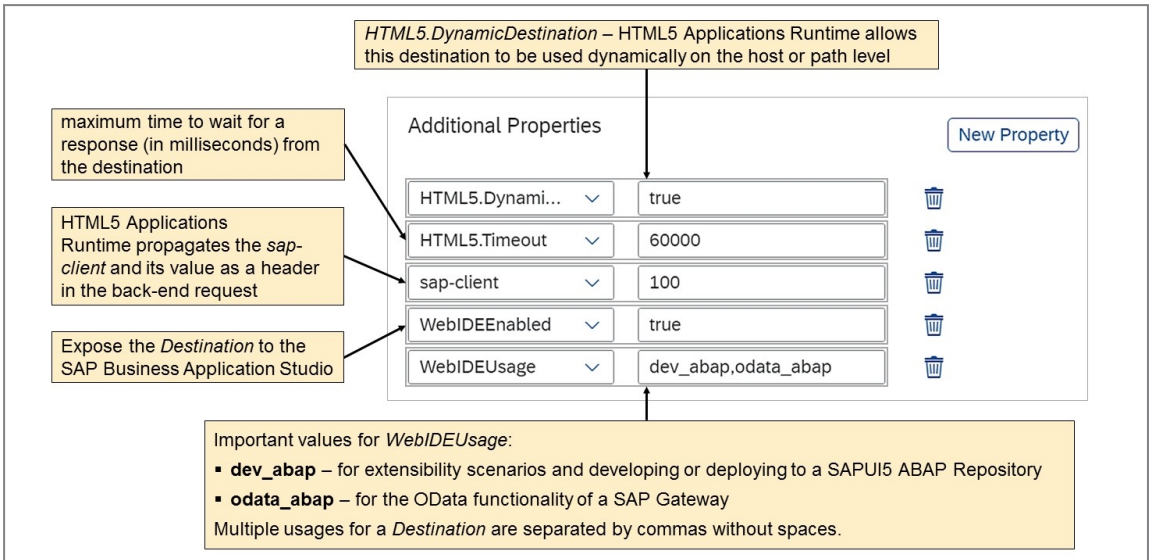


Figure 28: Destination Configuration for Establishing a Connection to an ABAP Back-end System (2)

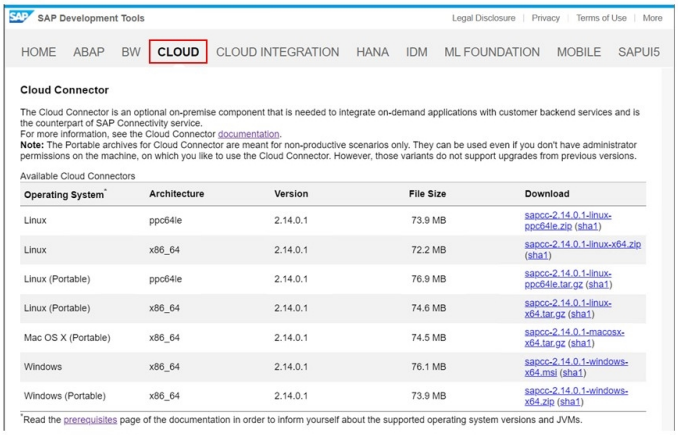
© Copyright. All rights reserved.

Installing the Cloud Connector on a Microsoft Windows OS

If you want to use the SAP Java Virtual Machine (JVM), you can download it from https://tools.hana.ondemand.com/#cloud.

The MSI-based installer variant of the Cloud Connector handles registration of the Cloud Connector as a Windows service, and in this way, automatically starts it after a machine reboot. Therefore, installation requires administration permissions.

The portable version does not support the automatic upgrade procedure.

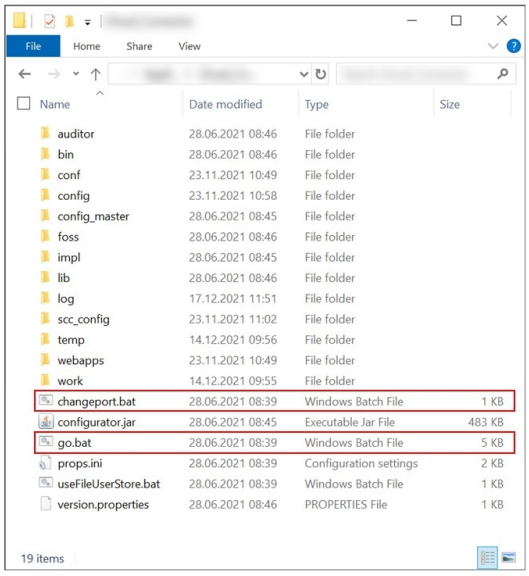


- The *Cloud Connector* can be downloaded from *https://tools.hana.ondemand.com/#cloud*.
- You can choose between a simple *portable* variant and the MSI-based *installer*.
- The *installer* is the generally recommended means that can be used for both developer and productive scenarios.
- Using the *portable* variant, you can run the *Cloud Connector* after a simple unzip (archive extraction). This might be useful if:
  - You cannot perform a true installation due to lack of permissions.
  - You need to use multiple versions of the *Cloud Connector* simultaneously on the same machine.

Figure 29: Installing the Cloud Connector on a Microsoft Windows OS

Using the Portable Variant of the Cloud Connector on a Microsoft Windows OS

Instructions for using the portable variant of the Cloud Connector on a Microsoft Windows OS are provided in the figure, Using the Portable Variant of the Cloud Connector on a Microsoft Windows OS.



- Set the environment variable *JAVA_HOME* to the installation directory of the JDK you want to use to run the *Cloud Connector*. (Alternatively, you can add the bin subdirectory of the JDK installation directory to the *PATH* environment variable.)
- Extract the ZIP file to an arbitrary directory on your local file system.
- Change to this directory and start the *Cloud Connector* via the **go.bat** batch file.
- Note:
  - When using the *portable* variant, the *Cloud Connector* is not started as a service, and hence will not automatically start after a reboot of your system.
  - Using the utility **changeport.bat,** you can change the *Cloud Connector* default port.

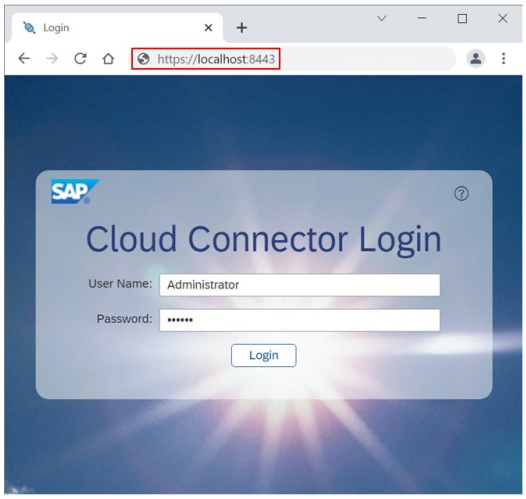
Figure 30: Using the Portable Variant of the Cloud Connector on a Microsoft Windows OS

© Copyright. All rights reserved.

Cloud Connector Initial Login

Once the Cloud Connector has been installed and the Cloud Connector daemon has been started, you can log in and perform the necessary customizing to make your Cloud Connector operational.

To administer the Cloud Connector, you need a Web browser. To log in to the Cloud Connector, in a web browser, enter **`https://<hostname>:<port>`**, where `hostname` refers to the machine on which the Cloud Connector has been installed (if installed on your machine, enter **`localhost`**), and `port` is the Cloud Connector port (the default port is 8443).



- In a web browser, enter *https://<hostname>:8443*, where *<hostname>* is the host name of the machine on which you have installed the *Cloud Connector*.

  If you access the *Cloud Connector* locally from the same machine, you can just enter *localhost*.

- For *User Name/Password* enter **Administrator / manage** (case sensitive).

  When you first log in, you must change the password before you continue.

Figure 31: Cloud Connector Initial Login

Cloud Connector Initial Configuration

When you first log in, the Cloud Connector requests the following information (some of which is optional):

- For *Region*, specify the SAP Business Technology Platform host that should be used. You can choose the one you need from the dropdown list.
- For *Subaccount*, *Login E-Mail*, and *Password*, enter the values that belong to your registration on the SAP Business Technology Platform.
- Optionally, specify a *Display Name*, which allows you to easily recognize a specific subaccount in the UI as an alternative to the technical *Subaccount Name*.
- Optionally, you can define a *Location ID* identifying the location of this Cloud Connector for a specific subaccount. Starting with Cloud Connector release 2.9.0, the *Location ID* is used as routing information and therefore it is possible to connect multiple Cloud Connectors to a single subaccount. If not specifying any value for *Location ID*, the default is used, which is consistent with the behavior of previous Cloud Connector versions. The *Location ID* must be unique per subaccount and should be an identifier that can be used in a URL. To route requests to a Cloud Connector with a *Location ID*, the *Location ID* must be configured in the respective Destinations.
- You need to specify an HTTPS proxy if your internal landscape is protected by a firewall that blocks any outgoing TCP traffic, so that the Cloud Connector can use the proxy to connect to the SAP Business Technology Platform. Typically, you use the same proxy settings as those used by your standard web browser.

© Copyright. All rights reserved.

- Optionally, you can provide a *Description* (free-text) of the subaccount. This is shown when you choose the *Details* icon in the *Actions* column of the *Subaccount Dashboard*. The *Description* helps you identify the particular Cloud Connector you use.

Once the initial setup has been completed successfully, the Cloud Connector starts a handshake with the SAP Business Technology Platform and attempts to establish a secure SSL tunnel to the subaccount in which your on-demand applications are running. However, no requests are yet allowed to pass from the cloud side to any of your internal back-end systems. To allow your on-demand applications to access specific internal back-end systems, proceed with the *Access Control* setup.

Once connected, you can also monitor the Cloud Connector in the *Connectivity* section of the SAP Business Technology Platform Cockpit. There, you can track attributes like version, description, and high-availability set up. Every Cloud Connector configured for your subaccount appears automatically in the *Connectivity* section.

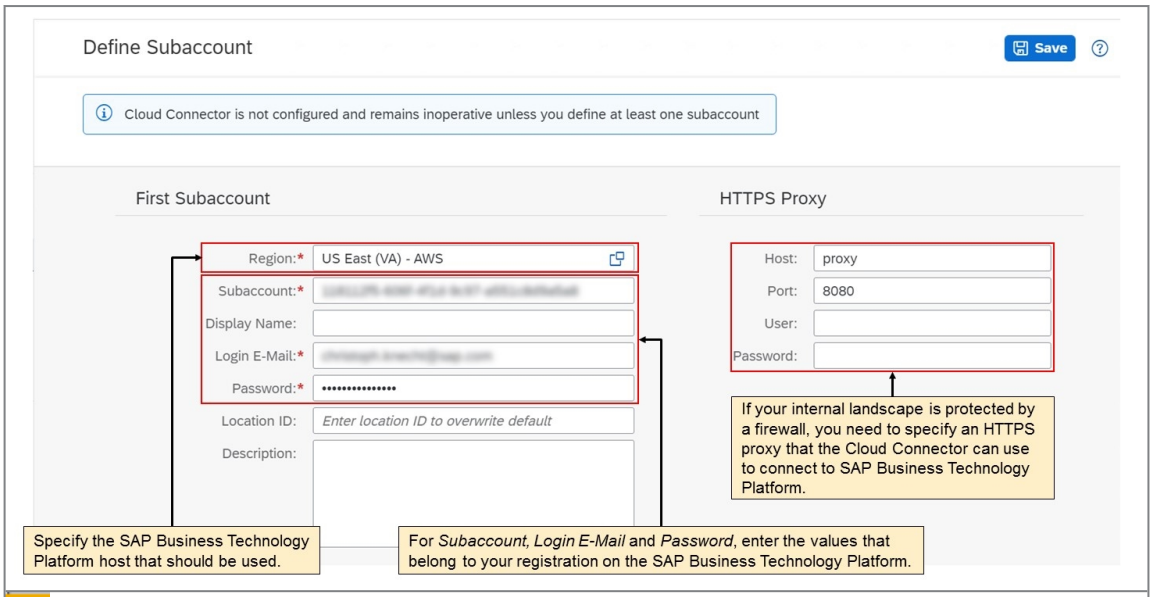


Figure 32: Cloud Connector Initial Configuration

Cloud Connector Configuring Access Control (HTTP)

In the *Cloud Connector Administration* screen:

1. Choose *Cloud To On-Premise* from your subaccount menu.
2. Choose the *Add* (+) button. A wizard opens and asks for the following required values:
   - *Back-end Type:*

     Select the description that best matches the addressed back-end system. This is important mainly for metering information; tunnel connections to any kind of SAP system are free of charge, while using the tunnel for connecting to a non-SAP system costs a fee. Furthermore, this selection defines which steps the wizard offers and which values are possible.
   - *Protocol*:

     This field allows you to decide whether the Cloud Connector should use HTTP or HTTPS for the connection to the back-end system. Note that this is completely

© Copyright. All rights reserved.

independent from the setting on the cloud side. Therefore, even if the HTTP destination on the cloud side specifies "http://" in its URL, you can select HTTPS. This way, you are ensured that the entire connection from the on-demand application to the actual back-end system (provided through the SSL tunnel) is SSL-encrypted. The only prerequisite is that the back-end system supports HTTPS on that port.

- *Internal Host* and *Internal Port*:

  Specify the actual host and port under which the target system can be reached within the intranet. The address must be an existing network address that can be resolved on the intranet and has network visibility for the Cloud Connector without any proxy. The Cloud Connector tries to forward the request to the network address specified by the internal host and port, so this address must be real.

- *Virtual Host* and *Virtual Port*:

  Specifies the host name and port exactly as it is specified as the URL property in the HTTP destination configuration in the SAP Business Technology Platform. The virtual host can be a fake name and does not need to exist. The *Virtual Port* allows you to distinguish between different entry points of your back-end system, for example, HTTP/80 and HTTPS/443, and have different sets of access control settings for each of them. For example, some non-critical resources may be accessed by HTTP, while some other critical resources are to be called using HTTPS only.

- *Principal Type:*

  Defines what kind of principal is used when configuring a destination on the cloud side using this system mapping with authentication type *Principal Propagation*. Regardless of the type you choose, you must ensure that the general configuration for the principal type has been performed for it to work correctly. For destinations using different authentication types, this setting is ignored. If you choose **`None`** as the principal type, it is not possible to use principal propagation to this system.

- *Host In Request Header:*

  Lets you define, which host is used in the host header that is sent to the target server. By choosing *Use Internal Host*, the actual host name is used. When choosing *Use Virtual Host*, the virtual host is used. In the first case, the virtual host is still sent via the *X-Forwarded-Host* header.

- *Description:*

  An optional parameter. The respective description will be shown when pressing the *Show details* button of the access control entry (table *Mapping Virtual to Internal System*).

© Copyright. All rights reserved.

- To allow your on-demand applications to access a certain back-end system on the intranet, you need to insert an extra line into the Cloud Connector ***Access Control*** management.

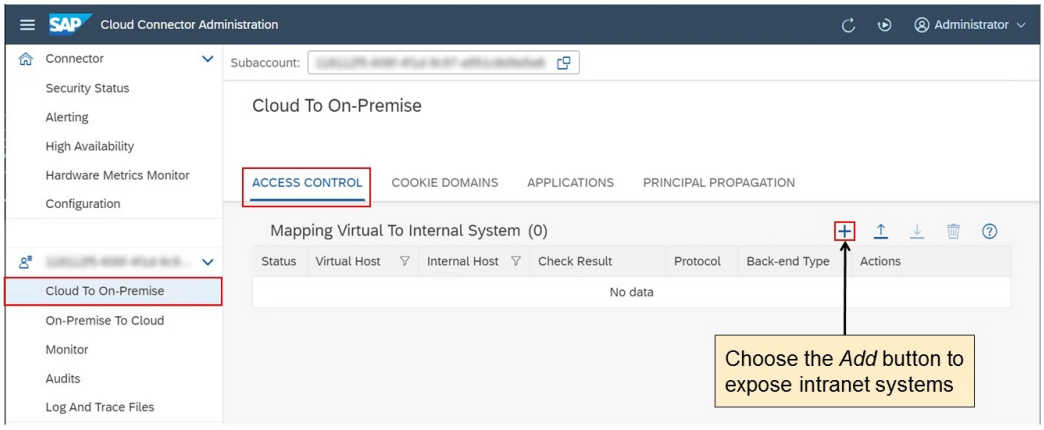


Figure 33: Cloud Connector Configuring Access Control (HTTP)

Cloud Connector – Exposing Intranet Systems

In addition to allowing access to a particular host and port, you also need to specify which URL paths (Resources) are allowed to be invoked on that host. The Cloud Connector uses strict white-lists for its access control, so only those URLs for which you explicitly grant access are allowed. All other HTTP(S) requests are denied by the Cloud Connector.

To define the permitted URLs (Resources) for a particular back-end system, choose the line corresponding to that back-end system and choose *Add* in the *Resources Of...* section, as shown in the figure, Cloud Connector — Exposing Intranet Systems. A dialog appears prompting you to enter the specific URL path that you want to allow to be invoked.

The Cloud Connector checks that the path part of the URL (up to but not including a possible question mark (?) that may denote the start of optional CGI-style query parameters) is exactly as specified in the configuration. If it is not, the request is denied. If you select the *Path And All Sub-Paths* option, the Cloud Connector allows all requests for which the URL path (not considering any query parameters) begins with the specified string.

The *Active* checkbox allows you to specify whether that resource is initially enabled or disabled. Enabling or disabling resources is useful for testing purposes, to temporarily disable certain resources without having to delete them from the configuration. This allows you to easily provide access again to these resources at a later point, without having to reenter everything.

© Copyright. All rights reserved.

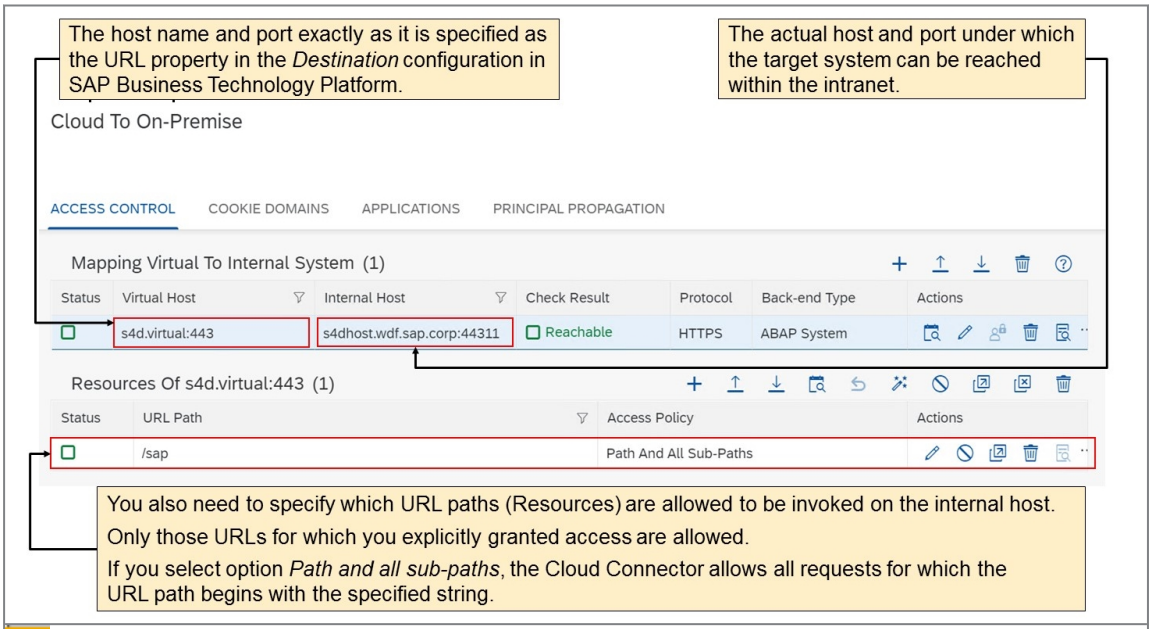


Figure 34: Cloud Connector – Exposing Intranet Systems

Deploying Applications to the SAPUI5 ABAP Repository

SAP Fiori tools support deployment to ABAP as well as deployment to Cloud Foundry on SAP Business Technology Platform.

For deployment to ABAP, the SAPUI5 Repository service exposed by the ABAP system is used to upload a deployment artifact. The ABAP back-end provides all functionality to run the application, such as hosting, routing, and authentication. As a result, the deployment artifact is just an SAP Fiori application. In other words, it is the zipped `dist` folder of the SAP Fiori tools project.

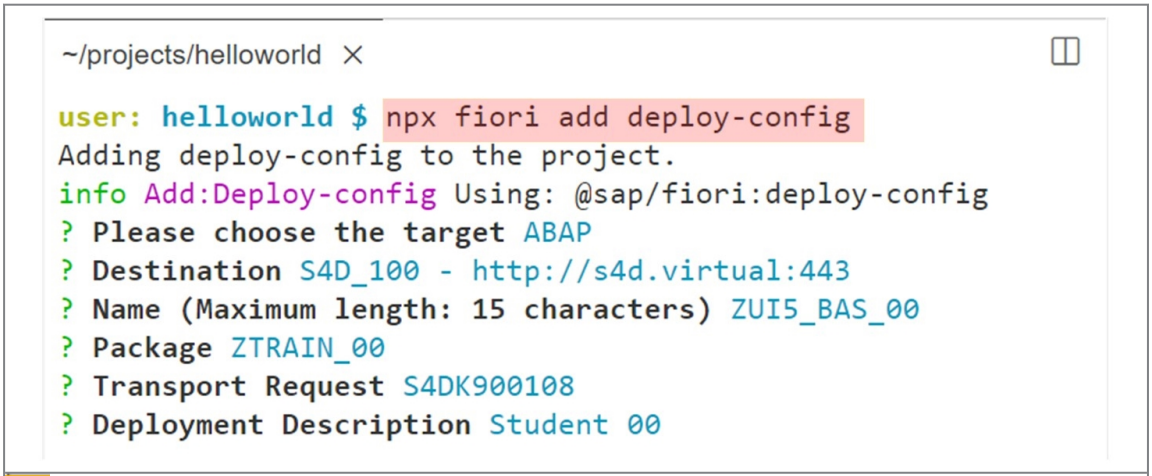


Figure 35: Generating a Deployment Configuration

To deploy an application, you need a so-called deployment configuration (file `ui5-deploy.yaml`). If the project does not yet contain such a file, you can create it using the following command: `npx fiori add deploy-config`. You are prompted for required information and then the file is created based on your input. In addition to creating the configuration, the create deployment configuration command will also update the `package.json` file in the project, so that you can execute `npm run deploy` afterwards to deploy the application.

© Copyright. All rights reserved.

```
~/projects/helloworld ×

user: helloworld $ npm run deploy
          [...]
Confirmation is required to deploy the app:

    Application Name: ZUI5_BAS_00
    Package: ZTRAIN_00
    Transport Request: S4DK900108
    Destination: S4D_100
    SCP: false


✓ Start deployment (Y/n)?

 … yes
          [...]
info builder:custom deploy-to-abap Deployment Successful.
```

Figure 36: Deploying an Application

To deploy an application to an ABAP system use the command `npm run deploy`. When prompted, check the deployment configuration and press **Y** to proceed. If authentication to the back-end is required, a prompt for username and password appears.

Please refer to the documentation for details on the deployment.

LESSON SUMMARY

You should now be able to:

- Explore the SAP Business Application Studio

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which of the following statements are true with respect to the SAP Business Application Studio?

   Choose the correct answers.

   - [ ] A SAP Business Application Studio is a browser-based integrated development environment
   - [ ] B SAP Business Application Studio is only available as on-premise edition
   - [ ] C SAP Business Application Studio is integrated in the SAP GUI
   - [ ] D SAP Business Application Studio is based on an open source IDE
   - [ ] E SAP Business Application Studio provides WYSIWYG tooling

2. What tool is used to connect the SAP Business Technology Platform to an on-premise system?

   Choose the correct answer.

   - [ ] A Destination Connector
   - [ ] B Cloud Connector
   - [ ] C Connector App
   - [ ] D Cloud Manager

3. Which of the following parameters must you provide when you create an initial Cloud Connector configuration?

   Choose the correct answers.

   - [ ] A Region
   - [ ] B Display Name
   - [ ] C Subaccount
   - [ ] D Password
   - [ ] E Location ID
   - [ ] F Description

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which of the following statements are true with respect to the SAP Business Application Studio?

   Choose the correct answers.

   [X] A SAP Business Application Studio is a browser-based integrated development environment

   [ ] B SAP Business Application Studio is only available as on-premise edition

   [ ] C SAP Business Application Studio is integrated in the SAP GUI

   [X] D SAP Business Application Studio is based on an open source IDE

   [X] E SAP Business Application Studio provides WYSIWYG tooling

   Correct. The SAP Business Application Studio is a browser-based integrated development environment that is based on an open source IDE, and it includes WYSIWYG tooling.

2. What tool is used to connect the SAP Business Technology Platform to an on-premise system?

   Choose the correct answer.

   [ ] A Destination Connector

   [X] B Cloud Connector

   [ ] C Connector App

   [ ] D Cloud Manager

   Correct. The Cloud Connector is used to connect the SAP Business Technology Platform with on-premise systems. The Cloud Connector is installed on-premise.

© Copyright. All rights reserved.

3. Which of the following parameters must you provide when you create an initial Cloud Connector configuration?

   Choose the correct answers.

   - [X] A Region
   - [ ] B Display Name
   - [X] C Subaccount
   - [X] D Password
   - [ ] E Location ID
   - [ ] F Description

   Correct. To configure the Cloud Connector, you must specify the region to which you want to connect. You must also specify the subaccount, and the corresponding password.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 4 Load and Initialize SAPUI5

Lesson 1

Loading and Initializing SAPUI5 55

UNIT OBJECTIVES

- Load and initialize SAPUI5

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 4
Lesson 1

Loading and Initializing SAPUI5

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Load and initialize SAPUI5

Bootstrapping

To use SAPUI5 features in your HTML page, you have to load and initialize the SAPUI5 library.

If you run your app as a 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 code snippet in the figure, Bootstrapping, shows a typical bootstrap script tag.

SAPUI5 provides several bootstrap files for different use cases. **`sap-ui-core.js`** is the standard bootstrap file, which SAP recommends to use for typical use cases.

SAPUI5 can either be loaded locally with a relative path from a Web server or externally from a Content Delivery Network (CDN).

> 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.

You can refer to a specific version on the CDN by using a versioned URL such as the following: `https://sapui5.hana.ondemand.com/1.84.20/resources/sap-ui-core.js`.

Here, the first segment of the URL after the host name is used to specify a concrete version.

Alternatively, you can use the default version of the SAPUI5 libraries which has the generic URL `https://sapui5.hana.ondemand.com/resources/sap-ui-core.js`.

> Note:
> 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.

© Copyright. All rights reserved.

- To use SAPUI5, you must load and initialize the SAPUI5 library.
- If you run your app standalone, you can add the SAPUI5 bootstrap script 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.
- **sap-ui-core.js** is the standard bootstrap file which contains the mimimum required parts of the SAPUI5 core.
- Bootstrap files can either be loaded locally with a relative path from a Web server or externally from a Content Delivery Network (CDN).

```
<script
    id="sap-ui-bootstrap"
    src="resources/sap-ui-core.js"
    data-sap-ui-libs="sap.m"
    data-sap-ui-theme="sap_fiori_3"
    data-sap-ui-resourceroots='{"student00.sap.training.helloworld": "./"}'
></script>
```

Standard bootstrap file

Figure 37: Bootstrapping

Configuration of the SAPUI5 Runtime

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.

You can use the following methods to provide configuration information:

- *Default Values:*

  The SAPUI5 runtime contains a default value for each configuration option.

- *Individual Script Tag Attributes:*

  For each configuration option, you can have one attribute in the bootstrap script tag. The attribute name is composed of the name of the configuration option and the `data-sap-ui-`prefix.

- *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.

- *Global Configuration Object*:

  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.

  As the configuration is evaluated during bootstrap, the configuration object must be created before SAPUI5 is bootstrapped. As a consequence, using the global configuration object requires another script tag in front of the bootstrap script tag.

- *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 `index.html?sap-ui-debug=true`.

  For security reasons, only some configuration options can be set via URL parameters.

© Copyright. All rights reserved.

- *Runtime Configuration Object:*

  The runtime configuration object provides set methods for a very limited set of configuration options that can be modified at runtime. The most prominent example for such a configuration option is the theme.

  You can access the runtime configuration object by using the `sap.ui.getCore().getConfiguration()` method.

- The SAPUI5 runtime can be configured using so-called **configuration options**.
- There are different ways to set values for these configuration options.
- For each configuration option, SAPUI5 accepts an attribute in the bootstrap script tag whose name is composed of the name of the configuration option and the prefix **data-sap-ui-**.

```
<script
    id="sap-ui-bootstrap"
    src="resources/sap-ui-core.js"
    data-sap-ui-libs="sap.m"
    data-sap-ui-theme="sap_fiori_3"
    data-sap-ui-resourceroots='{"student00.sap.training.helloworld": "./"}'
></script>
```

The configuration option **libs** defines a list of libraries that shall be loaded initially.

The configuration option **theme** defines the theme that shall be used.

Prefix required when using configuration options as attributes within the bootstrap script tag.

Figure 38: Configuration of the SAPUI5 Runtime

Application Script and UI Area

In the application script, you create controls such as `Button` and `Input` UI elements.

You can display your controls in an HTML element called UI area by invoking the `placeAt()` method (there can be multiple UI areas).

- After the bootstrap script tag, an **application script** can follow in which the UI5 application is written.
- Controls can be displayed in an HTML element called **UI area** by invoking the *placeAt()* method.

index.html ×

webapp > index.html > ...

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <script id="sap-ui-bootstrap" src="resources/sap-ui-core.js"
5          data-sap-ui-libs="sap.m"
6          data-sap-ui-theme="sap_fiori_3"
7          data-sap-ui-resourceroots='{"student00.sap.training.helloworld": "./"}'>
8      </script>
9      <script>
10         var oButton = new sap.m.Button({ text: "Hello World" });
11         oButton.placeAt("target");
12     </script>
13 </head>
14 <body class="sapUiBody sapUiSizeCompact" id="content">
15     <div id="target"></div>
16 </body>
17 </html>
```

Bootstrap script

Application script

UI area

Figure 39: Application Script and UI Area

© Copyright. All rights reserved.

LESSON SUMMARY

You should now be able to:

- Load and initialize SAPUI5

© Copyright. All rights reserved.

Learning Assessment

1. What is the initial starting phase of SAPUI5 called?

   Choose the correct answer.

   - ☐ A Bootstrapping
   - ☐ B AutoStart
   - ☐ C ConfigurationLoad
   - ☐ D Startup-phase

2. What is the name of the event that is fired after a successful SAPUI5 initialization?

   Choose the correct answer.

   - ☐ A `init`
   - ☐ B `startUpDone`
   - ☐ C `loadDone`
   - ☐ D `bootDone`

3. What is the prefix of the HTML custom attributes of SAPUI5?

   Choose the correct answer.

   - ☐ A `data-sap-ui5`
   - ☐ B `data-sap-ui`
   - ☐ C `data-sap-uiconfig`
   - ☐ D `data-sap-config-ui`

© Copyright. All rights reserved.

Learning Assessment - Answers

1. What is the initial starting phase of SAPUI5 called?

   Choose the correct answer.

   - [X] A Bootstrapping
   - [ ] B AutoStart
   - [ ] C ConfigurationLoad
   - [ ] D Startup-phase

   Correct. The initial starting phase of SAPUI5 is called “bootstrapping”. In the bootstrapping phase, all of the necessary runtime artefacts are instantiated and configured to execute the SAPUI5 application.

2. What is the name of the event that is fired after a successful SAPUI5 initialization?

   Choose the correct answer.

   - [X] A `init`
   - [ ] B `startUpDone`
   - [ ] C `loadDone`
   - [ ] D `bootDone`

   Correct. The event that is fired when the initialization phase is complete is called `init`.

3. What is the prefix of the HTML custom attributes of SAPUI5?

   Choose the correct answer.

   - [ ] A `data-sap-ui5`
   - [X] B `data-sap-ui`
   - [ ] C `data-sap-uiconfig`
   - [ ] D `data-sap-config-ui`

   Correct. All HTML custom attributes of SAPUI5 begin with the prefix `data-sap-ui`, where `data` is the common prefix of all HTML5 custom attributes, and `sap-ui` is the SAP-specific part.

© Copyright. All rights reserved.

UNIT 5 Dependency Management

Lesson 1

Managing Dependencies 63

UNIT OBJECTIVES

- Modularize SAPUI5 applications

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 5
Lesson 1

Managing Dependencies

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Modularize SAPUI5 applications

Modularization

The figure, Modularization, describes the concept of modularization in the SAPUI5 framework.

- The SAPUI5 framework has built-in support for modularizing JavaScript applications.
- An application can be split into smaller parts which can be loaded at runtime at the time when they are needed.
- These smaller individual files are called **modules**.
- Every module defines a list of dependencies that have to be resolved before the module can be used.

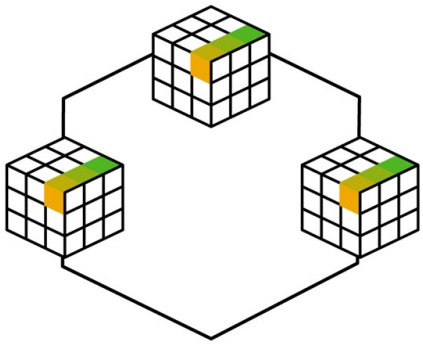


Figure 40: Modularization

sap.ui.require() and sap.ui.define()

`sap.ui.define()` is designed to support real Asynchronous Module Definitions (AMDs).

Synchronous loading of modules was possible using `jQuery.sap.require()`. Declaring a module was possible with `jQuery.sap.declare()`. Beginning with SAPUI5 version 1.52, both of these jQuery functions are deprecated.

To make **asynchronous loading** of modules and prerequisite dependencies possible, you can use *sap.ui.define*() and *sap.ui.require*().

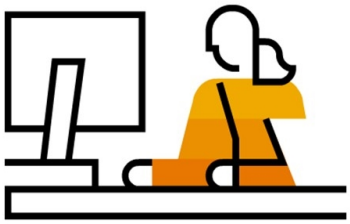


Figure 41: sap.ui.require() and sap.ui.define()

© Copyright. All rights reserved.

Defining a JavaScript Module

The typical, suggested usage of this method is to have one single, top-level call to `sap.ui.define()` in one Javascript resource (file). When a module is requested by its name for the first time, the corresponding resource is determined from the name and the current configuration. The resource is loaded and executed, which in turn executes the top-level `sap.ui.define()` call.

If the module name was omitted from that call, it is substituted by the name that was used to request the module. As a preparation step, the dependencies, in addition to their transitive dependencies, are loaded. Then, the module value is determined. If a static value (object, literal) was given, that value is the module value. If a function was given, that function is called (providing the module values of the declared dependencies as parameters to the function) and its return value is used as the module value. The framework internally associates the resulting value with the module name and provides it to the original requestor of the module.

The order of the parameters of the factory function matches the order of the modules in the dependencies array.

`sap.ui.define()` uses a simplified variant of the unified resource name syntax for the module's own name as well as for its dependencies. The only difference to that syntax is that for `sap.ui.define()` and `sap.ui.require()`, the extension (which is always `.js`) has to be omitted. Both methods add this extension internally.

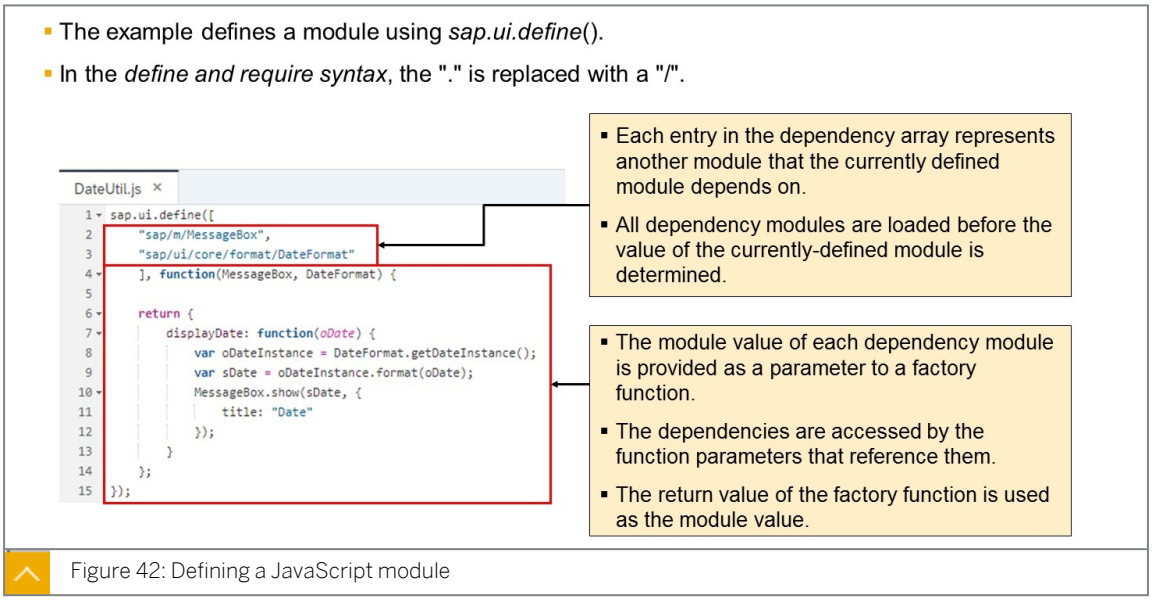


Figure 42: Defining a JavaScript module

Example from the SAPUI5 Library – sap.m.MessageBox

`sap.ui.define (sModuleName, aDependencies, vFactory, bExport)` defines a JavaScript module with its name, its dependencies, and a module value or factory.

The parameters are as follows:

- `{string}` *sModuleName*:

  Name of the module in simplified resource name syntax. When omitted, the loader determines the name from the request.

- `{string[]}` *aDependencies*:

  List of dependencies of the module.

© Copyright. All rights reserved.

- `{function|any}` *vFactory*:

  The module value or a function that calculates the value.

- `{boolean}` *bExport*:

  Whether an export to global names is required; should be used by SAP-owned code only.

As a convenience, the name of a dependency can start with the segment '`./`', which is replaced by the name of the package that contains the currently defined module (relative name).

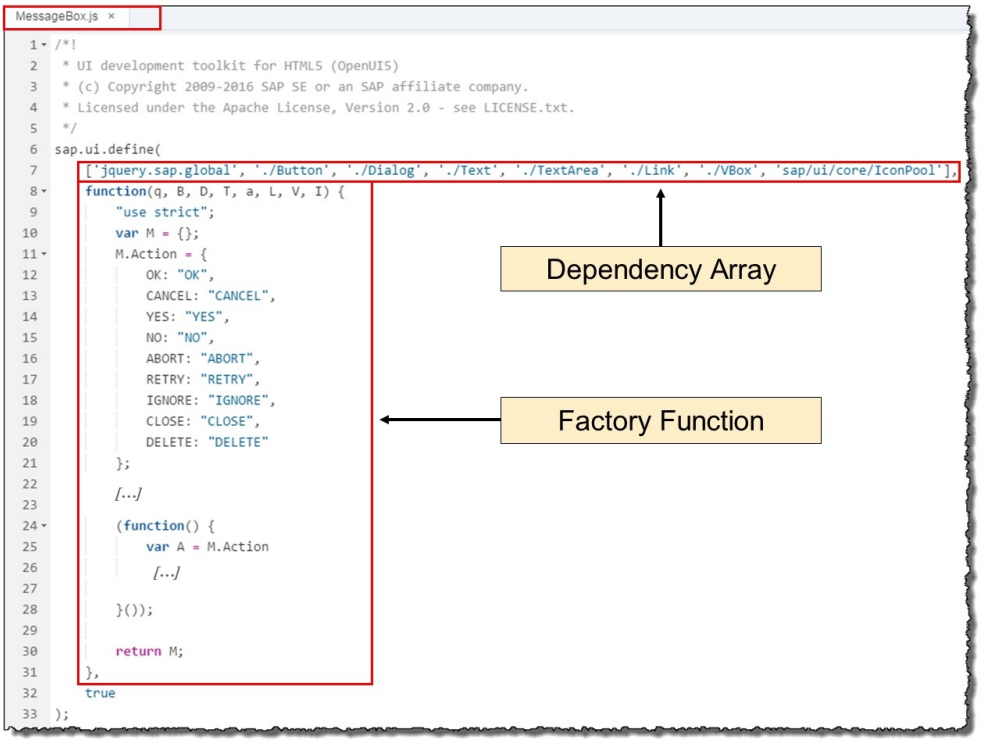


Figure 43: Example from the SAPUI5 Library – sap.m.MessageBox

Asynchronous Loading of Modules

The standard variant for bootstrapping loads all JavaScript modules of a library in advance with a single request for performance reasons.

A 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 round-trips since the single modules are bundled in one file.

> Note:
> An application must not reference the `library-preload.js`. If preload files exist, SAPUI5 automatically loads them.

As an alternative to the configuration option `async`, 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), **`sync`**, or **`auto`**

© Copyright. All rights reserved.

> Note:
> Using the **`async=true`** or **`preload=async`** configuration parameters requires extensive testing and cooperation on the application side to ensure a stable and fully-working application.

- For the SAPUI5 libraries (such as *sap.ui.core* or *sap.m*) there are so-called preload files named *library-preload.js* which contain all modules of a certain library.
- By setting the configuration option **async=true**, the module loader loads the modules and preload files asynchronously.
- The modules contained in a preload file are only loaded, but not executed until they are used by the application.

```
<script id="sap-ui-bootstrap"
    src="../../resources/sap-ui-core.js"
    data-sap-ui-libs="sap.m"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-resourceroots='{"student00.sap.training.moduledemo": "./"}'
    data-sap-ui-compatVersion="edge"
    data-sap-ui-async="true">
</script>
```

enables the module loader to load both modules and preload files asynchronously

Figure 44: Standard Variant for Bootstrapping (1)

The `attachInit()` method of the core registers a given function that is executed after the framework has been initialized. The given function will either be called as soon as the framework has been initialized or, if it has been initialized already, it will be called immediately.

- The initialization process of the SAPUI5 runtime comprises that all libraries and modules declared in the configuration as well as their dependencies are loaded.
- When all libraries are loaded and the document is ready, the *init* event of the core is fired.

To implement your app, register a function in the *application script* that is executed after the framework has been initialized.

```
<script>
    sap.ui.getCore().attachInit(function() { ... });
</script>
```

Figure 45: Standard Variant for Bootstrapping (2)

By default, SAPUI5 loads modules in relation to the folder from which the SAPUI5 runtime was loaded (the resources folder). This fails for modules that are contained in your web application. Using the configuration option `resourceRoots`, it is possible to redirect the request for the application-specific modules to the corresponding web application.

© Copyright. All rights reserved.

The strings in the array of the `sap.ui.require()` method are interpreted as module names and the corresponding modules (and their transitive dependencies) are loaded asynchronously.

The module values of the specified modules are provided as parameters to the callback function in the order in which they appear in the dependencies array. The callback function is called asynchronously after resolving the module dependencies.

The example uses *sap.ui.require()* to load the previously defined module and to work with it.

```
<script id="sap-ui-bootstrap"
    src="../../resources/sap-ui-core.js"
    data-sap-ui-libs="sap.m"
    data-sap-ui-theme="sap_belize"
    data-sap-ui-resourceroots='{"student00.sap.training.moduledemo": "./"}'
    data-sap-ui-compatVersion="edge"
    data-sap-ui-async="true">
</script>

<script>
    sap.ui.getCore().attachInit(function() {
        var oButton = new sap.m.Button({text: "Display Date" });

        oButton.attachPress(function() {
            sap.ui.require(["student00/sap/training/moduledemo/myLib/DateUtil"],
                           function(DateUtil) {
                                DateUtil.displayDate(new Date());
                           }
                           );
        });

        oButton.placeAt("content");
    });
</script>
```

redirects the module name prefix *student00.sap.training.moduledemo* to the local web app

loads ***./myLib/DateUtil.js*** asynchronously and calls the callback function as soon as the module is available

Figure 46: Asynchronous Loading of Multiple Modules

LESSON SUMMARY

You should now be able to:

- Modularize SAPUI5 applications

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which of the following features can be used to describe a module?

   Choose the correct answers.

   - [ ] A Name
   - [ ] B Association
   - [ ] C Definition
   - [ ] D Scope
   - [ ] E Dependencies

2. Which JavaScript function is called to define a module?

   Choose the correct answer.

   - [ ] A `jQuery.sap.require`
   - [ ] B `jQuery.sap.define`
   - [ ] C `sap.ui.define`
   - [ ] D `sap.ui.declare`

3. Which JavaScript function is used to load a module asynchronously?

   Choose the correct answer.

   - [ ] A `jQuery.sap.require`
   - [ ] B `sap.ui.require`
   - [ ] C `sap.ui.define`
   - [ ] D `sap.ui.loadModule`

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which of the following features can be used to describe a module?

   Choose the correct answers.

   - [X] A Name
   - [ ] B Association
   - [X] C Definition
   - [ ] D Scope
   - [X] E Dependencies

   Correct. When describing a module, it is common practice to provide the name, definition, and the dependencies of a module.

2. Which JavaScript function is called to define a module?

   Choose the correct answer.

   - [ ] A `jQuery.sap.require`
   - [ ] B `jQuery.sap.define`
   - [X] C `sap.ui.define`
   - [ ] D `sap.ui.declare`

   Correct. The `sap.ui.define` function defines a JavaScript module with its name, its dependencies, and a module value or factory. The typical, suggested usage of this method is to have one single, top-level call to `sap.ui.define` in one JavaScript resource (file).

© Copyright. All rights reserved.

3. Which JavaScript function is used to load a module asynchronously?

   Choose the correct answer.

   - [ ] A `jQuery.sap.require`
   - [x] B `sap.ui.require`
   - [ ] C `sap.ui.define`
   - [ ] D `sap.ui.loadModule`

   Correct. The `sap.ui.require` function resolves one or more module dependencies.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 6 Views and Controllers

Lesson 1

Using Views and Controllers 75

UNIT OBJECTIVES

- Use views and controllers

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Using Views and Controllers

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use views and controllers

Views

The figure, Views, shows the functions of a view and describes the available view types.

- In the Model View Controller concept, the view is responsible for defining and rendering the UI.
- The following predefined view types are available:
  - **XML view**
    The user interface is defined in an XML file or string.
  - **JS view**
    The user interface is constructed in a traditional manner.
  - **HTML view**
    The user interface is defined in an HTML file or string.
  - **JSON view**
    The user interface is defined in a file or string in JSON format.

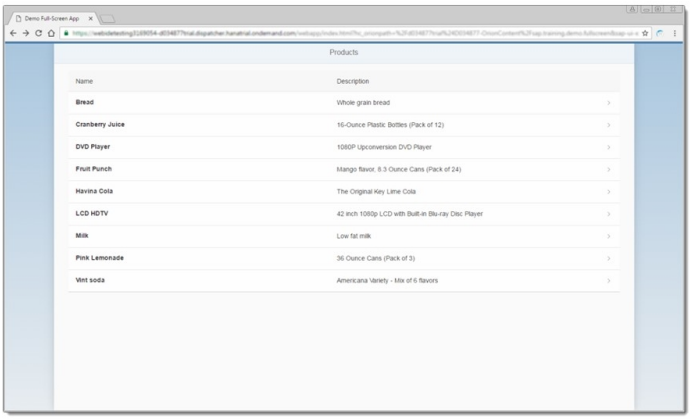


Figure 47: Views

View Instantiation

- To instantiate views asynchronously, SAPUI5 provides a specific factory method for each view type:
  - ***sap.ui.core.mvc.XMLView.create****(mOptions): Promise*
  - ***sap.ui.core.mvc.JSView.create****(mOptions): Promise*
  - ***sap.ui.core.mvc.JSONView.create****(mOptions): Promise*
  - ***sap.ui.core.mvc.HTMLView.create****(mOptions): Promise*
- In addition, SAPUI5 provides a generic factory function to instantiate views asynchronously:
  - ***sap.ui.core.mvc.View.create****(mOptions): Promise*
- With asynchronous loading of views, the view instance is not available at the moment of creation, instead you receive a ***Promise*** which resolves with the created view instance.

Figure 48: Available Methods for View Instantiation

© Copyright. All rights reserved.

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 receive a `Promise`.

`Promises` are useful for async success/failure, because you are less interested in the exact time something became available, and more interested in reacting to the outcome.

When a `Promise` is pending, it can transition to the **`fulfilled`** or **`rejected`** state. Once a `Promise` is **`fulfilled`** or **`rejected`**, however, it will never transition to any other state, and its value or failure reason will not change.

- A ***Promise*** represents the eventual result of an asynchronous operation. It is a placeholder into which the successful result value or reason for failure will materialize.
- A ***Promise*** can be in one of 3 states:
  - ***Pending*** – the *Promise's* outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet.
  - ***Fulfilled*** – the asynchronous operation has completed, and the *Promise* has a value.
  - ***Rejected*** – the asynchronous operation failed, and the *Promise* will never be *fulfilled*. In the *rejected* state, a *Promise* has a reason that indicates why the operation failed.

Figure 49: Elaboration: JavaScript Promises (1)

If a `Promise` has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier.

The code snippet in the figure, Elaboration: JavaScript Promises (2), shows how the view instance is available in the resolve function of the `Promise`.

Using Promises

- The primary way of interacting with a *Promise* is through its *then()* method, which registers callbacks to receive either a *Promise's* eventual value or the reason why the *Promise* cannot be fulfilled.
- *then()* takes two arguments, a callback for a success case, and another for the failure case. Both are optional, so you can add a callback for the success or failure case only.

```
View.create(
}).then(function (oView) {
    oView.placeAt("content");
});
```

value of the *Promise* in the *fulfilled* state

Figure 50: Elaboration: JavaScript Promises (2)

© Copyright. All rights reserved.

To pass the required information for view instantiation, a configuration object is used. The configuration object can have the following properties for the view instantiation:

- `id`:

  Specifies an ID for the view instance. If no ID is given, an ID will be generated.

- `viewName`:

  Name of the view resource in module name notation (without suffix).

- `type`:

  Specifies what kind of view will be instantiated. The type can be JSON, JS, XML, or HTML. All valid view types are listed in the enumeration `sap.ui.core.mvc.ViewType`.

Controllers and views are defined and instantiated with a name that equals an SAPUI5 module name within the define/require concept. When you instantiate a view or controller, the SAPUI5 runtime loads them relative to the `resources` folder of the machine on which the SAPUI5 runtime was loaded. To inform SAPUI5 runtime that your views and controllers are located on your local machine, you can use the configuration option `resourceRoots`.

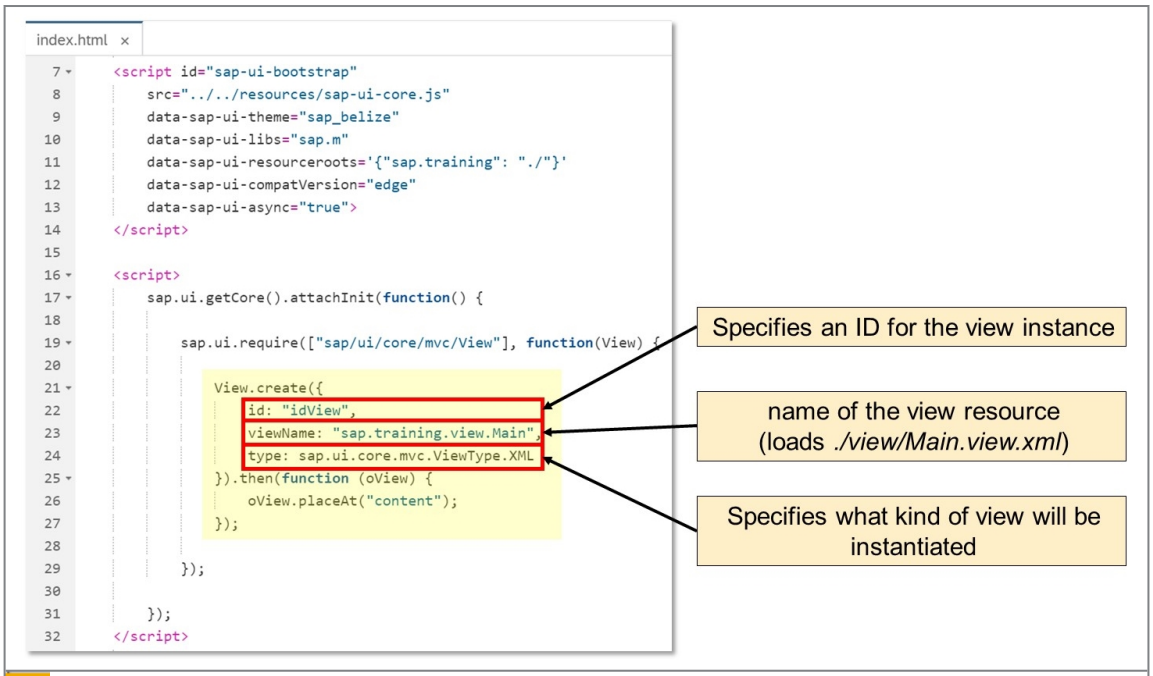


Figure 51: Instantiating Views: Example

Controllers

The methods used for controlling data flow are implemented in the controller. The suffix `.controller.js` is mandatory for controllers.

A controller can be defined by extending the `Controller` object of the SAPUI5 core. Once the `Controller` module is loaded, the callback function is called and you can make use of the `Controller` object by accessing the parameter passed to the function. This Asynchronous Module Definition (AMD) syntax enables the clear separation of module loading from code execution and improves the performance of the application. The browser determines when and how the resources are loaded prior to code execution.

The `use strict;` literal expression was introduced by JavaScript 1.8.5 (ECMAScript 5). It tells the browser to execute the code in a so-called “strict mode”. The strict mode helps the

© Copyright. All rights reserved.

detection of potential coding issues at an early stage of the development cycle. That means, for example, that there is an assurance that variables are declared before they are used. Strict mode helps to prevent common JavaScript pitfalls, and consequently it is a good practice to use it.

SAPUI5 provides the following lifecycle hooks:

- `onInit()`:

  This is called when a controller is instantiated and its view controls (if available) are already created. It can be used to modify the view before it is displayed, to bind event handlers, and do other one-time initialization tasks.

- `onExit()`:

  This is called when the controller is destroyed. Use this method to free resources and finalize activities.

- `onAfterRendering()`:

  This is called when the view has been rendered (so its HTML is part of the document). Post-rendering manipulations of the HTML can be done here. This hook is the same one that SAPUI5 controls get when they are rendered.

- `onBeforeRendering()`:

  This is 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 purpose).

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.

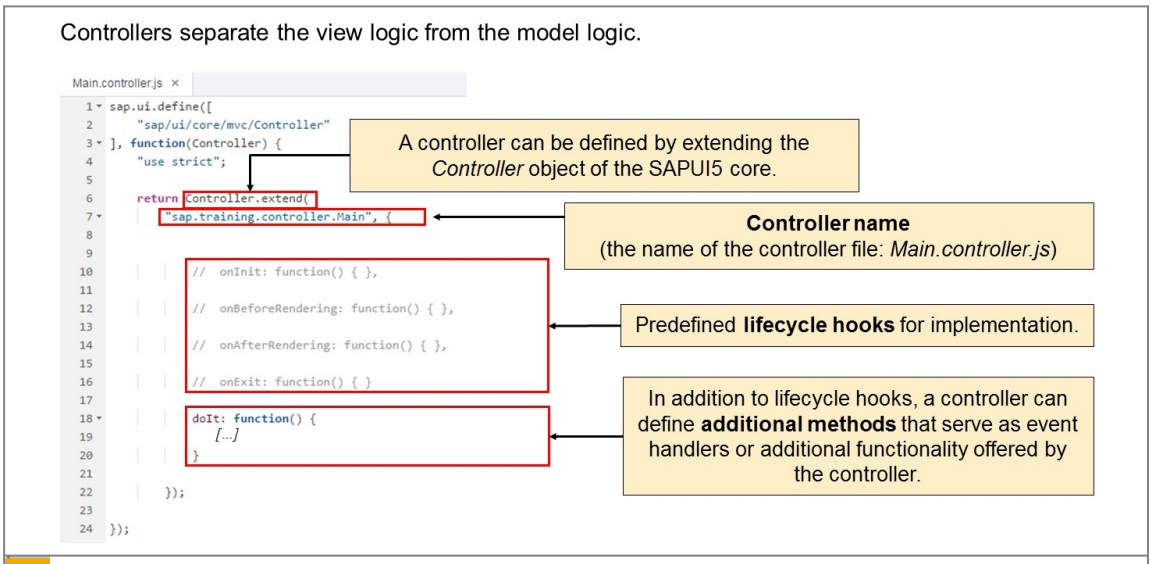


Figure 52: Controllers

JavaScript Views

A JavaScript view name equals the SAPUI5 module name within the require/declare concept.

For a JavaScript view file, the suffix `.view.js` is used.

SAPUI5 provides two default methods for the implementation of a JavaScript view, `getControllerName` and `createContent`, as shown in the figure, JavaScript Views.

© Copyright. All rights reserved.

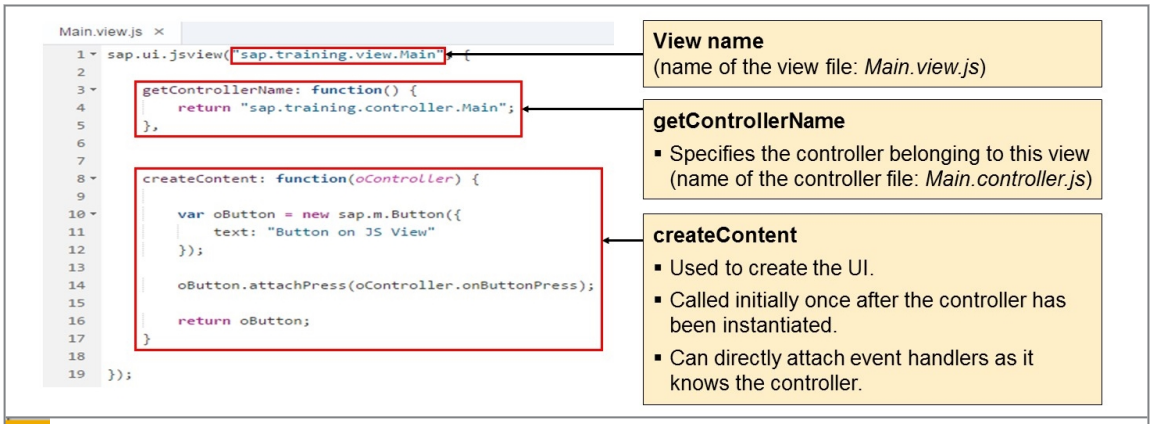


Figure 53: JavaScript Views

JavaScript Views – Support for Unique IDs

To guarantee the uniqueness of control IDs defined inside a JavaScript view when reusing views, you cannot use hardcoded IDs. You must give the view the opportunity to add its own instance ID as a prefix. This is done by using the `View.createId()` method. This is not required for declarative view types, as the view parser can manage this automatically.

If you want to modify the control with the view local `<ID>`, you can call the `byId(<ID>)` method on your view to get the correct control directly. You do not need to handle the prefix assignment on your own.

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 is usually expected to be denoted by `this`. The view developer can modify the `this` context as shown in the figure, JavaScript Views: Support for Unique IDs.

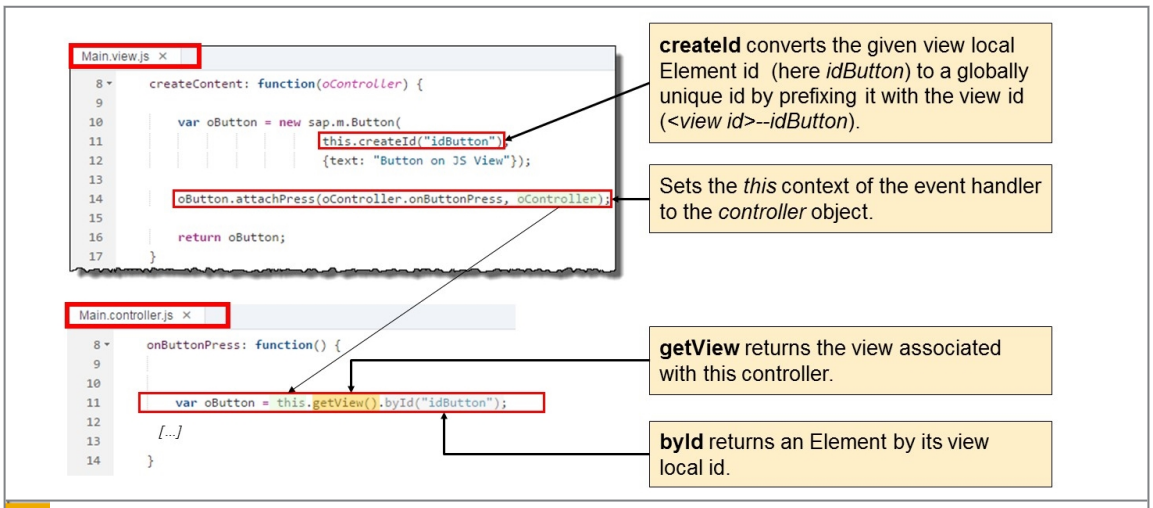


Figure 54: JavaScript Views: Support for Unique IDs

XML Views

The XML view type is defined in an XML file or as an XML string.

The name of the view is the SAPUI5 module name within the require/declare concept.

In the example shown in the figure, XML Views, the core namespace is defined with the prefix `core`. You can define any prefix you want. To keep the tag names shorter, for example, you can also use `c` instead of `core`.

© Copyright. All rights reserved.

One of the listed namespaces can be defined as the default namespace. The control tags for this namespace do not need a prefix.

If controls are located in a sub-package of a control library, for example `sap.ui.layout.Grid`, they must have their own XML namespace.

In XML views, child controls can be simply added as child tags.

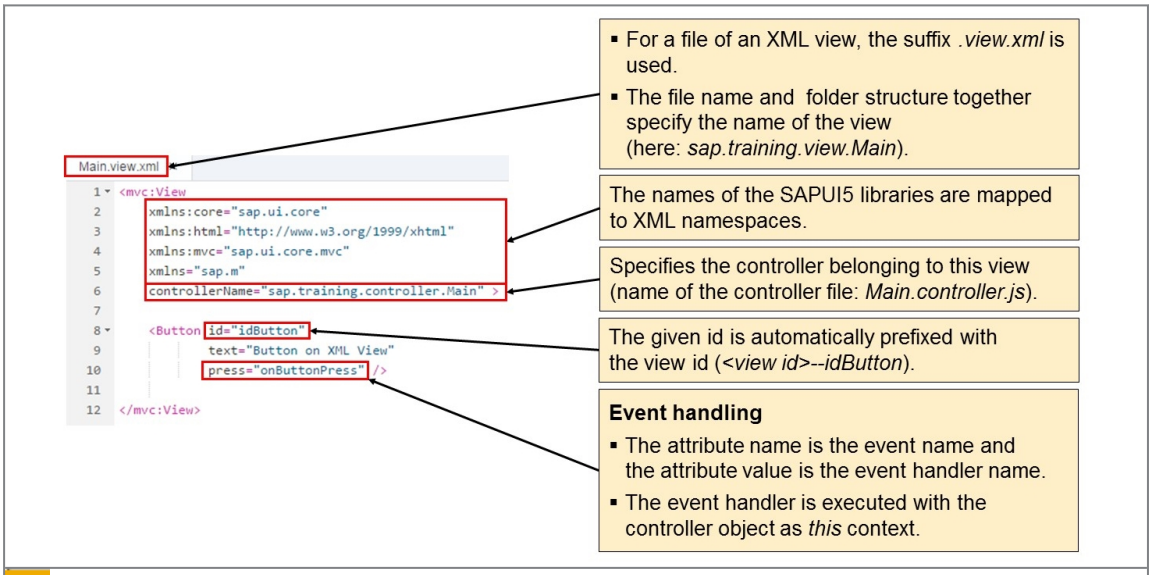


Figure 55: XML Views

HTML Views

An HTML view is defined by declarative HTML.

All view-specific properties can be added to the `<template>` tag as `data-*` attributes.

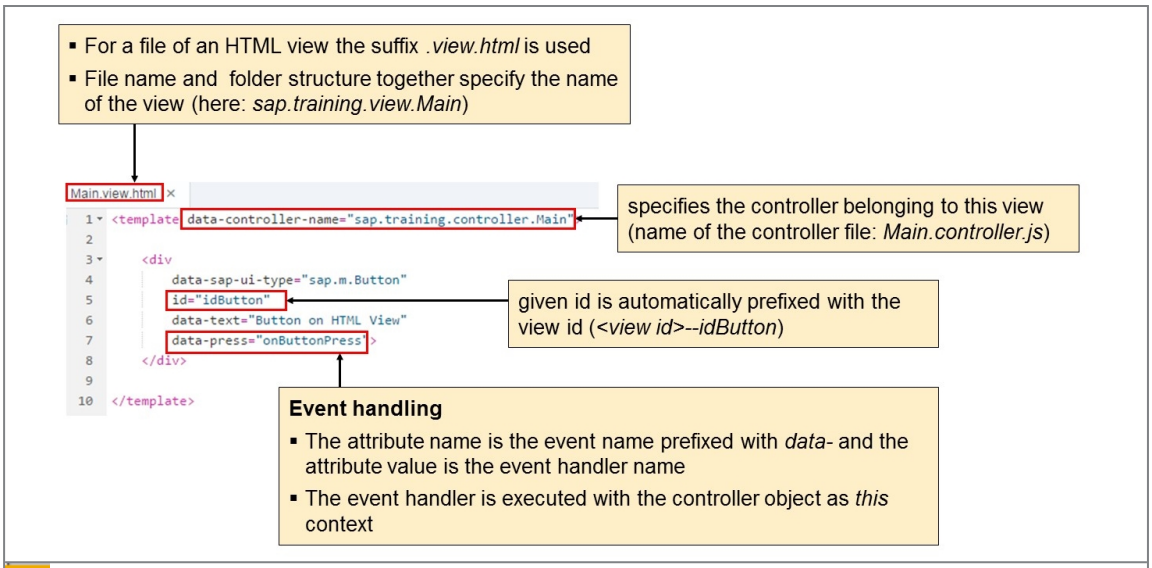


Figure 56: HTML Views

JSON Views

The JSON view type is defined in a file or as a JSON string.

The name of the view is the SAPUI5 module name within the modularization concept.

© Copyright. All rights reserved.

You add child controls as arrays. This is shown in the figure, JSON Views, where a button has been added to the view's `content` aggregation.

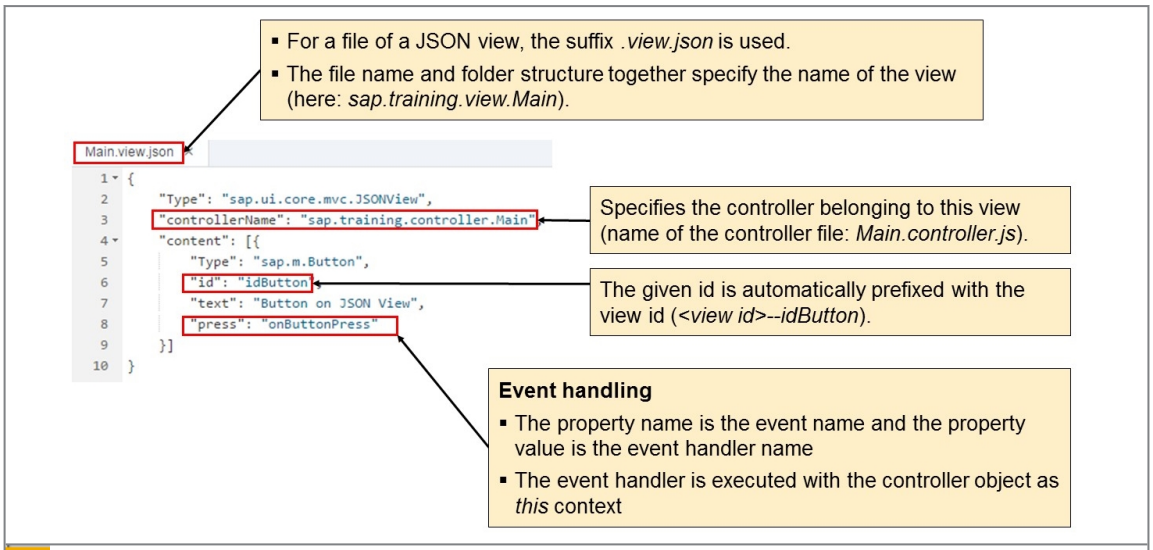


Figure 57: JSON Views

View Embedding in an XML View

The figure, View Embedding in an XML View, shows how the different view types can be embedded in an XML view.

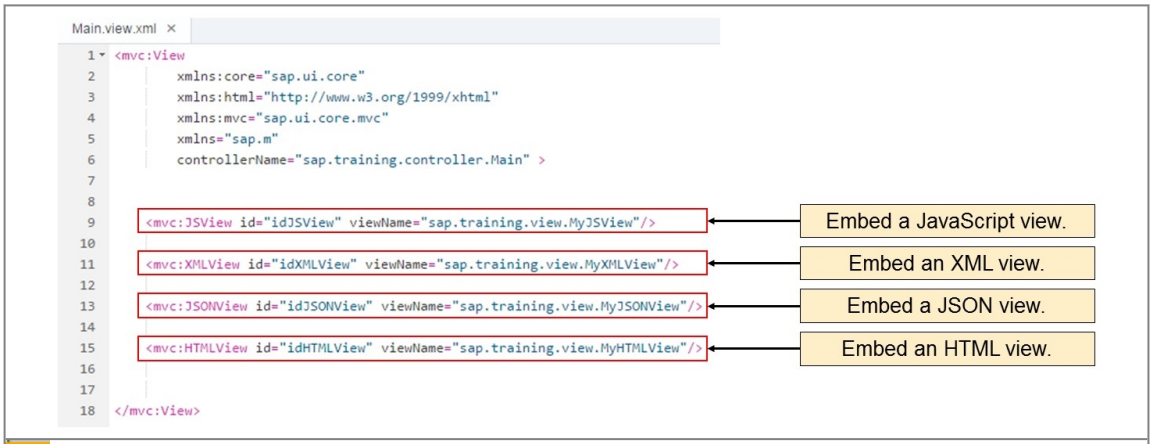


Figure 58: View Embedding in an XML View

LESSON SUMMARY

You should now be able to:

- Use views and controllers

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. What view type does SAP recommend?

   Choose the correct answer.

   - ☐ A JSON
   - ☐ B XML
   - ☐ C HTML
   - ☐ D JavaScript

2. Which function is provided to register a URL prefix for a module name prefix?

   Choose the correct answer.

   - ☐ A `jQuery.sap.mapModulePath()`
   - ☐ B `jQuery.sap.registerModulePath()`
   - ☐ C `sap.ui.registerModulePath()`
   - ☐ D `jQuery.sap.declareModulePath()`

3. Which class is normally used as the base class when implementing a SAPUI5 view controller?

   Choose the correct answer.

   - ☐ A `sap.ui.core.mvc.Controller`
   - ☐ B `sap.ui.view.Controller`
   - ☐ C `sap.m.Controller`
   - ☐ D `sap.ui.layout.Controller`

© Copyright. All rights reserved.

4. Which of the following lifecycle hook methods can be implemented by the developer in a view controller?

   Choose the correct answers.

   - [ ] A `onInit`
   - [ ] B `onRender`
   - [ ] C `onAfterRendering`
   - [ ] D `onExit`
   - [ ] E `onBeforeRndering`

© Copyright. All rights reserved.

Learning Assessment - Answers

1. What view type does SAP recommend?

   Choose the correct answer.

   - [ ] A JSON
   - [x] B XML
   - [ ] C HTML
   - [ ] D JavaScript

   Correct. SAP recommends the use of XML views. Using XML, you have a clearly-structured implementation.

2. Which function is provided to register a URL prefix for a module name prefix?

   Choose the correct answer.

   - [ ] A `jQuery.sap.mapModulePath()`
   - [x] B `jQuery.sap.registerModulePath()`
   - [ ] C `sap.ui.registerModulePath()`
   - [ ] D `jQuery.sap.declareModulePath()`

   Correct. The `jQuery.sap.registerModulePath()` function is provided by the framework for registration of an URL prefix for a module name prefix.

3. Which class is normally used as the base class when implementing a SAPUI5 view controller?

   Choose the correct answer.

   - [x] A `sap.ui.core.mvc.Controller`
   - [ ] B `sap.ui.view.Controller`
   - [ ] C `sap.m.Controller`
   - [ ] D `sap.ui.layout.Controller`

   Correct. The `sap.ui.core.mvc.Controller` class is normally used as a base class for custom view controller implementation.

© Copyright. All rights reserved.

4. Which of the following lifecycle hook methods can be implemented by the developer in a view controller?

   Choose the correct answers.

   [X] A `onInit`

   [ ] B `onRender`

   [X] C `onAfterRendering`

   [X] D `onExit`

   [X] E `onBeforeRndering`

   Correct. The developer can implement the `onInit` method to handle aspects that should be executed at controller initialization. The `onExit` method can be used to implement runtime aspects that should be executed when the view and the corresponding controller are discarded by the SAPUI5 runtime. The `onAfterRendering` and `onBeforeRendering` functions are implemented in the controller to influence the rendered HTML content.

© Copyright. All rights reserved.

UNIT 7 Layout Controls

Lesson 1

Using Layout Controls 89

UNIT OBJECTIVES

- Use layout controls

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Using Layout Controls

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use layout controls

Layout Controls Overview

The figure, Overview of Layout Controls, shows where to obtain samples of the available layout controls.

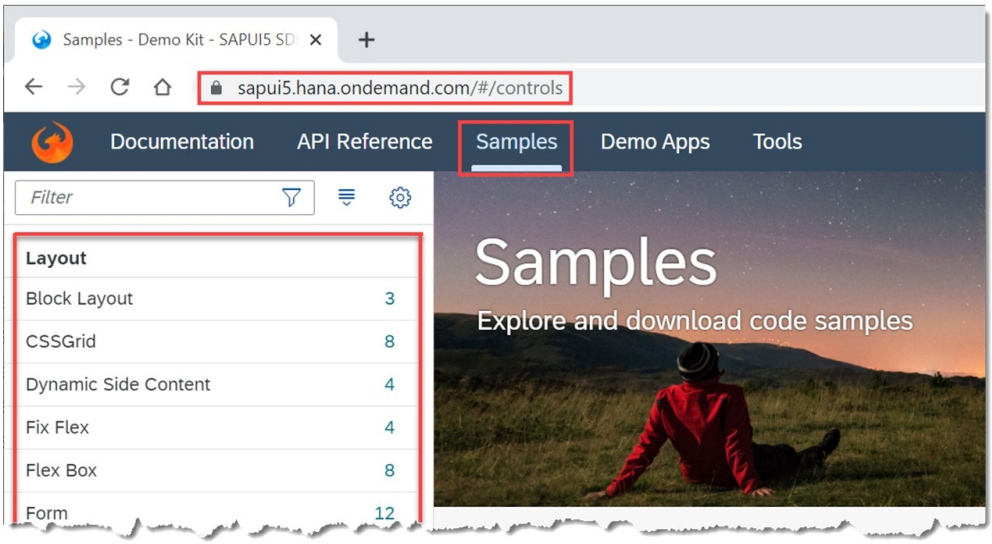


Figure 59: Overview of Layout Controls

sap.ui.layout.VerticalLayout Control

The figure, sap.ui.layout.VerticalLayout Control, provides general information about the `sap.ui.layout.VerticalLayout` control.

© Copyright. All rights reserved.

Figure 60: sap.ui.layout.VerticalLayout Control

sap.ui.layout.HorizontalLayout Control

To keep the *HorizontalLayout* control slim and light-weight, it cannot offer sophisticated layout options.

Spacing between the items is currently not adjustable in the *HorizontalLayout*, but can be added using CSS.

A specific use-case would be to use the *HorizontalLayout* control inside a *VerticalLayout* control to put several controls on one row.

Figure 61: sap.ui.layout.HorizontalLayout Control

sap.ui.layout.Grid Control

In the figure, sap.ui.layout.Grid Control (1), the form is laid out in a flexible fashion on large, medium, and small screen resolutions. The form layout adapts according to the available screen width. On small resolution screens, the form elements stack on top of each other.

The following typical screen widths are available by default:

- `Phone` (“small”):

  For screens smaller than 600 pixels

© Copyright. All rights reserved.

- `Tablet` (“medium”):

  For screens greater than or equal to 600 pixels and smaller than 1024 pixels

- `Desktop` (“large”):

  For screens greater than or equal to 1024 pixels and smaller than 1440 pixels

- `LargeDesktop` (“extra large”):

  For screens greater than or equal to 1440 pixels

For more information, see `sap.ui.Device.media.RANGESETS` in the API Reference documentation in the Demo Kit.

Using the **Grid** control, it is possible to achieve flexible layouts and line-breaks for extra large-, large-, medium- and small-sized screens, such as large desktop, desktop, tablet, and mobile.

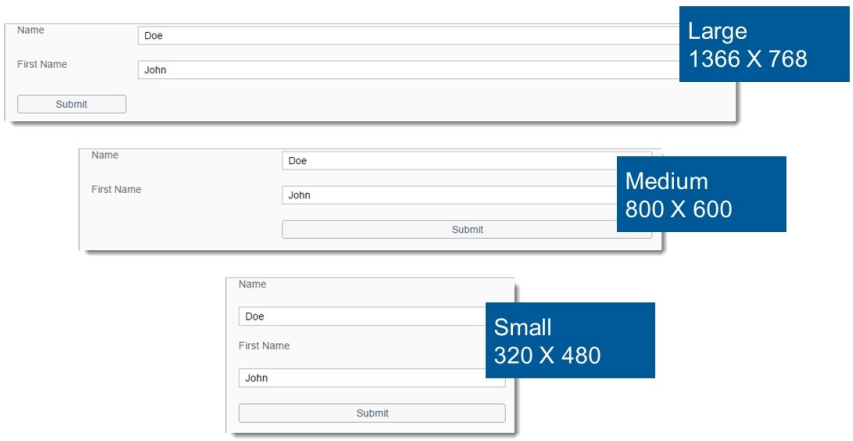


Figure 62: sap.ui.layout.Grid Control (1)

You can use the Grid control to make responsive table-free layouts.

The Grid control’s children can be specified to take on a variable amount of columns, depending on the available screen size (XL=extra large, L=large, M=medium, or S=small).

Note:
The values must be provided in the following order:

1. Extra large
2. Large
3. Medium
4. Small

© Copyright. All rights reserved.

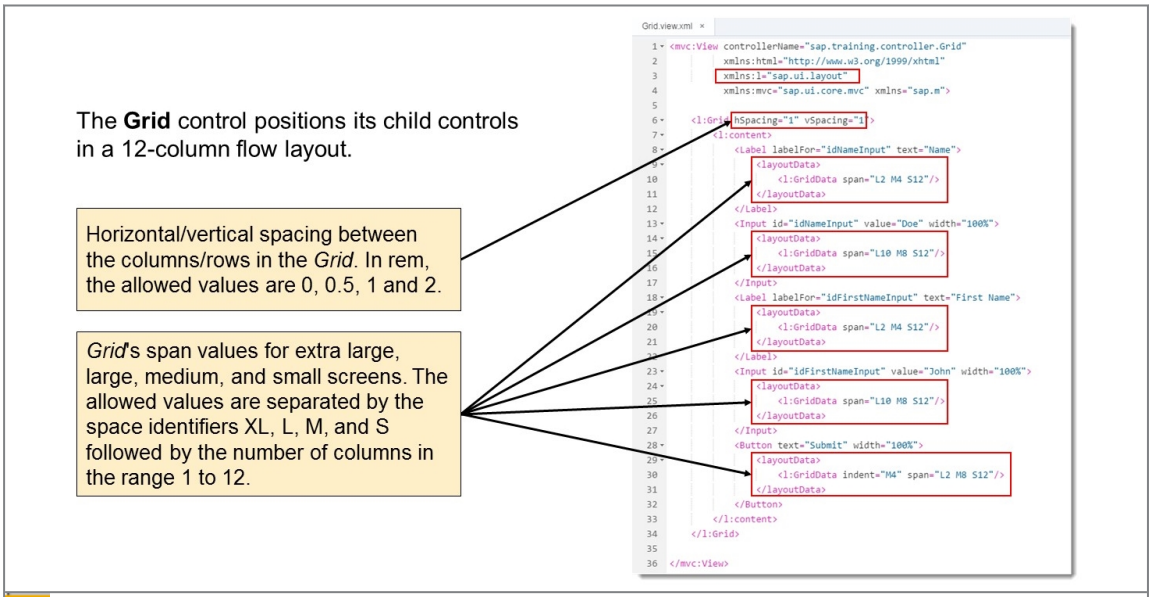


Figure 63: sap.ui.layout.Grid Control (2)

sap.ui.layout.ResponsiveFlowLayout Control

In HTML, the default behavior is that all items are positioned next to each other. If the space within the parent container becomes too small, the browser wraps the corresponding item to the next line. In this case, the extra space within a line is not used.

By contrast, the ResponsiveFlowLayout expands the added content to fit a whole line. If the window resizes, the controls are moved between the lines and resized again.

The figure, sap.ui.layout.ResponsiveFlowLayout Control (1), shows a small responsive form that is implemented by means of the *ResponsiveFlowLayout*. There are two *VerticalLayout* controls added to a *ResponsiveFlowLayout*. Each *VerticalLayout* contains a *Label* element and an *Input* UI element. If the space becomes too small, the second *VerticalLayout* is wrapped under the first one.

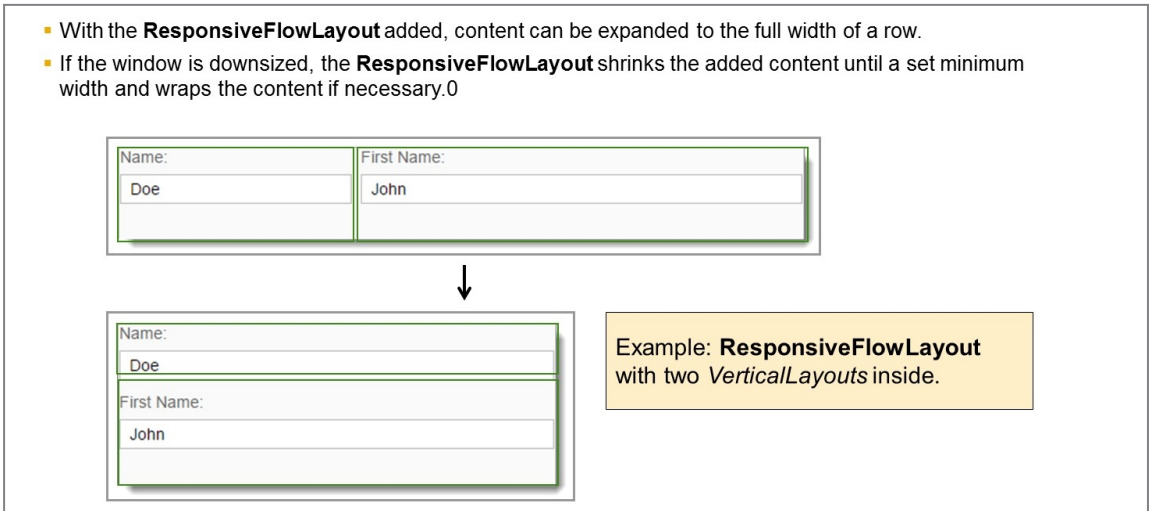


Figure 64: sap.ui.layout.ResponsiveFlowLayout Control (1)

The figure, sap.ui.layout.ResponsiveFlowLayout (2), shows an example of a ResponsiveFlowLayout view.

© Copyright. All rights reserved.

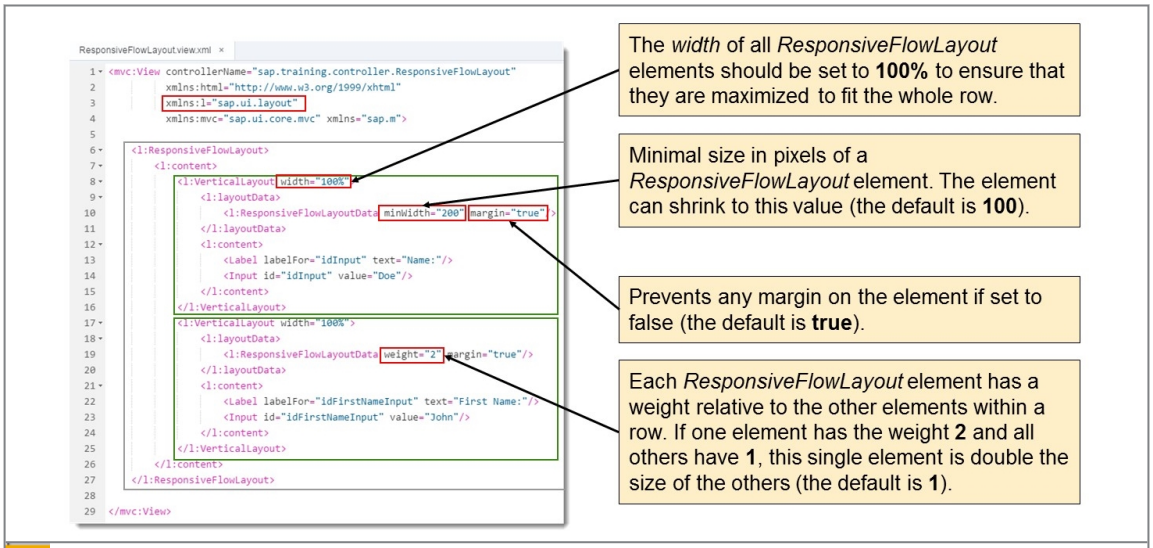


Figure 65: sap.ui.layout.ResponsiveFlowLayout Control (2)

sap.ui.layout.form.Form Control

A *FormContainer* can be seen as a panel. It can have a title and an expander.

The assignment of the label to the fields of a *FormElement* is done automatically.

Even though the *Form* is not part of `sap.m`, it can and should be used for building mobile user interfaces.

The *Form* supports a Navigation Mode and an Edit Mode for keyboard navigation. When you enter a *Form*, you are in Navigation Mode. If you start typing in an *Input* UI element for example, the mode switches to Edit Mode, which means that the arrow keys are navigating within the text. At the end of the text, pressing an arrow key results in the next field being selected, and a switch to Navigation Mode again. To explicitly switch between Edit Mode and Navigation Mode from within a field, you can press F2.

The Navigation Mode of the *Form* supports the following keyboard shortcuts:

- Arrows:

  You can navigate through *Form* fields using the arrow keys. Even read-only fields can be focused when using the arrow keys. At the end of a *Form*, you stay within the last field when using the arrow keys.

- Tab/Shift + Tab:

  You can navigate forward (Tab) and backwards (Shift + Tab) within a *Form*. Tab navigation ignores read-only fields. If a Container is expandable, you can select the expander icon within the *Container* header. The Container can then be expanded or collapsed using the Space key or the Enter key.

- Home:

  The first element of the Container is selected.

- End:

  The last element of the Container is selected.

- Ctrl + Home:

  The first element of the *Form* is selected.

© Copyright. All rights reserved.

- Ctrl + End:

  The last element of the *Form* is selected.

- F6:

  The first element of the next Container is selected. If you are within the last Container of a *Form*, the first element of the first Container is selected.

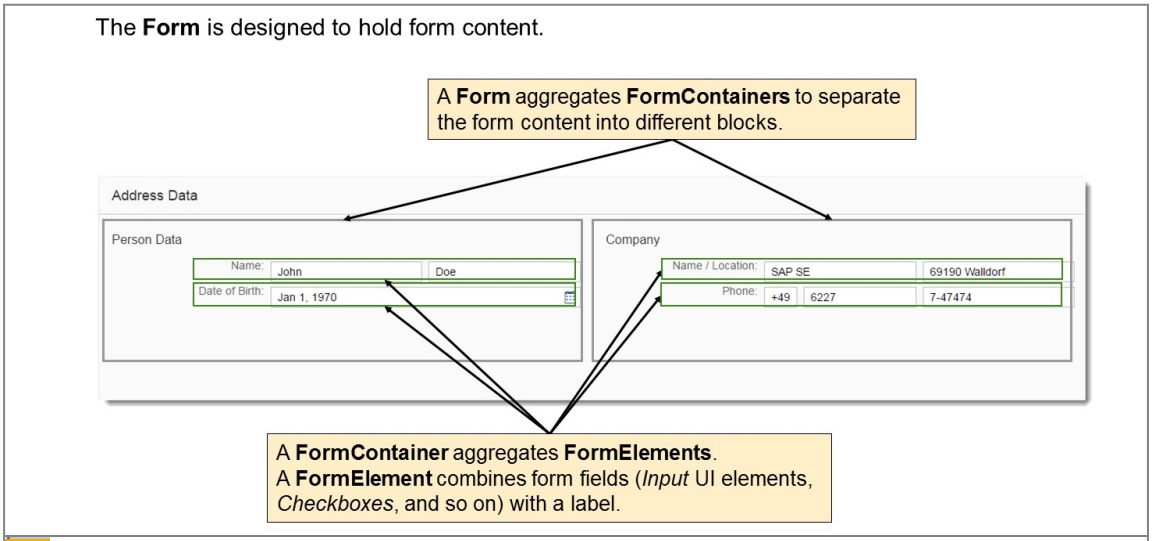


Figure 66: sap.ui.layout.form.Form Control (1)

The `layout` aggregation of the *Form* enables the *Form* to be shown in different ways, by just changing the *FormLayout*.

You can define several layouts and assign them to the *Form* on the fly. This way, it is possible to switch between layouts as a reaction to an event (for example, `resize`) or toggle between different layouts as per user request. To enable a simple switching of layouts, a *Form* supports *VariantLayoutData* for its content.

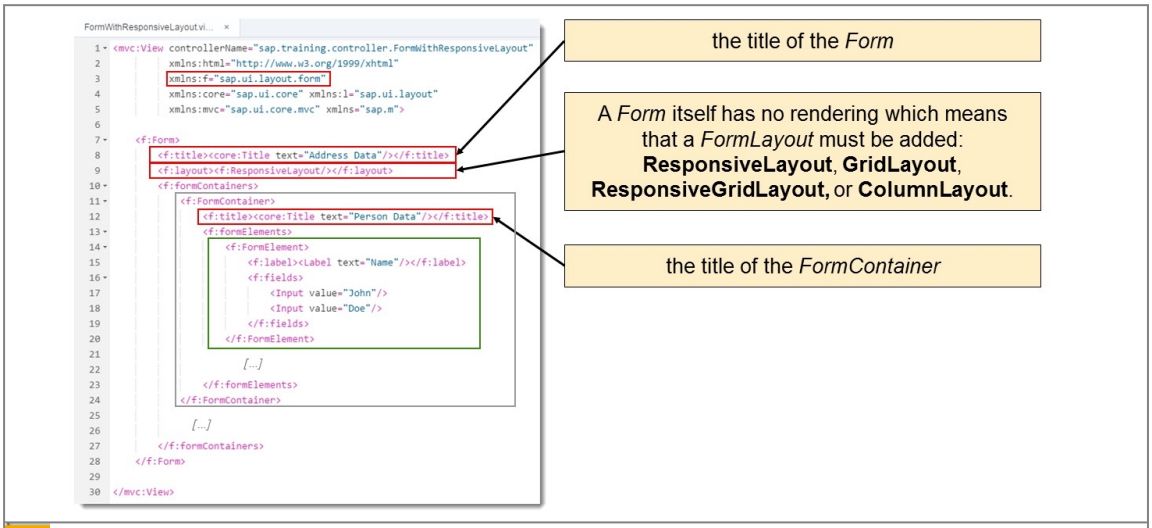


Figure 67: sap.ui.layout.form.Form Control (2)

The *ResponsiveLayout* can be attached to the *Form* control to arrange the elements in a responsive way (see the ResponsiveFlowLayout topic). The size and arrangement of the individual elements depends on the size of the screen.

© Copyright. All rights reserved.

According to the layout concept, *FormContainers*, *FormElements*, and the *field* controls can have *layoutData* associated with them. You can use *ResponsiveFlowLayoutData* to influence the linebreaks, minimum width, and weights of the individual elements.

If **ResponsiveLayout** is assigned to the *Form,* you can attach **ResponsiveFlowLayoutData** to *FormContainers*, *FormElements* and the field controls as in the *ResponsiveFlowLayout*.

```
<f:FormContainer>
    <f:title><core:Title text="Person Data"/></f:title>
    <f:formElements>
        <f:FormElement>
            <f:label><Label text="Name"/></f:label>
            <f:fields>
                <Input value="John"/>
                <Input value="Doe"/>
            </f:fields>
        </f:FormElement>
        <f:FormElement>
            <f:layoutData>
                <l:ResponsiveFlowLayoutData linebreak="true"/>
            </f:layoutData>
            <f:label><Label text="Date of Birth"/></f:label>
            <f:fields>
                <DatePicker value="1970-01-01">
                    <layoutData>
                        <l:ResponsiveFlowLayoutData weight="2"/>
                    </layoutData>
                </DatePicker>
            </f:fields>
        </f:FormElement>
    </f:formElements>
</f:FormContainer>
```

Person Data
Name: John Doe
Date of Birth: Jan 1, 1970

The *FormElement* is placed at the beginning of line.

The *DatePicker* is double the size of the other fields.

Figure 68: sap.ui.layout.form.Form Control (3)

The *GridLayout* supports the basic keyboard navigation of a *Form*. In addition, when using the arrow keys you can move up (Arrow Up) or down (Arrow Down) within the *Form* fields.

```
<f:Form>
    <f:title>
        <core:Title text="Address Data"/>
    </f:title>
    <f:layout>
        <f:GridLayout singleColumn="false"/>
    </f:layout>
    <f:formContainers>


    </f:formContainers>
</f:Form>
```

- The **GridLayout** can be attached to the *Form* control to arrange the elements in a cell-based layout.
- The **GridLayout** is by default a 16-cell grid (*singleColumn*="false").
- You can set the *singleColumn* property to *true* to get an 8-cell grid.

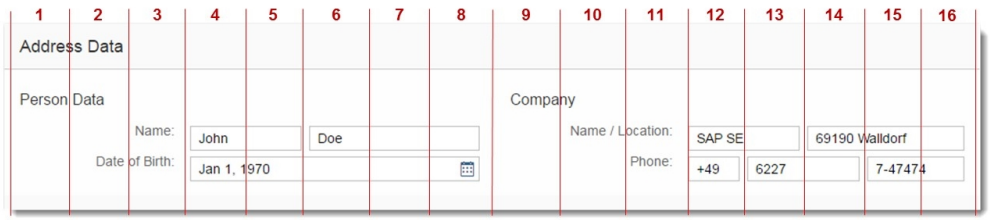


Figure 69: sap.ui.layout.form.Form Control (4)

To adjust the content inside the *GridLayout*, *GridContainerData* and *GridElementData* are used.

In a 16-cell grid, you can place two *FormContainers* in one row by setting *halfGrid* to `true` in the respective *GridContainerData* attached.

In an 8-cell grid, you cannot include *halfGrid* containers.

© Copyright. All rights reserved.

- If *halfgrid* is set for a *FormContainer* in the **GridLayout**, the container takes half the width of the *Form* (8 cells). If not, it takes the full width (16 cells).
- If the **GridLayout** is set to be *singleColumn*, the full width of the grid is only 8 cells. So containers are rendered only one per row.

Figure 70: sap.ui.layout.form.Form Control (5)

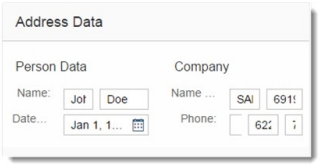
You can attach *GridElementData* to the labels and fields inside a *GridLayout*. Using the properties *hCells* and *vCells* of *GridElementData*, you can access different arrangements within the grid.

You can also add dummy *FormElements* to get spare rows or dummy *Text* UI elements to get spare cells.

If *hCells* is set to `auto`, the size is determined by the number of fields and the available cells. For *Labels*, the automatic size is 3 cells.

If *hCells* is set to `full`, only one field is allowed within the *FormElement*. It gets the full width of the row and the *Label* is displayed above. For *Labels*, `full` has no effect.

- To adjust the width of the labels and fields inside the **GridLayout**, specify the number of cells in the horizontal direction using the *hCells* property of *GridElementData*.
- The **GridLayout** is not responsive:



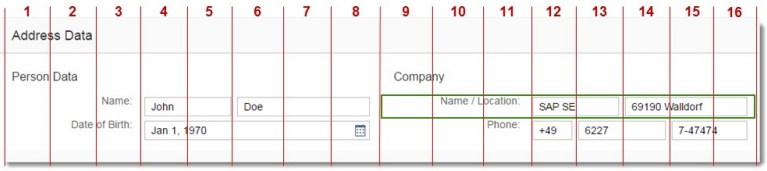


Figure 71: sap.ui.layout.form.Form Control (6)

The *ResponsiveGridLayout* can be attached to the *Form* control to arrange the elements in a responsive way (see the *Grid* topic). The size and arrangement of the individual elements depends on the size of the screen.

To customize the *ResponsiveGridLayout*, you can use the following properties:

- *breakpointXL:*

© Copyright. All rights reserved.

The breakpoint (in pixels) between the large size and the extra large (XL) size. The default value is 1440.

- *breakpointL:*

  The breakpoint between the medium size and the large size. The default value is 1024.

- *breakpointM:*

  The breakpoint between the small size and the medium size. The default value is 600.

- *columnsXL:*

  The number of *FormContainers* in one line for the extra large size. This number must not be smaller than the number of columns for the large size. The default value is -1.

> Note:
> If the default value -1 is not overwritten, the *columnsL* property value is used (for backward compatibility reasons).

- *columnsL:*

  The number of *FormContainers* in one line for the large size. This number must not be smaller than the number of columns for the medium size. The default value is 2.

- *columnsM:*

  The number of *FormContainers* in one line for the medium size. The default value is 1.

- *emptySpanXL:*

  The number of grid cells that are empty at the end of each line for the extra large size. The default value is -1.

> Note:
> If the default value -1 is not overwritten, the *emptySpanL* property value is used.

- *emptySpanL:*

  The number of grid cells that are empty at the end of each line for the large size. The default value is 0.

- *emptySpanM:*

  The number of grid cells that are empty at the end of each line for the medium size. The default value is 0.

- *emptySpanS:*

  The number of grid cells that are empty at the end of each line for the small size. The default value is 0.

- *labelSpanXL:*

  The default span for labels in the extra large size. The default value is -1.

© Copyright. All rights reserved.

> Note:
> If the default value -1 is not overwritten, the *labelSpanL* property value is used.

- *labelSpanL:*

  The default span for labels in the large size. The default value is 4.

> Note:
> If *adjustLabelSpan* is set, this property is only used if more than one *FormContainer* is in one line. If only one *FormContainer* is in the line, the *labelSpanM* property value is used.

- *labelSpanM:* The default span for labels in the medium size. The default value is 2.

> Note:
> If *adjustLabelSpan* is set, this property is used for full-size *FormContainers*. If more than one *FormContainer* is in one line, the *labelSpanL* property value is used.

- *labelSpanS:*

  The default span for labels in the small size. The default value is 12.

- *adjustLabelSpan:*

  If set, the usage of *labelSpanL* and *labelSpanM* are dependent on the number of *FormContainers* in one row. If only one *FormContainer* is displayed in one row, *labelSpanM* is used to define the size of the label. This is the same for medium and large *Form* controls. This is done to align the labels on forms where full-size *FormContainer* elements and multiple-column rows are used in the same *Form* (because every *FormContainer* has its own *Grid* inside). If not set, the usage of *labelSpanL* and *labelSpanM* are dependent on the *Form* size. The number of *FormContainer* elements does not matter in this case. The default value is `true`.

- *singleContainerFullSize:*

  If the *Form* contains only one single *FormContainer* and this property is set, the *FormContainer* is displayed using the full size of the *Form*. In this case, the properties *columnsXL*, *columnsL*, and *columnsM* are ignored. In all other cases, the *FormContainer* is displayed in the size of one column. The default value is `true`.

© Copyright. All rights reserved.

```
<f:Form>
    <f:title>
        <core:Title text="Address Data"/>
    </f:title>
    <f:layout>
        <f:ResponsiveGridLayout
                breakpointXL="1440"
                breakpointL="1024"
                breakpointM="600"
                columnsXL="-1"
                columnsL="2"
                columnsM="1"
                emptySpanXL="-1"
                emptySpanL="0"
                emptySpanM="0"
                emptySpanS="0"
                labelSpanXL="-1"
                labelSpanL="4"
                labelSpanM="2"
                labelSpanS="12"
                adjustLabelSpan="true"
                singleContainerFullSize="true"/>
    </f:layout>
    <f:formContainers>

        [...]

    </f:formContainers>
</f:Form>
```

- The **ResponsiveGridLayout** can be attached to the *Form* control in order to achieve flexible layouts for extra large-, large-, medium- and small-sized screens, such as large desktop, desktop, tablet, and mobile.
- Just like the *Grid control*, the ResponsiveGridLayout positions the form controls in a 12 column layout.

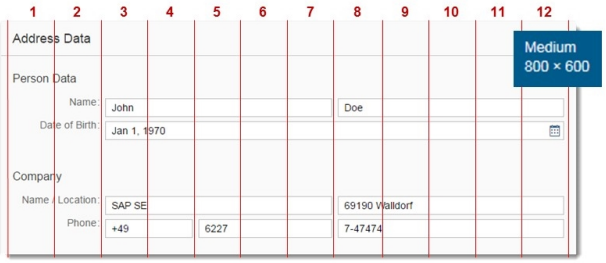


Figure 72: sap.ui.layout.form.Form Control (7)

To influence the rendering of the labels and fields, *GridData* must be set for these controls.

If no *GridData* is used on a *FormContainer* or control, the *ResponsiveGridLayout* tries to calculate the linebreaks and spans. If the provided *GridData* is very complex, this automation may not yield the expected result. In this case, *GridData* should be set on all affected controls.

On *FormElement* elements, *GridData* is not supported.

```
<f:FormElement>
    <f:label>
        <Label text="Phone"/>
    </f:label>
    <f:fields>
        <Input value="+49">
            <layoutData>
                <l:GridData span="L1 M2 S4"/>
            </layoutData>
        </Input>
        <Input value="6227">
            <layoutData>
                <l:GridData span="L3 M3 S8"/>
            </layoutData>
        </Input>
        <Input value="7-47474">
            <layoutData>
                <l:GridData span="L4 M5 S12"/>
            </layoutData>
        </Input>
    </f:fields>
</f:FormElement>
```

To influence the layout of the *Form* when using **ResponsiveGridLayout**, you can attach *GridData* to *FormContainers* and the content controls as in the *Grid control.*

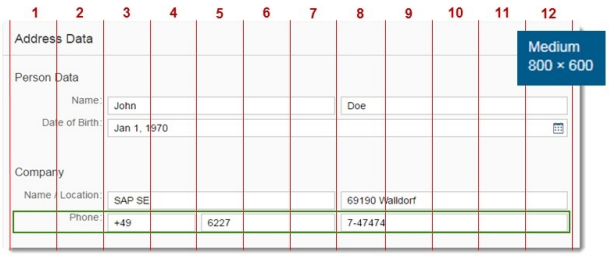


Figure 73: sap.ui.layout.form.Form Control (8)

sap.ui.layout.form.SimpleForm Control

If a more complex form is needed, the application must use the standard *Form* control and fill it according to the requirements. However, for simple forms without any special requirements, the *SimpleForm* control is the easier choice.

To customize the *SimpleForm*, you can use, among other things, the following properties:

© Copyright. All rights reserved.

> Note:
> Some of the properties in the following list are only used if a *ResponsiveGridLayout* is used as the layout. These properties are denoted by an asterisk (*) character next to the property name.

- *layout:*

  The *FormLayout* that is used to render the *SimpleForm*. We suggest using the *ResponsiveGridLayout* for rendering a *SimpleForm*, as its responsiveness uses the space available in the best way possible. The default value is `ResponsiveLayout`.

- *breakpointXL**:

  The breakpoint (in pixels) between the large size and the extra large (XL) size. The default value is 1440.

- *breakpointL**:

  The breakpoint between the medium size and the large size. The default value is 1024.

- *breakpointM**:

  The breakpoint between the small size and the medium size. The default value is 600.

- *columnsXL**:

  Form columns for the extra large size. The number of columns for the extra large size must not be smaller than the number of columns for large size. The default value is -1.

  > Note:
  > If the default value -1 is not overwritten, the *columnsL* value is used (for backward compatibility reasons).

- *columnsL**:

  Form columns for the large size. The number of columns for the large size must not be smaller than the number of columns for the medium size. The default value is 2.

- *columnsM**:

  Form columns for the medium size. The default value is 1.

- *emptySpanXL**:

  The number of grid cells that are empty at the end of each line on the extra large size. The default value is -1.

  > Note:
  > If the default value -1 is not overwritten, the *emptySpanL* value is used (for backward compatibility reasons).

- *emptySpanL**:

  The number of grid cells that are empty at the end of each line on the large size. The default value is 0.

© Copyright. All rights reserved.

- *emptySpanM**:

  The number of grid cells that are empty at the end of each line on the medium size. The default value is 0.

- *emptySpanS**:

  The number of grid cells that are empty at the end of each line on the small size. The default value is 0.

- *labelSpanXL**:

  The default span for labels in the extra large size. The default value is -1.

  > Note:
  > If the default value -1 is not overwritten, the *labelSpanL* value is used (for backward compatibility reasons).

- *labelSpanL**:

  The default span for labels in the large size. The default value is 4.

  > Note:
  > If *adjustLabelSpan* is set, this property is only used if more than one *FormContainer* is in one line. If only one *FormContainer* is in the line, then the *labelSpanM* value is used.

- *labelSpanM**:

  The default span for labels in the medium size. The default value is 2.

  > Note:
  > If *adjustLabelSpan* is set, this property is used for full-size *FormContainer* elements. If more than one *FormContainer* is in one line, *labelSpanL* is used.

- *labelSpanS**:

  The default span for labels in the small size. The default value is 12.

- *adjustLabelSpan**:

  If set, the usage of *labelSpanL* and *labelSpanM* are dependent on the number of *FormContainer* elements in one row. If only one *FormContainer* is displayed in one row, *labelSpanM* is used to define the size of the label. This is the same for the medium and large *Forms*. This is done to align the labels on forms where full-size *FormContainer* elements and multiple-column rows are used in the same *Form* (because every *FormContainer* has its own *Grid* inside). If not set, the usage of *labelSpanL* and *labelSpanM* are dependent on the *Form* size. The number of *FormContainer* elements does not matter in this case.

  > Note:
  > The default value is `true`.

© Copyright. All rights reserved.

- *singleContainerFullSize**:

  If the *Form* contains only one single *FormContainer* and this property is set, the *FormContainer* is displayed using the full size of the *Form*. In this case, the properties *columnsL* and *columnsM* are ignored. In all other cases, the *FormContainer* is displayed in the size of one column.

> Note:
> The default value is `true`.

- The **SimpleForm** is designed to allow an easy usage of the *Form control.*
- In the **SimpleForm** just the content controls are added.

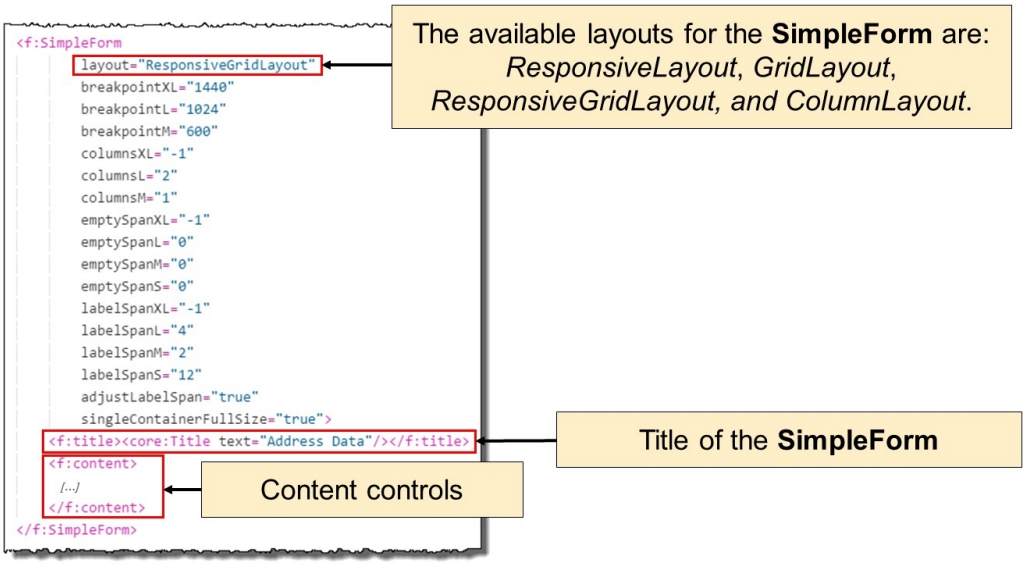


Figure 74: sap.ui.layout.form.SimpleForm Control (1)

The following content can be added:

- *Title* (based on `sap.ui.core.Title`)

  A new *FormContainer* is created. The content up to the next title is used as content of this *FormContainer*.

- *Label* (using interface `sap.ui.core.Label`)

  A new *FormElement* is created. This means that a new line inside the *FormContainer* starts. All subsequent content, up to the next *Label*, is part of this *FormElement*.

- All other content is used as a field of a *FormElement*. Here, all kinds of form controls can be added.

© Copyright. All rights reserved.

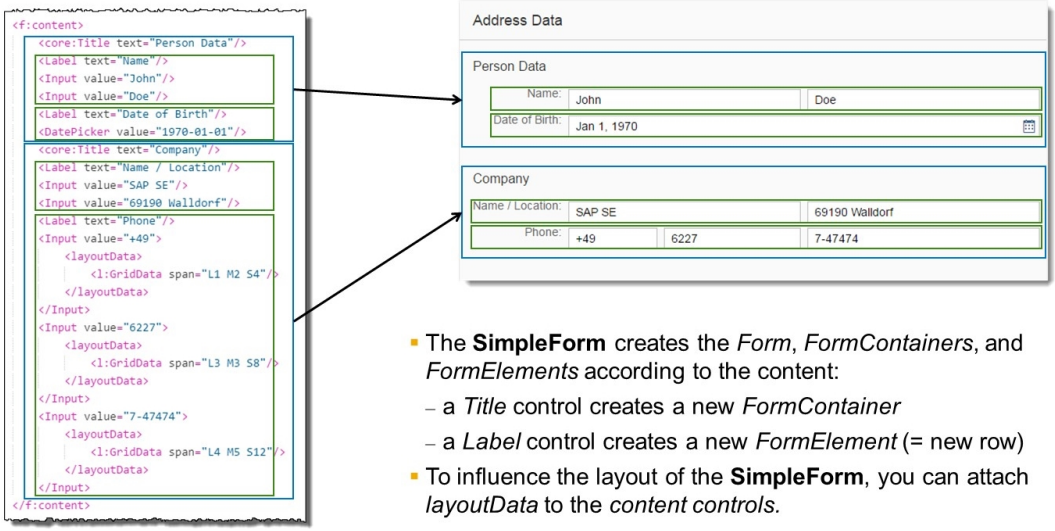


Figure 75: sap.ui.layout.form.SimpleForm Control (2)

sap.m.Panel Control

Panels are helpful for grouping custom content.

sap.m.Panel also provides the capability to expand/collapse its content (including the info toolbar, if available).

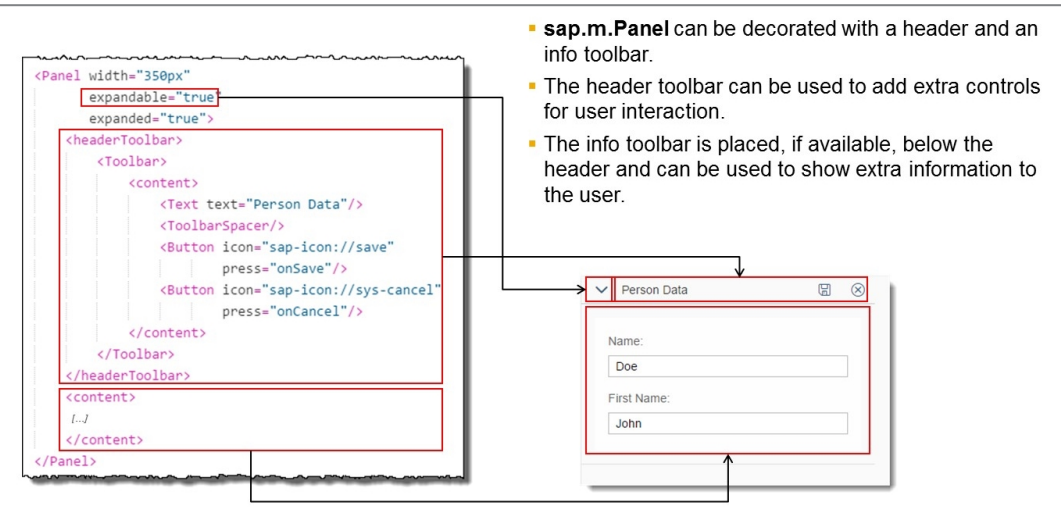


Figure 76: sap.m.Panel Control

Device Adaptation

SAPUI5 comes with several controls that 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 works out-of-the-box.

When using *sap.m.Table*, you can set a *minScreenWidth* property for the columns. This causes columns to show up only if a certain screen width is matched. You can define *minScreenWidth* in `px` or `rem` units, but you can also use the standard categories that come from the Device API (Phone, Tablet, or Desktop). Setting the additional property *demandPopin* to `true` for a column also reacts to the *minScreenWidth* you specify. In such a case, the column shows as a pop-in on smaller screens, instead of being completely hidden.

© Copyright. All rights reserved.

The Device API (`sap.ui.Device`) is an API that provides information about device specifics, like the operating system and its version, the browser and its version, the screen size, the current orientation, and support for specific features like, touch event support, orientation change, and so on.

Apart from the Device API, there is another way to check the operating system your application is running on. If an SAPUI5 application is running on a mobile device, a platform attribute is added to the HTML tag, which provides information about the current platform and version. In addition, the SAPUI5 adds a platform-dependent CSS class to the HTML tag of the page. The following are examples of this:

- `<html data-sap-ui-os="iOS9.1" […] class="sap-ios [...]">`
- `<html data-sap-ui-os="Android6.0" […] class="sap-android  [...]">`

The attributes and the CSS classes enable control or application developers to create platform-dependent styling for their controls or applications.

SAPUI5 supports several functions to adapt an application to operating systems and devices:

- **Controls with built-in device adaptation**, for example:
  - *sap.ui.layout.form.Form* – can adapt to the available screen size, particularly when used with *ResponsiveGridLayout*.
  - *sap.m.Table* (also called the "Responsive Table") – 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 hide columns completely.

Carriers

Id	Carrier	Currency	Url
LH	Lufthansa	EUR	http://www.lufthansa.com
JL	Japan Airlines	JPY	http://www.jal.co.jp
AZ	Alitalia	EUR	http://www.alitalia.it
DL	Delta Airlines	USD	http://www.delta-air.com

- **The Device API –** provides information about the used browser/device and cross platform support for certain events like orientation change or resizing (will be covered in the course later).

Figure 77: Device Adaptation (1)

For each device, you have a corresponding CSS class, which you can use to explicitly hide or show the particular control. The names of the CSS classes are generally self-explanatory.

Note:
The control is still part of the app, but hidden by CSS only. For managing visibility at a generic level, you can control the visible property with the Device API instead. This means the controls are not added to the DOM at all, but treated as invisible by SAPUI5.

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.

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. If your application is supposed to run on a smartphone, tablet, and desktop, it may be useful to choose your

© Copyright. All rights reserved.

margins depending on the screen width that is available. SAPUI5 comes with the CSS class `sapUiResponsiveMargin`, which does just that. It works with media queries to determine the available screen width and adapts its margin accordingly.

For many container controls in SAPUI5, such as *Dialog*, *Page*, *List*, *Table*, *Carousel*, *Panel*, *HorizontalLayout*, and *VerticalLayout*, 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. Setting the CSS class `sapUiResponsiveContentPadding` on the container control adds a padding based on the user's screen size.

It is also possible to add your own device-dependent or screen-size-dependent custom CSS.

There are several options available by using CSS:

- **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, and sapUiHideOnPhone*

```
<Button class="sapUiVisibleOnlyOnDesktop"
```

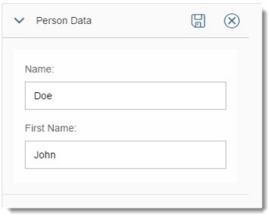
- **Responsive Margin and Padding Classes**
  - *sapUiResponsiveMargin – A margin clears an area around its respective control, outside of its border. If your application is supposed to run on a smartphone, tablet and desktop, it may be useful to choose your margins depending on the screen width that is available.*
  - *sapUiResponsiveContentPadding – A padding clears the area between the container layout and the controls that are displayed in the content area. Setting the CSS class sapUiResponsiveContentPadding on the container control adds a padding based on the user's screen size.*

Figure 78: Device Adaptation (2)

If you need to know which Content Densities are supported for a particular control, the best place to look is the Samples application, which is included as part of 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.

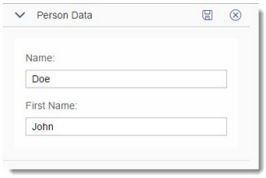
SAPUI5 contains different so-called **Content Densities** for controls to allow your application to adapt to the device in question:

- Content Density: **Cozy**
  - 'Large' design: Dimensions of the controls are optimized for touch-enabled devices.
  - Default Density for most controls, particularly those in the *sap.m library.*



Cozy Density

- Content Density: **Compact**
  - Reduced-size design: Dimensions of the controls and the spacing between them are reduced.
  - Mainly suitable for mouse-operated devices.



Compact Density

Figure 79: Device Adaptation (3)

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

© Copyright. All rights reserved.

scenario, we recommend that you set it at a high level, such as `<body>`, as in most cases, you need to set it for the whole app.

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, as the *Cozy Density* is exactly what you require. However, controls belonging to other libraries may also support a Cozy Design (for example, *sap.ui.table.Table*), but the default might be different (such as Compact Density). For this reason, if your application combines controls belonging to different libraries, we strongly recommend that you set the CSS class `sapUiSizeCozy` if you want to use *Cozy Density* (and similarly, to set the CSS class `sapUiSizeCompact` if you want to use *Compact Density*).

The CSS Density class only affects child controls. As dialogs and other popups are located at the root of the HTML document, they do not automatically inherit the respective Density. To set the Density for a dialog, either perform the same check as for the view, or use the convenience function `jQuery.sap.syncStyleClass()`. This convenience function synchronizes a style class between elements.

Beginning with SAPUI5 version 1.30, 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.

- *The Content Densities* **Cozy** and **Compact** are triggered by the CSS classes *sapUiSizeCozy* and *sapUiSizeCompact* respectively.
- Set the CSS class on the parent container of the UI region for which you want to use the corresponding *Content Density* (for example a view).

... controller.js ×

```
1  sap.ui.define([
2      "sap/ui/core/mvc/Controller",
3      "sap/ui/Device"
4  ], function(Controller, Device) {
5      "use strict";
6
7      return Controller.extend("   ...   ", {
8
9          onInit: function() {
10             // apply compact density if touch is not supported, the standard cozy design otherwise
11             this.getView().addStyleClass(Device.support.touch ? "sapUiSizeCozy" : "sapUiSizeCompact");
12         }
13
14         [...]
15     });
16 });
```

- In this example, the *Device API* is queried for touch support of the client.
- The CSS class *sapUiSizeCompact* is set for the view, if touch interaction is not supported and *sapUiSizeCozy* for all other cases.

Figure 80: Device Adaptation (4)

LESSON SUMMARY

You should now be able to:

- Use layout controls

© Copyright. All rights reserved.

Learning Assessment

1. Which layout control can help you group UI controls?

   Choose the correct answer.

   - [ ] A *sap.ui.layout.Panel*
   - [ ] B *sap.m.Panel*
   - [ ] C *sap.ui.layout.Group*
   - [ ] D *sap.m.GridGroup*

2. What is the preferred way, when you have the requirement, to hide certain UI controls on a specific form factor?

   Choose the correct answer.

   - [ ] A Get the form factor using the device and call the `setVisible()` function of the UI control.
   - [ ] B Assign a standard CSS class, such as `sapUiVisibleOnlyOnDesktop` of SAPUI5, to the UI control.
   - [ ] C Bind the *formFactor* property of the device model to the *visible* property of the UI control.

3. Which measurement unit is used for the property *hSpacing*?

   Choose the correct answer.

   - [ ] A em
   - [ ] B rem
   - [ ] C px
   - [ ] D percentage

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which layout control can help you group UI controls?

   Choose the correct answer.

   - [ ] A *sap.ui.layout.Panel*
   - [x] B *sap.m.Panel*
   - [ ] C *sap.ui.layout.Group*
   - [ ] D *sap.m.GridGroup*

   Correct. The *sap.m.Panel* layout control is a container for grouping and displaying information. It can be collapsed to save space on the screen.

2. What is the preferred way, when you have the requirement, to hide certain UI controls on a specific form factor?

   Choose the correct answer.

   - [ ] A Get the form factor using the device and call the `setVisible()` function of the UI control.
   - [x] B Assign a standard CSS class, such as `sapUiVisibleOnlyOnDesktop` of SAPUI5, to the UI control.
   - [ ] C Bind the *formFactor* property of the device model to the *visible* property of the UI control.

   Correct. Since version 1.28, SAPUI5 provides CSS classes to handle various layout aspects. `sapUiVisibleOnlyOnDesktop` is used to adapt the visibility of an UI control.

3. Which measurement unit is used for the property *hSpacing*?

   Choose the correct answer.

   - [ ] A em
   - [x] B rem
   - [ ] C px
   - [ ] D percentage

   Correct. The measurement unit rem is commonly used for SAPUI5.

© Copyright. All rights reserved.

UNIT 8 Fragments

Lesson 1

Using Fragments 111

UNIT OBJECTIVES

- Use fragments

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 8
Lesson 1

Using Fragments

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use fragments

Fragments Overview

Fragments support, reuse, and view modularization without adding overhead.

To support different use cases, and because some developers prefer declarative UIs while others prefer to programmatically construct them, SAPUI5 provides different types of fragments.

Defining a fragment is similar to defining views within a separate file. The fragments simply end with `*.fragment` instead of `*.view`. The same rules for file location apply.

Instead of defining fragments externally in a separate file, they can also be defined inline and can be instantiated immediately. The content definition and the instantiation syntax are almost the same for inline defined fragments. Inline definition of fragments can be used for prototyping, for dynamic fragment composition, or for loading fragment content from sources that are not accessible by the normal module loading mechanism. However, in general, the inline definition of fragments plays only a minor role.

- Fragments are light-weight UI parts (UI sub-trees) that can be reused.
- Fragments are defined in a manner similar to views, but do not have any controller or other behavior code involved.
- Fragments can connect to the existing controllers of the "owning" view.
- A fragment itself has no HTML representation when it is inserted into the UI tree. Instead, its content is inserted.
- Fragments are found by the SAPUI5 runtime using the module loading mechanism.
- The available fragment types are:
  - XML fragments
  - HTML fragments
  - JavaScript fragments

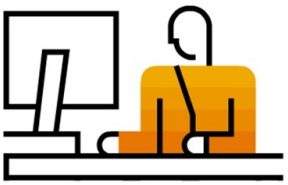


Figure 81: Fragments Overview

HTML Fragments

The figure, HTML Fragments, shows the syntax for HTML fragments.

© Copyright. All rights reserved.

HTML fragments have a syntax similar to that of HTML views, but without the <template> tag.

A HTML fragment file name has the suffix *.fragment.html*.

HtmlFrag.fragment.html

```
1 <div data-sap-ui-type="sap.m.Button"
2      data-press="onButtonPress"
3      data-text="Button on HTML Fragment">
4 </div>
```

HTML fragments can use controller methods for binding event handler methods, but only if the code instantiating them passes a controller.

Figure 82: HTML Fragments

XML Fragments

The XML fragment shown in the figure, XML Fragments, can only be instantiated with a controller that has a method called `onButtonPress`. If not, the code throws an error.

- XML fragments are similar to XML views, but have no <*View*> tag as the root element.
- The <*FragmentDefinition*> tag does not have any representation in HTML at runtime.

XmlFrag1.fragment.xml

```
1 <core:FragmentDefinition xmlns="sap.m"
2                          xmlns:core="sap.ui.core">
3
4     <Button press="onButtonPress"
5             text="Button on XML Fragment" />
6
7 </core:FragmentDefinition>
```

An XML fragment file name has the suffix *.fragment.xml* .

The event handler is bound to the *onButtonPress* method of a controller.

Figure 83: XML Fragments

JavaScript Fragments

The `createContent()` function is responsible for the UI definition and has to return a control. 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 between fragments and views, there are also differences. Firstly, 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 from 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. Controllers may expect certain controls to be in the fragment. This is in line with the purpose of fragments; that is, to be very light-weight, and re-use entities that provide little encapsulation. For more encapsulation, views, or even components, are better alternatives.

© Copyright. All rights reserved.

- 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.
- However, there is no *getControllerName()* method.

JsFrag.fragment.js

```
1  sap.ui.jsfragment("sap.training.view.JsFrag", {
2
3      createContent: function(oController) {
4
5          var oButton = new sap.m.Button({
6              text: "Button on JavaScript Fragment",
7              press: oController.onButtonPress
8          });
9
10         return oButton;
11     }
12 });
```

A JavaScript fragment file name has the suffix *.fragment.js*.

The *oController* object is passed by the code instantiating the fragment or it is *null*.

Figure 84: JavaScript Fragments

Instantiating Fragments Programmatically

When comparing fragments to views, there is one important difference; fragments are not 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.

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 this problem is to specify a unique fragment ID. This ID is then used as the prefix for all controls in the fragment instance.

- SAPUI5 provides the following method to load and instantiate a fragment:
  - ***sap.ui.core.Fragment.load****(mOptions): Promise*
- The *load()* method returns a *Promise* which resolves with the content of the fragment.
- Among other things, the following optional parameters can be passed to the *load()* method:
  - ***id***: ID of the fragment that will be used as prefix for the IDs of all contained controls.
  - ***name***: This name must be resolvable to the fragment file URL by the SAPUI5 module loading mechanism.
  - ***type***: The fragment type ("XML" (default), "JS", or "HTML")
  - ***controller***: The controller which should be used by the controls in the fragment.

Figure 85: Instantiating Fragments Programmatically

Instantiating Fragments Programmatically: Example

The figure, Instantiating Fragments Programatically: Example, shows how a controller instance is passed using the parameter `controller`. This is required because the particular fragment tries to bind the button `press` handler to the method `onButtonPress()` in the given controller. If no controller is given, this causes an error. Other fragments, not requiring a controller, can be instantiated without passing a controller.

For fragments that are used several times, an ID for the fragment can be provided optionally.

© Copyright. All rights reserved.

The content of the fragment (that is, the button) can be used as if created in a standard way. In the example, the button is added to a `HorizontalLayout` control, which is the content of the JS view.

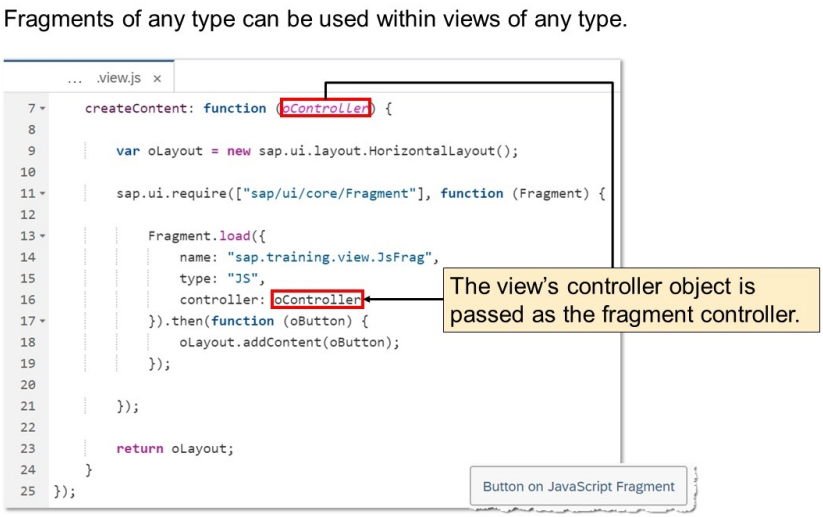


Figure 86: Instantiating Fragments Programmatically: Example

Using Other Objects Instead of Controllers

The controller 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.

The corresponding object is passed to the `createContent()` method of JavaScript fragments. In the case of the declarative fragment types, that is XML or HTML fragments, the event handler methods are searched on this object.

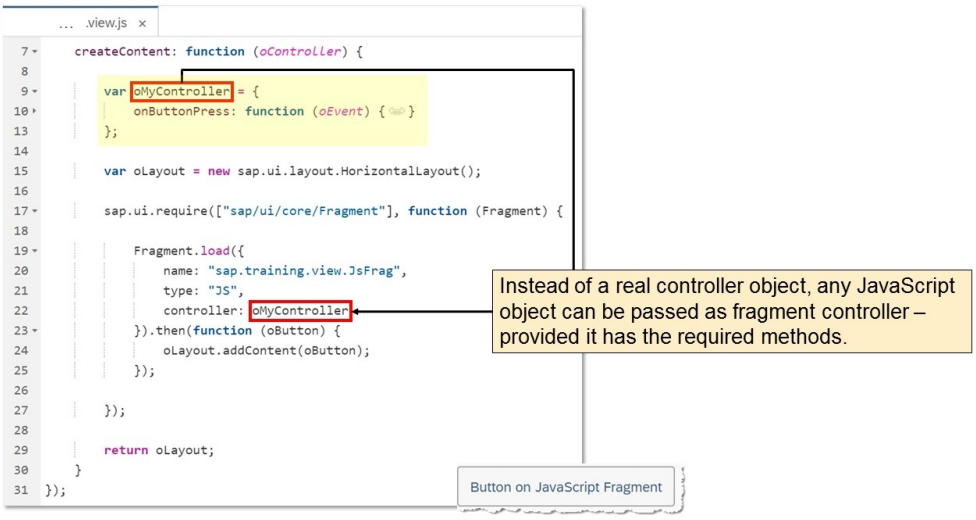


Figure 87: Using Other Objects Instead of Controllers

Instantiating Fragments in XML Views

In XML views, fragments are used like regular controls, or more precisely, like views.

The example in the figure, Instantiating Fragments in XML Views, shows an XML view that includes all three types of fragments; that is an XML fragment, a JavaScript fragment, and an

© Copyright. All rights reserved.

HTML fragment. The fragment references work like import statements including the fragment content controls.

- In XML views, fragments are instantiated using the *Fragment* tag.
- As fragment controller, the view's controller object is passed implicitly.

Main3.view.xml

```
1  <mvc:View controllerName="sap.training.controller.Main3"
2      xmlns:core="sap.ui.core"
3      xmlns:html="http://www.w3.org/1999/xhtml"
4      xmlns:l="sap.ui.layout"
5      xmlns:mvc="sap.ui.core.mvc"
6      xmlns="sap.m">
7
8      <l:HorizontalLayout>
9          <l:content>
10             <core:Fragment fragmentName="sap.training.view.XmlFrag1" type="XML"/>
11
12             <core:Fragment fragmentName="sap.training.view.HtmlFrag" type="HTML"/>
13
14             <core:Fragment fragmentName="sap.training.view.JsFrag" type="JS"/>
15         </l:content>
16     </l:HorizontalLayout>
17
18 </mvc:View>
```

Figure 88: Instantiating Fragments in XML Views

IDs in Fragments

The figure, IDs in Fragments, describes the consequences of including, or not including, an ID.

XmlFrag2.fragment.xml

```
1  <core:FragmentDefinition xmlns="sap.m"
2                           xmlns:core="sap.ui.core"
3                           xmlns:l="sap.ui.layout">
4
5      <l:HorizontalLayout>
6          <l:content>
7              <Input                  width="470px"/>
8              <Input id="idInput" width="470px"/>
9          </l:content>
10     </l:HorizontalLayout>
11
12 </core:FragmentDefinition>
```

If no *id* is given, SAPUI5 will always generate a unique *id* (for instance, *__input0*).

It is not possible to retrieve the control instance programmatically.

The control instance can be retrieved programmatically by its *id*.

If a fragment containing controls with explicit *ids* is instantiated twice without giving a fragment *id*, a duplicate *id* error occurs.

Figure 89: IDs in Fragments

ID Prefixing in XML Views

Static IDs in fragments are by default used as is; that is, they are not prefixed to make them unique. However, when a fragment is supposed 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 following rules apply for explicitly given control IDs in fragments:

© Copyright. All rights reserved.

- Given IDs are prefixed with only the view ID when no fragment ID is given.
- Given IDs are prefixed with both the view ID and the fragment ID when a fragment ID is given.

A control instance can be found in a fragment by means of its ID. Due to the described prefixing that guarantees unique IDs, there are different cases possible that require different calls. Assuming the control has the ID `idInput`, the fragment is embedded into a view and the code is inside a controller:

- When no fragment *id* is given:

  `oControl = this.getView().byId("idInput")`, where `this` is the controller object.

- When the fragment *id* `idFrag` is given:

  `oControl= this.getView().byId(sap.ui.core.Fragment.createId("idFrag", "idInput"))`, where `this` is the controller object.

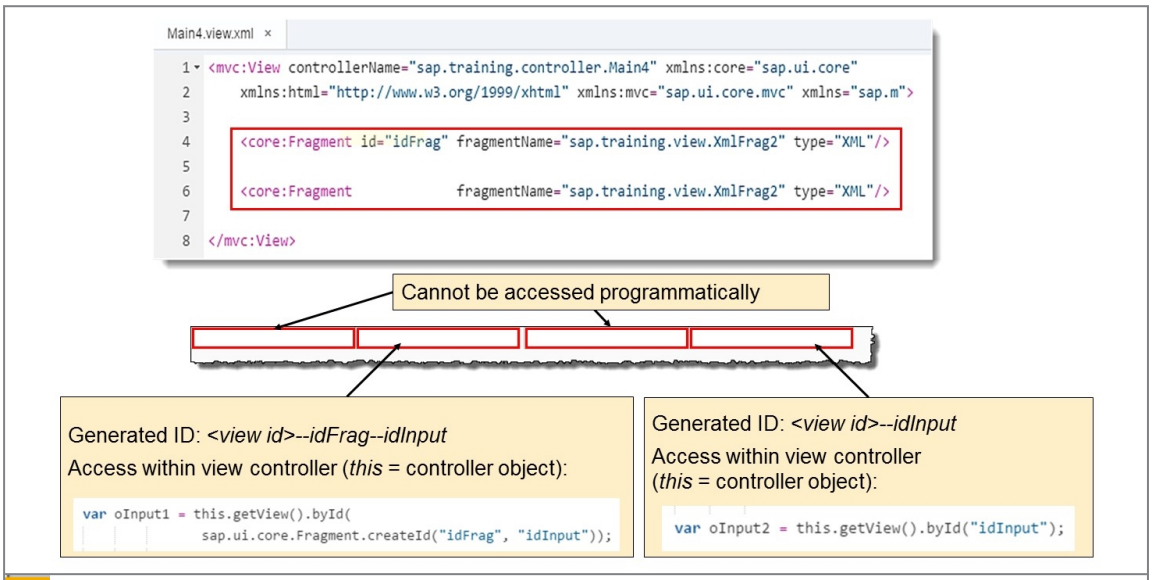


Figure 90: ID Prefixing in XML Views

Dialogs as XML Fragments

Dialogs are special, because they open on top of the regular app content and thus do not belong to a specific view. In addition, dialogs can be used in more than one view of an app. Therefore, fragments can be particularly useful for defining dialogs.

> Note:
> The dialog shown in the figure, Dialogs as XML Fragments, has the `ID` **`demoDialog`**. This `ID` is used later in the controller to access the dialog.

© Copyright. All rights reserved.

- You can use fragments to define dialogs and other pop-up controls that are not part of the normal page UI structure.
- To use fragments for defining pop-ups, just let the root control of the fragment be a dialog or similar control.
- JavaScript and HTML fragments are used the same way to define a dialog as a fragment.

Dialog.fragment.xml

```
1  <core:FragmentDefinition xmlns="sap.m"
2                           xmlns:core="sap.ui.core">
3
4      <Dialog id="demoDialog" title="XML Fragment Dialog">
5          <buttons>
6              <Button press="onCloseDialog" text="Close"/>
7          </buttons>
8
9          <content>
10             <Input id="idInput"/>
11         </content>
12     </Dialog>
13
14 </core:FragmentDefinition>
```

Figure 91: Dialogs as XML Fragments

Using Dialogs Defined as Fragments

To open the dialog, a button is used on the view. It calls the shown event handler function `onOpenDialog()` in the controller of the view.

If the dialog in the fragment does not exist yet, the fragment is instantiated by calling the `sap.ui.core.Fragment.load()` method with the following parameters:

- The `id` of the view:

  This parameter is used to prefix the IDs inside the fragment. This ensures that even if you instantiate the same fragment in other views in the same way, each dialog will have its own 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.

  On the other hand, using the view ID as fragment ID allows calling `this.byId()` in the view's controller to retrieve the controls inside the fragment (`Dialog` control with ID **`demoDialog`** and `Input` control with ID **`idInput`**). For example, you can access the dialog in the view controller by calling `this.byId("demoDialog")`.

- The `name` of the fragment definition
- The `type` of the fragment
- The `controller` object:

  There is a button on the dialog whose `press` event is referring to an event handler called `onCloseDialog`. The `controller` parameter is used to pass the view controller (`this`) as fragment controller. This means that the required event handler function `onCloseDialog` is put into the view controller and it closes the dialog by accessing it via `this.byId()`.

The `sap.ui.core.Fragment.load()` method returns a `Promise` that resolves with the fragment's root control, which is a `Dialog` control that can be used like any dialog. In the example shown in the figure, Using Dialogs Defined as Fragments, the dialog is opened using its `open()` method.

Always use the `addDependent()` method to connect the dialog to the lifecycle management and data binding of the view. A convenient side-effect is that the dialog will automatically be destroyed when the view is destroyed. Otherwise, you would have to destroy the dialog manually to free its resources.

© Copyright. All rights reserved.

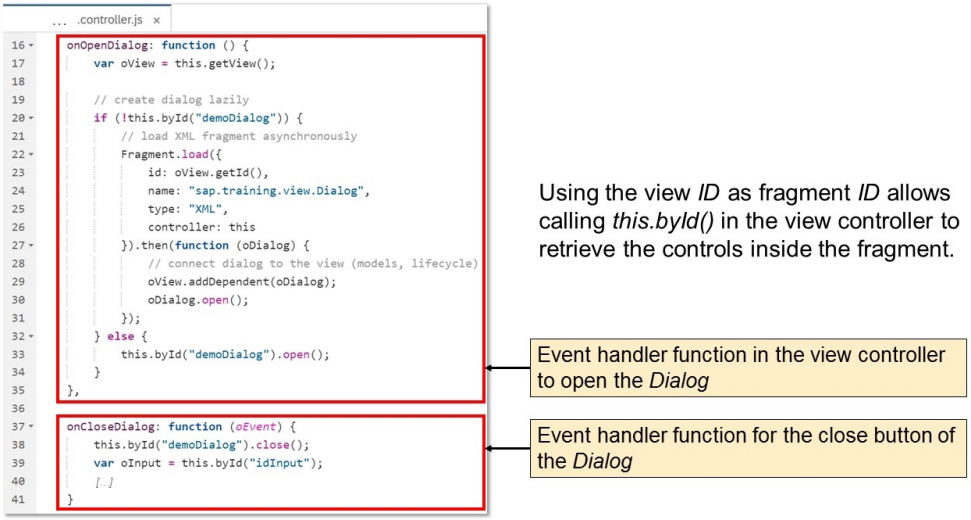


Figure 92: Using Dialogs Defined as Fragments

Fragments with Multiple Root Nodes

Because XML documents must have exactly one root node, to achieve XML fragments with multiple root nodes, an additional tag is needed. This `<FragmentDefinition>` tag has no representation in HTML at runtime; the children are added directly to wherever the fragment is placed.

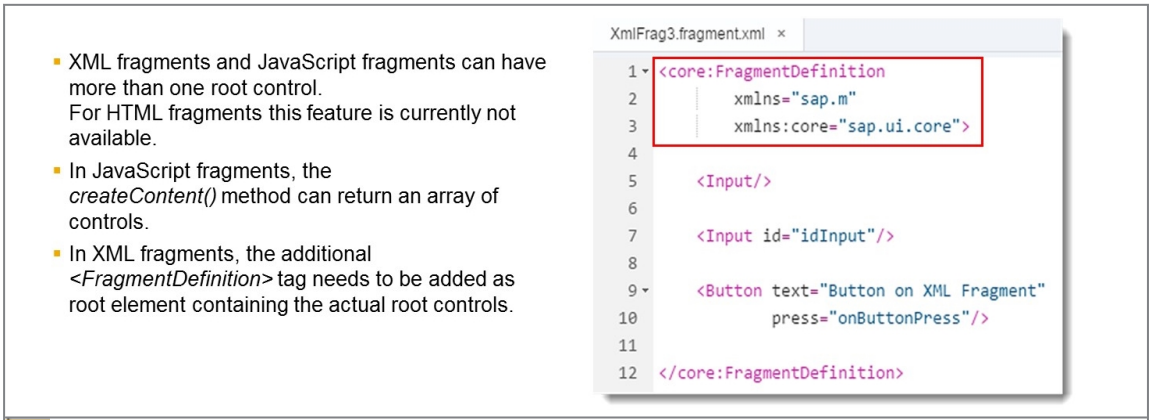


Figure 93: Fragments with Multiple Root Nodes

LESSON SUMMARY

You should now be able to:

- Use fragments

© Copyright. All rights reserved.

Learning Assessment

1. Which fragment type is not available in SAPUI5?

   Choose the correct answer.

   - ☐ A XML fragment
   - ☐ B JSON fragment
   - ☐ C HTML fragment
   - ☐ D JavaScript fragment

2. Which of the following statements are true with respect to fragments?

   Choose the correct answers.

   - ☐ A Fragments are defined in a manner similar to views.
   - ☐ B A fragment itself has a HTML representation.
   - ☐ C Fragments are light-weight UI parts.
   - ☐ D Fragments need a controller, similar to views.

3. When you implement a dialog using a fragment, you should bind the fragment to the lifecyle management of the consuming view. Which function do you need to call on the consuming view to connect the dialog to the lifecyle management of the view?

   Choose the correct answer.

   - ☐ A `addDependent()`
   - ☐ B `connectToView()`
   - ☐ C `addFragment()`
   - ☐ D `assignDependent()`

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which fragment type is not available in SAPUI5?

   Choose the correct answer.

   - [ ] A XML fragment
   - [x] B JSON fragment
   - [ ] C HTML fragment
   - [ ] D JavaScript fragment

   Correct. The fragment concept is implemented for XML, HTML, and JavaScript.

2. Which of the following statements are true with respect to fragments?

   Choose the correct answers.

   - [x] A Fragments are defined in a manner similar to views.
   - [ ] B A fragment itself has a HTML representation.
   - [x] C Fragments are light-weight UI parts.
   - [ ] D Fragments need a controller, similar to views.

   Correct. Fragments are light-weight UI parts (UI sub-trees) that can be reused, and are defined similar to views, but do not have any controller or other behavior code associated with them.

© Copyright. All rights reserved.

3. When you implement a dialog using a fragment, you should bind the fragment to the lifecyle management of the consuming view. Which function do you need to call on the consuming view to connect the dialog to the lifecyle management of the view?

   Choose the correct answer.

   - [x] A `addDependent()`
   - [ ] B `connectToView()`
   - [ ] C `addFragment()`
   - [ ] D `assignDependent()`

   Correct. You must call the `addDependent()` function to connect the dialog to the lifecyle management of the consuming view. Read more on this in the Using Fragments lesson of the course UX400.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 9 Models and Data Binding

Lesson 1

Exploring Models and Data Binding 125

UNIT OBJECTIVES

- Explore models and data binding

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 9
Lesson 1

Exploring Models and Data Binding

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explore models and data binding

Models

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 side without further server requests.

The OData Model is a server-side model, and only loads the data requested by the user interface from the server. Any changes in data binding or list operations require a new request to the server. This also means that sorting and filtering on the client side is not possible. For this, the client has to send a request to the server.

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.

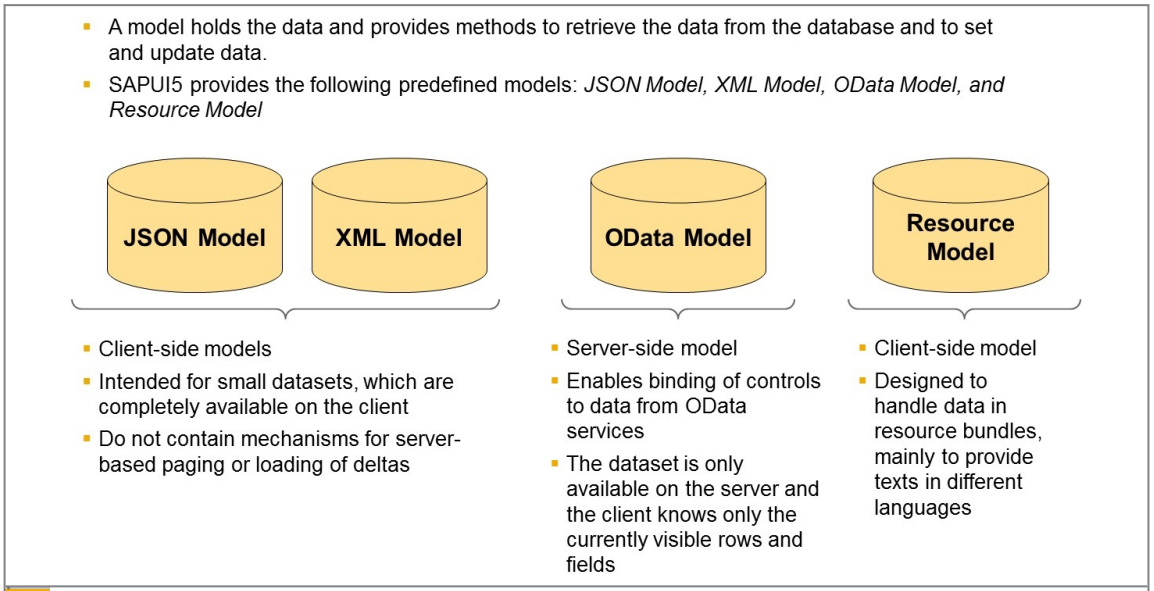


Figure 94: Models

Instantiating a JSON Model

After the model 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.

The JSON Model also has a `loadData()` method that loads the JSON data from the specified URL asynchronously and applies it to the model.

© Copyright. All rights reserved.

- The *JSON* Model can be used to bind controls to JavaScript object data, which is usually serialized in the JSON format.
- The first step when using a *JSON* Model is to instantiate the model and to provide the data for that model.

data.json ×

```
1 {
2     "firstName": "John",
3     "enabled": true
4 }
```

```
var oModel = new sap.ui.model.json.JSONModel();

oModel.setData({
    "firstName": "John",
    "enabled": true
});

//oModel.loadData("model/data.json");
```

Figure 95: Instantiating a JSON Model

Assigning the Model to the UI

To find the model it should bind to, a control looks up the parent hierarchy until it finds a control or UI area that has an assigned model.

- The SAPUI5 framework must know about a model instance, before you can bind controls to this model instance.
- You can define a model for a view that can be accessed by all controls embedded in the view by using the *setModel()* method on the SAPUI5 view object.

MyView.controller.js ×

```
10    onInit: function() {
11
12            [...]
13        var oView = this.getView();
14
15        oView.setModel(oModel);
16    }
```

- You can also define a **specific model** for sections within a UI area by means of the *setModel()* method available on any control.

Figure 96: Assigning the Model to the UI

Binding Controls to the Model

To retrieve data from the model for visualization in the UI, the controls must be bound to the model by means of a 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, while relative binding paths start with a name token and are resolved relative to the context of the control that is bound.

The UI uses data binding to bind UI controls to the model, which holds the application data. For data binding, you need a model and a data binding instance. The model instance holds the

© Copyright. All rights reserved.

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 data binding instance is created, which contains the binding information and provides an event that is fired whenever the bound data is changed. An element can listen to this event and update its visualization according to the new data.

For binding controls to the model, several binding types exist. In the example shown in the figure, Binding Controls to the Model, Property Binding is used. It 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 function or datatype to convert the data as needed.

- UI controls are bound to the model by means of a **binding path**.
- The **JSON Model** supports *two-way binding*:
  - Controls are updated automatically whenever application data is changed.
  - Changes in the controls cause updates in the underlying application data.

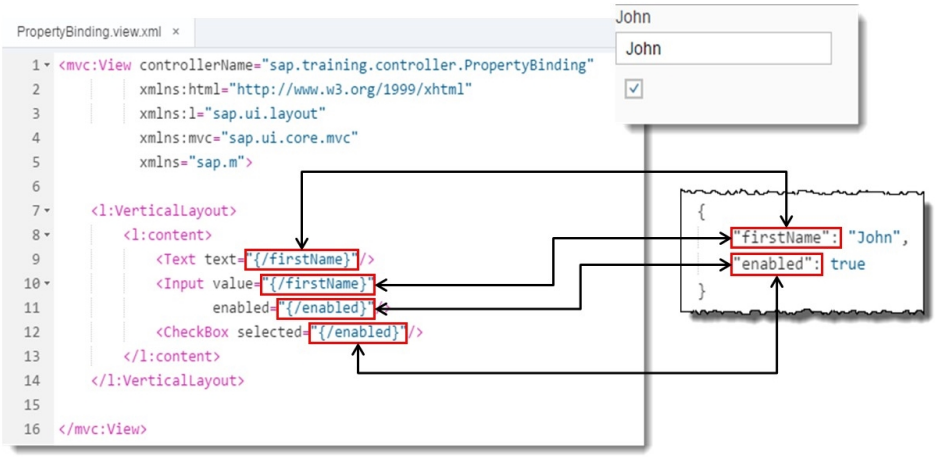


Figure 97: Binding Controls to the Model

Binding Modes

The table in the figure, Binding Modes, shows the binding modes that each of the respective models support.

The Resource Model only supports the one-time binding mode because it deals with static texts only.

The *binding mode* defines how the data sources are bound:

- **One-Way**: Binding from the model to the view; value changes in the model update all corresponding bindings and the view.
- **Two-Way**: Binding from the model to the view and from the view to the model; value changes in the model and in the view update all corresponding bindings and the view and model, respectively.
- **One-Time**: Binding from the model to the view once.

Model	One-way	Two-way	One-time
Resource Model	--	--	X
JSON Model	X	X	X
XML Model	X	X	X
OData Model	X	X	X

Figure 98: Binding Modes

© Copyright. All rights reserved.

Default Binding Mode of Models

The table in the figure, Default Binding Mode of Models, shows the default binding mode for each model implementation.

To change the default binding mode, call the `setDefaultBindingMode()` method on the model. In the example, all new bindings for the model have the one-way binding mode by default.

- When a model instance is created, the instance has a *default binding mode*. All bindings of this model instance have this binding mode as their default.

Model	Default Binding Mode
Resource Model	one-time
JSON Model	two-way
XML Model	two-way
OData Model	one-way

- You can overwrite the default binding mode after model creation.

```
var oModel = new sap.ui.model.json.JSONModel();

oModel.setDefaultBindingMode(sap.ui.model.BindingMode.OneWay);
```

Figure 99: Default Binding Mode of Models

Multimodel Support

The figure, Multimodel Support, describes how SAPUI5 supports multiple models for binding.

- SAPUI5 supports the use of more than one model for data binding.
- You can set an additional model for a view or for an element by specifying a name for the model to identify it.

MyView.controller.js

```
10    onInit: function() {
11
12        var oModel1 = new sap.ui.model.json.JSONModel();
13        oModel1.loadData("model/data_1.json");
14        this.getView().setModel(oModel1, "model_1");
15
16        var oModel2 = new sap.ui.model.json.JSONModel();
17        oModel2.loadData("model/data_2.json");
18        this.getView().setModel(oModel2, "model_2");
19    }
```

```
<Input value="{model_1>/firstName}"
       enabled="{model_1>/enabled}"/>
```

```
<Text text="{model_2>/street}"/>
```

- In the binding, you specify the model name and following the > sign, you specify the binding path to the values in the model that are to be bound.

Figure 100: Multimodel Support

© Copyright. All rights reserved.

Binding Types

When using Property Binding, you can only bind control properties to model properties of a matching type, or you use a formatter function or datatype to convert the data as needed. Once you have defined the Property Binding, the property is updated automatically every time the bound model property value is changed.

Aggregation Binding 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.

SAPUI5 supports three different *binding types*:

- **Property Binding**
  Property Binding allows properties of the control to get automatically initialized and updated from model data.
- **Aggregation Binding**
  Aggregation Binding can be used to automatically create child controls according to model data.
- **Element Binding**
  Element Binding allows to bind UI elements to a specific object in the model data, which will create a binding context and allows relative binding within the control and all of its children. This is especially helpful in master/detail scenarios.

Figure 101: Binding Types

Aggregation Binding Using Template Controls

Aggregation Binding is used to automatically create child controls according to model data.

Aggregation Binding requires the definition of a template, which is cloned for each bound entry of the model. For each clone that is created, the binding context is set to the respective model entry, so that all bindings of the template are resolved relative to the entry. The aggregated elements are destroyed and recreated whenever the bound model data is changed.

To bind an aggregation, you create a template or provide a factory function.

A template is not necessarily a single control, but can be a tree of controls. For each entry of the model, a deep clone of the template is created and added to the bound aggregation.

© Copyright. All rights reserved.

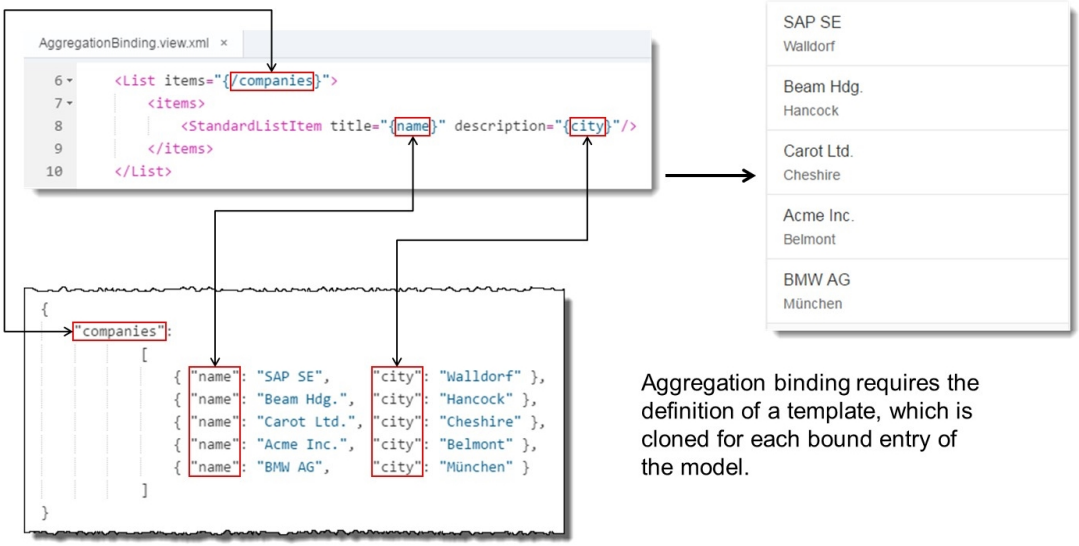


Figure 102: Aggregation Binding Using Template Controls

Element Binding

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 previously (for example in master/detail scenarios).

For server-side models, the Element Binding triggers the request to the server to load the referenced object in case it is not yet available on the client.

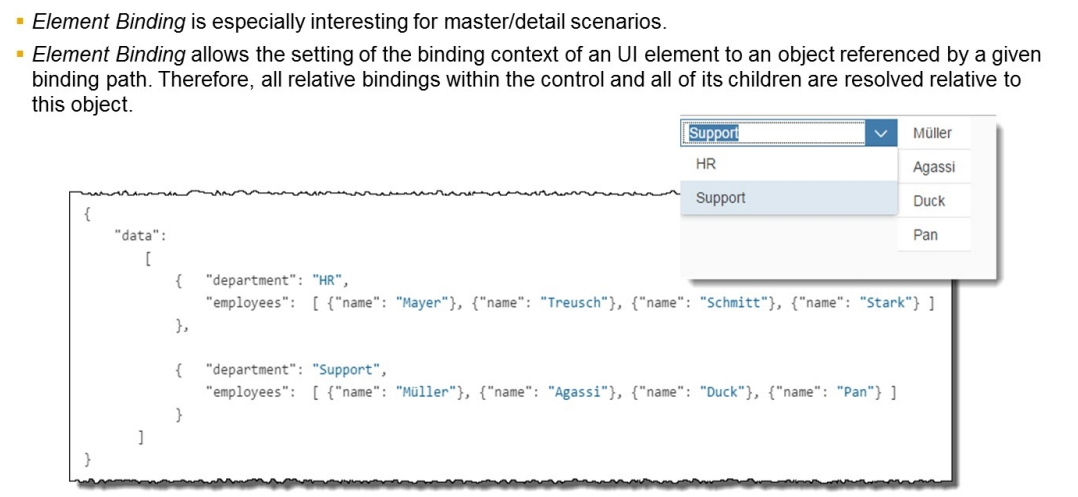


Figure 103: Element Binding (1)

The figure, Element Binding (2), shows how to define an Element Binding for a control.

© Copyright. All rights reserved.

To define an Element Binding, use the *bindElement()* method on a control.

ElementBinding.view.xml

```
11    <ComboBox items="{/data}" selectionChange="onSelectionChange">
12        <items>
13            <core:Item text="{department}"/>
14        </items>
15    </ComboBox>
16
17    <List id="idList" items="{employees}">
18        <items>
19            <StandardListItem title="{name}"/>
20        </items>
21    </List>
```

ElementBinding.controller.js

```
21    onSelectionChange: function(oEvent) {
22        var oItem = oEvent.getParameter("selectedItem");
23        var sPath = oItem.getBindingContext().getPath();
24
25        var oList = this.getView().byId("idList");
26        oList.bindElement(sPath);
27    }
```

Figure 104: Element Binding (2)

UI5 Inspector

The figure, UI5 Inspector (1), highlights the key features of the UI5 Inspector.

- With the **UI5 Inspector**, you can inspect, analyze, and support OpenUI5 and SAPUI5-based apps.
- Key features:
  - Inspect UI5 controls and review their properties, bindings, and data model.
  - Modify control properties on the fly and see how this affects the rendering and behavior.
  - Find relevant framework information for your OpenUI5/SAPUI5 app.
- You can download the **UI5 Inspector** as a standard extension from the Chrome Web Store.
- Once installed, the **UI5 Inspector** is available in Chrome DevTools (by choosing F12). It becomes active when a UI5 app is loaded.

Figure 105: UI5 Inspector (1)

The main UI features are highlighted in the figure, UI5 Inspector (2).

Clicking the *browser action* icon in the address bar provides you with the following:

- Information on the used UI5 version
- Links to the “What's New” section and the UI5 documentation

The *Control Inspector* tab page shows the structure and nesting of the UI5 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 UI5 control from the app and select *Inspect UI5 control*. This automatically selects the control in the tree and you can review its properties directly.

© Copyright. All rights reserved.

In the *Properties* tab page, 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 are validated against the framework and rendered dynamically. Errors from incorrectly set values are displayed in the Chrome browser console.

In the *Bindings* tab page, you can see the bindings for a specific control. The number of bindings is displayed in parentheses in the tab title. The model property holds a link to the corresponding binding file. Clicking on the link opens the *Model* section.

In the *Application Information* tab page, you can see overall information for your app, for example, the exact UI5 version you are running, the version of your browser, and the app URL.

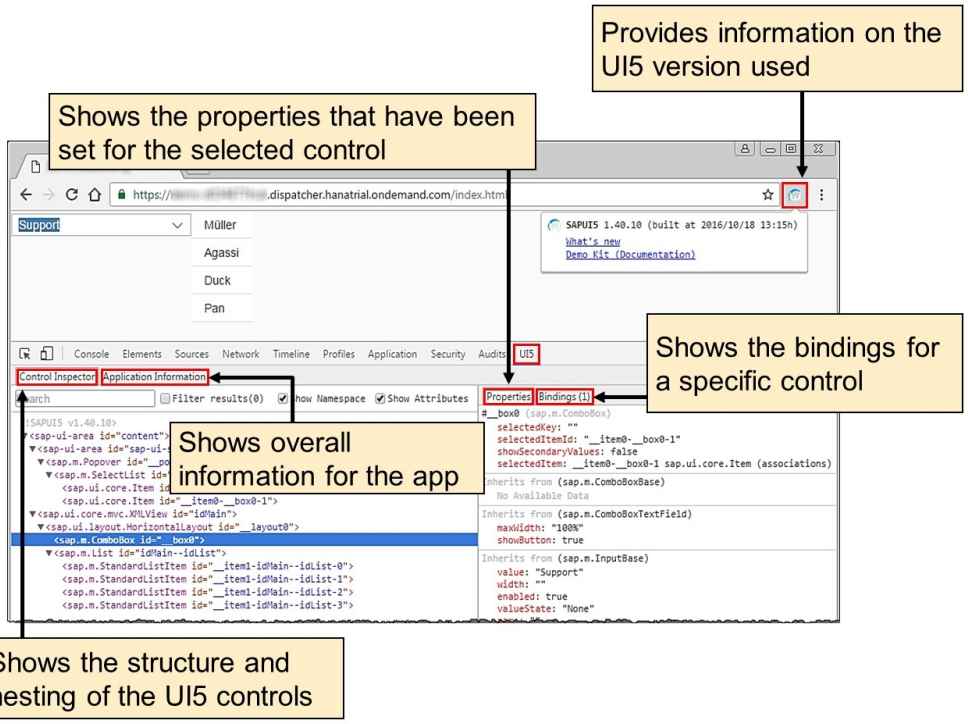


Figure 106: UI5 Inspector (2)

Data Binding Type System

The Simple Types use the formatter classes `sap.ui.core.format.DateFormat`, `sap.ui.core.format.NumberFormat`, and `sap.ui.core.format.FileSizeFormat`.

Whereas the Simple Types are not intended to be used without data binding, the formatter classes can be used standalone (for example, to format dates). The formatter classes do not provide validation.

The `DateFormat` class can be used to parse a string representing a date, time, or datetime into a JavaScript date object and vice versa.

The `NumberFormat` class can be used to parse a string representing a number (float or integer) into a JavaScript number and vice versa.

The `FileSizeFormat` class is a static class for formatting and parsing numeric file size values according to a set of format options.

© Copyright. All rights reserved.

- Data binding supports the definition of binding types that are handed over when binding properties.
- By means of **Simple Types**, you can define the data type of the properties used in the model.
- **Simple Types** provide the possibility to format and parse data, as well as validating if the entered data is within defined constraints.
- The following **Simple Types** are currently available for data binding in SAPUI5:
  - *sap.ui.model.type.Integer*
  - *sap.ui.model.type.Float*
  - *sap.ui.model.type.String*
  - *sap.ui.model.type.Boolean*
  - *sap.ui.model.type.Date*
  - *sap.ui.model.type.Time*
  - *sap.ui.model.type.DateTime*
  - *sap.ui.model.type.Currency*
  - *sap.ui.model.type.FileSize*

Figure 107: Data Binding Type System

Simple Types

The figure, Simple Types, shows the functions called when using simple types.

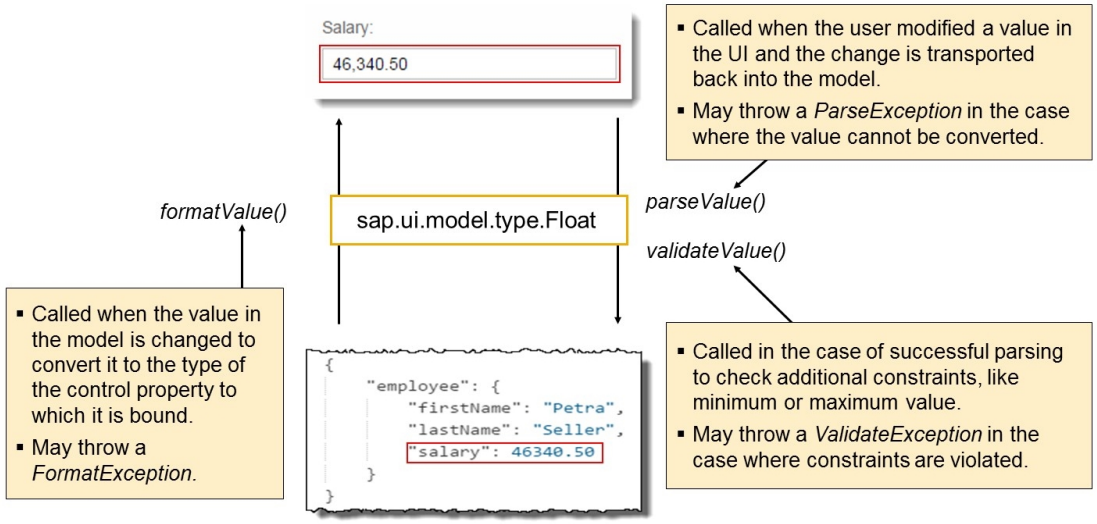


Figure 108: Simple Types

sap.ui.model.type.Float Data Type

The source value for the `Float` data type (the value given in the model), must be given as a number and is transformed, in one of the following ways, into the type of the bound control property:

- `float`:

  No transformation needed

- integer:

  The value is rounded using `Math.floor`

- `string`:

  The value is formatted or parsed according to the given output pattern.

For format options that are not specified, default values according to the type and locale settings are used.

To use the previously referenced Extended Syntax in XML views, the application must run with the complex binding syntax (described later).

© Copyright. All rights reserved.

- The **Float** data type supports the validation constraints *maximum* and *minimum*.
- The supported format options are listed in the *API Reference*, see class *sap.ui.core.format.NumberFormat*.

Datatypes.view.xml

```
64    <Label text="Salary"/>
65    <Input id="idSalaryInput"
66           value="{ path: '/employee/salary',
67                    type: 'sap.ui.model.type.Float',
68                    formatOptions: {minFractionDigits: 2, maxFractionDigits: 2},
69                    constraints: {minimum: 0}
70                  }"
71    />
```

Figure 109: sap.ui.model.type.Float Data Type

sap.ui.model.type.Integer Data Type

The source value for the `Integer` data type (the value given in the model), must be given as a number and is transformed, in one of the following ways, into the type of the bound control property:

- `float`:

  The value is rounded using `Math.floor`.

- `integer`:

  No transformation needed.

- `string`:

  The value is formatted or parsed according to the given output pattern.

For format options that are not specified, default values according to the type and locale settings are used.

- The **Integer** data type supports the validation constraints *maximum* and *minimum*.
- The supported format options are listed in the *API Reference*, see class *sap.ui.core.format.NumberFormat*.

Datatypes.view.xml

```
10    <Label text="Number of Employees"/>
11    <Input id="idNumOfEmployeesInput"
12           value="{ path: '/company/numOfEmployees',
13                    type: 'sap.ui.model.type.Integer',
14                    formatOptions: {groupingEnabled: true, groupingSeparator: '.'},
15                    constraints: {minimum: 1, maximum: 5000000}
16                  }"
17    />
```

```
{
    "company": {
        "numOfEmployees": 104867
    }
}
```

Number of Employees:

104.867

Figure 110: sap.ui.model.type.Integer Data Type

sap.ui.model.type.String Data Type

The source value for the `String` data type (value given in the model) must be given as a string and is transformed, in one of the following ways, into the type of the bound control property:

© Copyright. All rights reserved.

- `string`:

  No transformation needed.

- `integer/float`:

  The string is parsed accordingly.

- boolean:

  `true` or "`X`" is interpreted as "`true`", `false` or " " is interpreted as "`false`".

The `String` data type supports the following validation constraints:

- `maxLength`:

  The maximum length (in characters) that a string of this value may have.

- `minLength`:

  The minimum length (in characters) that a string of this value must have.

- `startsWith`:

  A prefix that any valid value must start with.

- `startsWithIgnoreCase`:

  A prefix that any valid value must start with, ignoring case.

- `endsWith`:

  A suffix that any valid value must end with.

- `endsWithIgnoreCase`:

  A suffix that any valid value must end with, ignoring case.

- `contains`:

  An infix that must be contained in any valid value.

- `equals`:

  The only value that is allowed.

- `search`:

  A regular expression that the value must match.

© Copyright. All rights reserved.

- The **String** data type does not have any format options.
- Among other things, the **String** data type supports the following validation constraints:
  - *maxLength*
    maximum length (in characters) that a string of this value may have
  - *minLength*
    minimum length (in characters) that a string of this value must have

Datatypes.view.xml ×

```
57    <Label text="Department"/>
58    <Input id="idDepartmentInput"
59            value="{ path: '/employee/department',
60                     type: 'sap.ui.model.type.String',
61                     constraints: {minLength : 1, maxLength : 20}
62                   }"
63    />
```

Figure 111: sap.ui.model.type.String Data Type

sap.ui.model.type.Date Data Type

The `Date` data type transforms a source value (given value in the model) into a formatted date string and vice versa.

The set of format options provided by the source property is used to create a `DateFormat` object for conversions between `string` values in the data source (for example, the model) and `Date`. The `DateFormat` object is used to convert from a model string to `Date` before converting the `Date` to a `string` with the primary format options. In a similar way, the source format is also used to format an already parsed external value (for example, user input) into the `string` format expected by the data source. The source format supports the same set of format options as the primary format.

- The **Date** data type represents a date without time.
- The source *pattern* must be defined in Unicode Locale Data Markup Language (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 automatically uses a locale-dependent date pattern.

Datatypes.view.xml ×

```
43    <Label text="Date of Birth" />
44    <Input id="idDOBInput"
45            value="{ path: '/employee/birthday',
46                     type: 'sap.ui.model.type.Date',
47                     formatOptions: { source: {pattern: 'yyyy/MM/dd'}, style: 'medium' }
48                   }"
49    />
```

```
{
    "employee": {
        "birthday": "1965/11/23"
    }
}
```

Date of Birth:
Nov 23, 1965

Figure 112: sap.ui.model.type.Date Data Type

Among other things, the `Date` data type supports the following format options:

- `pattern`:

  A data pattern in Locale Data Markup Language (LDML) format.

- `style`:

© Copyright. All rights reserved.

This can be `short`, `medium`, `long` or `full`. If no pattern is given, a locale-dependent default date pattern of that style is used from the `LocaleData` class.

- `UTC`:

  If true, the date is formatted and parsed as Coordinated Universal Time (UTC) instead of the local time zone.

The supported format options are listed in the API Reference. See class `sap.ui.core.format.DateFormat`.

The `Date` data type supports the following validation constraints:

- `maximum`:

  The largest value allowed for this type (expects a date presented in the source-pattern format).

- `minimum`:

  The smallest value allowed for this type (expects a date presented in the source-pattern format).

Defining Custom Types

The figure, Defining Custom Types, shows how to define custom data types.

- You can define **Custom Types** by inheriting from *sap.ui.model.SimpleType* and implementing the three methods *formatValue(), parseValue() and validateValue()*.

ZipCode.js

```
sap.ui.define([
    "sap/ui/model/SimpleType"
], function(SimpleType) {
    "use strict";
    return SimpleType.extend("sap.training.type.ZipCode", {

        /* formatValue() is called when the value in the model is changed to convert it to the type of the control property it is bound to.
        It may throw a sap.ui.model.FormatException.*/
        formatValue: function(oValue) {
            return oValue;
        },

        /* parseValue() is called, when the user modified a value in the UI and the change is transported back into the model.
        It may throw a sap.ui.model.ParseException in case the value cannot be converted.*/
        parseValue: function(oValue) {
            return oValue;
        },

        /* In case of successful parsing, validateValue() is called to check additional constraints, like minimum or maximum value.
        It throws a sap.ui.model.ValidateException in case constraints are violated.*/
        validateValue: function(oValue) {
            if (!/^(\d{5})?$/.test(oValue)) {
                throw new sap.ui.model.ValidateException("Zip code must have 5 digits!");
            }
        }
    });
});
```

Figure 113: Defining Custom Types

Using Custom Types

The figure, Using Custom Types, shows how custom type use with data binding is similar to simple type use with data binding.

© Copyright. All rights reserved.

Custom Types are used with data binding like the predefined Simple Types.

... .view.xml ×

```
1  <mvc:View xmlns:core="sap.ui.core"   ...   >
2
3      <Label text="Zip code"/>
4
5      <Input id="idZipCodeInput"
6             core:require="{ZipCode: 'sap/training/type/ZipCode'}"
7             value="{ path: '/employee/zipCode',
8                      type: 'sap.training.type.ZipCode'
9                    }"
10     />
```

Make sure the custom type is loaded

Figure 114: Using Custom Types

As of SAPUI5 1.69, 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. Such a declarative approach can help to avoid global variables and allows to reuse certain helper classes without a detour via controller code.

Handling Incorrect User Input

The figure, Handling Incorrect User Input, shows the capabilities available for handling incorrect user input.

- Event handlers can be attached to validate, parse, and format errors.
- To register event handlers, you can use the following functions of *sap.ui.getCore():*
  - **attachFormatError()**
    The *'formatError'* event is fired when a new value for a bound property should have been propagated from the model, but formatting the value failed with an exception.
  - **attachParseError()**
    The *'parseError'* event is fired when a new value for a bound property should have been propagated to the model, but parsing the value failed with an exception.
  - **attachValidationError()**
    The *'validationError'* event is fired when a new value for a bound property should have been propagated to the model, but validating the value failed with an exception.
  - **attachValidationSuccess()**
    The 'validationSuccess' event is fired when a value validation was successfully completed.
- You can also register for the corresponding events directly on the control or any parent control where the event is fired. The event bubbles up to the Core if it is not canceled in the event handler.

Figure 115: Handling Incorrect User Input

Handling Incorrect User Input: Example 1

The method `setValueState()` is used to visualize the state related to the input field's value. Possible value states are `Warning`, `Error`, `Success`, and `None`.

© Copyright. All rights reserved.

Attaching event handlers to the Core

```
... .controller.js ×
10  onInit: function() {
11
12      var oCore = sap.ui.getCore();
13
14      oCore.attachParseError( this.onError );
15
16      oCore.attachValidationError( this.onError );
17
18      oCore.attachValidationSuccess(function(oEvent) {
19          var oElement = oEvent.getParameter("element");
20          oElement.setValueState(sap.ui.core.ValueState.None);
21      });
22  },
23
24  onError: function(oEvent) {
25      var oElement = oEvent.getParameter("element");
26      var oException = oEvent.getParameter("exception");
27
28      oElement.setValueState(sap.ui.core.ValueState.Error);
29      sap.m.MessageToast.show(
30          oException.name + ": " + oException.message);
31  },
```

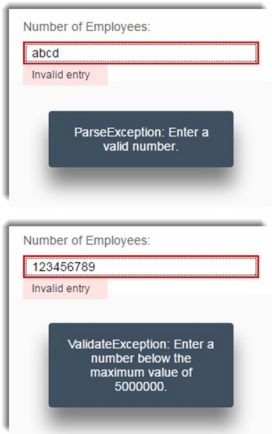


Figure 116: Handling Wrong User Input: Example 1

Handling Incorrect User Input: Example 2

The figure, Handling Incorrect User Input: Example 2 (1), shows how to attach event handlers to a control for handling incorrect user input.

Attaching event handlers directly to a control.

```
... .view.xml ×
10  <Label text="Number of Employees"/>
11  <Input id="idNumOfEmployeesInput"
12          value="{ path: '/company/numOfEmployees',
13                   type: 'sap.ui.model.type.Integer',
14                   formatOptions: {groupingEnabled: true, groupingSeparator: '.'},
15                   constraints: {minimum: 1, maximum: 5000000}
16                 }"
17          parseError="onInputError"
18          validationError="onInputError"
19          validationSuccess="onValidationSuccess"
20  />
21  <Text  id="idMessageText" text="" class="myMessageText"/>
```

```
style.css ×
1 .myMessageText {
2   font-weight: bold;
3   color: red;
4 }
```

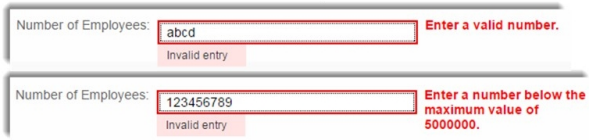


Figure 117: Handling Incorrect User Input: Example 2 (1)

The figure, Handling Incorrect User Input: Example 2 (2), shows the implementation of the `onInputError` and `onValidationSuccess` handlers.

© Copyright. All rights reserved.

Datatypes.controller.js

```
32    onInputError: function (oEvent) {
33
34        oEvent.getParameter("element").setValueState(sap.ui.core.ValueState.Error);
35
36        var oException = oEvent.getParameter("exception");
37        var oText = this.getView().byId("idMessageText");
38        oText.setText(oException.message);
39
40        oEvent.bCancelBubble = true;
41    },
42
43
44    onValidationSuccess: function (oEvent) {
45
46        oEvent.getParameter("element").setValueState(sap.ui.core.ValueState.None);
47
48        var oText = this.getView().byId("idMessageText");
49        oText.setText("");
50    }
```

Figure 118: Handling Incorrect User Input: Example 2 (2)

Validation Using the Message Manager

SAPUI5 provides a central management for messages; the Message Manager. The Message Manager is available globally as a Singleton by calling `sap.ui.getCore().getMessageManager()`.

In the context of message management, there are two different message types. These are as follows:

- UI messages, which are created from input validation and by the application itself.
- Server messages, which are managed by the server and are changed every time the back-end system responds to a request.

To activate the automatic message creation for input validation, several options exist.

You can activate automatic message generation in the component metadata or as a parameter when instantiating a component, in the `sap.ui5` section of the `manifest.json` file, and for individual controls by registering the controls in the Message Manager using the method `registerObject()`. The method `registerObject()` registers a managed object, so that its parse and validation errors are handled by the Message Manager. If you do not set the boolean parameter of `registerObject()` to `true`, the event is canceled without any message.

You can also create UI messages manually and add them to the Message Manager.

The central management for messages uses a Message Model that contains all messages. This Message Model can be retrieved from the Message Manager by calling the `getMessageModel()` method. You can use the Message Model like any other model in the application to bind to the messages to display them to the user.

In the example shown in the figure, Validation Using the Message Manager (1), a connection between the Message Manager and the view is established. As a result, parse and validation errors are handled by the Message Manager for all UI elements on the view. That is, any parse and validation error messages generated based on the user input are passed to the Message Manager, which in turn connects them to the appropriate control that caused the error.

© Copyright. All rights reserved.

- Data type parse and validation functions are managed by SAPUI5, which of itself has no mechanism for reporting error messages back to the UI.
- You can use the **Message Manager** for reporting these error messages back to the user.
- To do it, you must register the corresponding UI elements with the **Message Manager**.

In this example, the entire view is connected to the **Message Manager**.

index.html

```
20  <script>
21      sap.ui.require(["sap/ui/core/mvc/View"], function (View) {
22          // Create view
23          View.create(
27              }).then(function (oView) {
28                  // Insert the view into the DOM
29                  oView.placeAt("content");
30
31                  // Register the view with the message manager
32                  sap.ui.getCore().getMessageManager().registerObject(oView, true);
33              });
34      });
35  </script>
```

Figure 119: Validation Using the Message Manager (1)

A field in error has a red border, as shown in the figure, Validation Using the Message Manager (2). The error message itself is only displayed when the field has focus.

- If you press *Enter* or move the focus to a different UI control, SAPUI5 executes the *parse and validation* function belonging to the corresponding data type.
- Any error messages are picked up by the **Message Manager**, which in turn checks its list of registered objects and then passes the error message back to the correct UI element for display.

Number of Employees: abcd
Enter a valid number.

Number of Employees: 123456789
Enter a number below the maximum value of 5000000.

Message Manager

Main.view.xml

```
7   <Input id="idNumOfEmployeesInput"
8          value="{ path: '/company/numOfEmployees',
9                   type: 'sap.ui.model.type.Integer',
10                  constraints: {minimum: 1, maximum: 5000000} }"/>
11
```

Figure 120: Validation Using the Message Manager (2)

Formatter Functions

The figure, Formatter Functions, explains the concept of formatter functions.

© Copyright. All rights reserved.

- SAPUI5 provides two different options to convert model data to an external format for visual representation: **Data Types** and **Formatter Functions.**
- While **Data Types** provide the possibility to format, parse and validate data, **Formatter Functions** allow one-way conversion only.
- A **Formatter Function** has a single parameter, which is the value which needs to be formatted to an external representation.
- A **Formatter Function** can not only be used for formatting a value, but it can also do type conversion or calculate the results from a given value.

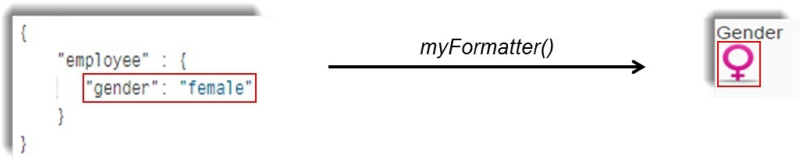


Figure 121: Formatter Functions

Defining Formatter Functions

The figure, Defining Formatter Functions, shows how to define formatter functions.

- **Formatter Functions** can be defined in a separate module:

formatter.js

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

    return {
        salutation: function(sGender) {
            return sGender === "female" ? "Mrs." : "Mr.";
        },

        genderIcon: function(sGender) {
            return sGender === "female" ? "img/Female-icon.png" : "img/Male-icon.png";
        }
    };

});
```

- … or locally in a View controller:

… .controller.js

```
    salutation: function(sGender) {
        return sGender === "female" ? "Mrs." : "Mr.";
    },

    genderIcon: function(sGender) {
        return sGender === "female" ? "img/Female-icon.png" : "img/Male-icon.png";
    }
```

Figure 122: Defining Formatter Functions

Using Locally-defined Formatter Functions

The figure, Using Locally Defined Formatter Functions, shows how to use locally-defined formatter functions.

© Copyright. All rights reserved.

- The name of the **Formatter Function in the View controller** has to be prefixed by a dot ('.'):

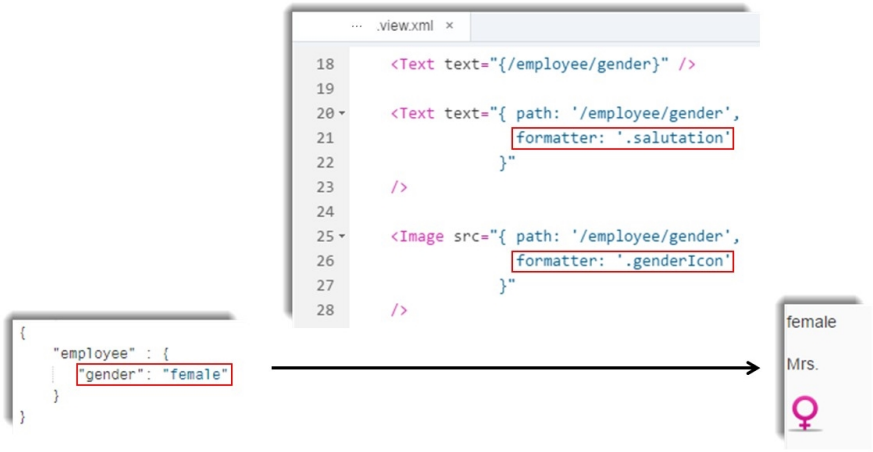


Figure 123: Using Locally Defined Formatter Functions

Using Separately-defined Formatter Functions

The figure, Using Separately-defined Formatter Functions, shows how to use separately-defined formatter functions.

A Formatter Function is specified with the reserved property *formatter* of the binding syntax.

Figure 124: Using Separately-defined Formatter Functions

As of SAPUI5 1.69, 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. Such a declarative approach can help to avoid global variables and allows to reuse certain helper classes without a detour via controller code.

Extended Syntax in XML Views

The figure, Extended Syntax In XML Views, shows how to set the complex binding syntax required when using the extended syntax in XML views.

© Copyright. All rights reserved.

- To use the previously shown *Extended Syntax* in XML views for locally or separately defined Formatter Functions, the application must run with the **complex binding syntax**.
- To use the **complex binding syntax**, set at least one of the following configuration flags in the bootstrap:
  - Set the bootstrap parameter *data-sap-ui-xx-bindingSyntax* to **complex:**
    (**Note**: Beginning with release 1.28, you can omit the "xx" in "data-sap-ui-xx-bindingSyntax".)

```
index.html ×
10   <script id="sap-ui-bootstrap"
11       [...]
12       data-sap-ui-xx-bindingSyntax="complex">
13   </script>
```

  - Set the bootstrap parameter *data-sap-ui-compatVersion* to **1.26** (or higher) or **edge**:

```
index.html ×
10   <script id="sap-ui-bootstrap"
11       [...]
12       data-sap-ui-compatVersion="edge">
13   </script>
```

Figure 125: Extended Syntax in XML Views

Sorting and Filtering for Aggregation Binding

It is important to keep in mind that the used Extended Syntax requires the application to run with the complex binding syntax.

You can define one or multiple filter objects. You can also define one or more sorter objects.

The following filter operators are available:

- `BT` (between)
- `Contains`
- `EndsWith`
- `EQ` (equals)
- `GE` (greater or equals)
- `GT` (greater than)
- `LE` (less or equals)
- `LT` (less than)
- `NE` (not equals)
- `StartsWith`

The figures, Sorting and Filtering for Aggregation Binding (1) — (5), show examples of implementation of sorting and filtering for aggregation binding.

© Copyright. All rights reserved.

The *Extended Syntax* in XML views can be used to provide initial sorting and filtering when specifying the aggregation binding.

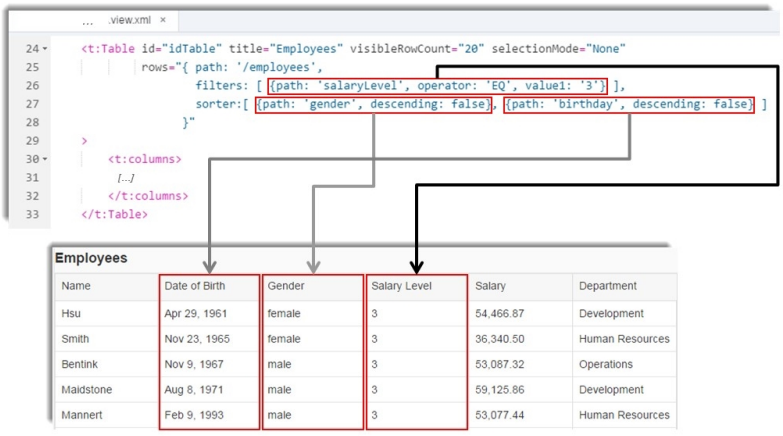


Figure 126: Sorting and Filtering for Aggregation Binding (1)

To sort/filter data manually after the aggregation binding is complete, you can access the corresponding *Binding object* and call the *sort()/filter()* function on it.

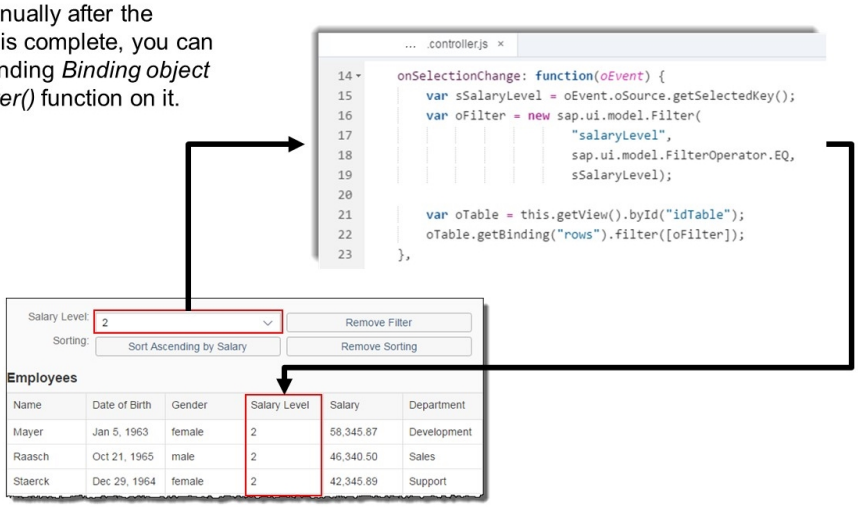


Figure 127: Sorting and Filtering for Aggregation Binding (2)

To sort/filter data manually after the aggregation binding is complete, you can access the corresponding *Binding object* and call the *sort()/filter()* function on it.

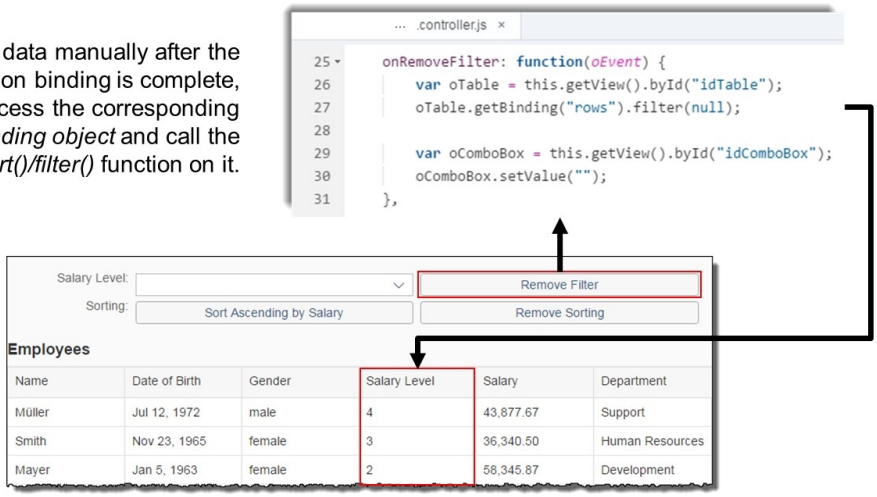


Figure 128: Sorting and Filtering for Aggregation Binding (3)

© Copyright. All rights reserved.

To sort/filter data manually after the aggregation binding is complete, you can access the corresponding *Binding object* and call the *sort()/filter()* function on it:

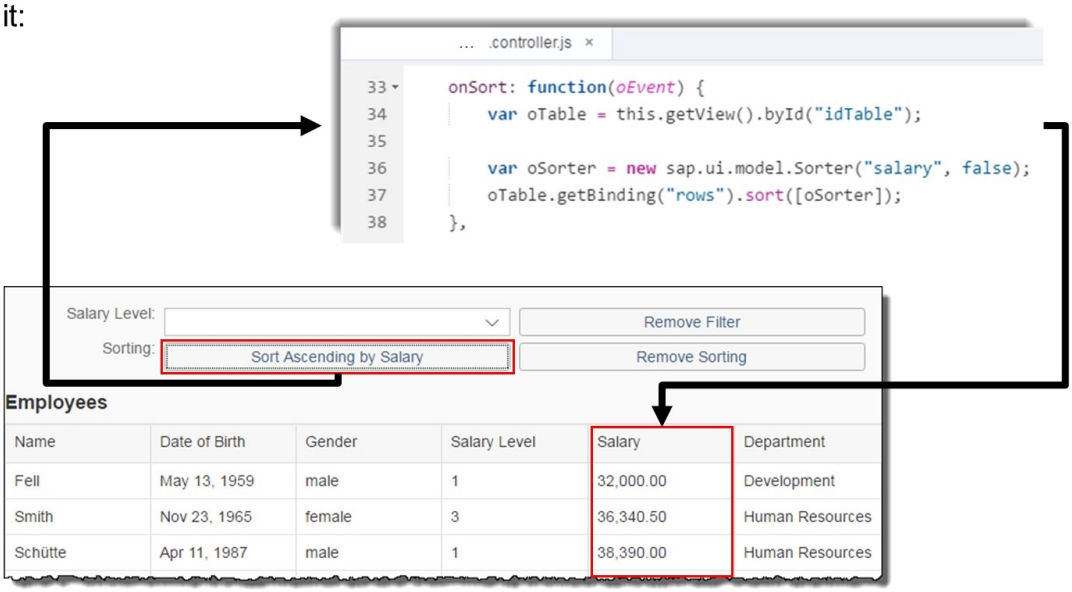


Figure 129: Sorting and Filtering for Aggregation Binding (4)

To sort/filter data manually after the aggregation binding is complete, you can access the corresponding *Binding object* and call the *sort()/filter()* function on it.

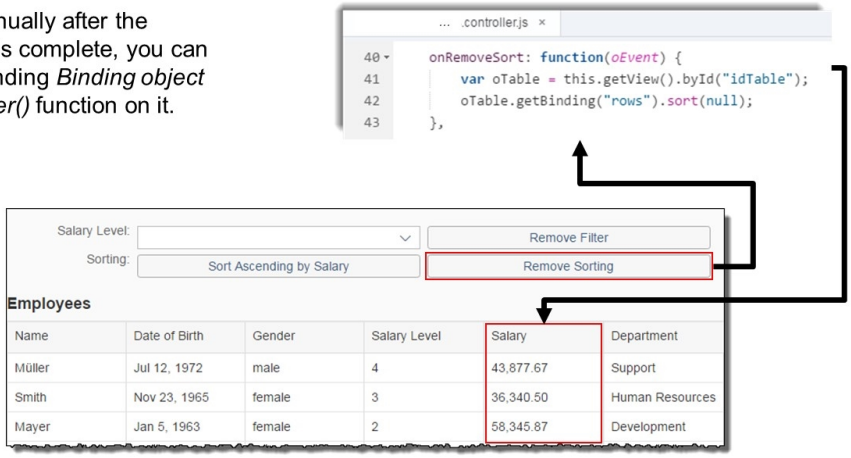


Figure 130: Sorting and Filtering for Aggregation Binding (5)

Calculated Fields For Data Binding

The *text* property of a *Text* UI element, 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, all values of the various parts are joined together as a string with a space separator.

Calculated Fields also work with a relative binding path, when you use them as a template in an aggregation binding.

Keep in mind that the used extended syntax requires the application to run with the complex binding syntax.

© Copyright. All rights reserved.

- **Calculated Fields** enable the binding of multiple properties in different models to a single property of a control. **Calculated Fields** work only from model to view (read-only).
- 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, like a *Type* or a *Formatter Function*.
- There are several syntax options to create multiple bindings for a control.

Figure 131: Calculated Fields For Data Binding

Custom Formatter Functions

The figure, Custom Formatter Functions, shows an example of a custom formatter function.

- SAPUI5 supports the definition of **Custom Formatter Functions.**
- **Custom Formatter Functions** can be defined *globally* (for the entire application), or *locally* in a View controller.

Figure 132: Custom Formatter Functions

Expression Binding

Expression Binding saves the overhead of defining a formatter function and is recommended in cases where the formatter function has a trivial implementation, such as the comparison of values. With Expression Binding, you only need the XML view and no controller logic.

It is recommended that you use formatter functions instead of complex and hard-to-read expressions.

To use Expression Binding, the application must run with the complex binding syntax.

To avoid escaping operators in XML, you can do one of the following:

- Rephrase the expression to make it more readable; for example, using `a > b` instead of `b < a`.

© Copyright. All rights reserved.

- Use a custom formatter function.

The following syntax elements can be used:

- Literals: number, object, string, `null`, `true`, and `false`
- Grouping: `{= (expression) }`
- Unary operators: `!`, `+`, `-`, `typeof`
- Multiplicative operators: `*`, `/`, `%`
- Additive operators: `+`, `-`
- Relational operators: `<`, `>`, `<=`, `>=`
- Strict equality: `===`, `!==`
- Binary logical operators: `&&`, `||`
- Conditional operator: `?`
- Member access with the `.` operator and function calls: You can use members and member methods on standard types like string as follows `{= ${/firstName}.length}`.
- Math: `RegExp` and `encodeURIComponent`.

- **Expression Binding** is an enhancement of the SAPUI5 binding syntax that allows for providing expressions instead of *Custom Formatter Functions.*
- An **Expression Binding** is specified using ***{= expression}*** in an XML view, where *expression* complies with a subset of the JavaScript expression syntax.
- Some characters such as the left angle bracket (<) and ampersand (&) that are used by operators must be escaped in XML.
- Bindings embedded in the *expression* are specified in the following way ***{= ${embedded} }***.

```
... .view.xml ×
4   <!-- Set to visible, if the salaryLevel equals 3 -->
5    <core:Icon src="sap-icon://message-warning" visible="{= ${/employee/salaryLevel} === 3 }"/>
```

Figure 133: Expression Binding (1)

The figure, Expression Binding (2), provides some examples of Expression Binding.

© Copyright. All rights reserved.

Examples:

... .view.xml ×

```
7   <!-- Set to visible, if salaryLevel equals 3 and salary is greater than 40000 (note escaping of &&) -->
8   <core:Icon src="sap-icon://message-warning"
9           visible="{= ${/employee/salaryLevel} === 3 &amp;&amp; ${/employee/salary} > 40000 }"/>
10
11  <!-- Text is 'Mrs.' if the gender is 'female', otherwise 'Mr.' -->
12  <Text text="{= ${/employee/gender} === 'female' ? 'Mrs.' : 'Mr.' }"/>
13
14  <!-- Set to visible, if the department equals 'Human Resources' or the salaryLevel is greater than 3 -->
15  <Button text="Save" press="onSave"
16          visible="{= ${/employee/department} === 'Human Resources' || ${/employee/salaryLevel} > 3 }"/>
17
18  <!-- Set to visible, if the zipCode is valid -->
19  <Text text="{/employee/zipCode}" visible="{= RegExp('^\\d{5}$').test(${/employee/zipCode}) }"/>
20
21  <!-- Text is maximum of 3 values -->
22  <Text text="{= Math.max(${/employees/0/salary}, ${/employees/1/salary}, ${/employees/2/salary}) }"/>
23
24  <!-- Control is enabled only if the department is set -->
25  <Button text="Save" press="onSave" enabled="{= ${/employee/department} !== null }"/>
```

Figure 134: Expression Binding (2)

Device API and Device Model

SAPUI5 applications are usually developed as what's known as components. Components are covered later in the course. In a component, the Device Model is typically created in the component's controller (file `Component.js`) and set for the complete component. The corresponding coding is automatically generated when creating a project.

- The **Device API** *(sap.ui.Device)* is an API that 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 such as, touch event support, orientation change, and so on.
- With a **Device Model,** you can make most properties of the **Device API** available as a *JSON Model*.

... .controller.js ×

```
1  sap.ui.define([
2      "sap/ui/core/mvc/Controller",
3      "sap/ui/model/json/JSONModel",
4      "sap/ui/Device",
5  ], function(Controller, JSONModel, Device) {
6
7      return Controller.extend("   ...   ", {
8
9          onInit: function() {
10             // set device model
11             var oDeviceModel = new JSONModel(Device);
12             oDeviceModel.setDefaultBindingMode("OneWay");
13             this.getView().setModel(oDeviceModel, "device");
14         }
15
16         [...]
17     });
18 });
```

To create a *Device Model,* add a dependency to *sap.ui.Device* to the controller.

Instantiate the *Device Model* in the initialization method of the controller by simply passing the loaded dependency *Device* to the constructor function of the *JSONModel*.

Set the *Binding Mode* to *OneWay* as the *Device Model* is read-only.

Set the *Device Model* as a named model.

Figure 135: Device API and Device Model (1)

The *expanded* property controls the state of the *Panel*. It is bound to the Device Model (model name: `device`), using the path `/system/desktop` to have the *Panel* expanded on desktops and to close it on all other devices. For the property *expandable* of the *Panel*, Expression Binding is used. The *Panel* cannot be expanded on desktops, but it can be expanded on all other devices.

The Device API of SAPUI5 offers more functionality to detect various device-specific settings. Refer to the documentation for more details.

© Copyright. All rights reserved.

The `sap.ui.Device` API detects the device type (phone, tablet, or desktop) based on the user agent and many other properties of the device. Therefore, simply reducing the screen size does not change the device type. To test the device-specific features of an app, you can enable device emulation in 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 may need to reload your page to see the results.

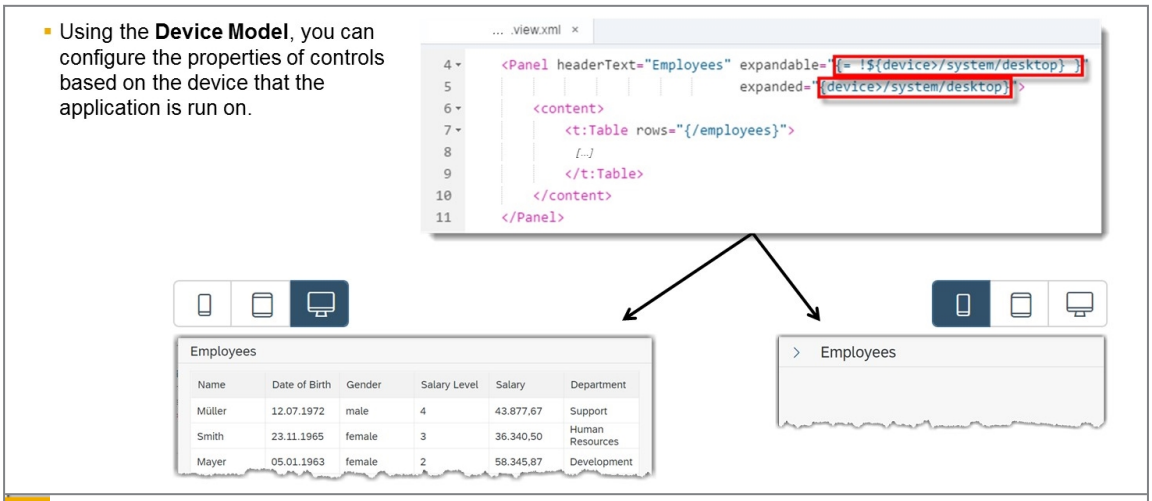


Figure 136: Device API and Device Model (2)

LESSON SUMMARY

You should now be able to:

- Explore models and data binding

© Copyright. All rights reserved.

Learning Assessment

1. Which of the following model types are supported by SAPUI5?

   Choose the correct answers.

   - ☐ A XML
   - ☐ B JSON
   - ☐ C OData
   - ☐ D SOAP

2. Which of the following are the binding types supported in SAPUI5?

   Choose the correct answers.

   - ☐ A Property binding
   - ☐ B Default binding
   - ☐ C Aggregation binding
   - ☐ D Elementary binding
   - ☐ E Element binding

3. Which enumeration defines the various binding modes provided by SAP?

   Choose the correct answer.

   - ☐ A `sap.ui.model.BindingMode`
   - ☐ B `sap.ui.binding.BindingMode`
   - ☐ C `sap.ui.bindingmode.Mode`
   - ☐ D `sap.ui.model.binding.Mode`

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which of the following model types are supported by SAPUI5?

   Choose the correct answers.

   [X] A XML

   [X] B JSON

   [X] C OData

   [ ] D SOAP

   Correct. XML, JSON, and OData are the supported model types when it comes to data handling. There is also a model type for handling resources such as i18n data, but SOAP is not a model type supported by SAPUI5. Web services are handled using an XML model.

2. Which of the following are the binding types supported in SAPUI5?

   Choose the correct answers.

   [X] A Property binding

   [ ] B Default binding

   [X] C Aggregation binding

   [ ] D Elementary binding

   [X] E Element binding

   Correct. Property binding, aggregation binding, and element binding are the supported binding types in SAPUI5. These binding types are used to bind the attributes of UI controls and elements to aspects of a data model.

© Copyright. All rights reserved.

3. Which enumeration defines the various binding modes provided by SAP?

   Choose the correct answer.

   - [x] A `sap.ui.model.BindingMode`
   - [ ] B `sap.ui.binding.BindingMode`
   - [ ] C `sap.ui.bindingmode.Mode`
   - [ ] D `sap.ui.model.binding.Mode`

   Correct. The enumeration `sap.ui.model.BindingMode` contains the binding type definitions.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 10 Localization

Lesson 1

Implementing Localization 157

UNIT OBJECTIVES

- Implement localization

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 10
Lesson 1

Implementing Localization

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement localization

Localization Overview

The figure, Localization Overview, provides information on resource bundle files and their role in localization.

- The framework concepts for text localization in SAPUI5 are aligned with the general concepts of the Java platform.
- The language-dependent texts are stored in so-called **resource bundle files**.
- A **resource bundle file** 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.

Figure 137: Localization Overview

Resource Bundles

Language suffixes are formed according to the older Java Development Kit (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 a more suitable language cannot be found.

© Copyright. All rights reserved.

- A **resource bundle file** has the file extension .properties.
- A **Resource Bundle** is a collection of *resource bundle files*: All files are named with the same base name (prefix identifying the *Resource Bundle*), and an optional suffix that identifies the language contained in each file.

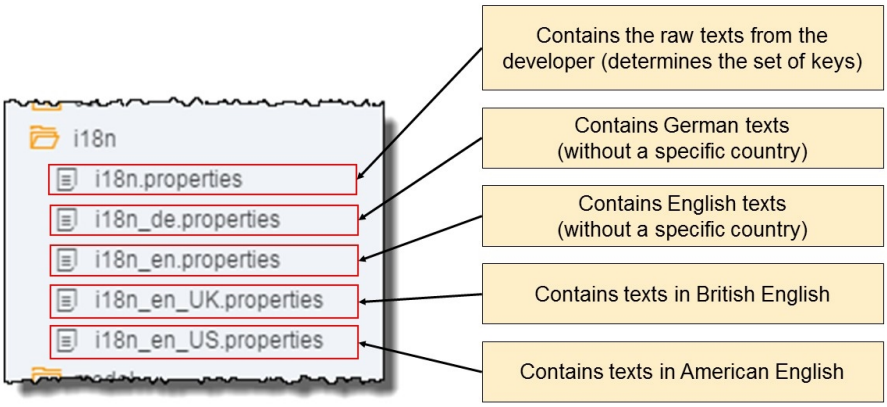


Figure 138: Resource Bundles

Current Language Code and Locale

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 is a known shortcoming in IE. To circumvent this, an additional server request is required, where IE provides the corresponding setting in the `Accept-Language` header. This server request, however, requires a back-end component. SAPUI5 must be able to run without a server component, and therefore, the server request is not implemented yet.

- When a localized text is needed, the application uses the SAPUI5 APIs to load the .properties file that matches **the Current Language Code/Locale** best.
- The **Current Language Code/Locale** is determined from the following sources of information. The sources are ordered increasingly by priority and the last available wins:
  - Hard-coded UI5 default locale (‘en’)
  - Potentially configured browser language (*window.navigator.browserLanguage*); for Internet Explorer this is the language of the Operating System
  - Potentially configured user language (*window.navigator.userLanguage*); for Internet Explorer this is the language in the Region settings
  - General language information from the browser (*window.navigator.language*)
  - Android: Language contained in the user agent string (*window.navigator.userAgent*)
  - Locale configured in the application coding (see documentation for class *sap.ui.core.Configuration*)
  - Locale configured using URL parameters (for example, *sap-ui-language*)

Figure 139: Current Language Code/Locale

Use of Localized Texts in Applications

The figure, Use of Localized Texts in Applications, introduces the options for the use of localized texts in applications.

© Copyright. All rights reserved.

- SAPUI5 provides two options to use localized texts in applications:
  - **sap/base/i18n/ResourceBundle**
    This module contains APIs to load a resource bundle file from a given URL and for a given locale.
  - **Data binding**
    You use the *ResourceModel* to bind texts for control properties to language-dependent resource bundle properties. The *ResourceModel* is a wrapper for resource bundles that exposes the localized texts as a model for data binding.

Figure 140: Use of Localized Texts in Applications

Using sap/base/i18n/ResourceBundle

If you need localized texts within your application, you can use the JavaScript module `sap/base/i18n/ResourceBundle` to load the locale-specific file from the server and access the texts of it.

To create an instance of `sap/base/i18n/ResourceBundle`, call `ResourceBundle.create()`. If no locale is passed, the default locale is "en" if the SAPUI5 framework is not available. Otherwise the locale is taken from the SAPUI5 configuration.

The `create()` method returns a new resource bundle, or a `Promise` on that bundle in asynchronous cases.

With the `getText()` method of the resource bundle, a locale-specific string value for a given key will be returned.

With the given locale, the resource bundle requests the locale-specific properties file (for example, “i18n_fr_FR.properties"). If no file is found for the requested locale, or if the file does not contain a text for the given key, a sequence of fall back locales is tried one by one. First, if the locale contains a region information (fr_FR), then the locale without the region is tried (fr). If that also can't be found, or doesn't contain the requested text, the English file is used (en - assuming that most development projects contain at least English texts). If that also fails, the file without locale (base URL of the bundle) is tried.

If none of the requested files can be found or none of them contains a text for the given key, then the key itself is returned as text.

© Copyright. All rights reserved.

Note:
From SAPUI5 1.77, a list of supported locales can be specified to restrict the number of locales for which resource bundles may be requested. This list is passed using the parameter `supportedLocales`. If the list is empty or not specified, all locales are supported.

The parameter `supportedLocales` works like a filter. Each locale in the fallback chain is checked against this list and only applied if it is contained there. This enables an application to only request particular resource bundles with language-specific texts. As requests, that were previously needed to check for the availability of language files, can now be avoided, this helps to improve the runtime performance of applications.

In addition, a fallback locale can be specified from SAPUI5 1.77 to be used after all locales derived from the given locale have been tried, but before the 'raw' bundle is used. English used to be the default locale, but now any fallback locale can be specified using the parameter `fallbackLocale`. If not specified, the fallback locale defaults to `en`.

For more information, see `ResourceBundle` in the API Reference.

The code snippet shows the use of the `ResourceBundle.create()` function to return a `Promise` which resolves with a `sap/base/i18n/ResourceBundle`. The `getText()` method is used to access a text in the loaded bundle by means of its key.

- *ResourceBundle.create()* returns an instance of *sap/base/i18n/ResourceBundle* (or a *Promise* on the bundle in asynchronous case) using the given *URL* and *locale*.
- If no *locale* is passed, the *locale* is taken from the SAPUI5 configuration.

```
// "ResourceBundle" required from module "sap/base/i18n/ResourceBundle"
ResourceBundle.create({
    url: "i18n/i18n.properties",
    async: true
}).then(function (oResourceBundle) {

    var oDateFormat = DateFormat.getDateInstance({
        style: "long"
    });

    var sDate = oDateFormat.format(new Date());

    var sDateText = oResourceBundle.getText("mainTextDate", [sDate]);
    oView.byId("idText2").setText(sDateText);

});
```

Load the best matching resource bundle file for the given *URL* and *locale*.

Access the text using the key and pass the value for the placeholder ({0}) using an array.

Figure 141: Using sap/base/i18n/ResourceBundle

Data Binding

You can use data binding to access localized texts. To do so, instantiate a `Resource Model` 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 given (parameter `bundleLocale`), the current session locale of SAPUI5 will be used (recommended).

© Copyright. All rights reserved.

Using the parameter `async` you can define that the language bundle is to be loaded asynchronously (recommended).

> Note:
> The current data binding implementation does not allow you 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.

After the instance has been created, you have a model containing the resource bundle texts as data. In contrast to other models, binding paths for a `ResourceModel` must not start with a slash. They are absolute by default and there's no further structure. Each key in the underlying resource bundle is a valid binding path.

For a complete overview of available methods and parameters, see `ResourceModel` in the API Reference in the Demo Kit.

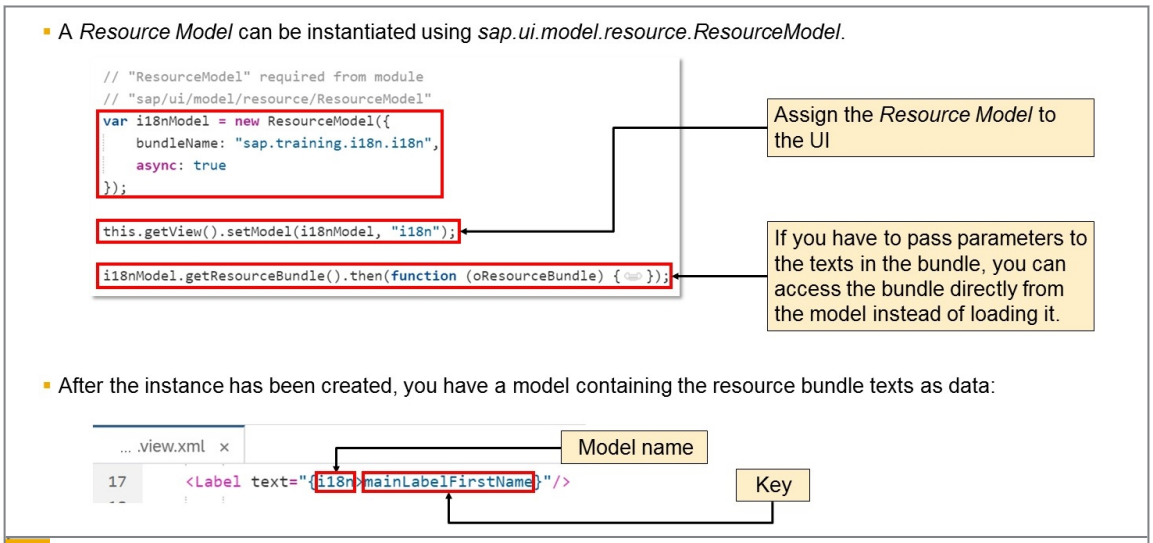


Figure 142: Data Binding

LESSON SUMMARY

You should now be able to:

- Implement localization

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which parameter do you need to provide to the `ResourceModel()` to specify the SAPUI5 module name of the `.properties` file?

   Choose the correct answer.

   - [ ] A *bundleName*
   - [ ] B *resourceBundleName*
   - [ ] C *resourceName*
   - [ ] D *bundlePath*

2. Which of the following are the two options provided by SAPUI5 for working with i18n texts?

   Choose the correct answers.

   - [ ] A Module `sap/base/i18n/ResourceBundle`
   - [ ] B Aggregation Binding
   - [ ] C Data Binding
   - [ ] D Resource Consumption

3. Assuming the input language in a SAPUI5 application is `de_CH`. What is the locale fallback chain used by SAPUI5 to load the suitable resource bundle file (where "" stands for the raw file)?

   Choose the correct answer.

   - [ ] A "de_CH" → "de" → "en" → " "
   - [ ] B "de_CH" → "de" → " " → "en"
   - [ ] C "de_CH" → "de" → " "

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which parameter do you need to provide to the `ResourceModel()` to specify the SAPUI5 module name of the `.properties` file?

   Choose the correct answer.

   - [X] A *bundleName*
   - [ ] B *resourceBundleName*
   - [ ] C *resourceName*
   - [ ] D *bundlePath*

   Correct. The parameter *bundleName* contains the SAPUI5 module name of the `.properties` file. This name is resolved to a path similar to the paths of normal SAPUI5 modules, and `.properties` is then appended.

2. Which of the following are the two options provided by SAPUI5 for working with i18n texts?

   Choose the correct answers.

   - [X] A Module `sap/base/i18n/ResourceBundle`
   - [ ] B Aggregation Binding
   - [X] C Data Binding
   - [ ] D Resource Consumption

   Correct. The module `sap/base/i18n/ResourceBundle` contains APIs to load a resource bundle file from a given URL. Inside a view, the developer can bind texts for control properties to language-dependent properties of the underlying resource model.

© Copyright. All rights reserved.

3. Assuming the input language in a SAPUI5 application is `de_CH`. What is the locale fallback chain used by SAPUI5 to load the suitable resource bundle file (where "" stands for the raw file)?

   Choose the correct answer.

   [X] A "de_CH" → "de" → "en" → " "

   [ ] B "de_CH" → "de" → " " → "en"

   [ ] C "de_CH" → "de" → " "

   Correct. If the file `i18n_de_CH.properties` is not found or if this file does not contain a text for the given key, then the locale without the region is tried ("de"). If that also can't be found, or doesn't contain the requested text, the English file is used ("en"). If that also fails, the file without locale is tried ("").

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 11 OData Services

Lesson 1

Exploring OData Services 169

UNIT OBJECTIVES

- Understand the Open Data Protocol

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 11
Lesson 1

Exploring OData Services

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand the Open Data Protocol

OData at a Glance

OData standardizes the way that data is accessed. Instead of learning new APIs to access data specific to each application's Web service, OData provides a common way to query and update data.

OData is consistent with the way the Web works. It makes a deep commitment to URIs for resource identification and commits to an HTTP-based, uniform interface for interacting with those resources, just like the Web. This commitment to core Web principles allows OData to enable a new level of data integration and interoperability across a broad range of clients, servers, services, and tools.

OData builds on the conventions of HTTP, popularized by AtomPub, which have simplified the process of sharing data, content, and information across independently developed systems. To simplify integration with HTML and JavaScript clients, OData defines an optional JSON representation of the protocol that complements the XML-based AtomPub format.

OData has been around for a number of years and more and more providers of data services are appearing on the Web as time passes.

- The *"Open Data Protocol"* – abbreviated OData – enables HTTP-based data services, which allow resources to be published and edited by Web clients using simple HTTP messages.
- The resources are defined in an abstract data model that is based on the **Entity Data Model**.
- OData uses URIs for resource identification.
- The HTTP-based, uniform interface for interacting with resources allows create, read, update and delete operations in a standardized way.
- OData builds on well known data formats: XML-based AtomPub and JSON.
- Hypermedia design pattern: Relationships between resources are hyperlinks between resources in the **OData** protocol.
- **OData** is published by Microsoft under the *Open Specification Promise,* so that anyone that wants to can build servers, clients, and tools without royalties or restrictions (see *http://www.odata.org*).

Figure 143: OData at a Glance

© Copyright. All rights reserved.

Metadata for OData Services

Each OData service is represented by a URI, called the Service Root URI. A URI is a uniform resource identifier, which is a string of characters used to identify a resource. This type of identification enables interaction with representations of the resource across a network using specific protocols like OData.

There are two types of documents associated with each OData service; the Service Document and the Service Metadata Document. Both documents can be retrieved from a server using an HTTP GET call.

OData services may provide two types of metadata documents to describe themselves:

- **The Service Metadata Document**
  - The *Service Metadata Document* describes the data model, that is, the structure and organization of all the resources exposed as HTTP endpoints by the service.
  - The *Service Metadata Document* provides a full description of the service's interface.
- **The Service Document**
  - The *Service Document* declares the resources accessible from the Service Root URI.

Figure 144: Metadata for OData Services

Service Metadata Document

The Service Metadata Document is available at the service metadata URI. The service metadata URI can be constructed easily by appending the string `$metadata` to the Service Root URI.

The purpose of the Service Metadata Document is to enable clients to discover the shape of an OData service, the structure of its resources, the known links between resources, and the service operations exposed.

A Service Metadata Document describes the Entity Data Model (EDM), for a given service. The EDM is the underlying abstract data model used by OData services to formalize the description of the resources it exposes.

An OData Service Metadata Document describes its data in EDM terms using an XML language for describing models called the Conceptual Schema Definition Language (CSDL).

© Copyright. All rights reserved.

- The **Service Metadata Document** describes the Entity Data Model (EDM) for a given service.

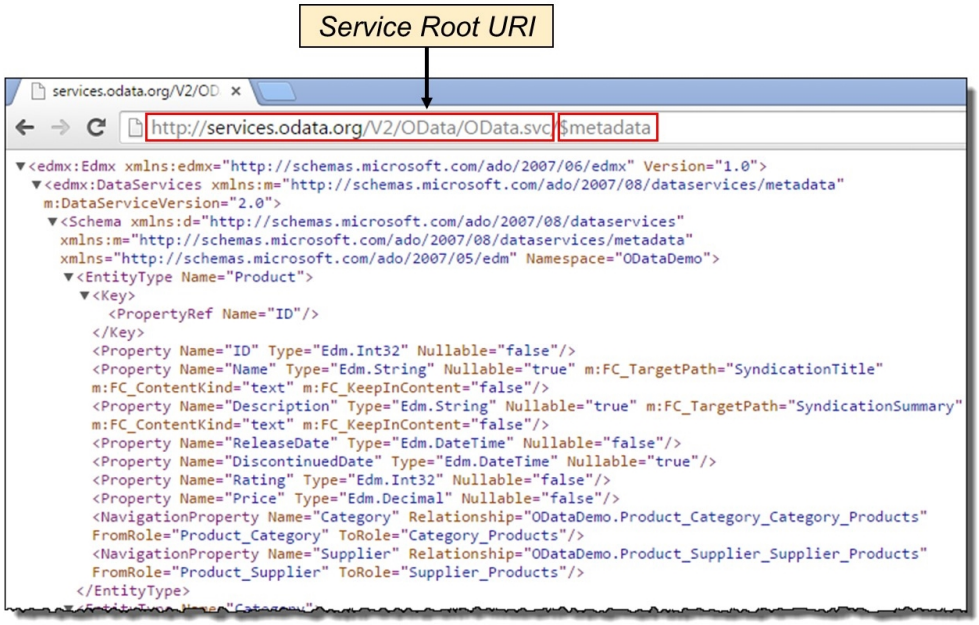


Figure 145: Service Metadata Document

OData Service Entity Data Model (EDM): Example

The figure, Example of an OData Service EDM, shows an example of an OData Service EDM.

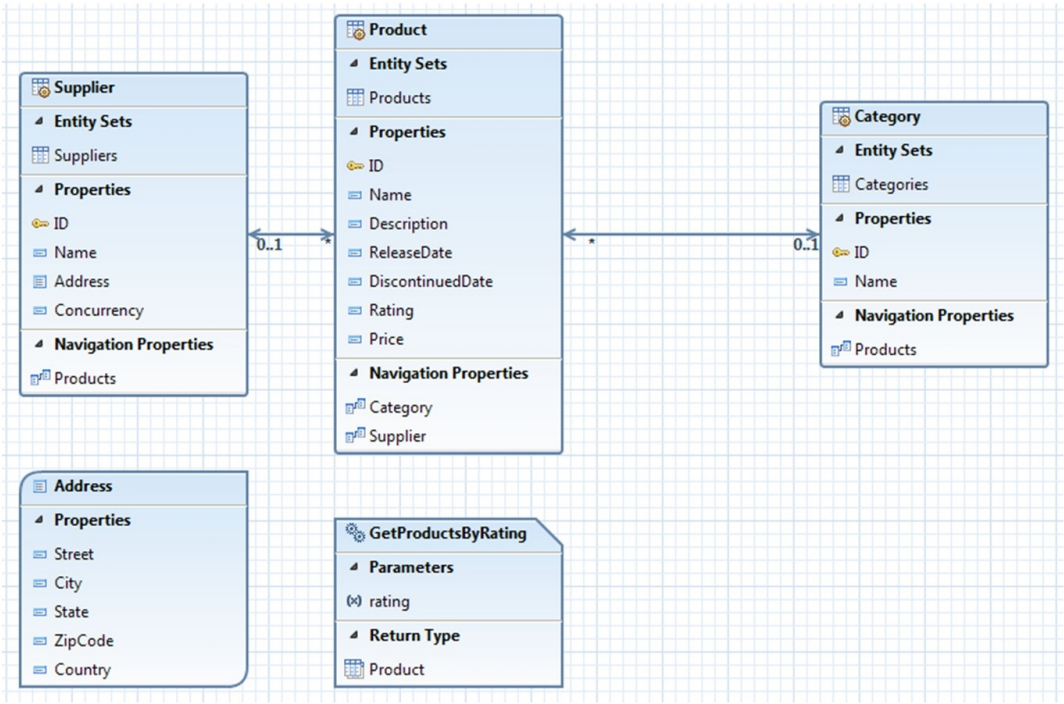


Figure 146: Example of an OData Service EDM

Entity Types and Entity Sets

Entity types are richly structured records with a key. The structure of an entity type is provided by its properties.

© Copyright. All rights reserved.

Entities are instances of entity types.

Entities are grouped in entity sets. For example, the Products entity set is a set of product instances. All entities of one entity set are of the same entity type.

An entity key is formed from a subset of the properties of the entity type. The key is a fundamental concept used to uniquely identify and persist entity instances. The key of an entity is defined by one or many properties. Entities can be addressed individually by means of key values, or as collections by means of query options.

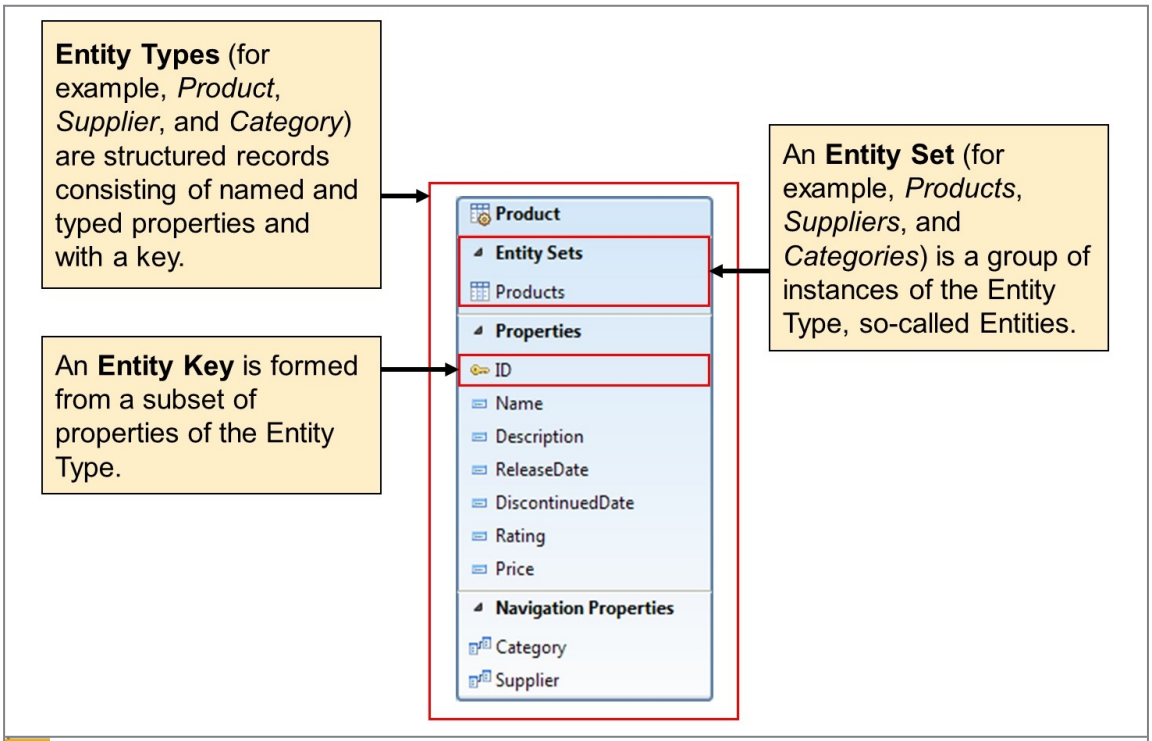


Figure 147: Entity Types and Entity Sets

Complex Types and Associations

A complex type is a structural type similar to an entity type. The main difference is that complex types have no identity and cannot support associations. For these reasons, complex types instances only exist as properties of entity types or other complex types. A complex type can be reused inside an OData service from several entity types.

An association is a named relationship between two entity types. Every association includes two association ends, which specify the entity types that are related, and the cardinality rules for each end of the association.

Entity types may include one or more navigation properties. The navigation property derives its name from the fact that it allows navigation from one entity to others. A navigation property is tied to an association and allows navigation from one end of an association, the entity type that declares the navigation property, to the other related end, which can be anything from 0 or more related entities. The *Supplier* entity type has exactly one navigation property named *Products*. The navigation property makes it possible to navigate from a *Supplier* instance to all the *Products* delivered by the supplier.

© Copyright. All rights reserved.

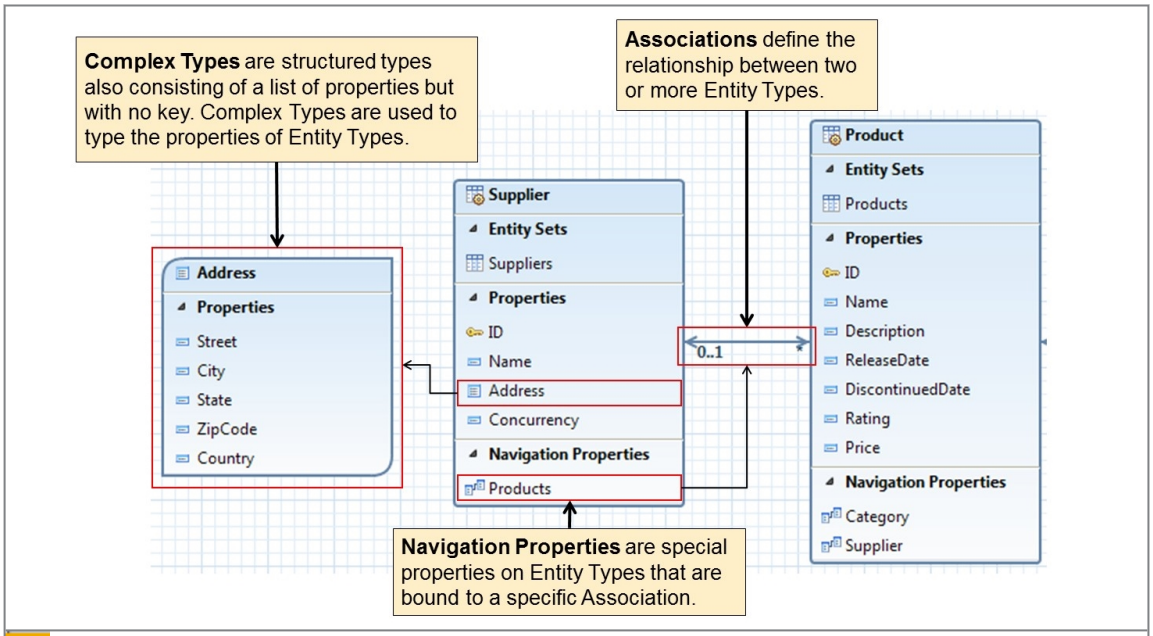


Figure 148: Complex Types and Associations

Service Operations

OData services can expose Service Operations, which are simple functions whose semantics are defined by the author of the function.

Service operations can be defined to take parameters, which are simple scalar values of one of the EDM primitive types.

Service operations can be defined to return a single primitive, single complex type, collection of primitives, collection of complex types, a single entity, an entity set, or void.

Service operations expose custom behaviors that do not map to the uniform interface.

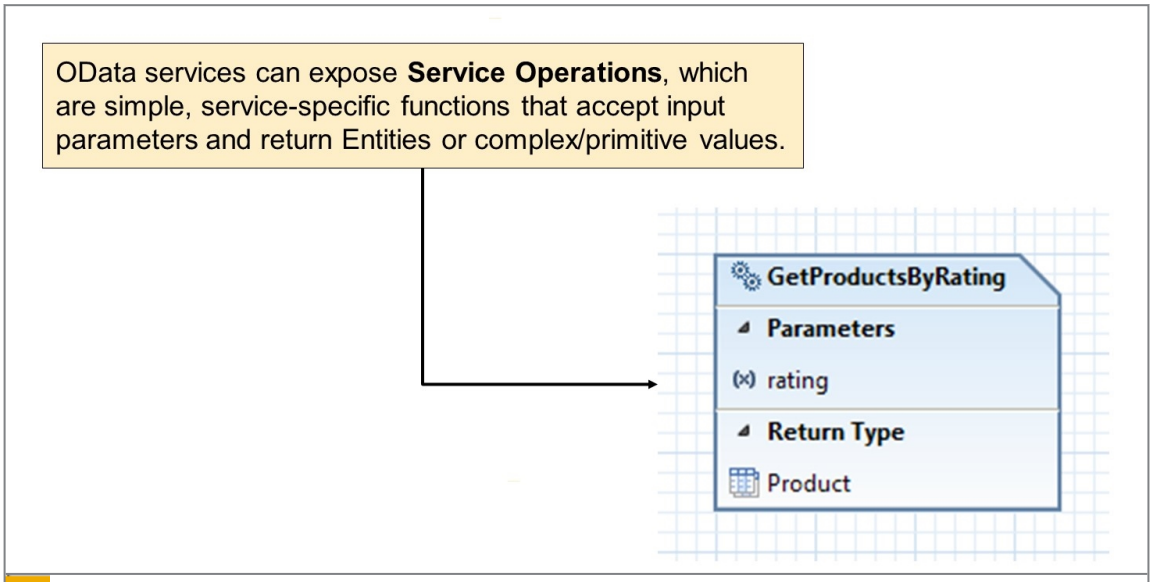


Figure 149: Service Operations

Entity Type Definition in the Service Metadata Document

This section covers the *Supplier*, and how this entity type definition looks in the Service Metadata Document.

© Copyright. All rights reserved.

All the properties of the *Supplier* are listed as XML elements named *Property*.

All key properties can be found as *PropertyRef* elements in the *Key* element. The *Supplier* is identified by one key field, the *ID*.

Similar to properties, the navigation properties are specified in a separate element called *NavigationProperty*. It defines the navigation property name, and binds the property to the corresponding *Association*.

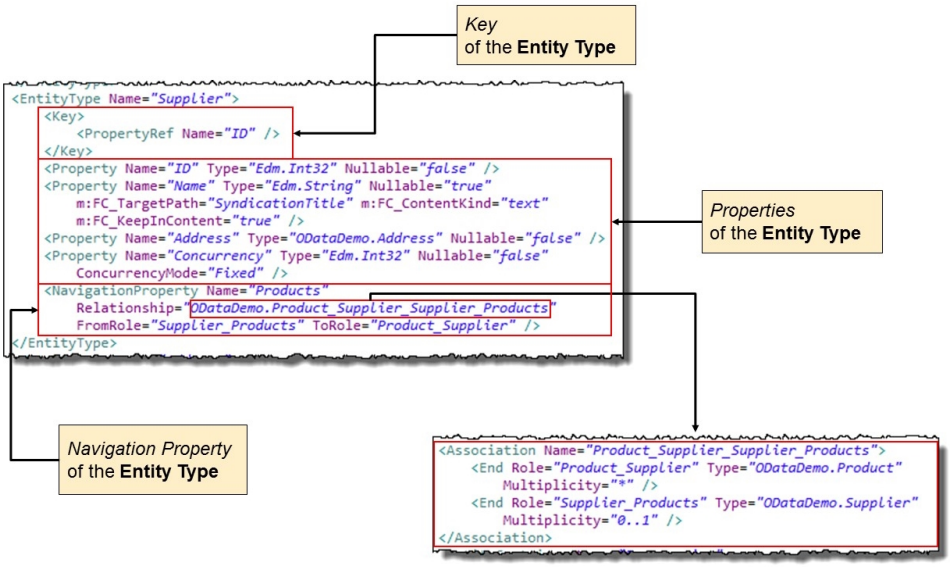


Figure 150: Entity Type Definition in the Service Metadata Document

Primitive Data Types

OData is strongly typed.

In the figure, Primitive Data Types, you can see how type definition looks in the Service Metadata Document. The type of the *Supplier*'s property *ID* is `Edm.Int32`, and the property *Name* is of type `Edm.String`. The property *Address* is typed with `ODataDemo.Address`, which is the fully-qualified name of the complex type `Address`. The namespace of this OData service is `ODataDemo`. The complex type `Address` has the properties *Street*, *City*, and so on. It is defined by a separate XML element called Complex Type.

© Copyright. All rights reserved.

- The EDM defines **Primitive Data Types**, such as *Edm.String, Edm.Int32*, and *Edm.Boolean*, which are the building blocks for structural types *like Entity Types* and *Complex Types*.

```
<ComplexType Name="Address">
    <Property Name="Street" Type="Edm.String" Nullable="true" />
    <Property Name="City" Type="Edm.String" Nullable="true" />
    <Property Name="State" Type="Edm.String" Nullable="true" />
    <Property Name="ZipCode" Type="Edm.String" Nullable="true" />
    <Property Name="Country" Type="Edm.String" Nullable="true" />
</ComplexType>
```

```
<EntityType Name="Supplier">
    <Key>
        <PropertyRef Name="ID" />
    </Key>
    <Property Name="ID" Type="Edm.Int32" Nullable="false" />
    <Property Name="Name" Type="Edm.String" Nullable="true"
        m:FC_TargetPath="SyndicationTitle" m:FC_ContentKind="text"
        m:FC_KeepInContent="true" />
    <Property Name="Address" Type="ODataDemo.Address" Nullable="false" />
    <Property Name="Concurrency" Type="Edm.Int32" Nullable="false"
        ConcurrencyMode="Fixed" />
    <NavigationProperty Name="Products"
        Relationship="ODataDemo.Product_Supplier_Supplier_Products"
        FromRole="Supplier_Products" ToRole="Product_Supplier" />
</EntityType>
```

Figure 151: Primitive Data Types

Entity Container

Frequently, simple entity models have one entity set per entity type. Other entity models may expose multiple entity sets per entity type. For instance, an entity model may have the following entity sets:

```
<EntitySet Name="Products" EntityType="ODataDemo.Product"/>
```

```
<EntitySet Name="DiscontinuedProducts" EntityType="ODataDemo.Product"/>
```

In this case, the *Products* entity set could expose products that have not been discontinued and the *DiscontinuedProducts* entity set could expose products that have been discontinued.

An Association Set disambiguates the entity set used by an association end. In the preceding example, an `ODataDemo.Product` could be exposed through the *Products* entity set or the *DiscontinuedProducts* entity set. An association set is required to clarify which entity set is used for any association end that refers to `ODataDemo.Product`.

In the OData Meta Model, service operations are called Function Imports. Service operations/ function imports are defined inside the entity container. Our example depicts the function import `GetProductsByRating`, which returns a collection of *Product* entities, and which has one input parameter named *rating*. On the server, the function import calculates the list of all products that exactly match the value of the *rating* parameter. The function must be called using the HTTP method GET, so it must not modify resources on the server.

© Copyright. All rights reserved.

The **Entity Container** aggregates *Entity Sets, Association Sets*, and *Function Imports.*

```
<EntityContainer Name="DemoService"
    m:IsDefaultEntityContainer="true">
    <EntitySet Name="Products" EntityType="ODataDemo.Product" />
    <EntitySet Name="Categories" EntityType="ODataDemo.Category" />
    <EntitySet Name="Suppliers" EntityType="ODataDemo.Supplier" />
    <AssociationSet Name="Products_Category_Categories"
        Association="ODataDemo.Product_Category_Category_Products">
        <End Role="Product_Category" EntitySet="Products" />
        <End Role="Category_Products" EntitySet="Categories" />
    </AssociationSet>
    <AssociationSet Name="Products_Supplier_Suppliers"
        Association="ODataDemo.Product_Supplier_Supplier_Products">
        <End Role="Product_Supplier" EntitySet="Products" />
        <End Role="Supplier_Products" EntitySet="Suppliers" />
    </AssociationSet>
    <FunctionImport Name="GetProductsByRating" EntitySet="Products"
        ReturnType="Collection(ODataDemo.Product)" m:HttpMethod="GET">
        <Parameter Name="rating" Type="Edm.Int32" Mode="In" />
    </FunctionImport>
</EntityContainer>
```

The *EntityContainer* defines all the *Entity Sets* belonging to one service.

An *Association Set* clarifies which *Entity Set* is used by an *Association* end.

A *Function Import* is used to expose functions that are defined in a data store.

Figure 152: Entity Container (1)

Service Document

If you enter the URL shown in the figure, Service Document, in a browser, you send an HTTP GET call to the server and the server responds with the Service Document in JSON format. The Service Document contains three entity sets: *Products*, *Categories*, and *Suppliers*.

- The **Service Document** is available at the *Service Root URI* and may be formatted in Atom or JSON.
- The **Service Document** lists all of the top-level Entity Sets. Therefore, clients can discover each Entity Set and identify its address.

Service Root URI

services.odata.org/V2/OD

http://services.odata.org/V2/OData/OData.svc/?$format=json

```
{
  - d: {
      - EntitySets: [
            "Products",
            "Categories",
            "Suppliers"
        ]
    }
}
```

Figure 153: Service Document

CRUD Operations

The figure, CRUD Operations, shows how OData can be used to perform Create, Read, Update, and Delete (CRUD) operations using HTTP methods.

© Copyright. All rights reserved.

- OData performs create, read, update, and delete operations on addressable resources by using existing HTTP methods.

Operation on Resource	HTTP Method
Create	POST
Read	GET
Update	PUT
Delete	DELETE

Figure 154: CRUD Operations

Exploring an OData Service

Because OData uses the HTTP protocol, any web browser can be used to start exploring OData. As of today, OData supports the Atom and the JSON formats.

In this section, we focus on the JSON format. To explore the OData protocol, we recommend that you use the Firefox or Chrome browser temporarily as your standard browser and install the JSONView plugin.

We start by reading the *Products* entity set from the OData sample service. To it, the name of the entity set, which can be found in the Service Document, must be appended to the Service Root URI.

In the figure, Exploring an OData Service (1), you can see the metadata section of the first product entity. This section contains an URI identifying this first entity, and the entity type. The section is followed by the properties and their values and the navigation properties, *Category* and *Supplier*.

We now follow the URI in the metadata section to read just the first Entity in the *Products* Entity Set, which has the *ID* 0.

© Copyright. All rights reserved.

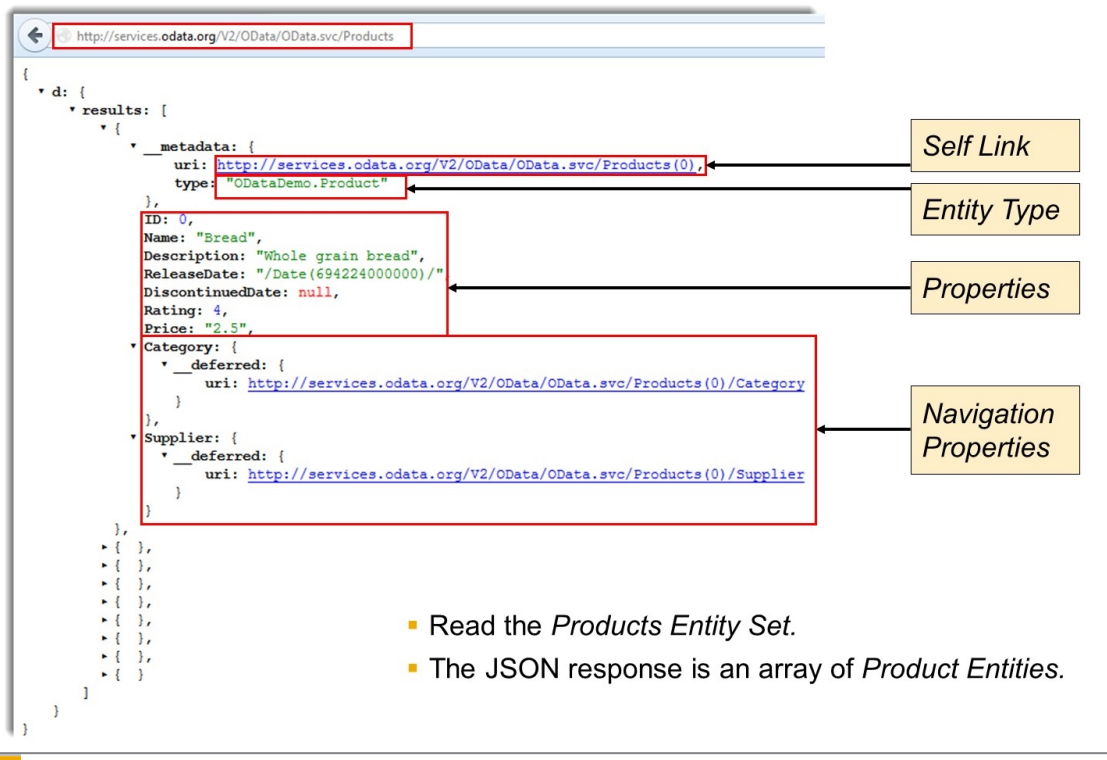


Figure 155: Exploring an OData Service (1)

If you compare the responses in the figures Exploring an OData Service (1) and (2), you observe that the entity payload is identical, but the entity set contains the surrounding array in addition.

At the end, you can see the two navigation properties pointing to the *Category* and to the *Supplier* of this product. All navigation properties are URIs. As a consumer of an OData service, you can just follow these URIs and navigate through the web of data. Let us do so and issue a GET call at the *Category* URI.

© Copyright. All rights reserved.

http://services.odata.org/V2/OData/OData.svc/Products(0)


```
{
  d: {
    __metadata: {
      uri: http://services.odata.org/V2/OData/OData.svc/Products(0),
      type: "ODataDemo.Product"
    },
    ID: 0,
    Name: "Bread",
    Description: "Whole grain bread",
    ReleaseDate: "/Date(694224000000)/",
    DiscontinuedDate: null,
    Rating: 4,
    Price: "2.5",
    Category: {
      __deferred: {
        uri: http://services.odata.org/V2/OData/OData.svc/Products(0)/Category
      }
    },
    Supplier: {
      __deferred: {
        uri: http://services.odata.org/V2/OData/OData.svc/Products(0)/Supplier
      }
    }
  }
}
```


- Read *Product Entity* with *ID* **0**.
- The JSON response is a *Product Entity*.

Figure 156: Exploring an OData Service (2)

The figure, Exploring an OData Service (3), shows the result. You can see the *Category* entity, which is related to the *Product* entity with *ID* 0.

The navigation property *URI* points to all products that are associated with this *Category* that have the *ID* 0. The resource addressed by this URI is an entity set. This can be derived from the cardinality of the association between the entity type *Category* and the entity type *Product* declared in the Service Metadata Document.

- Read Category of *Product Entity* with *ID* **0**.
- JSON response is a *Category Entity*.

http://services.odata.org/V2/OData/OData.svc/Products(0)/Category


```
{
  d: {
    __metadata: {
      uri: http://services.odata.org/V2/OData/OData.svc/Categories(0),
      type: "ODataDemo.Category"
    },
    ID: 0,
    Name: "Food",
    Products: {
      __deferred: {
        uri: http://services.odata.org/V2/OData/OData.svc/Categories(0)/Products
      }
    }
  }
}
```


Figure 157: Exploring an OData Service (3)

The *Product* entity contains a second navigation property *URI* addressing the supplier of the product. Looking at the supplier response, you can detect two new things:

© Copyright. All rights reserved.

- First, you see that the metadata section has a property called *ETag*. OData uses HTTP ETags for optimistic concurrency control. ETags can also be used for conditional GET requests.
- Second, notice that the complex type *Address* is embedded in the payload. Not surprisingly, the metadata section contains no URI. Complex types are not addressable because of the missing key property.

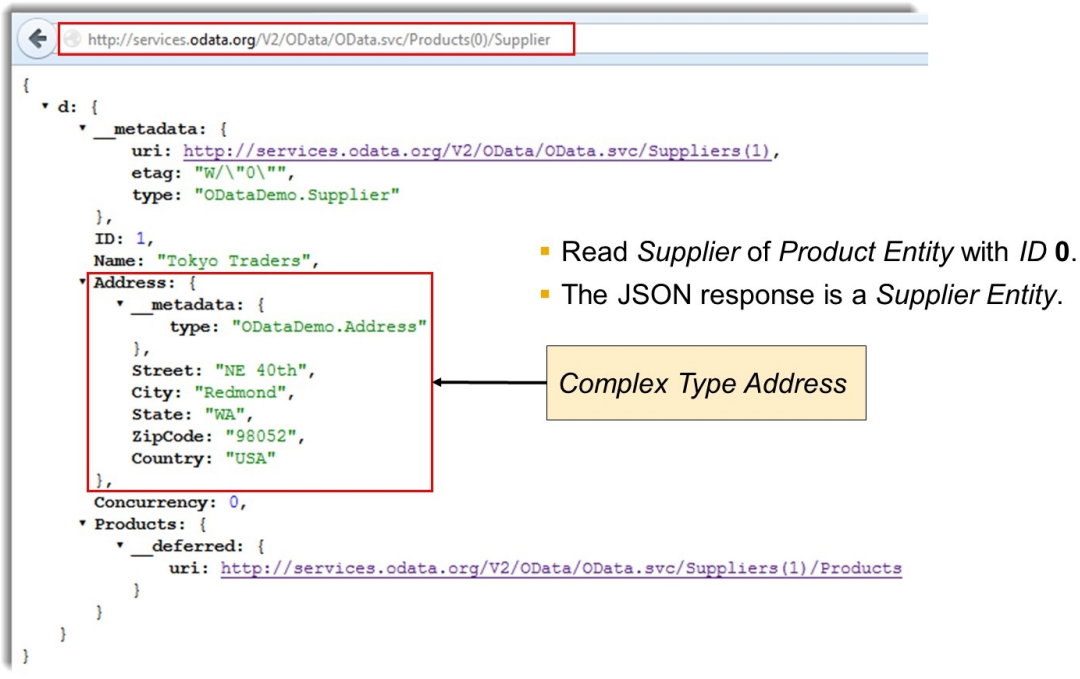


Figure 158: Exploring an OData Service (4)

The navigation property *URI* of the entity *Supplier* with *ID* 1 points to all products that are associated with this *Supplier*. The resource addressed by this URI is an entity set. This can be derived from the cardinality of the association between the entity type *Supplier* and the entity type *Product* declared in the Service Metadata Document.

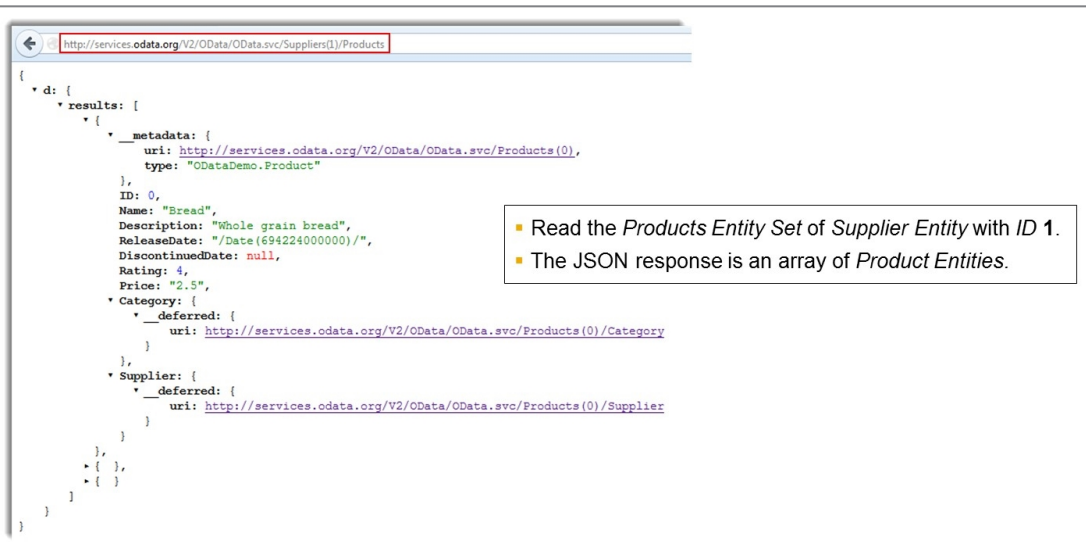


Figure 159: Exploring an OData Service (5)

© Copyright. All rights reserved.

URI Components

Custom Query Options provide an extension point for OData service-specific information to be placed in the query string portion of a URI. A Custom Query String Option is defined as any name/value pair query string parameter, where the name of the parameter does not begin with the `$` character. Any URI exposed by an OData service may include one or more custom query options. For example: `http://services.odata.org/OData/OData.svc/Products?x=y` identifies all *Product* entities and includes a custom query option `x` whose meaning is service specific.

Service operations represent functions exposed by an OData service. These functions may accept zero or more primitive type parameters. If a service operation requires one or more input parameters those parameters are passed by query string name/value pairs appended to the URI which identifies the service operation. For example, `http://services.odata.org/OData/OData.svc/GetProductsByRating?rating=5` identifies the `GetProductsByRating` Service Operation and specifies a value of `5` for the `rating` input parameter.

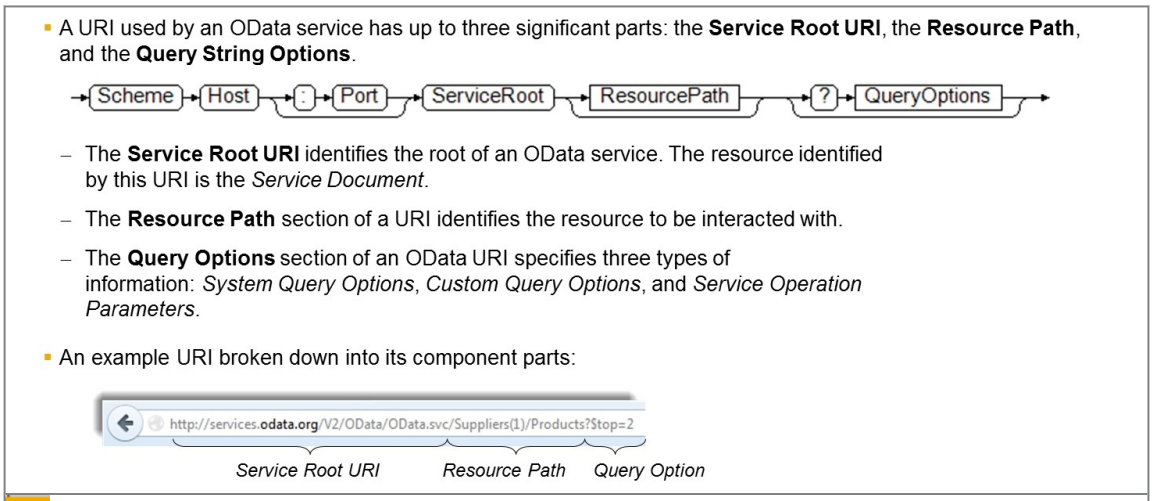


Figure 160: URI Components (1)

System Query Options

OData specifies a simple, but powerful, query language that allows a client to request arbitrary filtering, sorting, and paging.

The content in the following topics uses excerpts from http://www.odata.org. There you can find a comprehensive list of all Query Options. In this short introduction, we are focusing on the most important aspects only. We explain how to reduce the data volume transferred from the server to the client by filtering data sets and by selecting the relevant data properties only. Paging is another possibility to reduce data traffic, whereas sorting guarantees the right sequence of the entities within a requested entity set. A client can also instruct the server to also include associated entities into the response. This technique avoids subsequent read operations.

© Copyright. All rights reserved.

- **System Query Options** are query string parameters a client may specify to control the amount and the order of the data that an OData service returns for the resource identified by the URI.
- The names of all **System Query Options** are prefixed with a “$” character.
- The following is a list of the important **System Query Options**.

Operation	System Query Option
Filtering	$filter
Projecting	$select
Sorting	$orderby
Paging	$top and $skip
Inlining	$expand

Figure 161: System Query Options

Filter System Query Option ($filter)

A URI with a `$filter` system query option identifies a subset of the entities from the entity set identified by the resource path section of the URI. The subset is determined by selecting only the entities that satisfy the predicate expression specified by the query option.

The expression language that is used in `$filter` operators supports references to properties and literals. The literal values can be strings enclosed in single quotes, numbers, and the boolean values (`true` or `false`).

In addition to operators, a set of functions are also defined for use with the filter query string operator, for example, `endswith()`.

Filtering Examples

Logical Operator	
gt (greater than)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=Price gt 20
ge (greater than or equal)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=Price ge 10
lt (less than)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=Price lt 20
le (less than or equal)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=Price le 100
and (logical and)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=Price le 200 and Price gt 3.5
or (logical or)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=Price le 3.5 or Price gt 200
eq (equal)	http://services.odata.org/V2/OData/OData.svc/Suppliers?$filter=Address/City eq ‘Redmond‘
ne (not equal)	http://services.odata.org/V2/OData/OData.svc/Suppliers?$filter=Address/City ne ‘London‘
not (logical negotiation)	http://services.odata.org/V2/OData/OData.svc/Products?$filter=not endswith(Description, ‘milk‘)

Figure 162: Filter System Query Option ($filter)

Select System Query Option ($select)

A data service URI with a `$select` system query option identifies the same set of entities as a URI without a `$select` query option, however, the value of `$select` specifies that a response from an OData service should return a subset of the properties which would have been returned had the URI not included a `$select` query option.

The value of a `$select` system query option is a comma-separated list of selection clauses. Each selection clause may be a property name, navigation property name, or the `*` character.

© Copyright. All rights reserved.

If the `$select` query option lists a property that identifies a complex type, then all the properties defined on the complex type are returned.

Projecting Examples

- http://services.odata.org/V2/OData/OData.svc/Products?$select=Price,Name

  In a response from an OData service, only the *Price* and *Name* property values are returned for each product.

- http://services.odata.org/V2/OData/OData.svc/Products?$select=Name,Category

  In a response from an OData service, only the *Name* property value and a link to the related *Category* entity should be returned for each product.

Figure 163: Select System Query Option ($select)

Orderby System Query Option ($orderby)

A data service URI with a `$orderby` system query option specifies an expression for determining the values to use to order the entity set identified by the resource path section of the URI. This query option is only supported when the resource path identifies an entity set.

Sorting Examples

- http://services.odata.org/V2/OData/OData.svc/Products?$orderby=Rating

  All Product Entities are returned in ascending order sorted by the *Rating* property.

- http://services.odata.org/V2/OData/OData.svc/Products?$orderby=Rating asc

  Same as the previous example.

- http://services.odata.org/V2/OData/OData.svc/Products?$orderby=Rating,Category/Name desc

  Same as the previous URI except the set of Products is subsequently sorted (in descending order) by the *Name* property of the related *Category* entity.

Figure 164: Orderby System Query Option ($orderby)

Top and Skip System Query Options ($top and $skip)

A data service URI with a `$top` system query option identifies a subset of the entities in the entity set identified by the resource path section of the URI. This subset is formed by selecting only the first N items of the set, where N is a positive integer specified by this query option.

A data service URI with a `$skip` system query option identifies a subset of the entities in the entity set identified by the resource path section of the URI. That subset is defined by seeking N entities in the entity set and selecting only the remaining entities (starting with entity N+1). N is a positive integer as specified by this query option.

The combination of `$top` and `$skip` empowers the service consumer to page through the entity set identified by the resource path section of the URI.

© Copyright. All rights reserved.

Paging Examples

- http://services.odata.org/V2/OData/OData.svc/Products?$top=5
  The first five product entities returned where the *Products* entity set is sorted using a scheme determined by the OData service.
- http://services.odata.org/V2/OData/OData.svc/Products?$top=5&$orderby=Name desc
  The first five product entities returned in descending order when sorted by the *Name* property.
- http://services.odata.org/V2/OData/OData.svc/Categories(1)/Products?$skip=2
  The set of product entities (associated with the *category* entity identified by key value 1) starting with the third product.
- http://services.odata.org/V2/OData/OData.svc/Products?$skip=2&$top=2&$orderby=Rating
  The third and fourth product entity from the set of all products when the set is sorted by *Rating* (ascending).

Figure 165: Top and Skip System Query Options ($top and $skip)

Expand System Query Option ($expand)

A URI with a `$expand` system query option indicates that entities associated with the entity or entity set identified by the resource path section of the URI must be represented inline (that is, eagerly loaded). For example, if you want to identify a *Category* and its *Products*, you could use two URIs (and execute two requests), one for `/Categories(1)` and one for `/Categories(1)/Products`. The `$expand` option allows you to identify related entities with a single URI such that a graph of entities could be retrieved with a single HTTP request.

The syntax of a `$expand` query option is a comma-separated list of navigation properties. Additionally, each navigation property can be followed by a forward slash and another navigation property to enable the identification of a multi-level relationship.

All query options defined by OData can be combined and used in one URI. This creates a powerful query language that is fairly simple to use.

Inlining Examples

- http://services.odata.org/V2/OData/OData.svc/Categories?$expand=Products

  Identifies the *Categories* entity set as well as each of the *Products* associated with each *Category*.

- http://services.odata.org/V2/OData/OData.svc/Categories?$expand=Products/Supplier

  Identifies the *Categories* entity set as well as each of the *Products* associated with each *Category*. In addition, the URI also identifies the *Supplier* associated with each *Product*.

- http://services.odata.org/V2/OData/OData.svc/Products?$expand=Category,Supplier

  Identifies the set of *Products* as well as the *Category* and the *Supplier* associated with each *Product*.

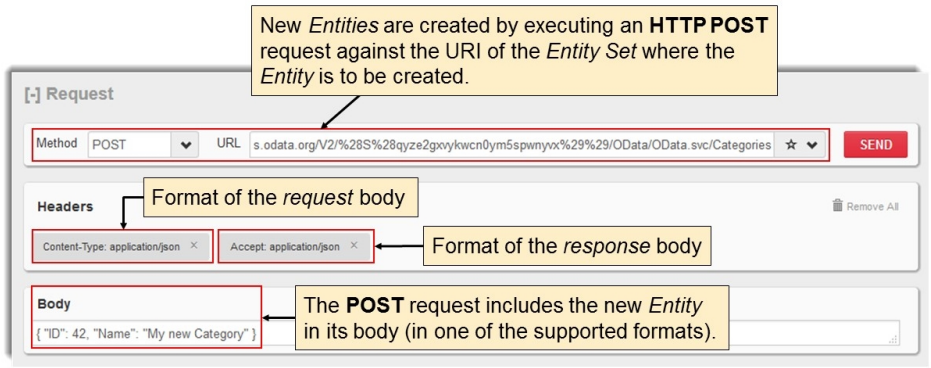
Figure 166: Expand System Query Option ($expand)

© Copyright. All rights reserved.

Creating New Entities

To demonstrate the data manipulating operations of the OData protocol, we use the RESTClient browser plugin.

The OData service used in this lesson offers read-only sessions as well as update sessions. So far, we used a read-only session, but update operations are not permitted for such a session. So, we have to connect to an update session, which means that the Service Root URI has to be adapted.

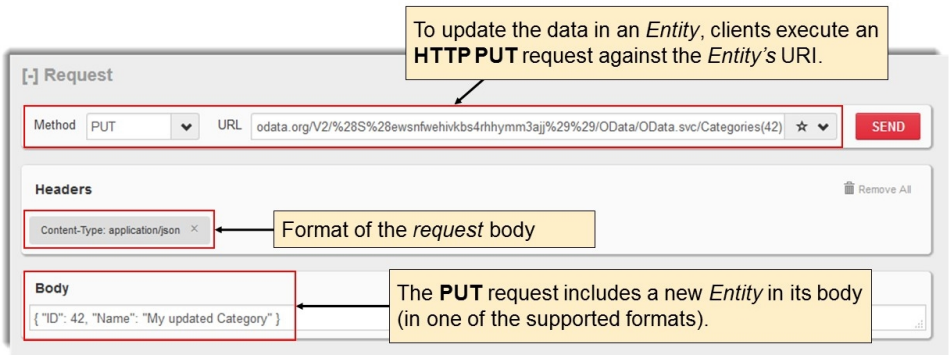


- The server processes the request by creating the resource, assigning default values to all the properties not indicated in the request that are optional, and sending the final state of the resource back to the client in the response, including a "location" header that contains the URL of the **Entity** that was created.
- **HTTP status code 201 (Created)** reflects that the *Entity* was created.

Figure 167: Creating New Entities

Updating Entities

The following figure shows the use of the HTTP PUT request to update an Entity.



- The **PUT** request replaces the existing *Entity*, so all property values in the *Entity* either take the values indicated in the request body, or are reset to their default value if not mentioned in the request.
- When processing a **PUT** request, the server returns **HTTP status code 204 (No Content)** to indicate success. There is no response body.

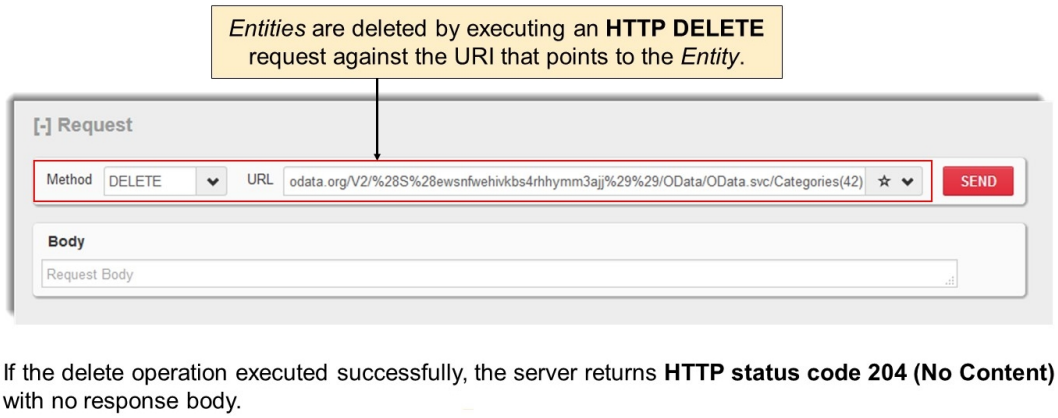
Figure 168: Updating Entities

Deleting Entities

If the entity has dependent entities, such as entities linked to it, it is up to the server to determine whether deletion should be cascaded, the operation should fail because of the presence of dependent entities, or if links are left dangling after the entity is deleted. This

© Copyright. All rights reserved.

largely depends on the server, storage model, and the actual application the OData service is a part of.



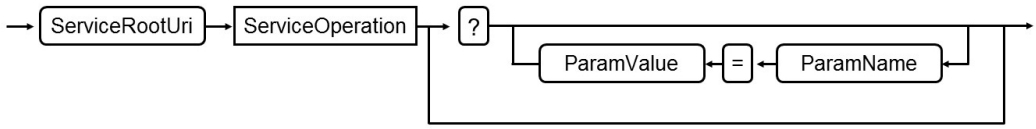
If the delete operation executed successfully, the server returns **HTTP status code 204 (No Content)** with no response body.

Figure 169: Deleting Entities

Invoking Service Operations

The following figure shows how to invoke service operations.

- *Service Operations* are identified using a URI.
- The basic rules for constructing URIs to address *Service Operations* and to pass parameters to them are shown in the following figure.



- If a particular *Service Operation* takes parameters, they are included as part of the query string in the request URI. The name of each parameter is used as the name of the query string parameter.
- Clients may invoke a *Service Operation* by issuing an HTTP GET or POST request against its URI.
- Server implementations should only allow invocation of *Service Operations* through GET when the operations do not cause side-effects in the system.

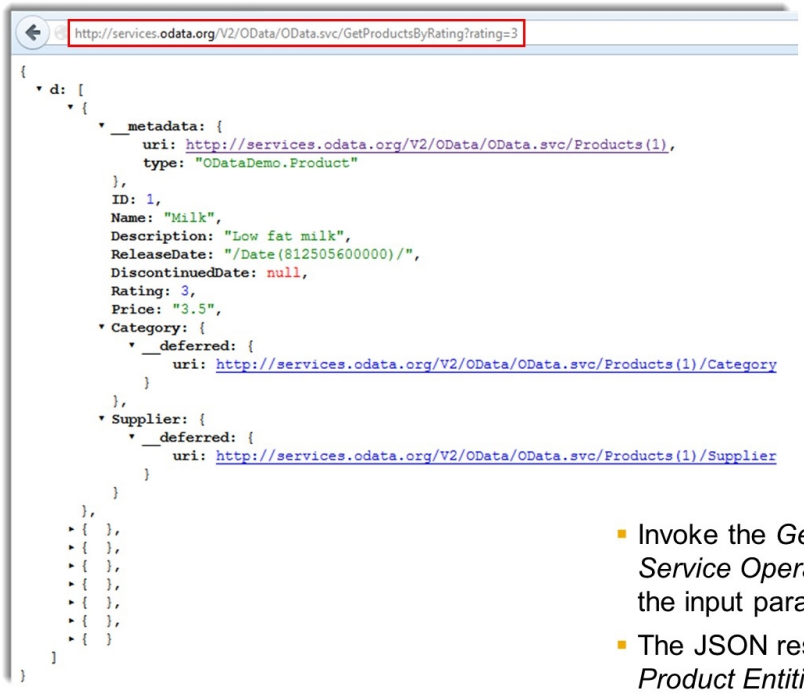
Figure 170: Invoking Service Operations

Invoking Service Operations – Example

The example in the following figure shows the JSON response of the service operation `GetProductsByRating` for a *rating* parameter that has the value 3.

Note:
The example service used in this lesson is provided by Microsoft. SAP has established a naming convention for service operation names to avoid the prefix `get`. So, in this example, the name `ProductsByRating` would be more precise. The HTTP method GET makes it clear that this is a get operation.

© Copyright. All rights reserved.



- Invoke the *GetProductsByRating Service Operation* and pass **3** for the input parameter *rating*.
- The JSON response is an array of *Product Entities*.

Figure 171: Invoking Service Operations - Example

LESSON SUMMARY

You should now be able to:

- Understand the Open Data Protocol

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. What does OData use for resource identification?

   Choose the correct answer.

   - [ ] A Email address
   - [ ] B URI
   - [ ] C Harmonized WSDL paths

2. Which design pattern is used for resolving relationships between resources?

   Choose the correct answer.

   - [ ] A Resource design pattern
   - [ ] B Hypermedia design pattern
   - [ ] C Metadata design pattern

3. Which of the following return types are supported by service operations?

   Choose the correct answers.

   - [ ] A Complex Types
   - [ ] B Primitive Types
   - [ ] C Aggregation Types
   - [ ] D Infinitive Types

© Copyright. All rights reserved.

Learning Assessment - Answers

1. What does OData use for resource identification?

   Choose the correct answer.

   - [ ] A Email address
   - [x] B URI
   - [ ] C Harmonized WSDL paths

   Correct. The URI is used to specify the path to a unique resource, such as the service document, an entity, or an entity collection.

2. Which design pattern is used for resolving relationships between resources?

   Choose the correct answer.

   - [ ] A Resource design pattern
   - [x] B Hypermedia design pattern
   - [ ] C Metadata design pattern

   Correct. To address aspects of the referenced OData service, the hypermedia design pattern is used.

3. Which of the following return types are supported by service operations?

   Choose the correct answers.

   - [x] A Complex Types
   - [x] B Primitive Types
   - [ ] C Aggregation Types
   - [ ] D Infinitive Types

   Correct. Data returned by an OData service operation may return complex types or primitive types.

© Copyright. All rights reserved.

UNIT 12 OData Model

Lesson 1

Consuming OData Services with SAPUI5 193

UNIT OBJECTIVES

- Use the SAPUI5 OData Model

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 12
Lesson 1

Consuming OData Services with SAPUI5

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use the SAPUI5 OData Model

SAPUI5 OData Model

The figure, SAPUI5 OData Model, provides an overview of the SAPUI5 OData Model.

- The **OData Model** enables binding of controls to data from OData services.
- The **OData Model** is a server-side model, meaning that the dataset 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.
- The requests to the service to fetch data are made automatically based on the data bindings that are defined for the controls.
- The **OData Model** currently supports the following OData versions:
  - OData Version 2.0
  - OData Version 4.0
- This course covers the **OData V2 Model**.
- Be aware of the Same-Origin-Policy security concept that prevents access to backend systems on different domains or sites.

OData Model

Figure 172: SAPUI5 OData Model

Adding an OData Service Connection to an SAP Business Application Studio Project

To consume an OData service in a SAPUI5 application, make sure that the required destination is configured in the SAP Business Technology Platform subaccount.

You can use the project wizard to configure an SAP Business Application Studio project to use a back-end OData service. To do this, select **`Connect to a System`** as *Data source* in the *Data Source and Service Selection* step. Also in this step, select the destination that was maintained in the SAP BTP Cockpit as *System*. You can then use the *Service* field to select the desired OData service from the back-end system mapped to the destination.

© Copyright. All rights reserved.

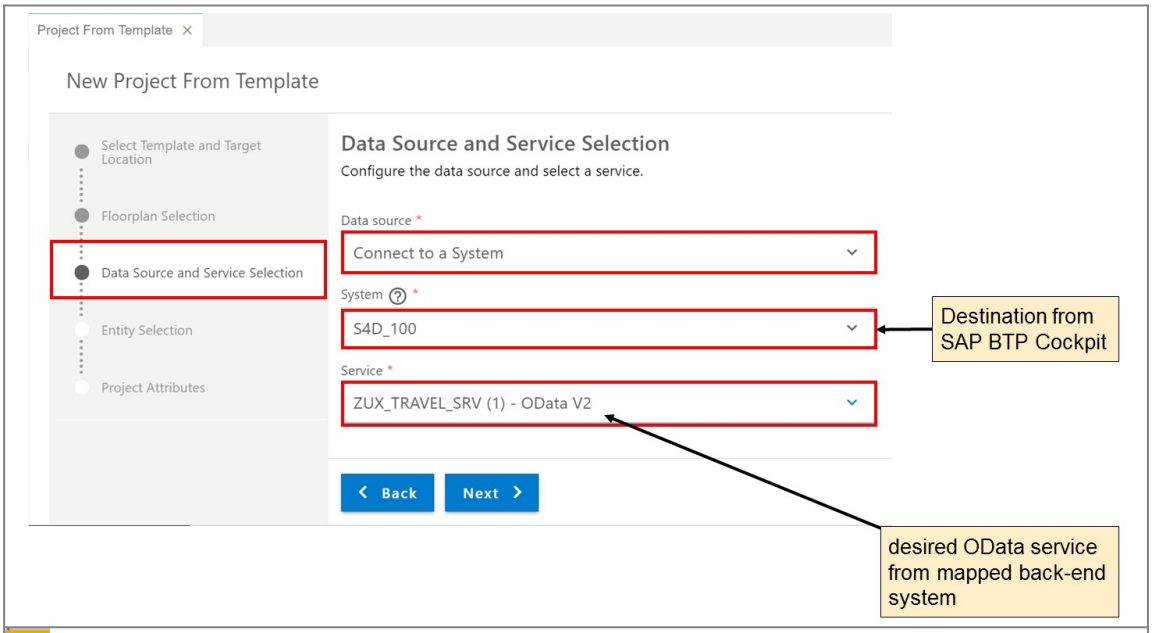


Figure 173: Adding an OData Service Connection to an SAP Business Application Studio Project

Result: The project is set up to consume the selected OData service, with the following changes:

- The application descriptor (`manifest.json` file) is updated: The property `dataSources` is added to the `sap.app` section, including the URI to the selected OData service.
- An OData model entry is added to the `models` section in the application descriptor.
- In the files `ui5.yaml` and `ui5-local.yaml`, proxy settings are added for accessing the back-end system using the selected destination.
- A file named `metadata.xml`, reflecting the metadata of the service entities, is added to the `localService` folder of the SAPUI5 project.

Creating the Model Instance

One OData model instance can only cover one OData service. To access multiple services, you have to create multiple OData model instances.

The Service URL can be passed as the first parameter, or within the `mParameters` map to the constructor.

© Copyright. All rights reserved.

- The only mandatory parameter when creating an *ODataModel* instance is the Service URL.

Main.controller.js ×

```
1  sap.ui.define([
2      "sap/ui/core/mvc/Controller",
3      "sap/ui/model/odata/v2/ODataModel"
4  ], function(Controller, ODataModel) {
5      "use strict";
6
7      return Controller.extend("sap.training.controller.Main", {
8
9          onInit: function() {
10             var sUrl = "/ODATA_ORG/V2/(S(quxwlgrwfkjy4erol4obk4nb))/OData/OData.svc/";
11             var oModel = new ODataModel(sUrl);
12
13             this.getView().setModel(oModel);
14         }
15
16     });
17 });
```

Figure 174: Creating the Model Instance

Addressing Entities – Binding Path Syntax

If 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 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.
- URL parameters, such as filters, cannot be added to a binding path.
- A binding path can be absolute or relative.

... .view.xml ×

```
8   <ComboBox items="{/Categories}" selectionChange="onCategoryChange">
9       <items>
10          <core:Item text="{Name}"/>
11      </items>
12  </ComboBox>
13
14  <List id="idProductList" items="{Products}">
15      <items>
16          <StandardListItem title="{Name}"/>
17      </items>
18  </List>
```

Absolute binding paths (starting with a slash ('/')) are resolved immediately.

A relative path can only be resolved if it can be automatically converted into an absolute binding path.

Figure 175: Addressing Entities - Binding Path Syntax

Binding-specific Parameters

The figures, Binding-specific Parameters (1) and (2), show examples of the use of binding-specific parameters.

© Copyright. All rights reserved.

- The **expand** parameter allows the application to read associated entities with their navigation properties. In this example, the Category and the Supplier are embedded inline in the server response for each *Product*.

```
... .view.xml
11    <t:Table rows="{path: '/Products', parameters: {expand: 'Category, Supplier'} }">
12        <t:columns>
13            [...]
14
15        </t:columns>
16    </t:Table>
```

- The **select** parameter allows the application to define a subset of properties that is read when requesting an entity. In this example, only the properties *Name* and *Description* are returned for each *Product*.

```
... .view.xml
11    <t:Table rows="{path: '/Products', parameters: {select:'Name, Description'} }">
12        <t:columns>
13            [...]
14
15        </t:columns>
16    </t:Table>
```

Figure 176: Binding-specific Parameters (1)

- In this example, the *Category* and the *Supplier* are embedded inline in the server response for each *Product*. Only the property *Name* is returned for the *Category* and the *Supplier*, whereas all *Product* properties are returned.

```
... .view.xml
11    <t:Table rows="{path: '/Products',
12                    parameters: {expand:'Category, Supplier', select:'*, Category/Name, Supplier/Name'} }">
13        <t:columns>
14
15            [...]
16        </t:columns>
17    </t:Table>
```

- You can use custom query options to specify the input parameters for service operations.

```
... .view.xml
11    <t:Table rows="{path: '/GetProductsByRating', parameters: {custom: {'rating': '3'} } }">
12        <t:columns>
13
14            [...]
15        </t:columns>
16    </t:Table>
```

Figure 177: Binding-specific Parameters (2)

Accessing Data from an OData Model

The figure, Accessing Data from an OData Model, shows how to access data from an OData model using the `getProperty()` method.

© Copyright. All rights reserved.

- The data requested from an OData service is cached in the OData model. It can be accessed by the *getProperty()* method.
- By means of *getProperty()*, you can access single entities and properties:

```
.controller.js
52    var oEntity = oModel.getProperty("/Products(2)");
```

```
.controller.js
54    var value = oModel.getProperty("/Products(2)/Name");
```

- *getProperty()* does not request data from the backend, so you can only access already requested and cached entities.
- The values returned by *getProperty()* are copies of the data in the model, not references as in the JSON model.

Figure 178: Accessing Data from an OData Model

Entity Creation

The application can choose the properties that are to be included in the created object and pass its own default values for these properties. Per the default, all property values are empty; that is, undefined. The entity set and the passed properties must exist in the metadata definition of the OData service.

- 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.

```
....controller.js
11    var oModel = new sap.ui.model.odata.v2.ODataModel(sUrl);
12    oModel.setDefaultBindingMode(sap.ui.model.BindingMode.TwoWay);
13
14    var oContext = oModel.createEntry("/Products");
15    var oSimpleForm = this.getView().byId("NewProductSF");
16    oSimpleForm.setBindingContext(oContext);
```

- To store the entities in the OData backend, the application calls *submitChanges()*. To reset the changes, the application can call the *deleteCreatedEntry(oContext)* method.

Figure 179: Creating Entities

CRUD Operations

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.

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

© Copyright. All rights reserved.

`refreshSecurityToken()` function on the OData model. The token is fetched with a request to the service document. To ensure the reception of a valid token, ensure that the service document is not cached.

- The OData model allows manual **CRUD** (*create, read, update, delete*) operations on the OData service.
- All operations have 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.

Figure 180: CRUD Operations

Creating Entities

The figure, Creating Entities, shows how to create an entity using the `create()` method.

- 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.

.controller.js

```
49      // Send OData Create request
50      oModel.create("/Products", oNewProduct, {
51          success: function(oData, response) {
52              [...]
53
54          },
55          error: function(oError) {
56              [...]
57
58          }
59      });
```

Figure 181: Creating Entities

Reading Entities

The figure, Reading Entities, shows how to read from entities using the `read()` method.

© Copyright. All rights reserved.

- The *read()* function triggers 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.

C_R_D.controller.js

```
59    ... //Get highest Product Id
60      oModel.read("/Products", {
61          urlParameters: {
62              "$top": 1,
63              "$orderby": "ID desc",
64              "$select": "ID"
65          },
66          success: function(oData, response) {
67              MessageToast.show("Highest Id: " + oData.results[0].ID);
68          },
69          error: function(oError) {
70              MessageToast.show(oError.message);
71          }
72      });
```

Figure 182: Reading Entities

Updating Entities

The `update()` method triggers a PUT/MERGE request to the OData service that was specified in the model constructor. The update method used is defined by the global *defaultUpdateMethod* parameter, which is `sap.ui.model.odata.UpdateMethod.Merge` by default.

To send update requests in a PUT request, you must set the parameter *defaultUpdateMethod* to `sap.ui.model.odata.UpdateMethod.Put` when instantiating the OData model.

- The *update()* function triggers a *PUT/MERGE* request to an OData service which was specified at creation of the OData model.

....controller.js

```
52    // Send OData Update request
53    oModel.update('/Products(' + oUpdatedProductData.ID + ')', oUpdatedProductData, {
54        success: function() {
55          [...]
56
57        },
58        error: function(oError) {
59          [...]
60
61        }
62    });
```

Figure 183: Updating Entities

Deleting Entities

The figure, Deleting Entities, shows how to delete entities using the `remove()` function.

© Copyright. All rights reserved.

- 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 *Entity* which should be deleted.

```
.controller.js
67      // Send OData Delete request
68      oModel.remove('/Products(' + productId + ')', {
69          success: function(oData, response) {
70              [...]
71
72          },
73          error: function(oError) {
74              [...]
75
76          }
77      });
```

Figure 184: Deleting Entities

Refreshing the Model

The figure, Refreshing the Model, provides information on the automatic and manual refresh of the OData model.

- The model provides a mechanism to automatically refresh bindings that depend on changed entities. If you carry out a **create, update** or **delete** function, the model identifies the bindings and triggers a refresh for these bindings.
- You can disable the auto refresh by calling
  oModel.setRefreshAfterChange(false);
  This disables automatic updates of all bindings after change operations.
- If the auto refresh is disabled, the application must take care of refreshing the respective bindings.
- The **refresh()** function refreshes all data within an OData model. Each binding reloads its data from the server. Data that has been imported using manual CRUD requests is not reloaded automatically.

Figure 185: Refreshing the Model

Two-way Binding

Filtering and sorting is not possible if two-way changes are present, because this would cause inconsistent data on the UI. Therefore, before you perform sorting or filtering, you must submit or reset the changes.

You can also reset only specific entities by calling `resetChanges()` with an array of entity paths.

© Copyright. All rights reserved.

- The ODataModel enables two-way binding.

```
... .controller.js ×
13  var oModel = new sap.ui.model.odata.v2.ODataModel(sUrl);
14  oModel.setDefaultBindingMode(sap.ui.model.BindingMode.TwoWay);
```

- Data changes are made on a data copy. To submit the changes, use *submitChanges()*.

```
... .controller.js ×
68  oModel.submitChanges({
69      success: function(oData)  {  [...]  },
70      error:   function(oError) {  [...]  }
71  });
```

- With *resetChanges()* you can reset all changes.

```
... .controller.js ×
93  oModel.resetChanges();
```

- Use *hasPendingChanges()* to check if pending changes exist in the model.

```
... .controller.js ×
52  if (oModel.hasPendingChanges()) {  [...]  }
```

Figure 186: Two-way Binding

LESSON SUMMARY

You should now be able to:

- Use the SAPUI5 OData Model

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which OData versions are currently supported by SAPUI5?

   Choose the correct answers.

   - [ ] A OData Version 1.0
   - [ ] B OData Version 2.0
   - [ ] C OData Version 3.0
   - [ ] D OData Version 4.0

2. Does the invocation of the `getProperty()` method on an OData model object trigger a request to the back-end?

   Choose the correct answer.

   - [ ] A Yes, for each `getProperty()` call, a synchronous request to the OData service is triggered to get the property value.
   - [ ] B No, a request is not triggered. The `getProperty()` method returns the current value from the OData model object at the client side. If the value is not available, `undefined` is returned.
   - [ ] C It depends on how the `getProperty()` method is parametrized by the developer. There is a property called *bFetchFromServer*. If this parameter is `true`, a request to the server-side is sent.

3. What function can be used to check whether the OData model contains changes?

   Choose the correct answer.

   - [ ] A `hasUpdatedValues()`
   - [ ] B `hasPendingChanges()`
   - [ ] C `hasPendingValues()`
   - [ ] D `hasUpdatedChanges()`

© Copyright. All rights reserved.

Unit 12

Learning Assessment - Answers

1. Which OData versions are currently supported by SAPUI5?

   Choose the correct answers.

   - [ ] A OData Version 1.0
   - [x] B OData Version 2.0
   - [ ] C OData Version 3.0
   - [x] D OData Version 4.0

   Correct. SAPUI5 supports OData Version 2.0 and 4.0.

2. Does the invocation of the `getProperty()` method on an OData model object trigger a request to the back-end?

   Choose the correct answer.

   - [ ] A Yes, for each `getProperty()` call, a synchronous request to the OData service is triggered to get the property value.
   - [x] B No, a request is not triggered. The `getProperty()` method returns the current value from the OData model object at the client side. If the value is not available, `undefined` is returned.
   - [ ] C It depends on how the `getProperty()` method is parametrized by the developer. There is a property called *bFetchFromServer*. If this parameter is `true`, a request to the server-side is sent.

   Correct. The `getProperty()` method does not request data from the back-end, so you can only access already requested and cached entities. The values returned by `getProperty()` are copies of the data in the model, not references as in the JSON model.

© Copyright. All rights reserved.

3. What function can be used to check whether the OData model contains changes?

   Choose the correct answer.

   - [ ] A `hasUpdatedValues()`
   - [x] B `hasPendingChanges()`
   - [ ] C `hasPendingValues()`
   - [ ] D `hasUpdatedChanges()`

   Correct. The function `hasPendingChanges()` checks if there are pending changes in the model created by the `setProperty()` method.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 13 Component Implementation

Lesson 1

Implementing Components 209

UNIT OBJECTIVES

- Implement a component

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 13
Lesson 1

Implementing Components

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement a component

Components Overview

An application can use components from locations that are different from the location at which the application is running. This allows components to be developed by different development teams and be used in different projects. Components also support the encapsulation of closely-related application parts into a particular component. This makes the structure of an application and its code easier to understand and easier to maintain.

The `sap.ui.core.Component` class is the base class for UI and faceless components, and it 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 independent and reusable parts used in SAPUI5 applications.
- SAPUI5 provides the following 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*)

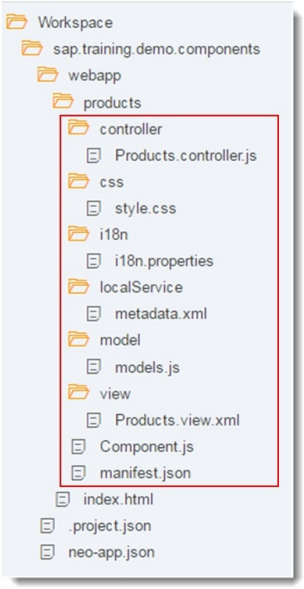
    UI Components extend *Faceless Components* and add rendering functionality to the component. They represent a screen area or element on the user interface, for example, a button or a shell, along with the respective settings and metadata.

Figure 187: Components Overview

Structure of a Component

The figure, Structure of a Component, shows the component structure.

© Copyright. All rights reserved.



- A *Component* is a folder.
- The *folder* name defines the *Component* name and contains all optional and required resources that are used in the *Component* except for the required SAPUI5 libraries and child components.
- Optional resources (for example): CSS files, internationalization files, views, and images.
- Mandatory files:
  - **Component.js**
    Component controller which provides the *Component* methods
  - **manifest.json**
    Descriptor for storing metadata associated with the *Component*

Figure 188: Structure of a Component

Component.js File

The figure, Component.js File, shows the `Component.js` file and explains its content.

- To create a *UI Component*, you extend the *UI Component*'s base class.
- Specify the modules that the *Component* depends on in the dependency array of *sap.ui.define()*.
- The module name of the *Component* is the package name + ***.Component***.
  You specify the module name in the *extend()* method.
- The *metadata* provides the information to ensure the completeness of the elements of which the *Component* consists of.

Component.js

```
sap.ui.define(["sap/ui/core/UIComponent", "sap/ui/model/resource/ResourceModel",
                "sap/ui/model/odata/v2/ODataModel"],
    function(UIComponent, ResourceModel, ODataModel) {
        "use strict";
        return UIComponent.extend("sap.training.products.Component", {

            metadata: {
                [...]
            }
            [...]
        });
    });
```

Figure 189: Component.js File

Properties Section in Component Metadata

The figure, Properties Section in Component Metadata, shows the `properties` section of a component.

© Copyright. All rights reserved.

- You can add a ***properties*** section to the metadata for all properties that can adopt different values at runtime.
- The *getters* and *setters* for these properties are generated automatically.
- You can overwrite the generated methods if you require additional functionality.

Component.js

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

        metadata: {
            properties: {
                text: {
                    type: "string"
                }
            }
        },

        // Override setText() method of the Component
        setText: function(sText) {
            this.setProperty("text", sText);
            [...]
            return this;
        }

    });
  });
```

Figure 190: Properties Section in Component Metadata

Methods Controlling the Initial Instantiation of a Component

The figures, Methods Controlling the Initial Instantiation (1) and (2), shows how to control the initial instantiation of a component.

- The *init()* method initializes the Component instance after creation.
- Subclasses of *Component* should override this hook to implement any necessary initialization.
- It is obligatory to make a super call to the *init()* function of the base class in the overridden *init()* method.
- Applications must not call this hook method directly, it is called by the framework while the constructor of the *Component* is executed.

Component.js

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

            init: function() {
                // call the init function of the parent
                UIComponent.prototype.init.apply(this, arguments);
                [...]
            }
        });
    });
```

Figure 191: Methods Controlling the Initial Instantiation (1)

© Copyright. All rights reserved.

- The *createContent()* method can be used to create the content (UI Control Tree) of the *Component*.
- This method has to be overwritten in the implementation of the *Component* if the root view is not declared in the *Component* metadata.
- Create all views, controls, and elements in a *UI Component* using the *this.createId()* method and retrieve them using the *this.byId()* method.

Component.js

```
1  sap.ui.define(["sap/ui/core/UIComponent", "sap/m/Button"],
2      function(UIComponent, Button) {
3          "use strict";
4          return UIComponent.extend("sap.training.button.Component", {
5              [...]
6
7              createContent: function() {
8                  this.oButton = new Button(this.createId("btn"));
9                  return this.oButton;
10             }
11         });
12     });
```

Figure 192: Methods Controlling the Initial Instantiation (2)

Using UI Components in Applications

To render UI Components, you must wrap them in a `ComponentContainer`. You cannot use the `placeAt()` method to place UI Components directly in a page. The `ComponentContainer` separates the application and the nested component.

To load and create a UI Component, you can use the following options:

- Create a new component instance asynchronously before creating the container using method `sap.ui.core.Component.create()`.
- Create the component instance asynchronously while creating the container using the constructor of `sap.ui.core.ComponentContainer`.

In both cases you can use the “manifest first” function, which enables SAPUI5 to load the `manifest.json` before the component instance is created. To enable this function, you can choose one of the following options:

- Set the `manifest` parameter to **`true`**.
- Specify the URL of the descriptor file (`manifest.json`) as the value for the parameter `manifest`.

SAP recommends the “manifest first” function; 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 to the component preload. Models can also be preloaded.

© Copyright. All rights reserved.

- To render *UI Components*, you must wrap them in a *ComponentContainer*.
- The *ComponentContainer* is a SAPUI5 control that can be used at any place within the SAPUI5 control tree.
- SAPUI5 provides different options for instantiating a *ComponentContainer* and a *UI Component* within an application.

```
// "ComponentContainer" required from
// module "sap/ui/core/ComponentContainer"
var oContainer = new ComponentContainer({
    name: "sap.training.button",
    manifest: true,
    async: true,
    settings: {
        id: "butComp",
        text: "Hello World"
    }
});
oContainer.placeAt("target");
```

- In the shown code snippet, the component instance is created asynchronously with “manifest first” mode while creating the container.
- The “manifest first” mode improves performance during the initial load since the loading process can be parallelized and optimized.

Figure 193: Using UI Components in Applications

With the declarative API, it is possible to define the initially-started component directly in the HTML markup. This is instead of the imperative way, using JavaScript.

To define the initial component in a declarative way, load and execute the `ComponentSupport` module after the initialization. For that, the configuration option `onInit` can be used. The passed string can be a reference to a function or a name of a module. Modules are indicated by the prefix **`module:`** (like **`"module:sap/ui/core/ComponentSupport"`**).

The function `sap/ui/core/ComponentSupport.run` is called automatically once the module has been required. It finds all DOM elements with the attribute `data-sap-ui-component` and parses the attributes from these DOM elements for the settings of the `ComponentContainer` which will be placed into these DOM elements.

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 some cases, the component initialization 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. The custom module can load dependencies and execute code before activating the `ComponentSupport` module.

© Copyright. All rights reserved.

- With the declarative API it is possible to define the initially started component directly in the HTML markup:
  - Use the configuration option *onInit* in the bootstrap script to load and execute the module **sap/ui/core/ComponentSupport**.
  - Use the custom data attribute *data-sap-ui-component* to place components within the DOM.

```
<head>
    <script id="sap-ui-bootstrap"
        [...]
        data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"
        data-sap-ui-async="true">
    </script>
</head>

<body class="sapUiBody">
    <div data-sap-ui-component data-name="sap.training.products"></div>
    <div data-sap-ui-component data-name="sap.training.button"
        data-id="container" data-settings='{"id": "butComp", "text": "Hello World"}'></div>
</body>
```

Figure 194: Declarative API for Initial Components

Application Descriptor

The descriptor for applications, components, and libraries is inspired by the Web Application Manifest concept introduced by the W3C.

The data is stored in JSON format in the `manifest.json` file.

The read metadata is used to define the dependencies that need to be loaded to start the component.

- The *Application Descriptor* provides a central, machine-readable and easy-to-access location for storing metadata associated with a *Component*.
- The existence of the *manifest.json* file must be declared in the *Component* metadata.
- The declaration in the *Component* metadata makes the *Component* load the *manifest.json* file and read the relevant entries for SAPUI5.

Component.js

```
1  sap.ui.define(["sap/ui/core/UIComponent", "sap/ui/model/resource/ResourceModel",
2                 "sap/ui/model/odata/v2/ODataModel"],
3      function(UIComponent, ResourceModel, ODataModel) {
4          "use strict";
5          return UIComponent.extend("sap.training.products.Component", {
6
7              metadata: {
8                  manifest: "json"
9              }
10             [...]
11         });
12     });
```

Figure 195: Application Descriptor

© Copyright. All rights reserved.

Structure of the Application Descriptor

The figures, Structure of the Application Designer (1) — (6) explain the structure of the application descriptor.

- The content for the *Application Descriptor* is structured using namespaces, amongst others, the **sap.app, sap.ui,** and **sap.ui5** namespaces.
  - The *sap.app* namespace contributes application-specific attributes.
  - The *sap.ui* namespace contributes UI-specific attributes.
  - The *sap.ui5* namespace contributes SAPUI5-specific attributes.

Figure 196: Structure of The Application Descriptor (1)

Using the property *i18n* of the section `sap.app`, you can define the path to a resource bundle file. You can refer to the resource bundle for the *title* and *description*, so that it can be translated to other languages.

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

For compatibility reasons, the root object and each of the sections state the descriptor version number 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.

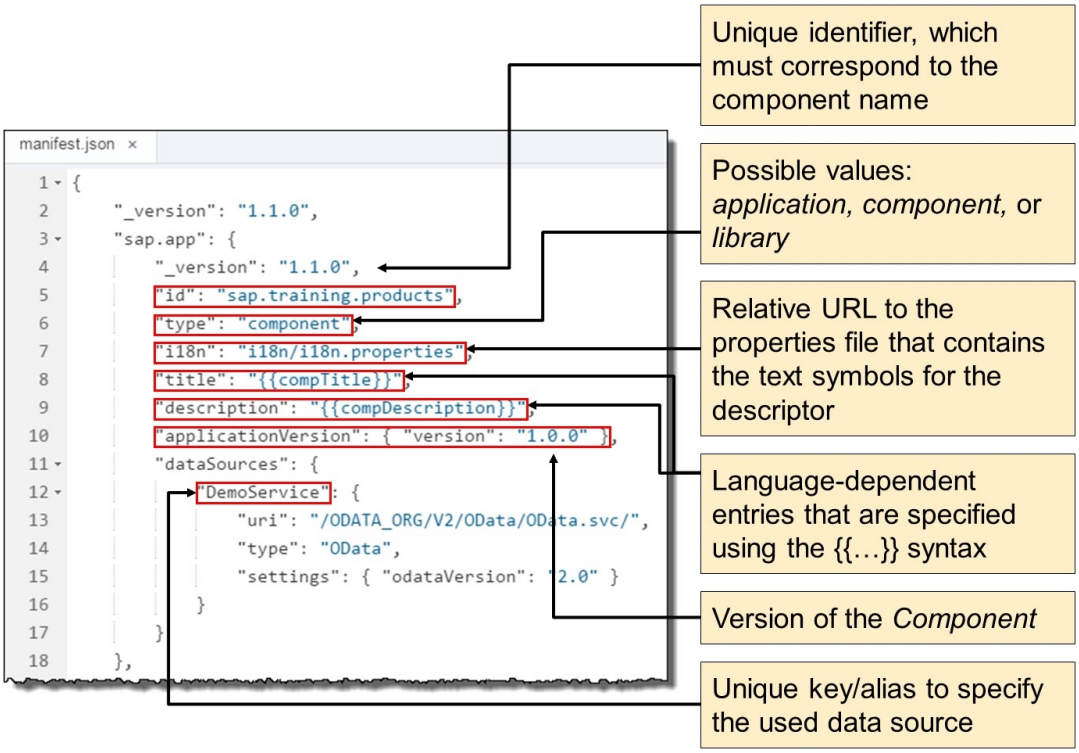


Figure 197: Structure of the Application Descriptor (2)

The *icons* property provides an object with the following app-specific icons:

© Copyright. All rights reserved.

- *icon*:

  Icon of the app

- *favIcon*:

  ICO file to be used inside the browser and for desktop shortcuts

- *phone*:

  57 x 57 pixel version for non-retina iPhones

- *phone@2*:

  114 x 114 pixel version for retina iPhones

- *tablet*:

  72 x 72 pixel version for non-retina iPads

- *tablet@2*:

  144 x 144 pixel version for retina iPads

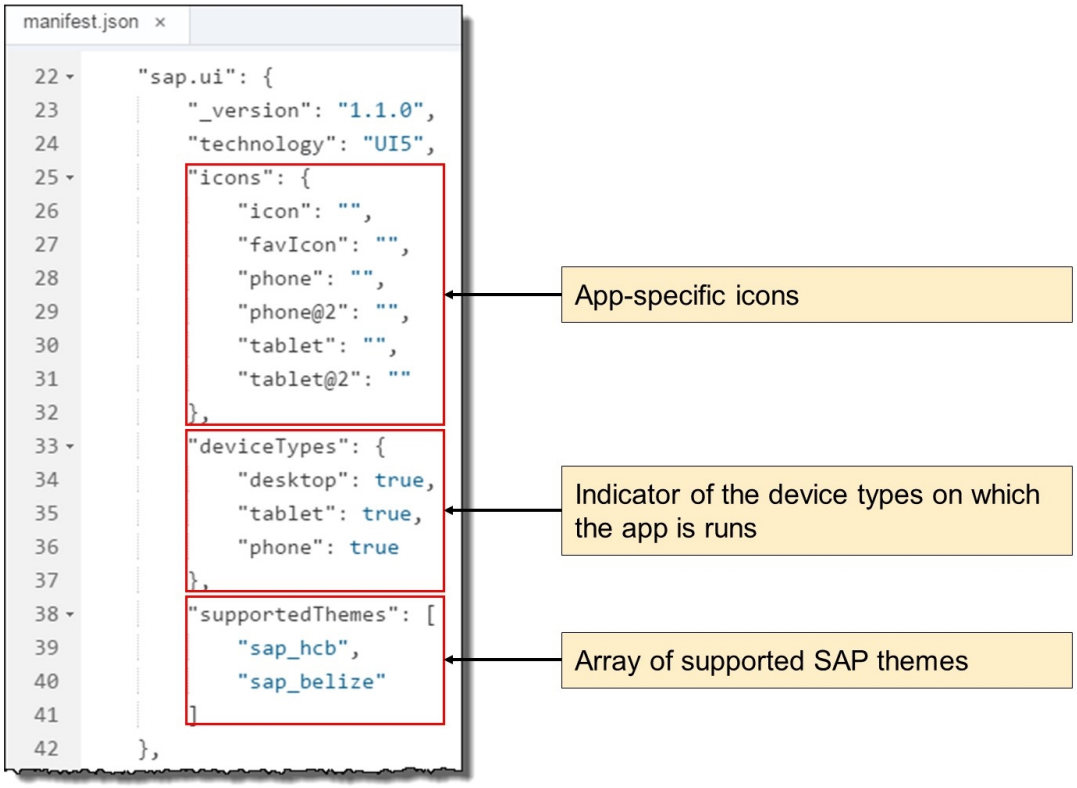


Figure 198: Structure of the Application Descriptor (3)

The `sap.ui5 namespace` is basically aligned with the previous component metadata and contributes SAPUI5-specific attributes for the application descriptor.

The *rootView* can be the view name as string for XML views or the view configuration object, with *viewName* for the view name as string, and *type* for the view type.

Using the *dependencies* parameter, you specify the external dependencies, such as libraries or components. The SAPUI5 core loads these dependencies before the component is initialized. You can use everything that is referenced here in your component code from the start. The information provided by *minUI5Version* helps you to ensure that the features of the

© Copyright. All rights reserved.

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.

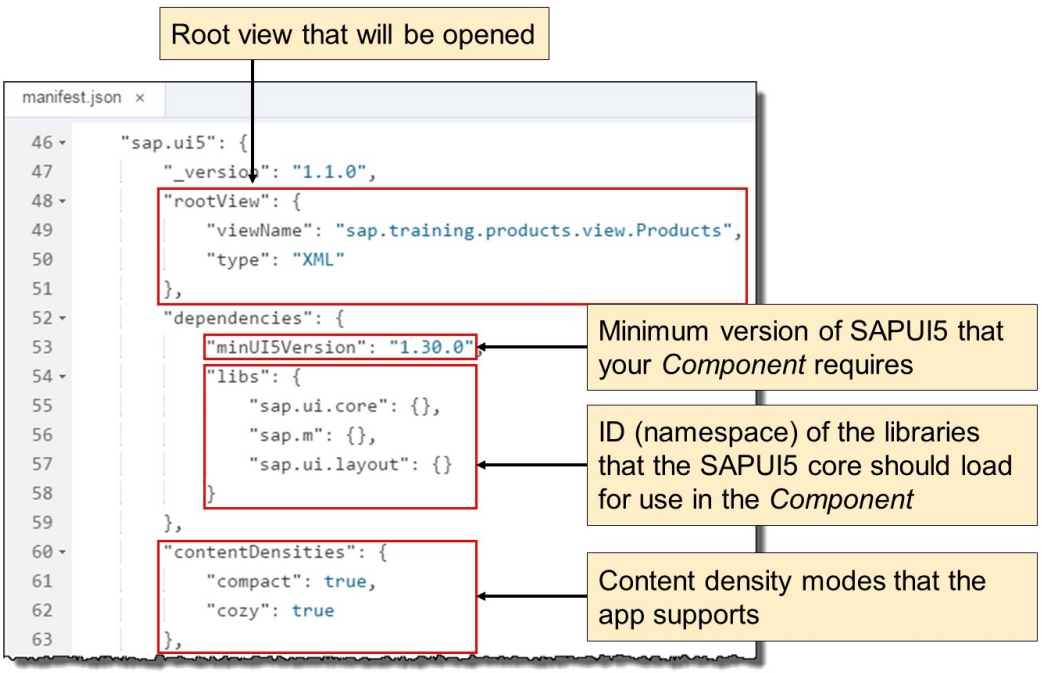


Figure 199: Structure of the Application Descriptor (4)

The `models` section allows automatic instantiation of models. The model key is used as the model name when the model is automatically instantiated during component initialization. Use an empty string (`""`) for the default model. The model *type* represents the model class name and the *uri* the URL of the model. The *dataSource* is the key/alias from `sap.app` *dataSources* to reference an existing data source. The `settings` object is passed to the model constructor.

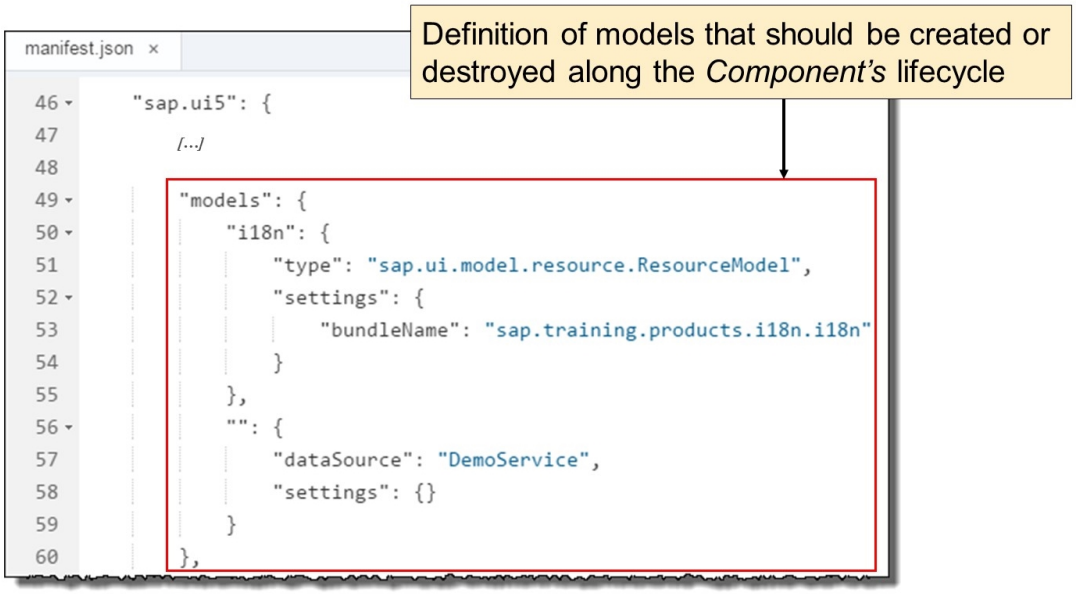


Figure 200: Structure of the Application Descriptor (5)

© Copyright. All rights reserved.

The `config` section is used for the definition of a static configuration for the *Component*. The *Component* configuration is part of the *Component* metadata. The configuration format is JSON.

In the `resources` section of the `sap.ui5 namespace`, additional `js` (JavaScript) and `css` resources for the *Component* can be loaded. The JavaScript files are loaded by the module loading mechanism. The CSS files are added to the head of the HTML page as a `link` tag and loaded by the browser automatically. The resources are resolved relative to the location of the `manifest.json` file.

The attribute *handleValidation* in the `sap.ui5` namespace can be used to enable or disable validation handling by the Message Manager for the component. Possible values for this attribute are `true` and `false` (default).

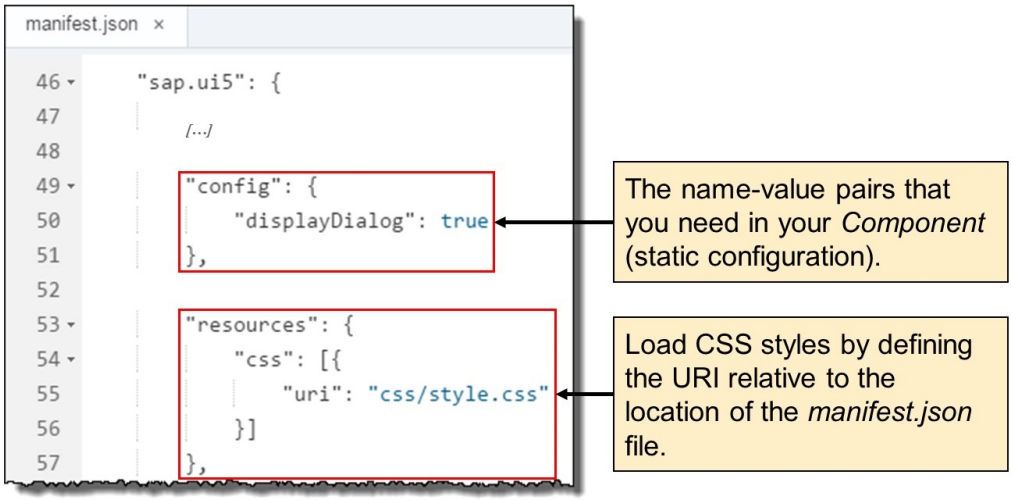


Figure 201: Structure of the Application Descriptor (6)

Application Descriptor API

The method `getManifest()` returns the manifest defined in the metadata of the component.

The method `getManifestEntry()` can be called in the following ways:

- by `section/namespace` (for example, **`oComponent.getManifestEntry("sap.ui5");`**)
- by `path` (for example, **`oComponent.getManifestEntry("/sap.ui5/config");`**)

Calling by `section/namespace` returns the configuration for the specified manifest section and by `path` allows the specification of a concrete path to a dedicated entry inside the manifest. The path syntax always starts with a slash (/).

© Copyright. All rights reserved.

At runtime, the *manifest.json* content can be accessed from the *Component* as shown in the following code snippet.

Component.js ×

```
12  init: function () {
13      [...]
14
15      this.mManifest = this.getManifest();
16
17      this.mAppData = this.getManifestEntry("sap.app");
18
19      this.mConfig = this.getManifestEntry("/sap.ui5/config");
20
21      [...]
22  }
```

- Get the complete manifest.
- Get a specific namespace.
- Get the static *Component* configuration.

Figure 202: Application Descriptor API

Accessing a Component Instance

The `getOwnerComponent()` method returns the component of the controller's view.

If there is no component connected to the view or the view is not connected to the controller, `undefined` is returned.

- Using the method *getOwnerComponent()*, you can access the *Component* instance, which is connected to a *View* from the *View's controller*.

... .controller.js ×

```
10  var oComponent = this.getOwnerComponent();
11
12  var mManifest = oComponent.getManifest();
13  var mAppData = oComponent.getManifestEntry("sap.app");
14  var mConfig = oComponent.getManifestEntry("/sap.ui5/config");
15  [...]
```

Figure 203: Accessing a Component Instance

LESSON SUMMARY

You should now be able to:

- Implement a component

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Learning Assessment

1. Which types of components are provided by SAPUI5?

   Choose the correct answers.

   - ☐ A Faceless Components
   - ☐ B UI Components
   - ☐ C Backend Components
   - ☐ D Agile Components

2. In which file is the Component controller implemented?

   Choose the correct answer.

   - ☐ A `Component.json`
   - ☐ B `Component.js`
   - ☐ C `UIComponent.js`
   - ☐ D `BaseComponent.js`

3. Which type of UI control is used to embed a component into a component consumer?

   Choose the correct answer.

   - ☐ A *sap.ui.core.ComponentGroup*
   - ☐ B *sap.ui.core.ComponentContainer*
   - ☐ C *sap.ui.core.ComponentPanel*
   - ☐ D *sap.ui.core.ComponentArea*

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which types of components are provided by SAPUI5?

   Choose the correct answers.

   - [X] A Faceless Components
   - [X] B UI Components
   - [ ] C Backend Components
   - [ ] D Agile Components

   Correct. SAPUI5 supports the implementation of faceless components that are used to hide non-UI aspects behind a component façade. For the implementation of UI aspects, UI components are supported.

2. In which file is the Component controller implemented?

   Choose the correct answer.

   - [ ] A `Component.json`
   - [X] B `Component.js`
   - [ ] C `UIComponent.js`
   - [ ] D `BaseComponent.js`

   Correct. For each component, a `Component.js` file is implemented and assigned to a unique namespace.

3. Which type of UI control is used to embed a component into a component consumer?

   Choose the correct answer.

   - [ ] A *sap.ui.core.ComponentGroup*
   - [X] B *sap.ui.core.ComponentContainer*
   - [ ] C *sap.ui.core.ComponentPanel*
   - [ ] D *sap.ui.core.ComponentArea*

   Correct. A *sap.ui.core.ComponentContainer* embeds a UI component in a control tree.

© Copyright. All rights reserved.

UNIT 14 Worklist App Development

Lesson 1

Developing Worklist Apps 225

UNIT OBJECTIVES

- Develop a Worklist app

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 14
Lesson 1

Developing Worklist Apps

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Develop a Worklist app

sap.m.App

Apps are often composed of several pages and the user can drill down to detail pages and go back up again. The transition between the pages when navigating is often animated (for example, with horizontal slide animation).

SAPUI5 supports this pattern by providing the control *sap.m.App*. The default type of transition when navigating is `slide`. Other predefined options are `fade`, `flip`, and `show`.

sap.m.App inherits the navigation capabilities from the *sap.m.NavContainer* control.

- When developing a Worklist app, **sap.m.App** can be used as the root element:
  - *sap.m.App* adds certain header tags to the HTML page that are necessary for proper display.
  - *sap.m.App* offers functionality to navigate between pages with animations.
  - The content entities between which *sap.m.App* navigates can be of type *sap.m.Page*, *sap.ui.core.mvc.View*, *sap.m.Carousel* or any other control with full-screen/page semantics.

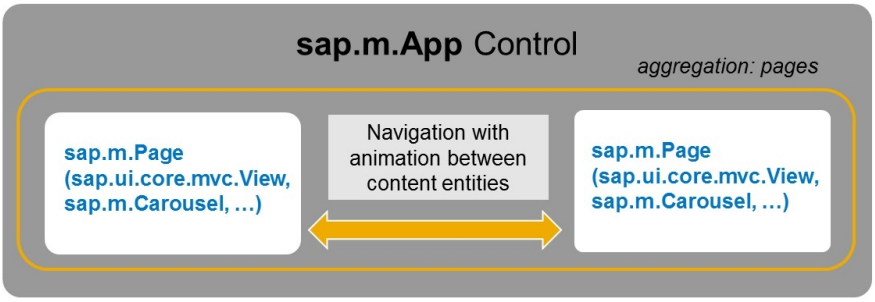


Figure 204: sap.m.App

Typical MVC Approach

The granularity of views can be chosen freely. One possible approach is to use one view per page. The following figure shows such a usage, a Model View Controller (MVC) concept used to decouple the different parts of the application. The figure is not intended to imply best practice, but rather to illustrate the interaction of views.

One XML view is used for the *sap.m.App* control, which forms the root level of the application. The remaining views (also XML views) are used for the pages.

The controls aggregated by the `pages` aggregation of *sap.m.App* receive navigation events, such as `beforeShow`. These navigation events are documented in the pseudo interface *sap.m.NavContainerChild*.

© Copyright. All rights reserved.

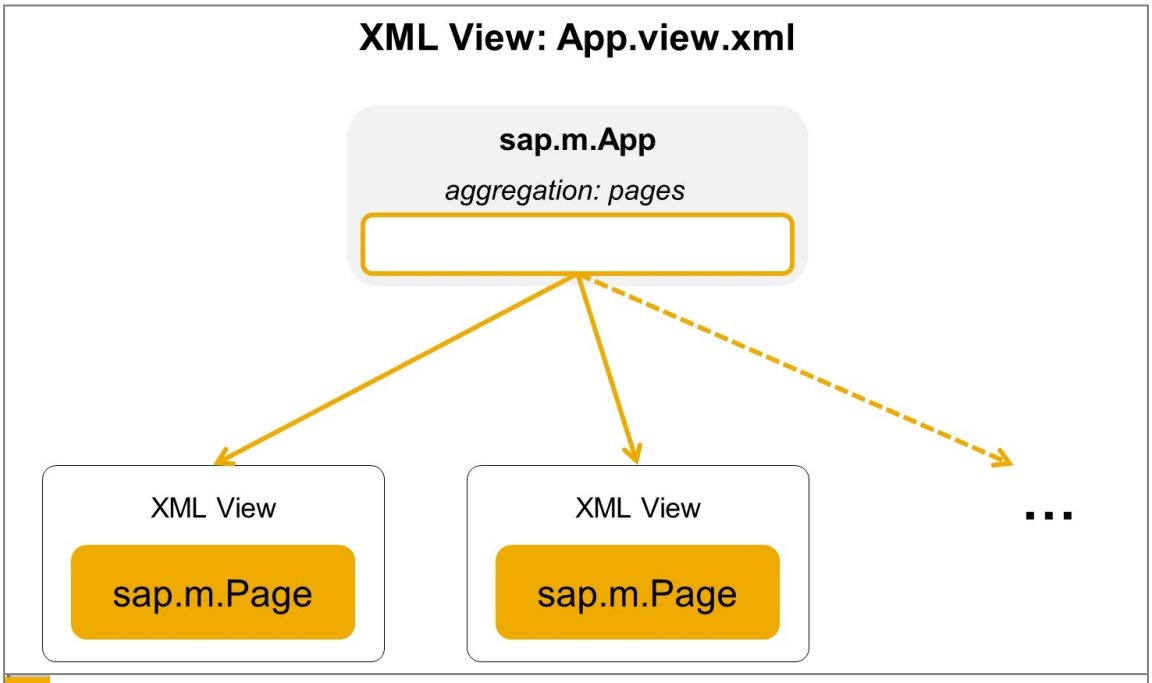


Figure 205: Typical MVC Approach

The App View

In the App view, put a *Shell* container around the *App* control and use it as the root element.

The use of the *Shell* is to control the width of the app on the desktop. On large monitors, a full-width rendering of an application can be too wide. Placing the app inside a *Shell* container gives you the appropriate left and right vertical margins. There are more options to customize the app, like setting a custom background image or color, and setting a custom logo. Check the API Reference documentation and the samples for more details.

- The *App View* contains the empty **App** tag.
- Place the **App** tag inside an *sap.m.Shell* to control the width of the app on the desktop: On large monitors, the *Shell* creates appropriate left and right vertical margins.
- Specify the *App View* in the *Application Descriptor* (*manifest.json*) as the root View to be opened.
- The pages are added automatically into the *App* control according to the current URL by the router.
- The router identifies the *App* control by means of the *id*.

App.view.xml

webapp > view > App.view.xml

You, a few seconds ago | 1 author (You)

```
1  <mvc:View controllerName="sap.training.fullscreen.controller.App"
2            xmlns:mvc="sap.ui.core.mvc"
3            displayBlock="true"
4            xmlns="sap.m">
5
6      <Shell id="shell">
7          <App id="app"/>
8      </Shell>
9
10 </mvc:View>
```

displayBlock="true" prevents vertical scrollbars appearing with *Views* that are set to 100% height.

Figure 206: The App View

© Copyright. All rights reserved.

Example Page

sap.m.Page provides a basic container for a mobile application screen. Typically, one page is displayed at any one time in landscape mode (or on tablets, depending on the layout, two pages may be displayed side-by-side).

- *sap.m.Page* provides an aggregation to 0..N other controls called *content*.

Overview.view.xml

webapp > view > Overview.view.xml

```
1  <mvc:View controllerName="sap.training.fullscreen.controller.Overview"
2            xmlns:mvc="sap.ui.core.mvc" xmlns="sap.m">
3      <Page title="{i18n>overviewPageTitle}">
4          <content>
5              <Table items="{ path : 'product>/Products', sorter : { path : 'Name' } }">
6                  <columns> <Column><Text text="{i18n>columnName}"/></Column>
7                            <Column><Text text="{i18n>columnDescription}"/></Column>
8                  </columns>
9                  <items>
10                     <ColumnListItem press="onPress" type="Navigation">
11                         <cells> <ObjectIdentifier title="{product>Name}"/>
12                                 <Text text="{product>Description}"/>
13                         </cells>
14                     </ColumnListItem>
15                 </items>
16             </Table>
17         </content>
18     </Page>
19 </mvc:View>
```

The *title* attribute is displayed in a header section on top of the content.

Figure 207: Example Page

index.html

In the index.html file, the `<style>` tag is used to set a height of 100%. This ensures that the component uses the entire screen height. In application containers like the SAP Fiori launchpad, this is done automatically.

- Use the declarative API to embed the worklist component into the HTML page.

index.html

webapp > index.html > ...

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      [...]
5      <style>
6          html, body, body > div, #container, #container-uiarea {
7              height: 100%;
8          }
9      </style>
10     [...]
11 </head>
12 <body class="sapUiBody sapUiSizeCompact" id="content">
13     <div data-sap-ui-component data-name="sap.training.fullscreen"
14                                data-id="container"
15                                data-settings='{"id" : "sap.training.fullscreen"}'
16                                data-handle-validation="true"></div>
17 </body>
18 </html>
```

Make use of the whole screen height

Figure 208: index.html

Routing and Navigation

The following figure describes the hash-based navigation concept.

© Copyright. All rights reserved.

- SAPUI5 allows building apps where the navigation is done by changing the hash.
- A *route* is used to notify your application that the hash has changed to a certain value. The pattern of the *route* is matched against the hash. If the pattern matches, the *route* provides the data passed over the hash to a defined handler.
- Hash-based navigation enables you to use the current hash as a bookmarkable URL.

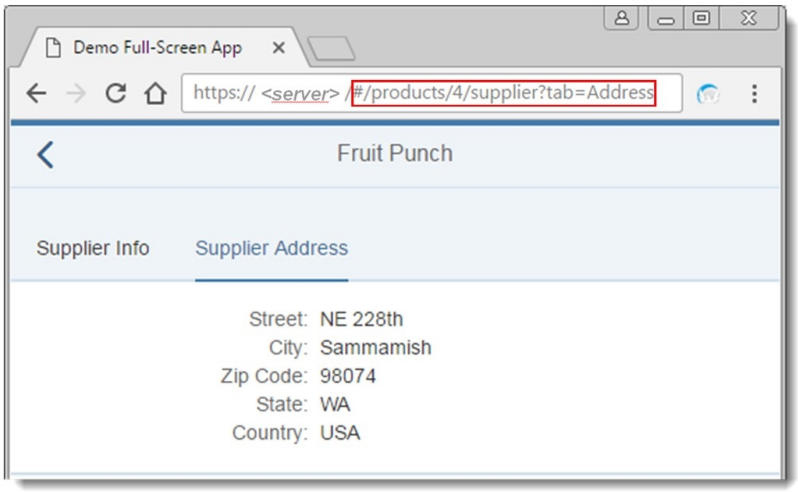


Figure 209: Routing and Navigation

Routing Configuration

The pattern of a route is basically the hash part of the URL that matches the route.

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 a different syntax.

- SAPUI5 applications can use a so-called **router** to dispatch hash-based URLs to one or more views of the app.
- To configure the ***router***, add a *routing* section to the *sap.ui5* section in *manifest.json*.
- There are three properties to control the application flow when the user triggers a navigation action or opens a link to the application directly.

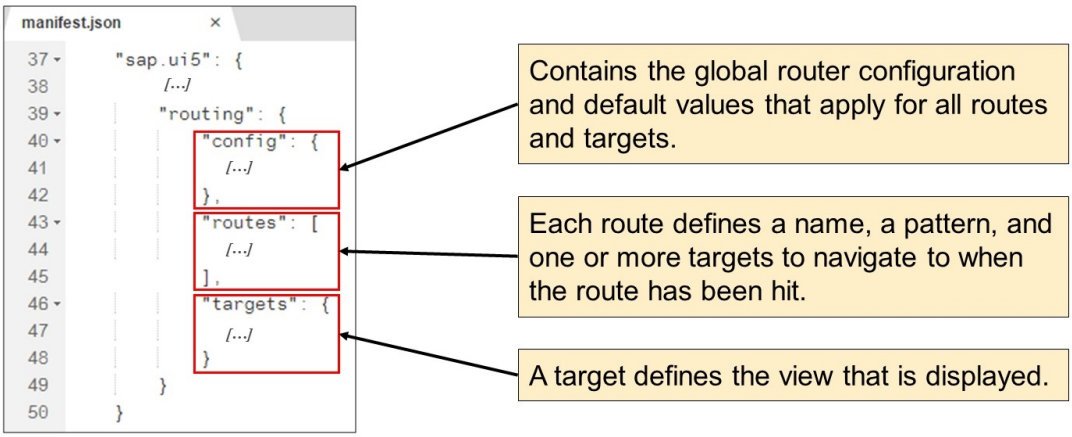


Figure 210: Routing Configuration

© Copyright. All rights reserved.

Config Section – Example

The default value for `routerClass` is `sap.ui.core.routing.Router`. In this example, the `routerClass` is set to `sap.m.routing.Router`, because we implement an app based on `sap.m`. 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 every route or target created by the `sap.m.routing.Router`. Refer to the API Reference documentation for more information.

All parameters in the *config* section can be overruled in the individual route and target definitions, if needed.

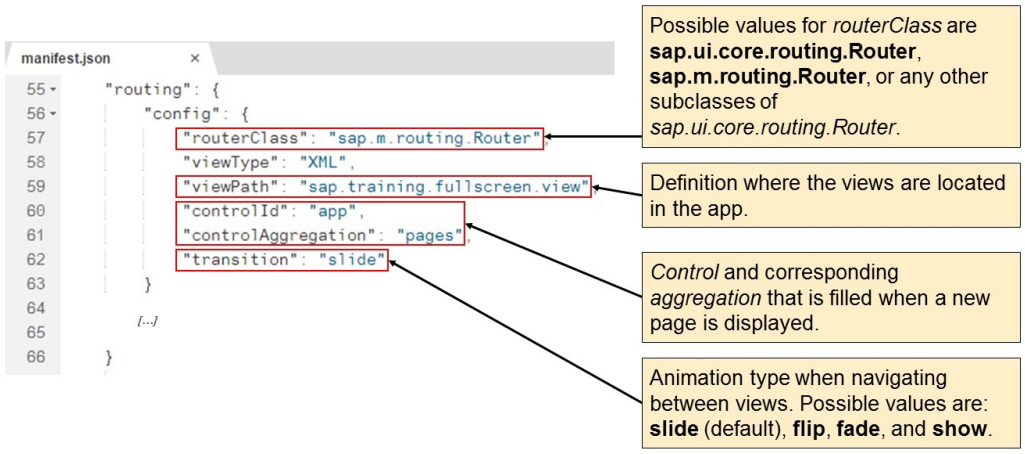


Figure 211: Config Section - Example

> Note:
> In version 1.69, the default value `slide` for the transition type was visually updated to behave as a slide & fade animation, and the classic slide animation was no longer an option. In version 1.74, the previous slide behavior was added as a new type of transition. To use it, set the transition type to `baseSlide`. For more information, refer to the API Reference.

Routes Section – Example

The following figure provides an example of the `routes` section in a `manifest.json` file.

The sequence of the routes in the *routes* section is important because only the first matched route is used by the router.

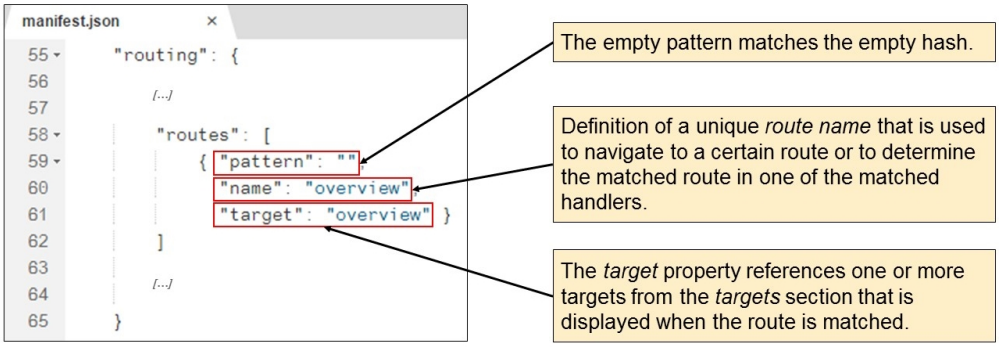


Figure 212: Routes Section - Example

© Copyright. All rights reserved.

Targets Section – Example

The following figure shows an example of the `targets` section of the `manifest.json` file.

Whenever a `target` is displayed, the corresponding view is loaded and added to the aggregation configured with the `controlAggregation` option of the control. The control is configured using *controlId*.

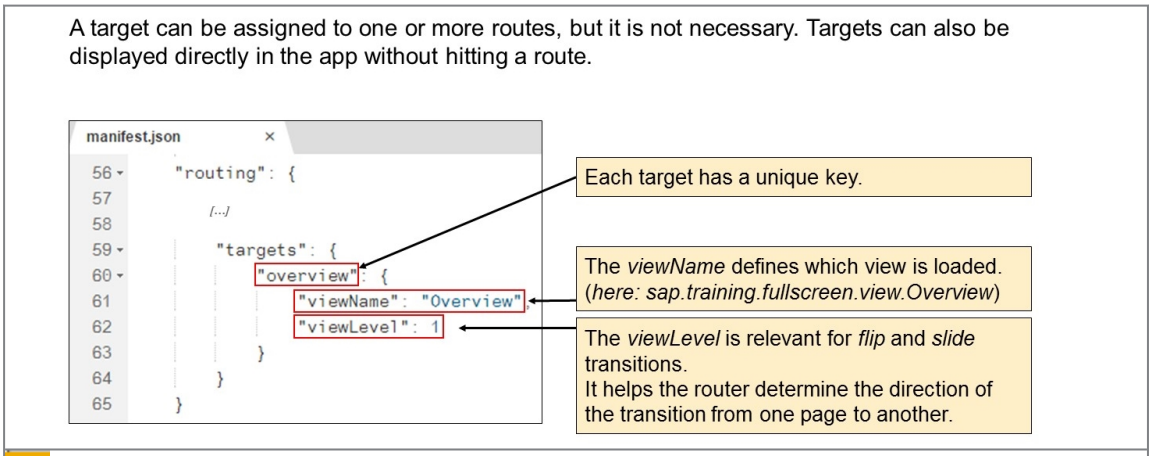


Figure 213: Targets Section - Example

Initializing the Router

You do not need to instantiate the *Router* manually, it is instantiated automatically, based on the configuration in `manifest.json` and assigned to the *Component*.

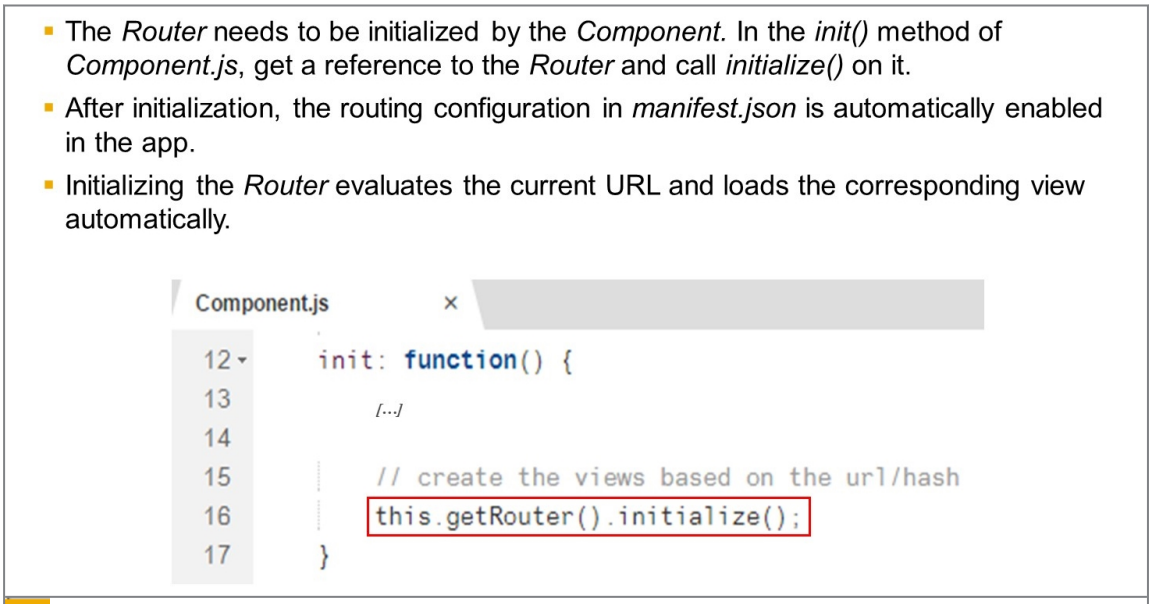


Figure 214: Initializing the Router

Catching Invalid Hashes

The following figure shows an example of the type of behavior desired when a user tries to access an invalid route.

© Copyright. All rights reserved.

If a user tries to access an invalid pattern, which does not match any of the configured routes, the user should get an indication that the requested resource was not found.

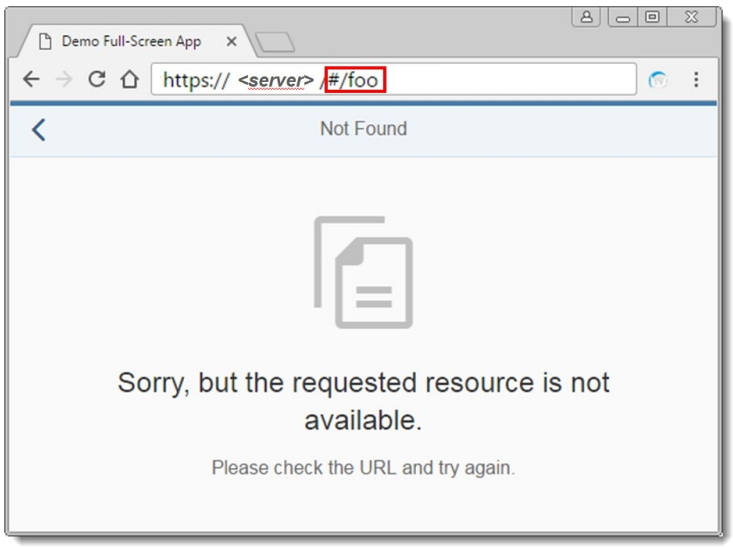


Figure 215: Catching Invalid Hashes

Catching Invalid Hashes – Configuration

The following figure shows how to set up behavior for catching invalid hashes.

manifest.json

```
56    "routing": {
57        "config": {
58            [...]
59
60            "bypassed": {
61                "target": "notFound"
62            }
63        },
64
65        [...]
66        "targets": {
67            [...]
68
69            "notFound": {
70                "viewName": "NotFound",
71                "transition": "show"
72            }
73        }
74    }
```

Using the *bypassed* property, you tell the *Router* to display a special *target* (*here: notFound*) in case no route was matched to the current hash.

The *notFound* target configures a *NotFound* view with a *show* transition.

Figure 216: Catching Invalid Hashes - Configuration

NotFound View

Routing related error messages can be displayed in a *Notfound* view.

© Copyright. All rights reserved.

Use the *sap.m.MessagePage* control to display routing related error messages:

NotFound.view.xml

```
1  <mvc:View
2        controllerName=" [...] "
3        xmlns="sap.m"
4        xmlns:mvc="sap.ui.core.mvc">
5     <MessagePage
6        title="{i18n>notFoundMessageTitle}"
7        text="{i18n>notFoundMessageText}"
8        description="{i18n>notFoundMessageDescription}"
9        showNavButton="true"
10       navButtonPress="onNavBack"/>
11 </mvc:View>
```

A back button is rendered in the header of the page.

navButtonPress is fired when the back button is pressed.

i18n.properties

```
27 # NotFound View
28 notFoundMessageTitle=Not Found
29 notFoundMessageText=Sorry, but the requested resource is not available.
30 notFoundMessageDescription=Please check the URL and try again.
```

Figure 217: NotFound View

Implementing the Back Button on the Notfound Page

The following figure shows how to implement a *Back* button on the *Notfound* page.

NotFound.controller.js

```
3    getRouter: function() {
4        return sap.ui.core.UIComponent.getRouterFor(this);
5    },
6
7    onNavBack: function(oEvent) {
8        var oHistory, sPreviousHash, oRouter;
9
10       oHistory = sap.ui.core.routing.History.getInstance();
11       sPreviousHash = oHistory.getPreviousHash();
12
13       if (sPreviousHash !== undefined) {
14           window.history.go(-1);
15       } else {
16           oRouter = this.getRouter();
17           oRouter.navTo("overview", true /*no history*/ );
18       }
19   }
```

Returns a reference to the *Router* instance.

Check if there is a previous hash value in the app history:

- If so, redirect to the previous hash via the browser's native History API.
- If not, use the *Router* to navigate to the route *overview* which is the *Overview* view.

Name of the *route*.

true: The hash is replaced (no browser history entry).
false: The hash is set (browser history entry).

Figure 218: Implementing the Back Button on the Notfound Page

URL Patterns for Routes

The following figure shows an example of a *pattern* parameter for a route and defines the allowable content of a pattern.

© Copyright. All rights reserved.

manifest.json

```
56  "routing": {
57      [...]
58      "routes": [
59          [...]
60          {
61              "pattern": "products/{productId}",
62              "name": "product",
63              "target": "product"
64          }
65      ]
66      [...]
67  }
```

- The *pattern* parameter of a *Route* defines the string that is matched against the hash.
- A *pattern* may consist of the following:
  - Hardcoded parts
  - Mandatory parameters
  - Optional parameters
  - Mandatory query parameters
  - Optional query parameters

Figure 219: URL Patterns for Routes (1 of 3)

The following two figures show the valid pattern types, together with examples and notes.

Pattern Type	Example	Notes
hardcoded parts	**"products/settings"**	▪ This pattern only matches if the hash of the browser is *products/settings*. ▪ No arguments are passed to the events of the *Route*.
mandatory parameters	**"products/{productId}"**	▪ *{productId}* is a mandatory parameter. ▪ For example, the following hashes would match: *products/5* and *products/3*. ▪ The *matched* event of the *Route* gets 5 or 3 passed as *productId* in its arguments. ▪ The hash *products/* does not match.
optional parameters	**"products/{productId}/detail/:detailId:"**	▪ *:detailId:* is an optional parameter. ▪ For example, the following hashes would match: *products/5/detail* and *products/3/detail/2*.

Figure 220: URL Patterns for Routes (2 of 3)

Pattern Type	Example	Notes
mandatory query parameters	**"products{?query}"**	▪ *{?query}* allows you to pass queries with any parameters. ▪ For example, the following hashes match: *products?first=value1* *products?first=value1&second=value2*
optional query parameters	**"products/{productId}/supplier:?query:"**	▪ *:?query:* is an optional query parameter. ▪ For example, the following hashes match: *products/5/supplier?tab=Address* *products/3/supplier*

Figure 221: URL Patterns for Routes (3 of 3)

Navigate to Routes with Mandatory Parameters – Example

In the example shown in the following figure, we implement a feature that allows the user to click a product in the list to see additional details of the product.

The detail page must read the ID of the product from the URL to fetch and display the product data from the server. If the product was not found, for example, because an invalid product ID

© Copyright. All rights reserved.

was passed, we want to inform the user by displaying the *notFound* target. In addition, the back navigation must also work for the detail page.

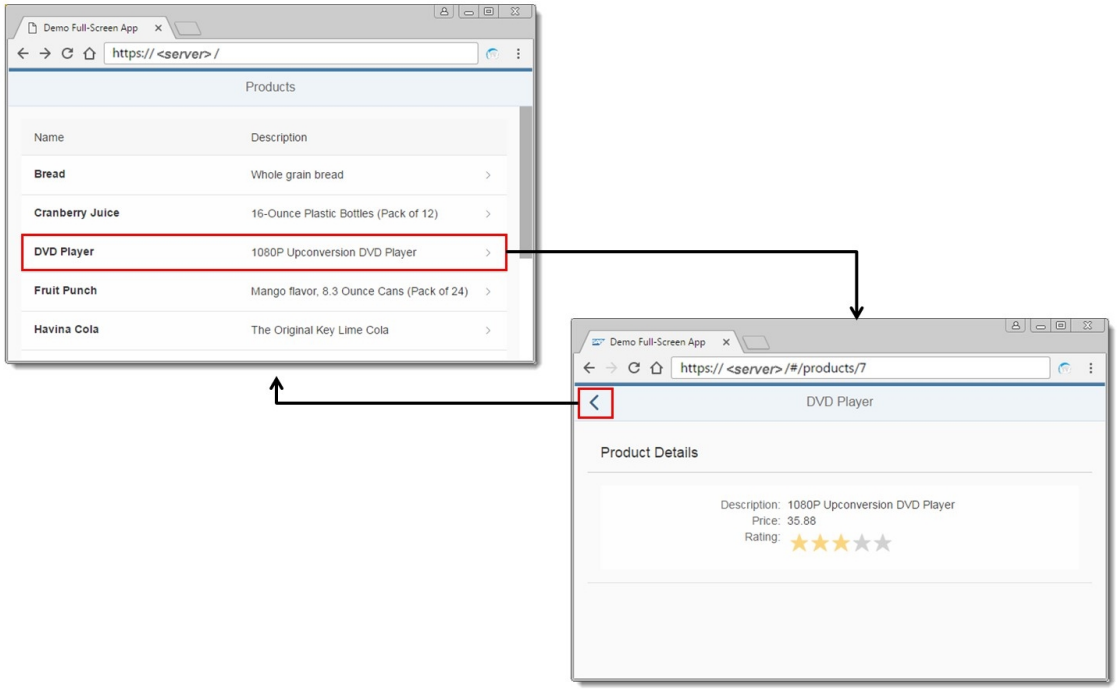


Figure 222: Example - Navigate to Routes with Mandatory Parameters

Definition of the Route

Each *Product* entity is identified by an ID.

The `{productId}` part of the pattern is mandatory, as indicated by the curly brackets. The hash that contains an actual product ID is matched against that pattern at runtime.

The target of the route is *product*. We create the target product with *viewLevel* 2. With that, we ensure that we have the correct slide animation direction.

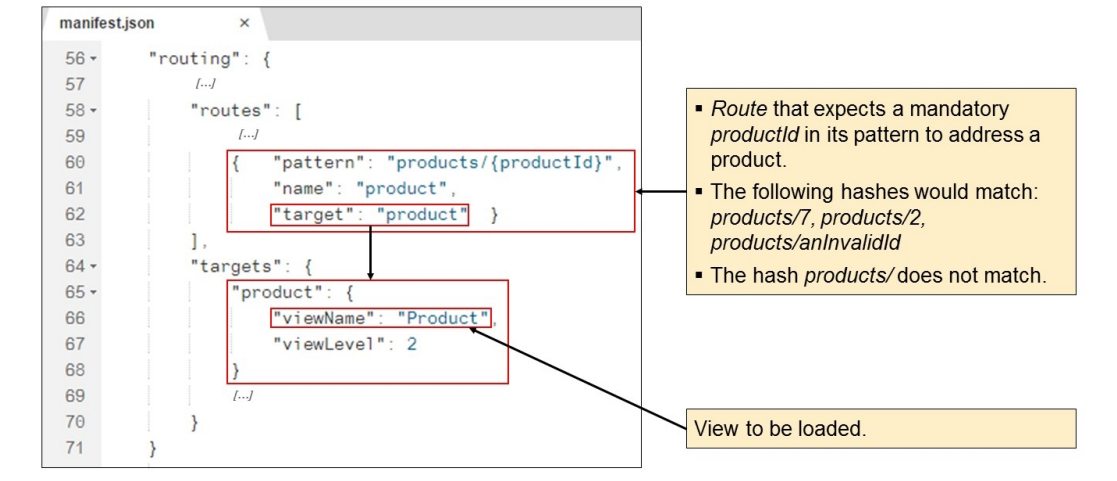


Figure 223: Definition of the Route

© Copyright. All rights reserved.

Handler for the Press Event in the View Controller

The following figure shows how to implement the handler for the `press` event in the view controller.

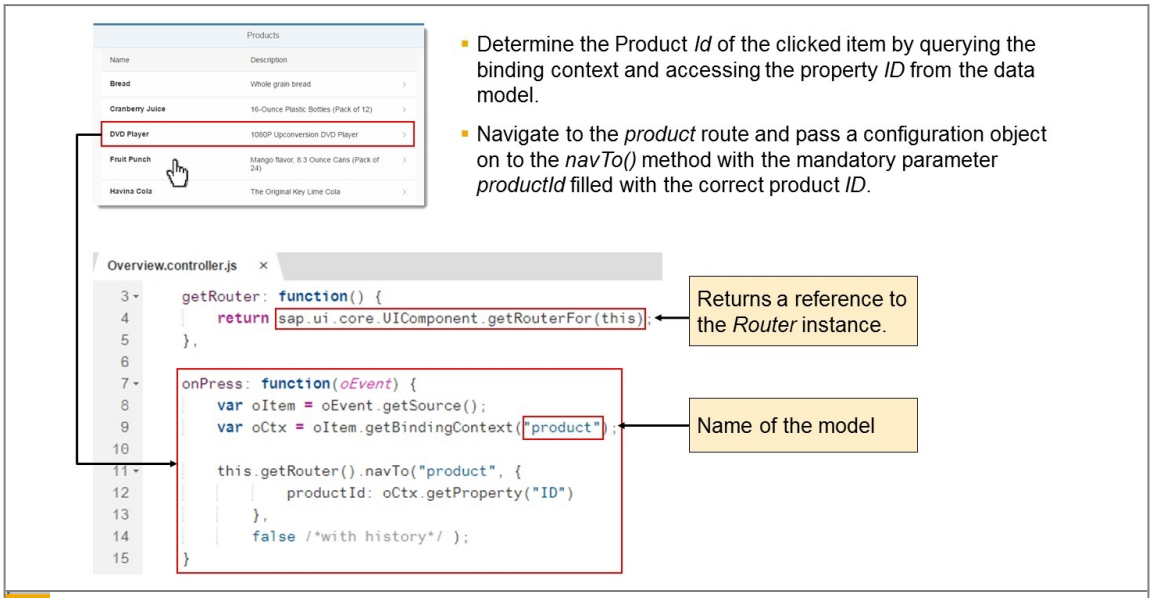


Figure 224: Handler for the Press Event in the View Controller

Detect the Product in the Product View Controller

In the *Product View Controller*, we want to detect which product is displayed to show the product's data in the view.

The `ODataModel` handles the necessary data requests to the backend 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.

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 do not use the latter because we would have to implement an additional check for making sure that the current route is the route that has been matched. We want to avoid this extra overhead and register on the *route* instead.

© Copyright. All rights reserved.

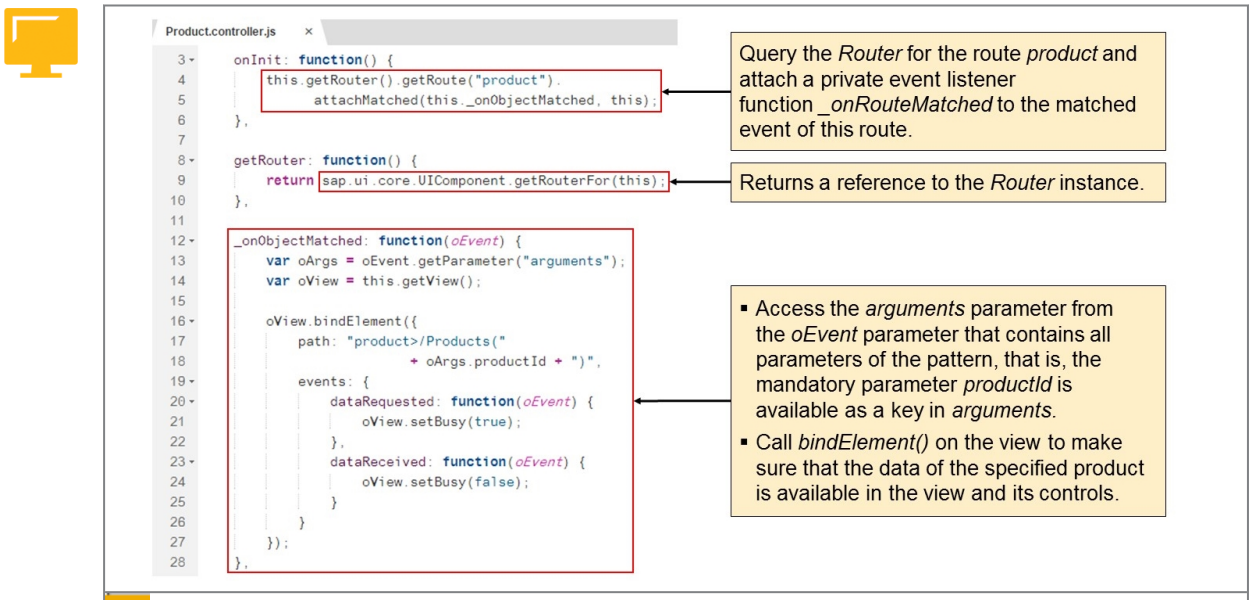


Figure 225: Detect the Product in the Product View Controller

Implementation of the Product View

This *Product* view displays master data for a product in a *Panel* with a *SimpleForm* control: *description*, *price*, and *rating*. The data comes from a relative data binding that is set on the view level in the *Product* view controller.

Figure 226: Implementation of the Product View

Implementing the Back Button on the Product Page

The displayed `onNavBack()` function is the same as the corresponding function on the controller of the NotFound page (as described earlier). Also, the `getRouter()` function shown is duplicated in the view controllers.

Because you should not repeat yourself, it is better to create a base controller from which all other controllers inherit. The base controller should implement the set of functions that are reused by its subclasses (for example, `onNavBack()` and `getRouter()`) instead of adding these functions directly to the view controllers.

© Copyright. All rights reserved.

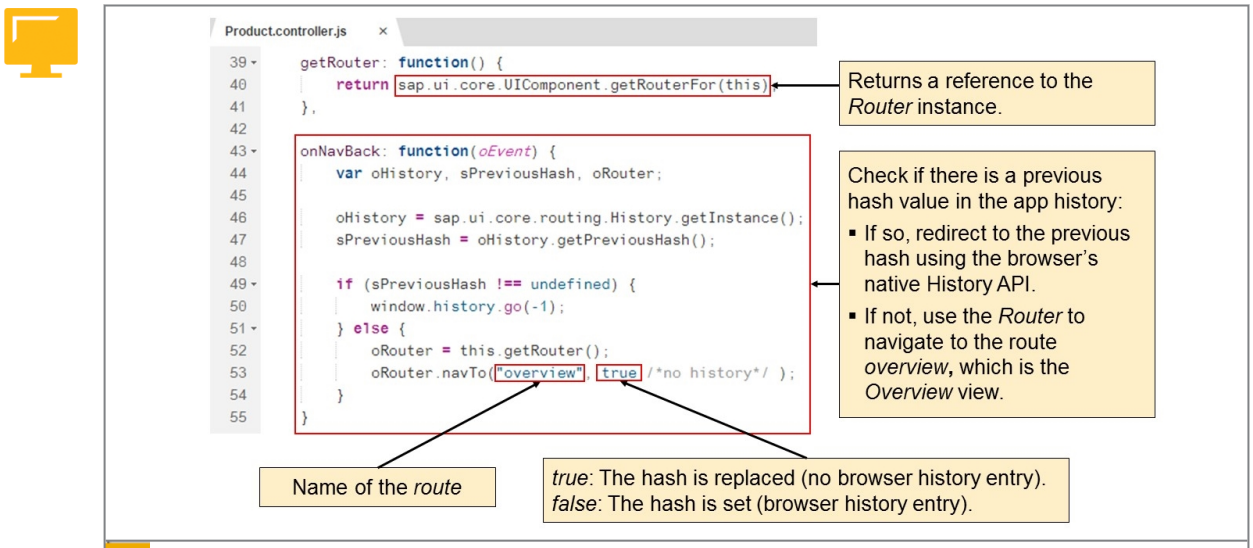


Figure 227: Implementing the Back Button on the Product Page

Display the NotFound Page for Invalid Product IDs

You can display targets manually without referencing them in a navigation route. Good examples of this are temporary errors, switching to an edit page for a business object, or going to a *Settings* page.

In the `_onBindingChange()` handler, we get a reference to the *Targets* helper object of the router and simply call `display("notFound")`. The view associated with the target with the name `notFound` from the routing configuration is displayed by the router without changing the hash.

The *sap.m.routing.Targets* object is retrieved by calling `getTargets()` on the router. However, you could also get a reference to the *sap.m.routing.Targets* object by calling `this.getOwnerComponent().getRouter().getTargets()` or `this.getOwnerComponent().getTargets()`.

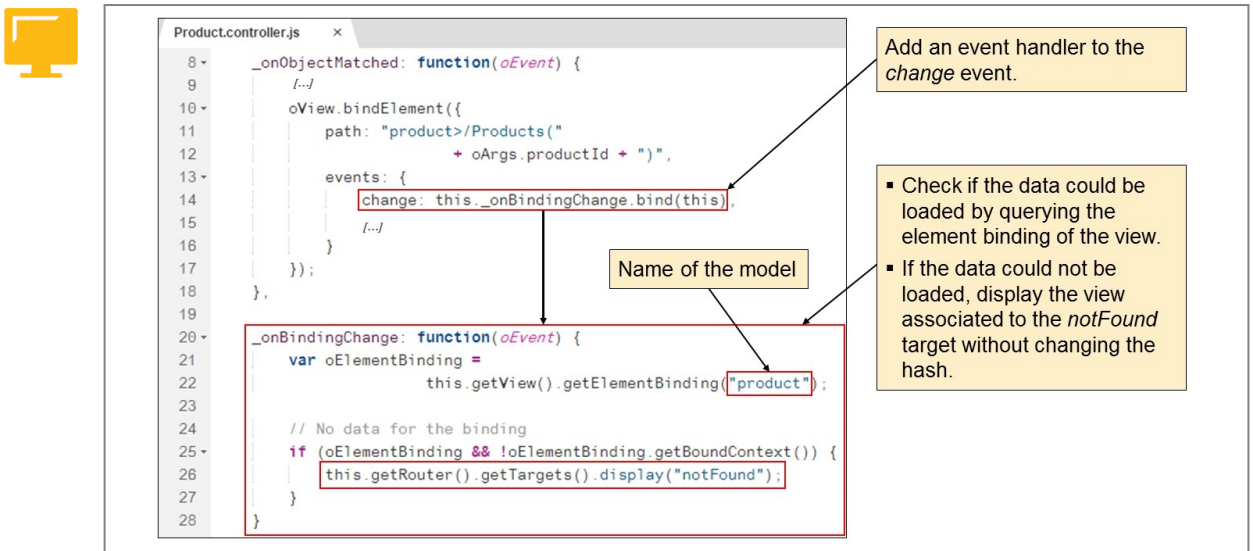


Figure 228: Display the NotFound Page for Invalid Product IDs

© Copyright. All rights reserved.

LESSON SUMMARY

You should now be able to:

- Develop a Worklist app

© Copyright. All rights reserved.

Learning Assessment

1. Which root element can be used when developing a worklist app?

   Choose the correct answer.

   - ☐ A *sap.m.App*
   - ☐ B *sap.m.FullScreenApp*
   - ☐ C *sap.m.BaseApp*
   - ☐ D *sap.m.SplitApp*

2. Which function invocation is necessary to set up the routing inside a Worklist app?

   Choose the correct answer.

   - ☐ A `this.getRouter().initialize();`
   - ☐ B `this.getRouter().setup();`
   - ☐ C `this.getRouter().init();`
   - ☐ D `this.getRouter().registerRoutes();`

3. How do you catch invalid hashes in your routing configuration?

   Choose the correct answer.

   - ☐ A Handle the invalid hashes during component initialization and forward them to a central routing target.
   - ☐ B Use the bypassed attribute inside the routing configuration.
   - ☐ C Register an event handler to handle invalid hashes, and navigate to the correct target when the `invalidHashOccurs` event is raised.

© Copyright. All rights reserved.

Learning Assessment - Answers

1. Which root element can be used when developing a worklist app?

   Choose the correct answer.

   [X] A *sap.m.App*

   [ ] B *sap.m.FullScreenApp*

   [ ] C *sap.m.BaseApp*

   [ ] D *sap.m.SplitApp*

   Correct. The *sap.m.App* control is one of the possible root elements of a SAPUI5 mobile app. *sap.m.App* inherits from *sap.m.NavContainer*, which provides its navigation capabilities. *sap.m.App* adds certain header tags to the HTML page that are considered useful for mobile apps.

2. Which function invocation is necessary to set up the routing inside a Worklist app?

   Choose the correct answer.

   [X] A `this.getRouter().initialize();`

   [ ] B `this.getRouter().setup();`

   [ ] C `this.getRouter().init();`

   [ ] D `this.getRouter().registerRoutes();`

   Correct. To attach the referenced router object to the hash changer, you must invoke the `this.getRouter().initialize()` function.

© Copyright. All rights reserved.

3. How do you catch invalid hashes in your routing configuration?

   Choose the correct answer.

   - [ ] A Handle the invalid hashes during component initialization and forward them to a central routing target.
   - [x] B Use the bypassed attribute inside the routing configuration.
   - [ ] C Register an event handler to handle invalid hashes, and navigate to the correct target when the `invalidHashOccurs` event is raised.

   Correct. Developers can catch invalid hashes using the bypassed attribute in the routing configuration.

© Copyright. All rights reserved.

© Copyright. All rights reserved.

UNIT 15 Appendix - OData V4 Model

Lesson 1

Instantiating a Model 245

Lesson 2

Using Data Binding 249

Lesson 3

Filtering and Sorting Model Data 261

Lesson 4

Consuming OData Operations 267

Lesson 5

Implementing Batch Control 273

Lesson 6

Creating and Deleting Entities 277

UNIT OBJECTIVES

- Instantiate an OData V4 Model
- Use Data Binding
- Filter and sort Model Data
- Consume OData Operations
- Implement Batch Control
- Create and delete entities

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 15
Lesson 1

Instantiating a Model

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Instantiate an OData V4 Model

OData V4 Model instantiation

The `sap.ui.model.odata.v4.ODataModel` is the model implementation for consuming an OData V4 service.

The OData V4 model supports the following:

- Read access
- Updating properties of OData entities via two-way-binding
- Deleting entities
- Operation (function and action) execution
- Grouping data requests in a batch request
- Server-side sorting and filtering

The OData V4 model only supports data access using bindings. It does not provide any direct access to the data.

The OData V4 model is primarily designed for OData V4 services. Nevertheless, OData V2 services may be used through an adapter as well. The adapter takes care to convert the metadata and the data in a way that the application developer writes its application as they 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 full list of the limitations can be found in the documentation. One restriction is that the OData V4 model can consume an OData V2 service for read scenarios only.

SAPUI5 uses the concept of a "current language". This language is automatically propagated to the OData service by the OData V4 model.

Smart Controls (`sap.ui.comp`) and Fiori Elements do not support the SAPUI5 OData V4 model in general as well as controls like `TreeTable` and `AnalyticalTable`.

One OData V4 model instance can only cover one OData service.

The property `serviceUrl` contains the root URL of the service to request data from. The path part of the URL must end with a forward slash according to OData V4 specification ABNF, rule "serviceRoot".

© Copyright. All rights reserved.

- When creating an OData V4 model instance, the only parameter actually needed is a map.
- This map must contain at least the properties *serviceUrl* and *synchronizationMode.*
- *synchronizationMode* controls synchronization between different bindings which refer to the same data for a case where data changes in one binding. It must be set to *'None'* which means bindings are not synchronized at all; all other values are not supported and lead to an error.

MyModule.js ×

```
sap.ui.define(["sap/ui/model/odata/v4/ODataModel"], function (ODataModel) {

    return {
        createODataModel: function () {

            var oModel = new ODataModel({
                serviceUrl: "/ODATA_ORG/TripPinRESTierService/(S(sv1mz4tmvvmdy4bxrxry2vke))/",
                synchronizationMode: "None"
            });

            return oModel;
        }
    };
});
```

Figure 229: OData V4 Model Instantiation

Instantiating an OData V4 Model using manifest.json

The code sample in the figure, Instantiating an OData V4 Model using manifest.json, shows the parts of the Application Descriptor (`manifest.json`) that are relevant for instantiating an OData V4 model.

© Copyright. All rights reserved.

manifest.json

```
1  {
2      "sap.app": {
3
4          "dataSources": {
5              "demoService": {
6                  "uri": "/ODATA_ORG/TripPinRESTierService/(S(sv1mz4tmvvmdy4bxrxry2vke))/",
7                  "type": "OData",
8                  "settings": {
9                      "odataVersion": "4.0",
10                     "localUri": "localService/metadata.xml"
11                 }
12             }
13         }
14     },
15
16     "sap.ui5": {
17
18         "models": {
19             "": {
20                 "type": "sap.ui.model.odata.v4.ODataModel",
21                 "settings": {
22                     "synchronizationMode": "None"
23                 },
24                 "dataSource": "demoService"
25             }
26         }
27     }
28 }
```

Figure 230: Instantiating an OData V4 Model using manifest.json

LESSON SUMMARY

You should now be able to:

- Instantiate an OData V4 Model

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 15
Lesson 2

Using Data Binding

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use Data Binding

Binding Modes and Binding Types

The OData V4 model supports one-time binding, one-way binding and two-way binding. 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.

Bindings connect SAPUI5 view elements to model data, allowing changes in the model to be reflected in the view element and vice versa.

Every resource path (relative to the service root URL, no query options) is a valid data binding path within an OData model if a leading slash is added. For example, you can use `"/People('angelhuffman')"` to access an entity instance with key "angelhuffman".

The OData V4 model supports the following **binding modes**:

- **two-way** (default binding mode), **one-way**, and **one-time**

The OData V4 model supports the following **binding types**:

- **List bindings**
  represent a collection of OData entities (such as */People)*, or a collection of complex or primitive types
- **Context bindings**
  represent a single entity (such as */People('angelhuffman')*), or a structural property with complex type
- **Property bindings**
  represent a single, primitive type property in an entity (such as */People('angelhuffman')/FirstName*) or in a complex type

Figure 231: Binding Modes and Binding Types

Absolute and Relative Bindings

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.

© Copyright. All rights reserved.

The figure, Absolute and Relative Bindings, shows an absolute list binding; A table's `items` aggregation is bound to **`/People`**. The columns define relative bindings with paths **`FirstName`**, **`LastName`**, and **`Age`**, with the absolute list binding as parent binding.

You can suspend an absolute list binding or context binding with its `suspend` method. A suspended binding does not send data service requests. You can resume a suspended list or context binding with its `resume` method.

Typical use cases for suspend and resume methods are situations where you may want to suppress OData requests triggered by an OData V4 binding for a certain period of time. In this way you can trigger read requests for controls in the view later; not when the view is initialized. This could be useful for value help dialogs, for example, to delay the data service request until the dialog is displayed. The controller code to open the dialog resumes the list binding (suspended by default) and thus triggers the request.

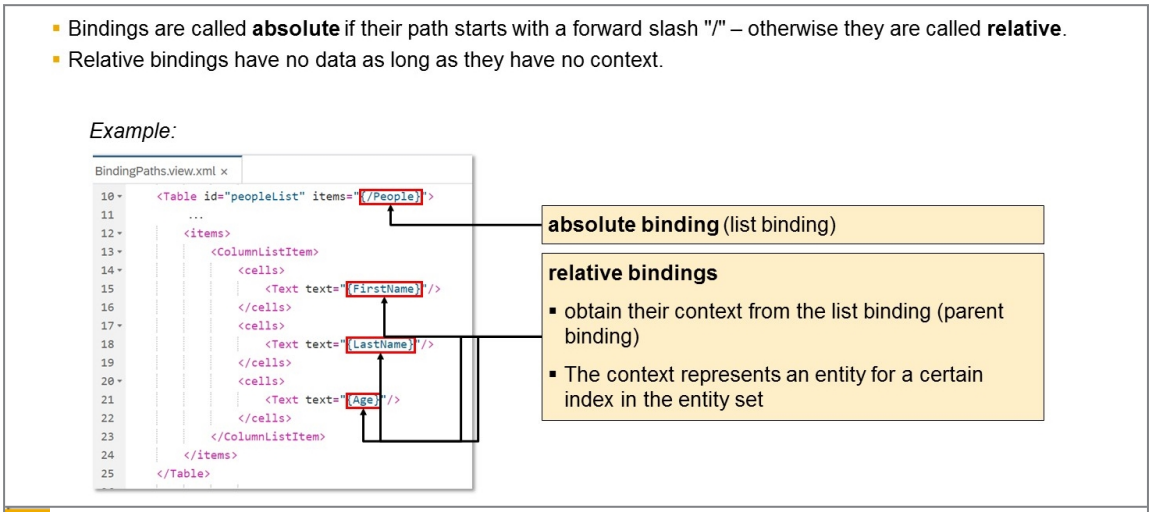


Figure 232: Absolute and Relative Bindings

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.
- Binding-specific parameters, which start with `$$` and influence the behavior of the binding (such as `$$groupId`, `$$updateGroupId`, `$$operationMode`).

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 triggered 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-`.
- 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`. The OData V4 model allows for binding the inline count of the entity collection read by an

© Copyright. All rights reserved.

`ODataListBinding`, which has the parameter `$count` set to **`true`**. You just have to define a relative property binding with path `$count`.

All query options, except query option `$count`, can be specified with a string value. In addition to strings, the following alternatives are possible:

- `$select` can be specified as an array of strings where each string specifies a select item, or the value '*' to select all properties.
- `$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:
  1. `true` or `null` if no expand options are required
  2. 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.

To change the binding's parameters and refresh the binding, the JavaScript methods `sap.ui.model.odata.v4.ODataContextBinding#changeParameters` and `sap.ui.model.odata.v4.ODataListBinding#changeParameters` can be used. They allow you 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. Changing, adding, or deleting is possible at the same time. The binding is refreshed as soon as the parameter changes are applied.

When creating a binding, you can provide a parameter map which can contain the following:

- **OData query options**
- **Binding-specific parameters** (such as *$$groupId*, *$$updateGroupId*, *$$operationMode*)

Parameters.view.xml ×

```
10  <Table id="peopleList"
11         items="{ path: '/People',
12                  parameters: {
13                                $select: [ 'FirstName', 'LastName', 'Age'],
14                                $filter : 'LastName ge \'M\'',
15                                $orderby : 'LastName asc',
16                                $expand: { BestFriend: { $select: ['FirstName', 'LastName'] } }
17                              }
18                  }"
19  >   ···   </Table>
```

Figure 233: Parameters

Relative Binding with Data from Parent Binding

Once it has a context, a relative binding creates a data service request in the following cases:

- The relative binding has parameters, such as OData query options like `$select`. If you want a relative binding to create its own data service request, you specify a `$select` query option as binding parameter in order to reduce the response size. If you do not want to specify a `$select`, but still want the binding to create its own request, you can specify

© Copyright. All rights reserved.

`$select` with value `*`. Alternatively, you can set the binding-specific parameter `$$ownRequest` to **`true`** in the parameter map for the relative binding. (All values other than `true` lead to an error). This also ensures the binding uses an own service request to read data.

- You specify a dynamic filter or dynamic sorter for a list binding and use `sap.ui.model.odata.OperationMode.Server`.

In all other cases, a relative binding reads data from its parent binding that created the context.

In the example shown in the figure, Relative Binding with Data from Parent Binding, the lower table for the trips reads data from the parent binding, that is, from the list binding of the upper table whose `items` aggregation is bound to **`/People`**. To provide the required trips data, the list binding of the upper table uses the `$expand` and `$select` query options as binding parameters.

Per default, a relative binding reads data from its parent binding that created the context, i.e. it does not create its own data service request.

DataFromParentBinding.view.xml

```
10  <Table id="peopleList" mode="SingleSelectMaster" growing="true" growingThreshold="5"
11         selectionChange="onPersonChange"
12         items="{ path: '/People',
13                  parameters: { $expand: { Trips: { $select: ['Name', 'Description', 'StartsAt', 'EndsAt'] } } }
14                }"
15  >        </Table>
16
17  <Table id="tripList" items="{Trips}" growing="true" growingThreshold="5">        </Table>
```

relative binding which reads data from parent binding

```
onPersonChange: function (oEvent) {
    var oBindingContext = this.byId("peopleList").getSelectedItem().getBindingContext();
    this.byId("tripList").setBindingContext(oBindingContext);
}
```

Figure 234: Relative Binding with Data from Parent Binding

Relative Binding with own Data Service Request

In the example shown in the figure, Relative Binding with own Data Service Request, the lower table for the trips still has a relative binding. As it now has parameters defined, it triggers its own data service request once it receives its binding context.

Bindings that 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.

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. You can refresh all bindings with `ODataModel.refresh`.

© Copyright. All rights reserved.

If you want a relative binding to create its own data service request, you can specify a *$select* query option as binding parameter (see code example below).

DataFromOwnRequest.view.xml ×

```
10    <Table id="peopleList" mode="SingleSelectMaster" growing="true" growingThreshold="5"
11        selectionChange="onPersonChange"
12        items="{/People}"
13    >   ...   </Table>
14
15    <Table id="tripList" growing="true" growingThreshold="5"
16            items="{ path: 'Trips',
17                     parameters: { $select: ['Name', 'Description', 'StartsAt', 'EndsAt'] } }"
18    >   ...   </Table>
```

relative binding with own data service request

```
onPersonChange: function (oEvent) {
    var oBindingContext = this.byId("peopleList").getSelectedItem().getBindingContext();
    this.byId("tripList").setBindingContext(oBindingContext);
}
```

Figure 235: Relative Binding with own Data Service Request

Automatic Determination of $expand and $select

Auto-$expand/$select has the following advantages:

- You don't have to add or change `$select` or `$expand` in the binding parameters yourself.
- Auto-$expand/$select only selects data needed for the UI, so that you get a minimal response size and improved performance.

It is still possible to specify `$expand` and `$select` in the binding parameters. This is useful if you need to access properties that are not bound on the UI. When auto-$expand/$select is switched on, 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:

- If the request for the parent binding is not sent and the child binding has only OData system query options in its parameters.
- 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): `People?`
`$select=FirstName,LastName,UserName&`
`$expand=BestFriend($select=FirstName,LastName,UserName)`

```
<Table items="{/People}">
        ...
    <items>
        <ColumnListItem>
```

© Copyright. All rights reserved.

```
            <cells>
                <Text text="{FirstName}"/>
                <Text text="{LastName}"/>
                <Text text="{BestFriend/FirstName} {BestFriend/
LastName}"/>
            </cells>
        </ColumnListItem>
    </items>
</Table>
```

- 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.
- auto-$expand/$select is switched on by setting the flag *autoExpandSelect* during model construction.

```
var oModel = new ODataModel({
    serviceUrl: "        ...        ",
    synchronizationMode: "None",
    autoExpandSelect: true
});
```

manifest.json ×

```
48    "sap.ui5": {
49
50        "models": {
51            "": {
52                "type": "sap.ui.model.odata.v4.ODataModel",
53                "settings": {
54                    "synchronizationMode": "None",
55                    "autoExpandSelect": true
56                },
57                "dataSource": "demoService"
58            }
59        },
```

Figure 236: Automatic Determination of $expand and $select

Refresh a Single Entity

The method `onRefreshSelectedPerson`, shown in the figure, Refresh a Single Entity, is a handler for the event `selectionChange` of an `sap.m.Table`. In this handler the OData V4 context for the selected entity is requested and the `refresh` method is called on it. The refresh is only allowed if there are no pending changes for the context and all its dependent bindings.

Only contexts of an absolute `sap.ui.model.odata.v4.ODataListBinding` can be refreshed. Calling the `refresh` method on a context also refreshes all dependent bindings of the context.

After refreshing an entity, it may no longer match the query options for the collection that loaded the entity, in particular `$filter`. You can decide whether the refresh should ignore the query options or not. You can have the corresponding context removed from the list binding for the collection by setting the parameter `bAllowRemoval` of the `refresh` method to **`true`**.

© Copyright. All rights reserved.

- You can refresh a single entity in a collection by calling the *refresh* method on the context of an *sap.ui.model.odata.v4.ODataListBinding* which corresponds to this entity.
- Refresh is only allowed if there are no pending changes for the context.

```
onRefreshSelectedPerson: function () {

    var oPersonContext = this.byId("peopleList2").getSelectedItem().getBindingContext();

    if (!oPersonContext.hasPendingChanges()) {
        oPersonContext.refresh();
    }

}
```

Figure 237: Refresh a Single Entity

Automatic Data Type Detection

The OData V4 model provides automatic data type detection to parse, validate, and format user entries.

The property binding automatically determines the appropriate type depending on the property's metadata. For example, for `<Input value="{Age}"/>` the SAPUI5 type `sap.ui.model.odata.type.Int64` is used, which corresponds to the OData type `Edm.Int64`.

The setting **`"handleValidation": true`** in the `manifest.json` descriptor file makes sure that any validation errors that are detected by the SAPUI5 types are shown on the UI using the message manager.

If user input does not match the type and constraints given in the metadata file, the corresponding fields are highlighted and an error message is displayed. For example, enter the string **`abc`** in the `Age` field, which requires an integer input, or remove the entry from the `FirstName` field, which is mandatory.

If you explicitly specify a type in the binding 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.odata.type.String'}"/>`, the `String` type will be used, not the `Int64` type from the service metadata. You cannot specify format options or constraints unless you also hardcode the type.

© Copyright. All rights reserved.

- The service metadata contains type information for the properties of each entity.

```
metadata.xml ×
8   <EntityType Name="Person">
9       ...
10      <Property Name="FirstName" Type="Edm.String" Nullable="false"/>
11      <Property Name="Age" Type="Edm.Int64"/>
12  </EntityType>
```

- 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.

```
TypeDetermination.view.xml ×
29  <Input value="{FirstName}"/>
30  <Input value="{Age}"/>
```

- In the Application Descriptor you can enable validation handling by the message manager:

```
manifest.json ×
47  "sap.ui5": {
48      "handleValidation": true,
```

- As a result, fields with incorrect entries are highlighted and an error message is displayed:

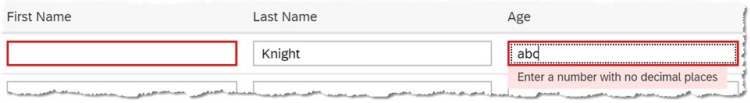


Figure 238: Automatic Data Type Detection

Context API

The context `sap.ui.model.odata.v4.Context` is a pointer to model data as returned by a query from an `sap.ui.model.odata.v4.ODataContextBinding` or an `sap.ui.model.odata.v4.ODataListBinding`. Contexts are always, and only, created by such bindings. A context for a context binding points to the complete query result. A context for a list binding points to one specific entry in the binding's collection. A property binding does not have a context; you can access its value via `sap.ui.model.odata.v4.ODataPropertyBinding#getValue`.

Applications can access model data only via a context, either synchronously with the risk that the values are not available yet (`#getProperty` and `#getObject`), or asynchronously (`#requestProperty` and `#requestObject`).

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 metadata of the property, and not by the type or formatter specified for the binding.

© Copyright. All rights reserved.

The OData V4 model's list and context bindings create *sap.ui.model.odata.v4.Context* objects which provide the following methods:

- *getObject* or *getProperty* for synchronous and *requestObject* or *requestProperty* for asynchronous access to values.
- The methods to access a property can provide the value in internal or external format.

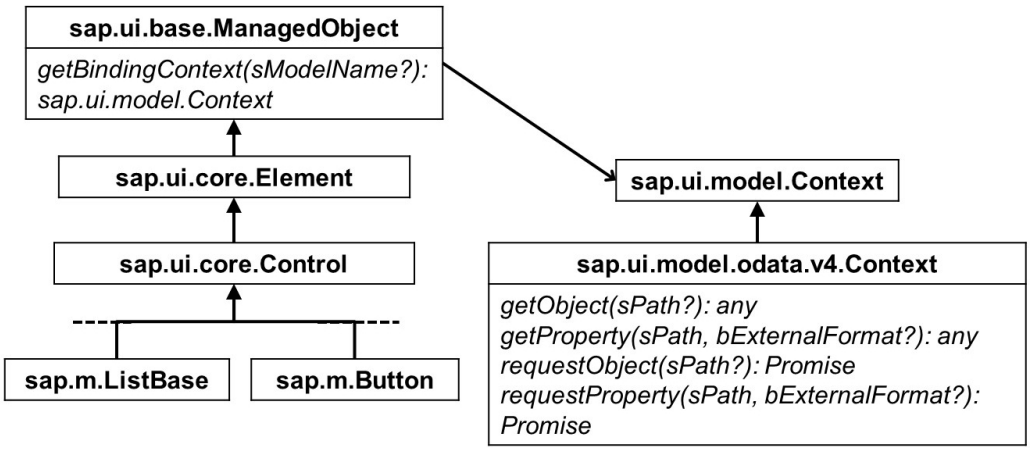


Figure 239: Context API

Context API – Example

On `sap.ui.model.odata.v4.Context` objects, the following methods are available:

- `getBinding:`

  To retrieve the binding that created the context

- `getIndex:`

  To return the context's list index provided the context has been created by an `ODataListBinding`

© Copyright. All rights reserved.

- When a button is pressed in an item of the shown table, the property *LastName* in the same item is accessed first synchronously and then asynchronously in the event handler of the button's press event.

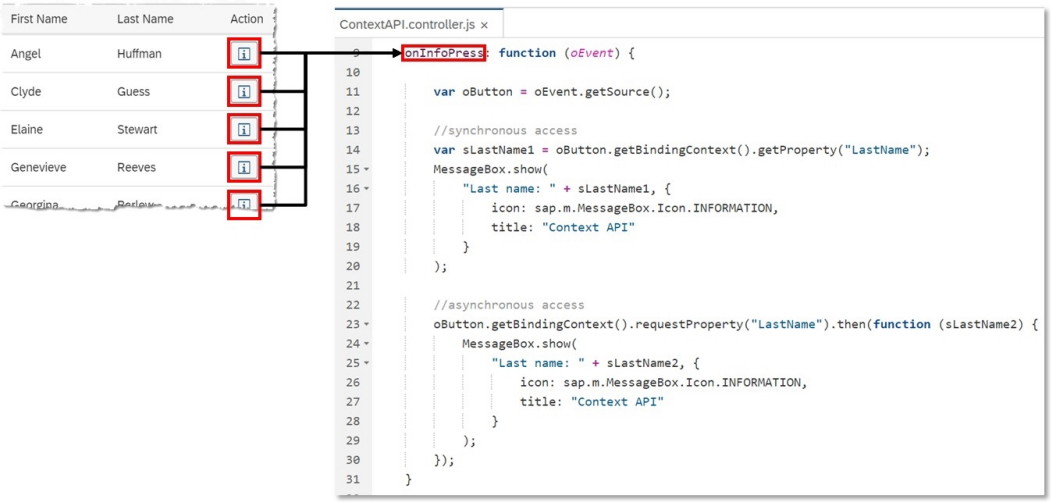


Figure 240: Context API: Example

Binding Events

The OData V4 model supports certain events intended for applications and others that are to be used for controls (like the `change` event).

For applications, the OData V4 model supports the following events: `dataRequested`, `dataReceived`, `patchSent`, and `patchCompleted`.

Typically, the `dataRequested` and `dataReceived` events are used 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:
> 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.

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 their changes are being saved and when saving is finished.

© Copyright. All rights reserved.

BindingEvents.controller.js

```
9     onUserChange: function (oEvent) {
10
11        var sUser = ···
12        var oView = this.getView();
13
14        var oSimpleForm = this.byId("userData");
15        oSimpleForm.bindElement({
16            path: "/People('" + sUser + "')",
17            events: {
18                dataRequested: function (oEvent) {
19                    oView.setBusy(true);
20                },
21
22                dataReceived: function (oEvent) {
23                    oView.setBusy(false);
24                    var oError = oEvent.getParameter("error");
25                    if (oError) {
26                        MessageBox.error(oError.message, {
27                            title: oError.statusText
28                        });
29                    }
30                }
31            }
32        });
33    }
```

- The *dataRequested* event can be used by applications for example to switch on a busy indicator.
- The *dataReceived* event can be used by applications for example to switch off a busy indicator or to process an error.
- If a back-end request fails, the *dataReceived* event provides the corresponding error in the *error* event parameter.

Figure 241: Binding Events

LESSON SUMMARY

You should now be able to:

- Use Data Binding

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 15
Lesson 3

Filtering and Sorting Model Data

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Filter and sort Model Data

Operation Mode

The OData V4 Model supports server-side filtering and server-side sorting on lists.

To use server-side filtering and server-side sorting, set the operation mode to **`sap.ui.model.odata.OperationMode.Server`**. This can be done in the following ways:

- For a single `ODataListBinding` instance, set the binding parameter `$$operationMode`.
- For all list bindings of the model, set the model parameter `operationMode`.

- To use server side filtering and server side sorting, set the operation mode to ***sap.ui.model.odata.OperationMode.Server***.
- The operation mode can be set for all list bindings of the model using the model parameter *operationMode* in manifest.json:

```
manifest.json ×
66    "models": {
67        "": {
68            "type": "sap.ui.model.odata.v4.ODataModel",
69            "settings": {
70                "synchronizationMode": "None",
71                "operationMode": "Server"
72            },
73            "dataSource": "demoService"
74        }
75    },
```

- It can also be set for a specific list binding using the binding parameter *$$operationMode*:

```
<Table items="{ path: '/People',
                parameters: {
                    $$operationMode: 'Server',
                    $filter: 'LastName ge \'B\'
                }
            }">
```

Figure 242: Operation Mode

Static and Dynamic Filters

The `ODataListBinding` allows you 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.

© Copyright. All rights reserved.

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.

The example shown in the figure, Static and Dynamic Filters, filters the `People` entity set by **`LastName`** (static filter) and **`FirstName`** (dynamic filter, initial value).

The list binding allows the setting of *static* and *dynamic filters*:

- The *static filter* value is sent with every data service request for the binding. The *static filter* cannot be overwritten for an existing binding.
- To set the *dynamic filter*, use the method *ODataListBinding.filter()*. This overwrites the initial value of the *dynamic filter* specified on binding construction.
- The list binding combines the *dynamic filter* and *static filter* with a logical AND.

```
<Table  items="{ path: '/People',
                 parameters : {
                    $filter : 'LastName ge \'B\''
                 },
                 filters : {
                    path : 'FirstName',
                    operator : 'LE',
                    value1 : 'W'
                 }
              }">
```

static filter

dynamic filter initial value

Figure 243: Static and Dynamic Filters

Filtering with Any

The OData V4 model supports the Lambda Operators `any` and `all` as defined in the OData Version 4.0 specification. They are represented by `sap.ui.model.Filter` objects with filter `operator`, either `sap.ui.model.FilterOperator.Any` or `sap.ui.model.FilterOperator.All`.

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. It is a `variable` for the current element of the collection referenced by `path` and needs to be part of the `path` of a nested filter `condition`. That is, the `path` of the nested filter `condition` contains the `variable` as prefix to reference the current element of the collection.

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.

© Copyright. All rights reserved.

- Get all the people who have at least one trip assigned:

Filtering.controller.js

```
14    onFilter1: function (oEvent) {
15        var oBinding = this.byId("peopleList").getBinding("items");
16
17        oBinding.filter(
18            new sap.ui.model.Filter({
19                path: "Trips",
20                operator: sap.ui.model.FilterOperator.Any
21            }));
22    },
```

resulting request:
http://host/service/People?$filter=Trips/any()

- Get all the people who have at least one trip with a budget of more than 2,700:

Filtering.controller.js

```
24    onFilter2: function (oEvent) {
25        var oBinding = this.byId("peopleList").getBinding("items");
26
27        oBinding.filter(
28            new sap.ui.model.Filter({
29                path: "Trips",
30                operator: sap.ui.model.FilterOperator.Any,
31                variable: "trip",
32                condition: new sap.ui.model.Filter("trip/Budget", sap.ui.model.FilterOperator.GT, 2700)
33            }));
34    },
```

resulting request:
http://host/service/People?$filter=Trips/any(trip:trip/Budget gt 2700)

Figure 244: Filtering with Any

Filtering with 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.

Get all the people whose trips all have a budget of more than 2,700:

Filtering.controller.js

```
36    onFilter3: function (oEvent) {
37        var oBinding = this.byId("peopleList").getBinding("items");
38
39        oBinding.filter(
40            new sap.ui.model.Filter({
41                path: "Trips",
42                operator: sap.ui.model.FilterOperator.All,
43                variable: "trip",
44                condition: new sap.ui.model.Filter("trip/Budget", sap.ui.model.FilterOperator.GT, 2700)
45            }));
46    }
```

resulting request:
http://host/service/People?$filter=Trips/all(trip:trip/Budget gt 2700)

Figure 245: Filtering with All

Static and Dynamic Sorters

`ODataListBinding` allows you 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.

© Copyright. All rights reserved.

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.

Dynamic sorters are transformed to an OData `$orderby` system query option value and the static sorters are always appended as secondary sort criterion.

In the example shown in the figure, Static and Dynamic Sorters, the `People` entity set is first ordered as descending by **`LastName`** (dynamic sorter) and then as ascending by **`FirstName`** (secondary sort criterion, static sorter).

The list binding allows the setting of *static* and *dynamic sorters*:

- The *static sorter* value is sent with every data service request for the binding. The *static sorter* cannot be overwritten for an existing binding.
- To set the *dynamic sorter*, use the method *ODataListBinding.sort()*. This overwrites the initial value of the *dynamic sorter* 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.

```
<Table items="{ path: '/People',
                parameters: {
                    $orderby: 'FirstName'
                },
                sorter: {
                    path: 'LastName',
                    descending: true
                }
            }">
```

static sorter

dynamic sorter; can be overwritten by calling *sort()* on the list binding

Figure 246: Static and Dynamic Sorters

Setting the Dynamic Sorter – Example

The dynamic sorter is a `sap.ui.model.Sorter` instance, or an array thereof in which case the sorters are concatenated. For setting the dynamic sorter, the `ODataListBinding.sort()` method is used. The sorter that is shown in the figure, Setting the Dynamic Sorter: Example, overwrites the initial value specified on binding construction.

- Set an ascending order by *LastName* as *dynamic sorter*:

Sorting.controller.js

```
9   onSort: function (oEvent) {
10      var oBinding = this.byId("peopleList").getBinding("items");
11
12      oBinding.sort(
13          new sap.ui.model.Sorter("LastName", false));
14  },
```

Figure 247: Setting the Dynamic Sorter: Example

© Copyright. All rights reserved.

LESSON SUMMARY

You should now be able to:

- Filter and sort Model Data

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 15
Lesson 4

Consuming OData Operations

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Consume OData Operations

Simple Function Binding

OData V4 Operations: Summary

OData supports custom operations: Actions and Functions.

Functions are operations exposed by an OData service that must return data and must have no observable side effects.

Actions are operations exposed by an OData service that may have side effects when invoked. Actions may return data but must not be further composed with additional path segments.

Actions and Functions may be bound to an entity type, primitive type, complex type, or a collection. The first parameter of a bound operation is the binding parameter. The namespace, or alias-qualified name of a bound operation, may be appended to any URL that identifies a resource whose type matches, or is derived from, the type of the binding parameter. The resource identified by that URL is used as the binding parameter value.

To invoke a function bound to a resource, the client issues a get request to a function URL. Any non-binding parameter values are specified using inline parameter syntax: Parameter values are specified by appending a comma-separated list of parameter values, enclosed by parenthesis to the function name. Each parameter value is represented as a name/value pair in the format `Name=`**`Value`**, where `Name` is the name of the parameter to the function and **`Value`** is the parameter value.

To invoke a function through a function import, the client issues a get request to a URL identifying the function import and passing parameter values using inline parameter syntax.

To invoke an action bound to a resource, the client issues a post request to an action URL. Any non-binding parameter values are passed in the request.

To invoke an action through an action import, the client issues a post request to a URL identifying the action import. When invoking an action through an action import, all parameter values must be passed in the request body.

Using OData V4 Operations in the SAPUI5 OData V4 Model

The SAPUI5 OData V4 model supports OData V4 operations (Action Import, Function Import, bound Actions and bound Functions).

You can gain access to a Function Import 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, as shown in the figure, Simple Function Binding. The binding path represents the function's return value. The model calls the function immediately when a control requests this value.

© Copyright. All rights reserved.

- You can gain access to a *Function Import* by simply binding it to a view element.

Example 1:

```
<Text text="{/GetNumberOfAvailableItems()}"/>
```

Example 2:

```
<f:SimpleForm binding="{/GetPersonWithMostFriends()}">
    <f:content>
        <Label text="{i18n>firstNameLabelText}"/>
        <Text text="{FirstName}"/>
        <Label text="{i18n>lastNameLabelText}"/>
        <Text text="{LastName}"/>
    </f:content>
</f:SimpleForm>
```

Figure 248: Simple Function Binding

Deferred Function Binding

It is often not feasible for a function to be called immediately. For example, there may be 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 `<SimpleForm>`. Mark the function as deferred by inserting an ellipsis ("...") in the brackets, for example **`/GetNearestAirport(...)`**. To access the return value from child elements use relative bindings.

If the function binding defers function execution, you need to call its `execute()` method to execute the function.

In the example shown in the figure, Deferred Function Binding, the function import `GetNearestAirport`, which has the parameters `lat` and `lon`, is bound to a simple form (with id `nearestAirport`). The `Text` field showing a property of the result is a child of this form. It has a relative binding to the property `Name`. To specify the values for the function parameters `lat` and `lon`, the method `setParameter()` is called on the function binding.

If a function returns a primitive value or a collection, the binding path for the result must be `"{value}"`.

© Copyright. All rights reserved.

- By inserting an ellipses (“…”) in the brackets, a function binding is marked as ***deferred***.
- To execute a deferred function binding, call its *execute()* method.
- Function parameters can be set by calling the method *setParameter()* on the function binding.

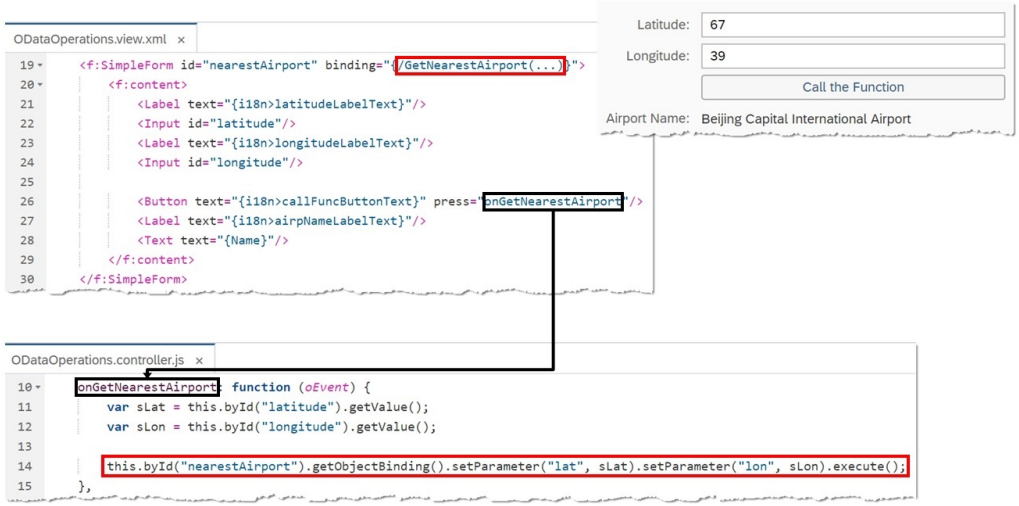


Figure 249: Deferred Function Binding

Action Binding

A deferred action binding is declared exactly like a deferred function binding. You insert an ellipsis in the brackets "(...)" even though the action's resource URL does not contain it. However, it is necessary to mark the binding as deferred.

When calling the `execute()` method, the binding uses the metadata to distinguish between action and function, and to build the correct operation resource path. The `execute()` method returns a `Promise` which is `resolved` if the operation was successful and `rejected` with an error if this was not the case.

A 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).

© Copyright. All rights reserved.

- 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.

Figure 250: Action Binding

Bound Actions and Functions

So far we have covered examples that 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 sample shown in the figure, Bound Actions and Figures, calls the `GetFavoriteAirline` function on the person selected in the corresponding table.

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.

The operation binding's `execute()` method returns a `Promise` that is `resolved` without data or a return value context when the operation call succeeded, or `rejected` with an instance of `Error` in case of failure. A return value context is a `sap.ui.model.odata.v4.Context` which represents a bound operation response. It is created only if the operation is bound and has a single entity return value from the same entity set as the operation's binding parameter and has a parent context which is a `sap.ui.model.odata.v4.Context` and points to an entity from an entity set.

Operation results can be accessed by calling `getObject()` from the bound context.

© Copyright. All rights reserved.

- It is possible to bind an action or a function to another resource of the service.
- Bound actions or functions are controlled in the same way as unbound operations.
- Operation results can be accessed by calling *getObject()* from the bound context.

Figure 251: Bound Actions and Functions

LESSON SUMMARY

You should now be able to:

- Consume OData Operations

© Copyright. All rights reserved.

© Copyright. All rights reserved.

Unit 15
Lesson 5

Implementing Batch Control

LESSON OBJECTIVES
After completing this lesson, you will be able to:

- Implement Batch Control

Group IDs

OData V4 allows you to group multiple operations into a single HTTP request payload.

The OData V4 model sends requests in the following cases:

- Implicit read requests to retrieve data for a binding.

  For example, a list binding with the absolute path `/People` triggers a `GET People` to read data.

- Implicit update requests via two-way binding.

  For example, update a person's first name through a property binding with the relative path `FirstName`, which has a context with path `/People('russellwhyte')` triggering `PATCH People('russellwhyte')` with the first name'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,

© Copyright. All rights reserved.

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`**.

- 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 ID` has the submit mode **`sap.ui.model.odata.v4.SubmitMode.API.`**

  On construction of the OData model, it is possible to define different submit modes 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 OData model's use of **batch requests** is controlled by *Group IDs*. Valid *Group IDs* are:

- ***$auto and $auto.****
  - Requests associated with the *Group ID* ***$auto*** or ***$auto.**** are bundled in one batch request which is triggered automatically before rendering.
  - You can use different ***$auto.**** *Group IDs* to use different batch requests. The suffix may be any non-empty string consisting of alphanumeric characters from the basic Latin alphabet, including the underscore.
- ***$direct:***
  - Requests associated with the *Group ID* ***$direct*** are sent directly without batch.
- ***Application Group IDs:***
  - An *Application Group ID* is a non-empty string (defined by the developer) consisting of alphanumeric characters from the basic Latin alphabet, including the underscore.
  - By default, requests associated with an *Application Group ID* are sent in a batch request by calling the method *sap.ui.model.odata.v4.ODataModel#submitBatch(sGroupId)*.
  - It is possible to use a different submit mode.

Figure 252: Group IDs

Usage of Group IDs in XML views

In the example shown in the figure, Usage of Group IDs in XML views, updates for the first name and last name through two-way binding are sent directly without batch. Likewise, data is read without batch.

The default for the Group ID of the OData model is `$auto`. The value of Group ID is used as default for the Update Group ID of the OData model.

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.

© Copyright. All rights reserved.

- To specify a *Group ID*, use the following parameters for the binding which triggers the request:
  - **$$groupId** – *Group ID* for read requests
  - **$$updateGroupId** – *Group ID* for update requests

BatchControl.view.xml

```
39  <f:SimpleForm binding="{ path:       '/People(\'clydeguess\')',
40                 parameters: {$$updateGroupId: '$direct', $$groupId: '$direct'} }"
41              editable="true">
42
43     <Label text="{i18n>firstNameLabelText}"/>
44     <Input value="{FirstName}"/>
45     <Label text="{i18n>lastNameLabelText}"/>
46     <Input value="{LastName}"/>
47
48  </f:SimpleForm>
```

- On instantiation of the OData model, you can also provide both a *Group ID* and an *Update Group ID*; they are used as defaults if the corresponding binding parameters are not specified.

Figure 253: Usage of Group IDs in XML views

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()`.

> 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.

© Copyright. All rights reserved.

BatchControl.view.xml

```
54    <f:SimpleForm binding="{ path:        '/People(\'willieashmore\')',
55                             parameters: {$$updateGroupId: 'PersonUpdateGroup'} }"
56                  editable="true">
57        <Label text="{i18n>firstNameLabelText}"/>
58        <Input value="{FirstName}"/>
59        <Label text="{i18n>lastNameLabelText}"/>
60        <Input value="{LastName}"/>
61        <f:toolbar>
62            <Toolbar>
63                <content>
64                    <ToolbarSpacer/>
65                    <Button icon="sap-icon://sys-cancel-2" press="onCancelPerson"/>
66                    <Button icon="sap-icon://save" press="onSavePerson"/>
67                </content>
68            </Toolbar>
69        </f:toolbar>
70    </f:SimpleForm>
```

First Name:
Willie
Last Name:
Ashmore

```
onSavePerson: function (oEvent) {
    this.getView().getModel().submitBatch("PersonUpdateGroup");
},
```

```
onCancelPerson: function (oEvent) {
    this.getView().getModel().resetChanges("PersonUpdateGroup");
},
```

Figure 254: Resetting Property Changes

LESSON SUMMARY

You should now be able to:

- Implement Batch Control

© Copyright. All rights reserved.

Creating and Deleting Entities

LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create and delete entities

Creating an Entity

In the example shown in the figure, Creating an Entity: Example (1), it is possible to create new person data from the user interface and send the data to the backend.

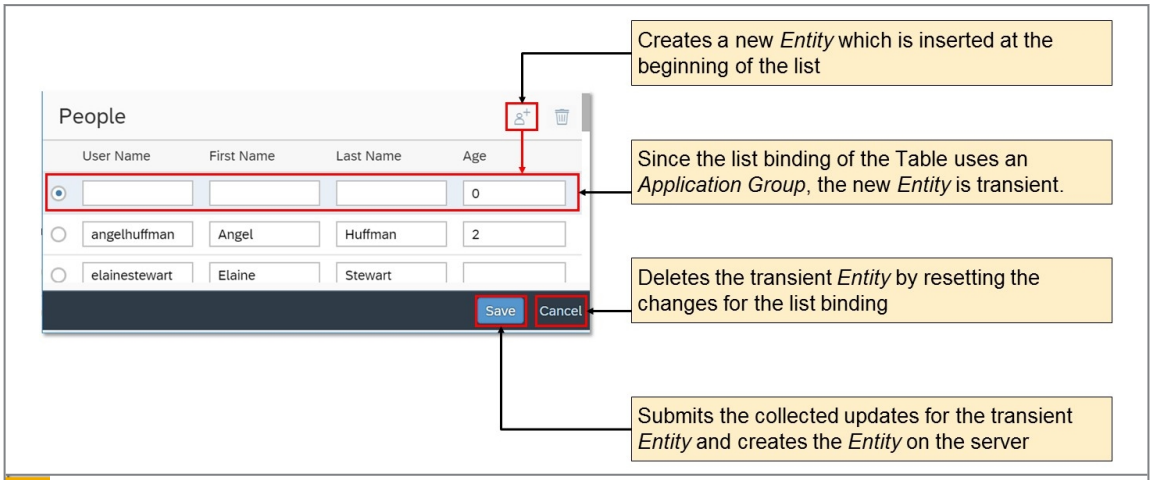


Figure 255: Creating an Entity: Example (1)

The `onAddPerson` event handler responds to the `press` event of the `Add Person` button. The implementation of this event handler uses the `create()` method of the `ODataListBinding` API to create a new person with initial data and insert it at the top of the table.

As long as the binding contains an entity created via the `create()` function, you cannot create another entity. This is only possible after the creation of the entity has been successfully sent to the server and you have called the `refresh()` method of the binding or the new entity is deleted in between.

For creating the new entity, the binding's Update Group ID is used.

The `create()` method returns the binding context of the new person. That context provides a `created()` method, which returns a `Promise`. The `Promise` is resolved when the new person is successfully transferred to the OData service and has been created in the backend. When that happens, the `refresh()` method of the `ODataListBinding` is called. Only then can you add more persons to the list binding later.

The `Promise` returned by the `created()` method is `rejected` in all cases where the created entity is deleted before it is created in the backend.

© Copyright. All rights reserved.

- **sap.ui.model.odata.v4.ODataListBinding#create** creates a new entity and inserts it at the beginning of the list.
- For creating the new entity, the binding's *Update Group ID* is used.
- **sap.ui.model.odata.v4.Context#created** returns a *Promise* that is resolved when the entity has been created in the backend.
- Calling the *refresh()* method of the list binding ensures that more people can be added later.

```
onAddPerson: function () {
    var oBinding = this.byId("peopleList").getBinding("items");

    var oContext = oBinding.create({
        "UserName": "",
        "FirstName": "",
        "LastName": "",
        "Age": ""
    });

    oContext.created().then(function () {
        oBinding.refresh();
    });
}
```

Figure 256: Creating an Entity: Example (2)

You can call `sap.ui.model.odata.v4.Context#delete` to delete a created context again. As long as the context is transient, `sap.ui.model.odata.v4.ODataListBinding#resetChanges`, and a call to `sap.ui.model.odata.v4.ODataModel#resetChanges` with the Update Group ID as parameter, also delete the created context together with other changes.

> Note:
> 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.

If you have called `sap.ui.model.odata.v4.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 `sap.ui.core.message.MessageManager` and the `Promise` you get via `Context#created` is not `rejected`.

© Copyright. All rights reserved.

- **sap.ui.model.odata.v4.ODataListBinding#resetChanges** deletes a transient context from the binding.
- **sap.ui.model.odata.v4.ODataModel#submitBatch(sGroupId)** creates the entity on the server.
- It is recommended to lock the UI until *submitBatch()* is resolved, because updates during a running POST request result in errors.

```
onResetChanges: function () {
    this.byId("peopleList").getBinding("items").resetChanges();
}
```

```
onSave: function () {
    var oView = this.getView();

    function resetBusy() {
        oView.setBusy(false);
    }

    oView.setBusy(true);

    oView.getModel().submitBatch("peopleGroup").then(resetBusy, resetBusy);
}
```

Figure 257: Creating an Entity: Example (3)

Deleting an Entity

In the event handler `onDeletePerson`, a check is performed to see if an item is selected in the table and if so, the related data is deleted from the model. To do that, the binding context of the selection is retrieved and its `delete()` method is called.

For the deletion, the Update Group ID is set explicitly 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 was defined for the table's list binding. But the `delete()` function currently does not support deferred batch processing.

The `Promise` returned by the `delete()` function is `rejected` with an instance of `Error` in case of failure, for example, if the given context does not point to an entity, if it is not part of a list binding, if there are pending changes for the context's binding, if the resulting Group ID has `SubmitMode.API`, or if the deletion on the server fails.

The error instance is flagged with `isConcurrentModification` in case a concurrent modification (for example, by another user) of the entity between loading and deletion has been detected; this should be shown to the user who needs to decide whether to try the deletion again. If the entity does not exist, it is assumed that it has already been deleted by someone else and success is reported.

© Copyright. All rights reserved.

- **sap.ui.model.odata.v4.Context#delete(sGroupId?)** deletes the OData entity the context points to on the server and updates the user interface accordingly.
- The *delete()* function returns a *Promise* which is resolved without a result in case of success, or rejected with an instance of *Error* in case of failure.

```
onDeletePerson: function () {
    var oItem = this.byId("peopleList").getSelectedItem();

    if (oItem) {
        var oPersonContext = oItem.getBindingContext();
        oPersonContext.delete("$auto").then(function () {
                //...
        }, function (oError) {
                //...
        });
    }
}
```

Figure 258: Deleting an Entity

LESSON SUMMARY

You should now be able to:

- Create and delete entities

© Copyright. All rights reserved.