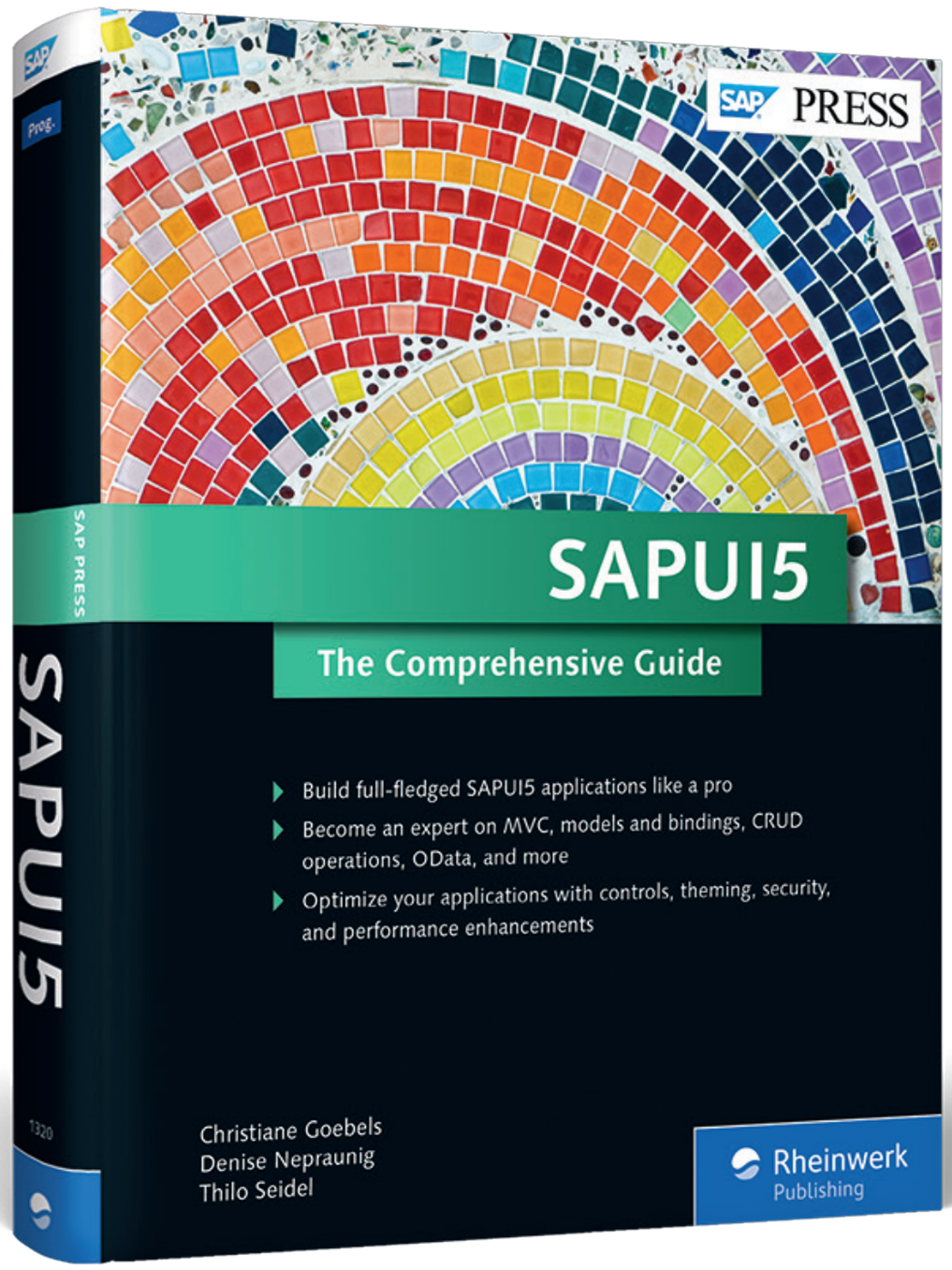
SAP PRESS
First-hand knowledge.
Rheinwerk Publishing



Reading Sample

User experience plays an important role when creating and conceptualizing applications. In this reading sample, we'll discuss some of the most widely used SAPUI5 application patterns and their attributes. We'll begin by looking at the different layouts and floorplans that can be implemented, then provide steps for running these applications in SAP Fiori Launchpad.

Christiane Goebels, Denise Nepraunig, Thilo Seidel

SAPUI5: The Comprehensive Guide

672 Pages, 2016, $79.95
ISBN 978-1-4932-1320-7

www.sap-press.com/3980

Application development in general must close the gap between technological feasibility and the best possible support for a given usage scenario. Therefore, we must not only know about technology but also have a deep understanding of user requirements and constraints. In this chapter, we'll approach the topic of application patterns from both design and technical perspectives.

8 Application Patterns and Examples

Application development with SAPUI5 benefits from the well-defined design patterns and overall application concepts found in the SAP Fiori design guidelines, available at *https://experience.sap.com/fiori-design/*. From a design perspective, this information provides clear guidance on how to structure your content, define usage patterns, and define interaction flows, allowing you to concentrate on your specific scenario implementation, building on top of best practices. From an application developer's point of view, SAPUI5 supports the implementation of these guidelines by providing controls and the right APIs that are built based on the overall design requirements.

Although it's been said that good user experience can never be achieved simply by technology alone, technical aspects and decisions do play an important role. As previously stated, SAP Fiori design concepts and SAPUI5 grew up together. While SAP Fiori emphasized the implementation of small, single-purpose applications, SAPUI5 served as the tailored technology for these application.

For us, this means that we should always try to build individual and focused applications. For example, in a scenario in which our users can create, approve, and analyze leave requests, we should create three applications.

In this chapter, we'll explore general application concepts and patterns found in SAPUI5. We'll start with general application layouts, then dig deeper and explore more detailed floorplans. Finally, we'll look into specific application types and explore shared application features. We'll always start from a design perspective and build knowledge for use cases and underlying assumptions. Then, we'll start

to explore technical assets like controls that are provided by SAPUI5 and that ease the implementation of these design patterns.

In the final section of this chapter, you'll learn how SAP Fiori Launchpad serves as the central access point for SAPUI5 applications in many scenarios and will gain some hands-on experience with its developer features.

8.1 Layouts

Laying out applications generally happens at different levels. Think of a grid used to cut the screen into pieces that will later be assigned individual content. This concept is common in web development.

When building full-blown applications, you might still use a grid-based approach. However, you should first think about the general cut of your application, meaning the overall number of content areas you'll need to leverage to enrich user experience and to streamline the tasks your users will have to complete using the application you build. Therefore, the first decision you make should be simply whether you want to build a *full-screen* or *split-screen application*. Differentiating between full-screen and split-screen options might seem like a no-brainer at first glance, but we'll discuss these difference to a greater extent in this section. The choice isn't as simple as it may seem initially.

These applications can be derived from the task, sequence of usage, and target group of your application. This first decision will ultimately help you understand the underlying usage scenario of your application in greater detail. We'll outline the important questions to ask when choosing a layout in this section, and then we'll build example implementations using SAPUI5 controls. To begin, we'll use a simple application skeleton that can be generated from a template in the SAP Web IDE.

The generation of templates in the SAP Web IDE is covered in Appendix D of this book. Please look up the general wizard functionality there. What we want to generate now is the SAPUI5 Application template (see Figure 8.1).

This template provides the right folder structure and all the files needed to build our first prototypes. Most of it should look familiar from the previous examples in the book. We use this template frequently to test new controls or even to test

complex patterns isolated from the actual project we're working on. With some small changes, it can also serve as a base for application development.

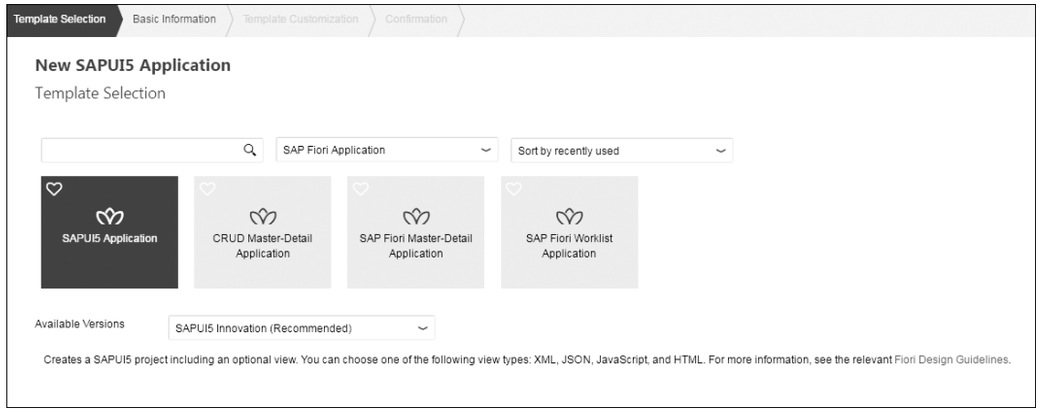


Figure 8.1 SAPUI5 Template in Template Wizard

Let's first look into the Main.view.xml file in the *view* folder of the project. This is defined as the `rootView` in manifest.json and will therefore be loaded at application startup (see Listing 8.1).

```
<mvc:View
  controllerName="my.app.controller.Main"
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">
  <App>
    <pages>
      <Page title="{i18n>title}">
        <content></content>
      </Page>
    </pages>
  </App>
</mvc:View>
```

Listing 8.1 Initial Main.view.xml

The app control serves as a root control for the template application. However, it already has a `sap.m.Page` element prefilled in its `pages` aggregation. In application development, we use routing in SAPUI5 to display individual views and can therefore delete the page and all its content. In addition, we'll add an ID to the root control that we can use later in the routing configuration (see Listing 8.2).

```
<mvc:View
  controllerName="my.app.controller.Main"
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">
  <App id="rootControl"/>
</mvc:View>
```

Listing 8.2 Main.view.xml: Adapted

We still need to set up some basic routing configuration to enable the dynamic display of content in the root control. For this, we'll add a routing block into the `sap.ui5` namespace in manifest.json (see Listing 8.3). This block should hold the ID of the root control and some generic settings, such as `controlAggregation` and the path to the root view. Refer back to Chapter 4 for more details.

```
"sap.ui5": {
  "_version": "1.1.0",
  "rootView": {
    "viewName": "my.app.view.Main",
    "type": "XML"
  },
  "routing": {
    "config": {
      "routerClass": "sap.m.routing.Router",
      "controlId": "rootControl",
      "controlAggregation" : "pages",
      "viewPath": "my.app.view",
      "viewType": "XML",
      "async": true
    }
  },
```

Listing 8.3 Basic Routing Configuration

Although the SAPUI5 controls from the `sap.m` library we will use come with built-in support for different form factors, like mobile and desktop devices, we still have to tell the toolkit for what device it should optimize the display. This will happen dynamically based on what the `sap.ui.Device` API has identified. To enable this functionality, we'll add the check shown in Listing 8.4 to the `onInit` event of the main controller.

```
onInit : function() {
  var sContentDensityClass = "";
  if (jQuery(document.body).hasClass("sapUiSizeCozy") || jQuery(documen
t.body).hasClass("sapUiSizeCompact")) {
    sContentDensityClass = "";
```



```
  } else if (!Device.support.touch) {
    sContentDensityClass = "sapUiSizeCompact";
  } else {
    sContentDensityClass = "sapUiSizeCozy";
  }
  this.getView().addStyleClass(sContentDensityClass);
}
```

Listing 8.4 Content Density Check in Main.controller.js

Finally, we'll add a configuration for the creation of a `sap.ui.model.odata.v2.ODataModel` instance that uses an SAP NetWeaver demo OData service provided by SAP as `dataSource` into manifest.json. We'll use this model later to display real data when building the floorplans and example applications. For now, it will be created silently without any effect.

```
{
  "sap.app" : {
    "dataSources": {
      "mainService": {
        "uri": "/destinations/ES4/sap/opu/odata/IWBEP/GWSAMPLE_BASIC/",
        "type": "OData",
        "settings": {
          "odataVersion": "2.0"
        }
      }
    }
..},
  "sap.ui5":{
    "models": {
      "": {
        "dataSource": "mainService",
        "settings": {
          "metadataUrlParams": {
            "sap-documentation": "heading, quickinfo"
          }
        }
      }
    }
  }
}
```

Listing 8.5 Excerpt from manifest.json with OData Model Creation

The result should now look like Figure 8.2: a simple, letterboxed `sap.ui.core.UIComponent` display that's still unspectacular. However, with this foundation in

place, we're well prepared to later implement specific layouts and floorplans and then start the real application development.

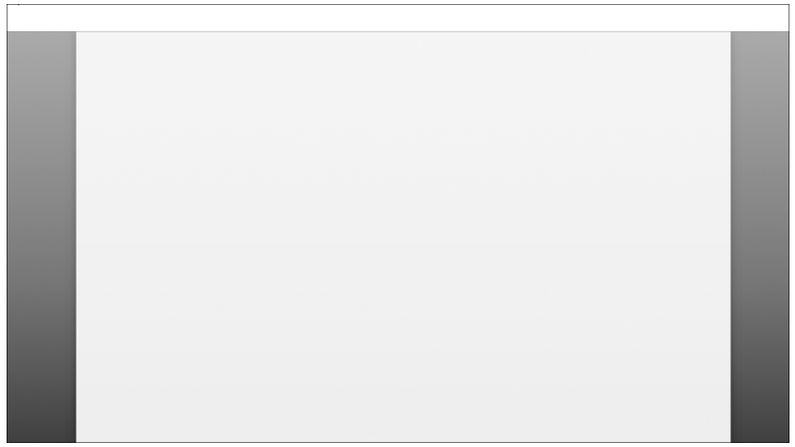


Figure 8.2 Application Starter Template Display

In the following two subsections, we'll look at guidelines for creating both a full-screen and a split-screen layout.

8.1.1 Full-Screen Layout: sap.m.App

Naturally, full-screen apps make use of the entire screen. You can still decide if you want to have your app in a letterboxed display or not (see Section 8.3.4 for details), but the main characteristic of a full-screen layout from a programming point of view is that it contains a single content area.

The term *content area* might need some explanation. Just think of one, single-purpose area on your screen. This could be a list of items that is displayed, for example, or details about a specific item. This will become clearer when you learn more about the split-screen layout in Section 8.1.2.

For the full-screen layout, it's important to understand that there should be only one purpose per screen (like the display of object details), although this could still mean that you mix information from different data sources and even use different types of display. This can include charts, textual information, and even a list of related items. Therefore, a full-screen layout is clearly purpose-oriented and has nothing to do with data origin or media.

The following are some guiding questions you should ask yourself when using a full-screen layout:

- Do I want to display a high number of facets related to a single entity with minimal navigation?
- Does the content require maximal space (e.g., charts or images)?
- Do I want to display a list in combination with complex filtering options?

Technically, a full-screen layout uses the `sap.m.App` control as a root control. Based on the routing configuration, different views can be placed into its `pages` aggregation. Because the `sap.m.App` control inherits from `sap.m.NavContainer`, transitions are fully supported, and routing-specific events can be attached and handled based on the existing API.

Pay attention to *responsive behavior* for full-screen applications. Later, you'll see that the control used as a root control for the split-screen layout introduces some responsiveness out of the box. This is not the case for the app control, however, because of the single content area. That's why we will have to take care of enabling responsive behavior directly for the full-screen layout. Luckily, SAPUI5 provides controls that include the necessary intelligence to handle different form factors, which is why we'll use pages from `sap.m.semantic` when building applications. For this example, we'll use `sap.m.semantic.FullscreenPage`, which provides overflow handling for header and footer areas in the full-screen layout. We'll revisit headers and foots in Section 8.3.5.

Let's now enhance the starter application by adding a first view and additional routing configuration so that it can serve as a first, simple, full-screen-layout application (see Listing 8.6 and Listing 8.7). This results in a simple full-screen display as in Figure 8.3.

```
<mvc:View
  controllerName="my.app.controller.Main"
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
  xmlns:semantic="sap.m.semantic">
  <semantic:FullscreenPage title="Fullscreen">
    <!-- Enough space for your content here -->
  </semantic:FullscreenPage>
</mvc:View>
```

Listing 8.6 webapp/view/Home.view.xml

```
"routing": {
  "config": {
    "controlId": "rootControl",
    "controlAggregation" : "pages",
    "viewPath": "my.app.view",
    "viewType": "XML"
  },
  "routes": [{
    "name" : "home",
    "pattern": "",
    "target": ["home"]
  }],
  "targets": {
    "home": {
      "viewName": "Home"
    }
  }
},
```

Listing 8.7 Simple Full-Screen Routing Configuration in manifest.json

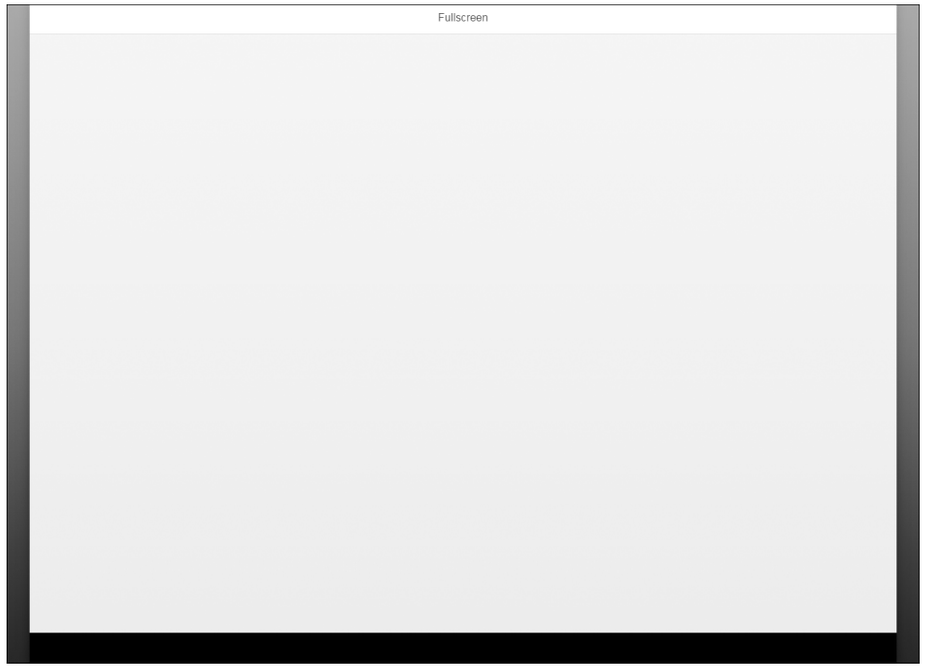


Figure 8.3 Simple Full-Screen Layout

This is obviously not rocket science: You could easily build upon this foundation with what you've learned in this book already and extend this view now with controls and content.

Now, let`s look at what floorplans, defined by the SAP Fiori design guidelines, make use of the full-screen layout:

- **Initial page**
  Single object display based on user input (search, barcode scanning).
- **Worklist**
  See Section 8.2.1.
- **List report**
  Multi-object display with extended filtering/sorting capabilities.

8.1.2 Split Screen Layout: sap.m.SplitApp

Now, let's turn our attention to the split-screen layout in SAPUI5. A split screen consists of at least two content areas displayed side by side. However, this does not mean that the two areas are separate from each other; in fact, both content areas need to be orchestrated such that they're dependent on each other. One frequently used and well-established floorplan in SAP Fiori is the master-detail pattern. The selection in the master list determines the display of details of the selected item in the object view. We will look into this pattern in more detail in Section 8.2.2.

One use case that benefits the most from using a split-screen layout is one in which you expect your application users to review a high number of items—for example, in approval scenarios. In this case, you generally want to assure that users do not have to execute a high number of back and forth navigations and therefore want to display the list to select from next to the details to review, all on one screen. Most of us use this pattern on a daily basis; for example, it's a default setting in most of the local email clients available.

The general build-up of a split-screen layout is similar to what you've already seen for the full-screen layout in Section 8.1.1, with some slight modifications. We can again make use of the application starter template we created in Section 8.1.

We'll start by using a different root control and some other slight modifications down the line. First, we'll use the `sap.m.SplitApp` control in Main.view.xml (see Listing 8.8).

```
<mvc:View
  controllerName="my.app.controller.Main"
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">
  <SplitApp id="rootControl"/>
</mvc:View>
```

Listing 8.8 Main.view.xml for Split-Screen Layout

The `sap.m.SplitApp` control is a pretty clever composite control that provides two `sap.m.NavContainer` elements internally as hidden aggregations that can be populated by making use of two public aggregations: `masterPages` and `detailPages`. We can therefore use the routing configuration to handle the placement of views into these aggregations and again use routing events if needed. The `masterPages` and `detailPages` are derived from the internal navigation containers that are wrapped and exposed by the `sap.m.SplitApp` control. Before we look into the routing configuration in detail, let's first create two views. For the split screen with `sap.m.SplitApp`, we can use specific semantic page controls—one for the `masterPages` aggregation (see Listing 8.9), and one for `detailPages` (see Listing 8.10).

```
<mvc:View
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
  xmlns:semantic="sap.m.semantic">
  <semantic:MasterPage title="Master">
    <!-- Enough space for your content here -->
  </semantic:MasterPage>
</mvc:View>
```

Listing 8.9 Master.view.xml with sap.m.semantic.MasterPage

```
<mvc:View
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
  xmlns:semantic="sap.m.semantic">
  <semantic:DetailPage title="Detail">
    <!-- Enough space for your content here -->
  </semantic:DetailPage>
</mvc:View>
```

Listing 8.10 Detail.view.xml with sap.m.semantic.DetailPage

Now, let's modify and enhance the existing routing configuration. We want to ensure that both Master.view.xml and Detail.view.xml are displayed in the respective aggregations of the root control when the application is started.

To achieve this, let's quickly revisit what you learned in Chapter 4, Section 4.7 about how routing works in SAPUI5. The routing configuration is built up by configuring the router globally in the `config` setting and can then be enriched for specific routes and targets. In that sense, the configuration for targets is more specific than the one for routes, and configuration options can even be overridden. For the current scenario, we'll therefore have specific targets that define their own aggregations to address the two content areas in `sap.m.SplitApp` accordingly. Compared to the routing configuration for the full-screen layout, we'll have two additional targets for every route. Here, the sequence makes a difference. This is because `sap.m.SplitApp` handles the display of views based on the current screen size and therefore includes responsiveness across form factors out of the box. Figure 8.4 shows that the control displays differently across device types.

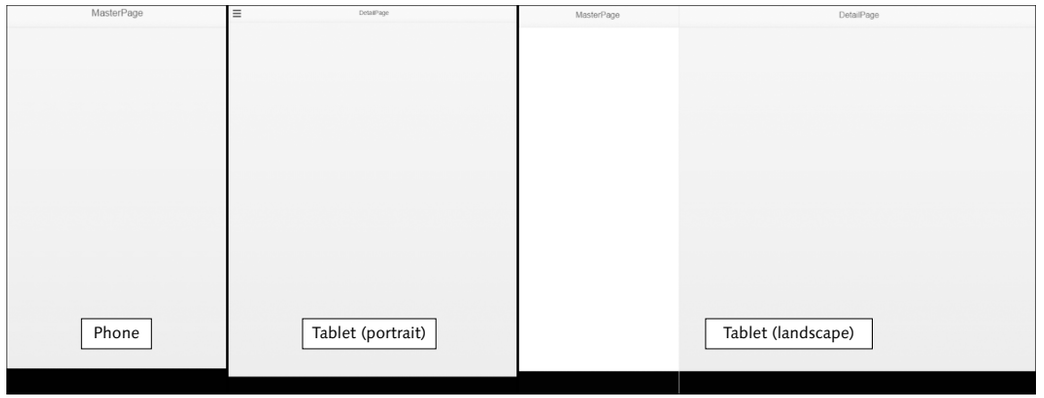


Figure 8.4 Responsiveness of sap.m.SplitApp

You can influence this control behavior with the routing configuration. To do so, define the targets per route in the right sequence with the target you want to have displayed on a phone, on which only one content area will be displayed, for this

route in the array of targets. For the default route, with an empty hash, you'll most likely choose the master view. In that case, on a tablet in portrait mode, you'll see the details view and a button in the header; clicking on that button will slide in the master view (see Listing 8.11).

```
"routing": {
  "config": {
    "controlId": "rootControl",
    "viewPath": "my.app.view",
    "viewType": "XML"
  },
  "routes" : [
    {
        "pattern" : "",
        "name" : "main",
        "target" : ["detail", "master"]

    }
  ],
  "targets" : {
    "master" : {
      "viewName" : "Master",
      "controlAggregation" : "masterPages"
    },
    "detail" : {
      "viewName" : "Detail",
      "controlAggregation" : "detailPages"
    }
  }
},
```

Listing 8.11 Routing Configuration in manifest.json

sap.m.SplitAppModes

In addition to its default behavior, `sap.m.SplitApp` offers four different modes for handling the `masterPages` aggregation display on mobile devices. The `mode` property can be set either as static on the declaration of the control in the XML or as dynamic in JavaScript using the default setter. The modes include the following:

- **`ShowHideMode` (default)**
  Master hidden in portrait mode
- **`StretchCompressMode`**
  Master in a compressed version in portrait mode
- **`PopoverMode`**
  Master shown in a popover in portrait mode

- **`HideMode`**
  Master initially hidden in portrait and landscape

In SAPUI5, there are several controls that can be used to create an application with more than one content area. Most of these examples are part of the `sap.ui.layout` library:

- `sap.ui.layout.Splitter`
- `sap.ui.layout.DynamicSideContent`
- `sap.ui.layout.ResponsiveSplitter`

In this section, we walked through the split-screen layout. In the next section, we'll use the skeleton layouts of our full- and split-screen layouts in floorplans.

8.2 Floorplans

In this section, we'll take the layout skeletons we built in Section 8.1 and extend them to match their respective floorplans with all the needed functionality. We'll actually take this one step further and build two applications that we can later use in Section 8.4 to integrate into SAP Fiori Launchpad and make use of some of the features the launchpad provides for cross-application navigation.

In Section 8.2.1, we'll build a worklist that displays data from `SalesOrder entitySet` in a demo service. Because each `SalesOrder` item is associated with a specific `BusinessPartner` in the service, we'll also build a business partner address book in Section 8.2.2 using the master-detail layout.

Note

In the following sections, we outline the most important features and cornerstones of SAPUI5 application development. Because application development with SAPUI5 could easily fill more than a single chapter, we'll only give examples of certain application patterns here. We'll also describe some shared application features in Section 8.3. Here is a list of application best practices that should be followed but could not be described or used in the scope of this chapter:

- **Usage of i18n texts**
  Do not use hard-coded strings in XML or JavaScript to be displayed in the view. Always use texts that can be translated centrally.

- **Usage of fixed IDs for controls**
  Always add a fixed ID to all controls that are not used as templates in aggregations.

8.2.1 Worklist

In this section, we'll take the full-screen layout we created in Section 8.1.1 and extend the coding to match the worklist floorplan. The *worklist floorplan* can be used for all applications that should display a number of work items. *Work items* are items that need to be processed by the user. For example, stock management is a use case in which users have to ensure a balanced stock level and can trigger actions on individual items. Applications should display the most relevant information in a list of all items on the first screen, allow users to review more detailed information per item on a second screen, and generally offer processing options. If we stay with the stock management use case, these processing options could include reordering or discontinuing items. The SAPUI5 Demo Kit includes a tutorial covering how to build this use case.

We'll now lay the foundation for a worklist by creating the views and adding the essential controls.

Worklist Table

The actual worklist is technically a responsive table (`sap.m.Table`). We'll add the table to the *Home.view.xml* file created in Section 8.1, but will rename it to Worklist.view.xml. The user should be offered additional options to limit or refine the results displayed in the worklist. This can be achieved by using filters, search, or sorting capabilities, which can be triggered by controls displayed via `sap.m.Toolbar`. `sap.m.Toolbar` can be added to the `headerToolbar` aggregation of the responsive table.

For a nice display, we'll also add a `responsive-margin-css` class provided by SAPUI5 and bind it to `SaleOrderSet` in the OData service. To have a minimal footprint on the screen, we'll also show some bound properties via `sap.m.ColumnListItem` and add a custom action to the table using `sap.m.Button`.

The simple version shown in Listing 8.12 leads to the display shown in Figure 8.5.

Note

Custom actions on the worklist are an optional way to provide direct access to commonly used functionality for the user. You can decide to add the actions directly on the list based on whether the information available initially justifies an action to be triggered. Another option is to require actions to be performed initially in every case.

```
<Table
  id="table"
  class="sapUiResponsiveMargin"
  width="auto"
  items="{
    path : '/SalesOrderSet'
  }">
  <headerToolbar>
    <Toolbar>
      <Title
        Id="title"
        text="Manage Sales Orders"/>
      <ToolbarSpacer/>
      <SearchField
        width="auto"/>
      <OverflowToolbarButton icon="sap-icon://filter"/>
      <OverflowToolbarButton icon="sap-icon://sort"/>
    </Toolbar>
  </headerToolbar>
  <columns>
    <Column>
      <Text text="Customer"/>
    </Column>
    <Column>
      <Text text="Net Amount"/>
    </Column>
    <Column/>
  </columns>
  <items>
    <ColumnListItem vAlign="Middle">
      <cells>
        <Link text="{CustomerName}"/>
        <Text text="{NetAmount}"/>
        <Button text="Create Incoive"/>
      </cells>
    </ColumnListItem>
  </items>
</Table>
```

Listing 8.12 sap.m.Table as Worklist

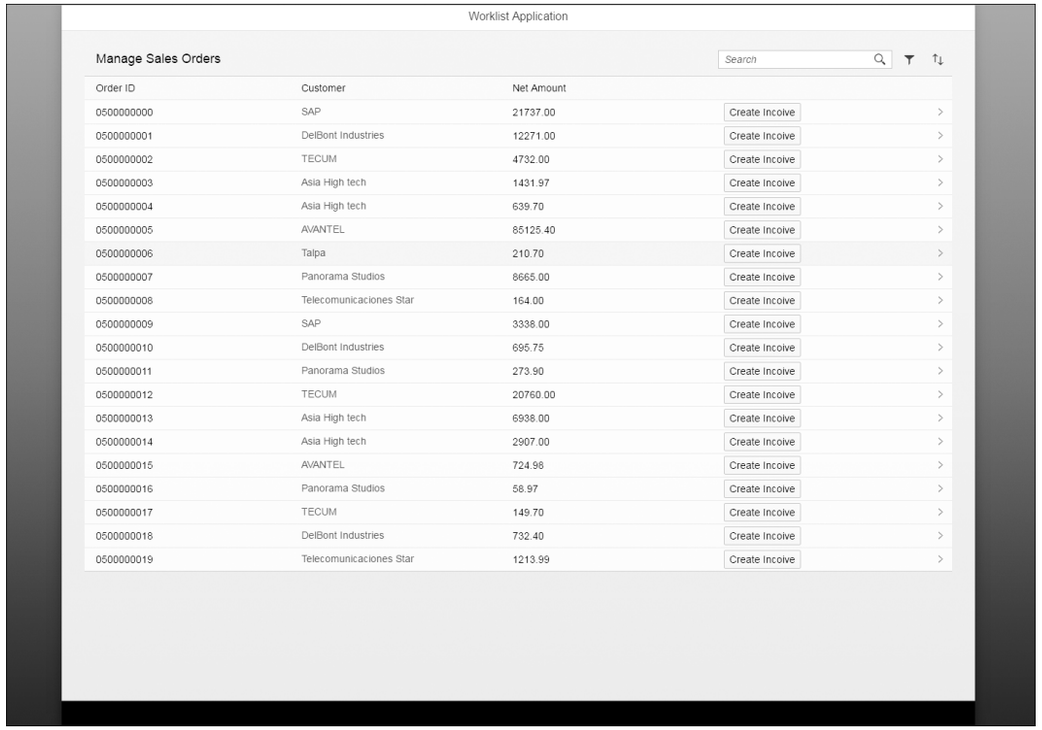


Figure 8.5 Simple Worklist Page

Now, let`s add two more things to the application in this step: an item count in the list indicated next to the table title and search functionality.

Item Count in Table Title

Here, we need to update the displayed item count number whenever the binding of the responsive table is updated. Luckily, this event exists on the table control, and we can simply attach to it by adding `updateFinished="onTableUpdateFinished"` to the control constructor in the view. We can then implement the handler function on the controller, as shown in Listing 8.13. Here, we can receive the total count of items available on the backend based on the current filter as a parameter from the argument of the callback. With this information, we can update the title control.

To achieve a nice display, as shown in Figure 8.6, we need two numbers here. In addition to the total number of items available in this collection, we also can display

the number of items currently displayed on the screen. This makes sense if the `growing` feature of the list is enabled and if your users most likely will have to deal with a high number of items.

```
onTableUpdateFinished : function(oEvent) {
  var sTitle = "Sales Orders",
      oTable = this.getView().byId("table");
  //catch cases where the backend is not supporting remote count
  if(oTable.getBinding("items").isLengthFinal()) {
    var iCount = oEvent.getParameter("total"),
        iItems = oTable.getItems().length;
    sTitle += " (" + iItems + "/" + iCount + ")";
  }
  this.getView().byId("title").setText(sTitle);
}
```

Listing 8.13 Event Handler Function to Set Number of Items

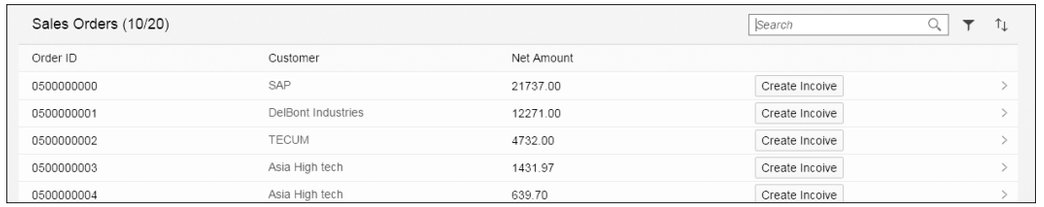


Figure 8.6 Item Count with Two Numbers

Handle Search Input and Filter the Table

Search capabilities give users the feeling of direct control over the displayed list. To increase the effect of this capability even more, we'll use the `liveSearch` event that `sap.m.SearchFiled` provides and will pass a handler function to it by adding `liveChange="onSearch"` to the constructor in the XML. Technically, we'll use filtering on the binding in this handler function. These work equally as well as simply implementing a predefined filter with the buttons directly.

In the handler function to be implemented on the controller (see Listing 8.14), we'll receive the query string entered and instantiate a new `sap.ui.model.Filter` object that will get this query string, a `sap.ui.model.FilterOperator` element of choice and the property to be filtered against. Because it's likely that application users do not want to search on only one column, we'll create a filter that will perform a search on several columns. The buildup is a little more complex, but it's really nothing more than wrapping several `sap.ui.model.Filter` objects into one, which is later handed over to the `filter` function on the binding. For this

filter function, we can also choose between the filter modes. In Figure 8.15, we'll set it to `Application`, which will come at the cost of an additional round-trip to the server with every new filter request. This can be costly, especially when live search is used, and might lead to a bad user experience, especially for applications mostly used on mobile devices. The alternative method is to use `Client`, which would trigger only local filtering. The result is shown in Figure 8.7.

```
onSearch : function(oEvent) {
  var sSearchValue = oEvent.getSource().getValue(),
  aFilters = [];
  if(sSearchValue.length > 0) {
    var oFilterName = new Filter("CustomerName", sap.ui.model.
      FilterOperator.Contains, sSearchValue);
    var oFilterID = new Filter("SalesOrderID", sap.ui.model.
      FilterOperator.Contains, sSearchValue);
    aFilters.push(new Filter({
      filters : [oFilterID, oFilterName],
      And : false}));
  }
  this.getView().byId("table").getBinding("items").filter(aFilters,
    "Application");
}
```

Listing 8.14 Handler Function for Search Functionality

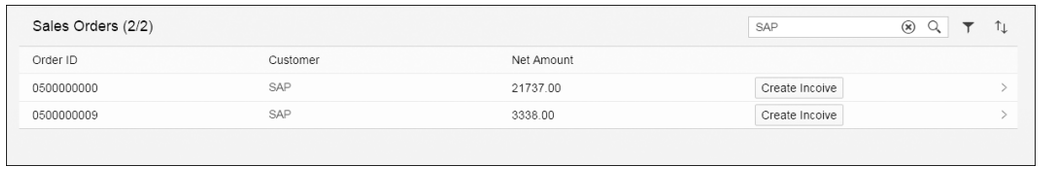


Figure 8.7 Search Handling in Worklist

Now that we've added the worklist table functionality for the worklist floorplan, in the next section, we'll provide functionality for navigation to the detail view.

Navigation and Detail View

In general, a worklist can offer two different types of navigation: *Inner-application navigation*, triggered by clicking on one of the list items, which brings the user to a second screen within the application that shows details for the selected item; and *cross-application navigation*, which can jump to a second application. We'll look into cross-application navigation in more detail in Section 8.4. Jumping to

an external website triggered by clicking a link can be a valid use case for a worklist, but this functionality should not be seen as mandatory and should be implemented based on user requirements.

Now, let's build a simple second screen and set up the routing to ensure that navigation within the application based on a click as well as deep links is possible. We've covered how to do this in code multiple times up to this point throughout the book. However, we'll now perform these functions based on the Descriptor Editor provided by the SAP Web IDE. This tool offers UI-based configuration of the manifest.json file and opens by default when opening any manifest.json file in the SAP Web IDE. Based on the work we've done already, when you open the Descriptor Editor and click on the ROUTING tab, the Descriptor Editor should look like Figure 8.8.

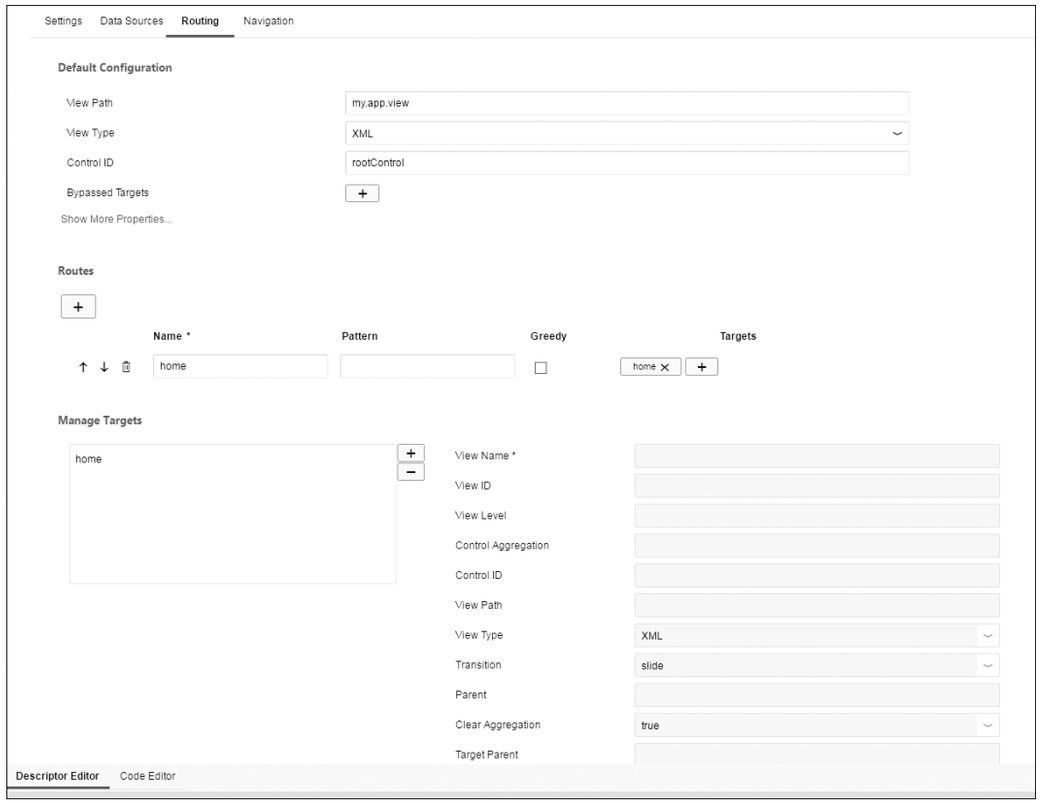


Figure 8.8 Descriptor Editor in SAP Web IDE

First, let's add a new route. To do so, click on the + button in the ROUTES section, and a new route will appear. Change the name for the new route; let's call it `salesOrder`. Because we want to have deep link capabilities for the new route, we'll also define a pattern here. Any string would work here, but we suggest making the link transparent to the user and calling it `SalesOrder/{SalesOrderID}`. The identifier in curly brackets now will be used to identify the distinct sales order to be displayed and handed over to the navigation step. This route now needs a target.

Note

The number of identifiers used in application patterns is determined by the number of identifiers defined in the metadata for the specific entity set. Otherwise, single entities cannot be addressed correctly.

In the MANAGE TARGETS section (refer to Figure 8.8), click on + and a popup will open (see Figure 8.9) in which you can define the name for the new target. Let's call it `salesOrder`.

Figure 8.9 Add Target

This target is created instantly, so we can configure it now. Here, all we have to do is define the VIEW NAME (SalesOrder) and we're done. Finally, we need to associate this target with the route. To do so, click on + in line with the route; a popup will open, and you can select `salesOrder` (see Figure 8.10).

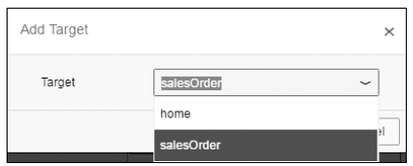


Figure 8.10 Target Assignment in the Descriptor Editor

Now, save the changes and run the application with a hash like so: `#SalesOrder/4711`. You'll see an error in the console indicating that SalesOrder.view.xml could not be loaded, which tells us that we did everything correctly and have to create the view now.

We've performed similar tasks related to navigation in previous chapters (see Chapter 4), so we don't want to repeat the individual steps here; instead, try to implement it on your own. When doing so, please keep in mind to separate concerns. The navigation step in particular may tempt you to build a close interaction between the two controllers. Use the router here to abstract the interaction by calling the `navTo` function on one controller and attaching two `patternMatched` events on the other controller. The complete code can be found in the Git repository that accompanies this book. Here's a brief outline of the steps to follow:

1. Create a new view called SalesOrder.view.xml in the *view* folder, and add some controls and relative binding. Don't forget to add BACK button handling.
2. Attach the `patternMatched` event in the controller for this view and bind the view to the `SalesOrderID` in the hash. Ensure that the metadata is already loaded (use `metadataLoaded` promise on the OData Model; see Listing 8.15).

```
this.getOwnerComponent().getRouter().getRoute("salesOrder").attachPa
tternMatched(function(oEvent) {
    var that = this;
    var sSalesOrderID =
 oEvent.getParameter("arguments").SalesOrderID;
    this.getView().getModel().metadataLoaded().then(function(){
    var sObjectPath =
 that.getView().getModel().createKey("SalesOrderSet", {
      SalesOrderID : sSalesOrderID
    });
    that.getView().bindElement({
        path : "/" + sObjectPath,
        parameters : {
            expand : "ToLineItems"
        }
    });
  });
}.bind(this));
```

Listing 8.15 Handling Binding on the SalesOrder.controller.js

3. Implement a press handler function that triggers navigation when an item in the worklist is clicked on.

Figure 8.11 shows how the final result looks like, based on the coding in the Git repository. Still, there are lots of variations possible, and the service we're using allows for displaying lots of related and additional information. For example, you could display the list of products associated in the sales orders here, the geo information of the supplier, and much more.

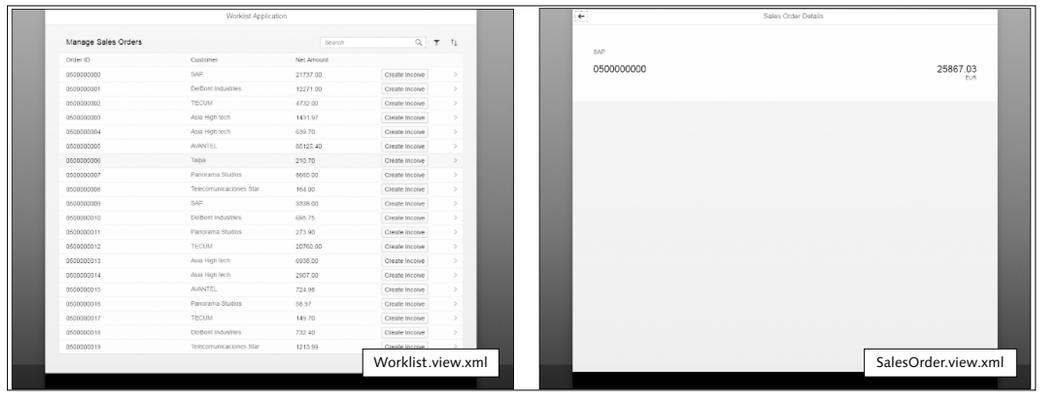


Figure 8.11 Worklist and SalesOrder Views

8.2.2 Master-Detail

In this section, we'll extend the split-screen layout we built in Section 8.1.2 and extend it to a master-detail floorplan. The master-detail floorplan, because of its different dependent content areas, is complex to implement, and its details and pitfalls could fill an entire book on its own. Therefore, we'll only explore its complexity and learn how to overcome some trouble areas of the master-detail floorplan—just enough to get a better understanding of SAPUI5 application development for this complex pattern overall. If you want to build a master-detail application in one of your projects, we highly recommend using the template available in the SAP Web IDE. The SAP Web IDE covers best practice implementation for all these little, but sometimes annoying details.

Before we dive deeper into the technical details for this floorplan, let's first discuss valid use cases in which master-detail should be your floorplan of choice. The buildup is pretty simple: We always have a list in the master section that displays a set of items. Based on what's selected in this list, a detail area provides more relevant information for the selected item. If you think of software you use on a daily basis, you'll find some examples of master-detail floorplans in action.

Think of email clients, local as well as web-based: Most of them have a list of emails on the left showing the most important information, and when one email is selected, the entire email text appears in a bigger content area on the right. Or, if you're an iPad user, you can see the master-detail pattern at work in your device settings.

From these examples, we can derive some golden rules for deciding when to use master-detail floorplans in applications. First, master-detail is helpful in cases that require minimal navigation, such as when you want your application users to be able to quickly switch between different business objects while always keeping the overview of the complete set of objects available.

However, this makes only sense if the amount of data displayed on the details screen is easy to consume. In the email client, the email content displayed upon selection is something a user can handle. The user clicks on an email stub and sees the entire email displayed. This pattern would simply not work if not only this email but six other related emails were displayed at once. This means that we should only use the master-detail floorplan if the amount of data to be displayed on the details side is strongly related to the content to be displayed on the master list.

Master List

Now, let's move into some hands-on work with the master-detail floorplan by creating the master list. We'll start by creating the controllers for the master view and the detail view, then we'll register them in the view, and then add the files to the controller folder. (We created the application skeleton with a `sap.m.SplitApp` control and basic routing that displays the empty master and detail pages in Section 8.1.)

In this section, we'll first concentrate on the master list and extend Master.view.xml (see Listing 8.16) with `sap.m.List` that we'll bind to `BusinessPartnerSet` in the OData service. `sap.m.List` offers several modes for single or multi select and some that affect general appearance. We'll use `sap.m.List` as `SingleSelectMaster`, the optimized mode for single selection on desktop devices. Individual items in this list will be displayed using `sap.m.ObjectListItem` to show a minimal set of details per item in a nice, card-like display. We'll also add a search field in the `subHeader` of the semantic page. (We covered search handling in Section 8.2.1.) Listing 8.14 provides the binding and event handlers.

```
<semantic:subHeader>
  <Bar id="headerBar">
    <contentMiddle>
    <SearchField id="searchField" search="onSearch"
      width="100%"/>
    </contentMiddle>
  </Bar>
</semantic:subHeader>
</semantic:content>
<List
  id="list"
  selectionChange="onItemPressed"
  mode="SingleSelectMaster"
  growing="true"
  growingScrollToLoad="true
    items="{
      path: '/BusinessPartnerSet'
    }">
  <items>
    <ObjectListItem
      title="{CcompanyName}"
      intro="{WebAddress}"/>
  </items>
</List>
</semantic:content>
```

Listing 8.16 Master List with Binding and Event Handlers in Place

Object View

For Detail.view.xml, we'll opt for a minimal display for now and will add `sap.m.ObjectHeader` with one bound property, which we'll bind later to the model relative to the selected item. Doing this requires one simple line of code in the content aggregation of `sap.m.semantic.SemanticDetailPage`: `<ObjectHeader title="{CompanyName}"/>`.

Synchronize Master and Detail

Because we now have some basic content for our two content areas in place, we need to orchestrate these two content areas in such a way that any selection in the master view reflects the content that is displayed in the detail view. To do so, we need to implement three features: handling of master list selections, full support for deep links, and handling of the default route. Finally, we have to follow some steps to ensure the master-detail floorplan can function for mobile scenarios as well.

Note

For now, we'll only address ideal cases. Error handling and "not found" handling scenarios will be covered in Section 8.3.

Handling of Master List Selections

Currently, the application has a master list with data, but no visible details about this data and no selectable content. In this section, we'll add some depth to the master list by providing details for its data upon selection. The first thing we want to do is create a new route, called `detail`, which will use a pattern from which we can extract the item ID later (see Listing 8.17). The targets we established previously can be reused; only the sequence is important. In Section 8.1.2, you learned that the first target defined in routes that are used with `sap.m.SplitApp` is to be displayed on mobile devices. It can be assumed that a user opening an application with a deep link wants to see the details page and not the master page first, so we'll add the detail target first and the master target second into the array.

```
{
  "name": "detail",
  "pattern": "BusinessPartner/{BusinessPartnerID}",
  "greedy": false,
  "target": ["master", "detail"]
}
```

Listing 8.17 Master-Detail Route for Deep Links

We will now add the function (`onItemPressed`) to handle selections in Master.controller.js (see Listing 8.18). We've done something similar several other times in this book (see Chapter 4). One particular function of the `selectionChange` event that we're using now is that you get the list item that was pressed as a parameter in the callback argument instead of calling `oEvent.getTarget()`. From this list item, we get the entity ID from the binding context and trigger navigation to the detail route that gets this ID as a parameter.

```
onItemPressed : function(oEvent) {
  var oItem = oEvent.getParameter("listItem");
  var sID = oItem.getBindingContext().getProperty("BusinessPartnerID");
  this.oRouter.navTo("detail", {
    BusinessPartnerID : sID
  }, false);
},
```

Listing 8.18 Handling of Press Event on Master List

We'll now attach to the `patternMatched` event in the Detail.controller.js and bind the view based on the parameter we just received (see Listing 8.19). Because `sap.ui.model.OData.V2.ODataModel` offers some functionality to create the key that can be used to bind the view (which is handy, especially for entity sets with more than one key), we can use this function. You just have to be aware that the actual key generated is dependent on the metadata.xml file already loaded and processed. We can use a promise provided by `sap.ui.model.OData.V2.ODataModel` here to secure this.

```
onInit : function() {
  this.oRouter = this.getOwnerComponent().getRouter();
   this.oRouter.getRoute("detail").attachPatternMatched(this.
     onDetailRouteHit.bind(this));
},

onDetailRouteHit : function(oEvent) {
  var sID =  oEvent.getParameter("arguments").BusinessPartnerID;
  this.getView().getModel().metadataLoaded().then(function(){
    var sObjectPath =
 this.getView().getModel().createKey("BusinessPartnerSet", {
      BusinessPartnerID : sID
    });
    this.getView().bindElement({
      path: "/" + sObjectPath,
    });
  }.bind(this)
}
```

Listing 8.19 Binding of Detail.view.xml Based on Navigation

Full Support for Deep Links

If you run what we have so far, it will appear as if nothing has changed. The master list appears with all the items, and no details are displayed. However, once you select an item in the list, the detail content area will be updated and will display what we have bound to the list item. We can even see in the URL that the pattern we defined before is filled, and the ID of the selected object is included there. If you now click Refresh in the browser, the detail matching the browser is displayed, but the focus on the master list for the selected item is not set. Now, let's select any item again. It becomes even more obvious that we missed something if we change the browser hash manually (e.g., from *#/BusinessPartner/0100000000* to *#/BusinessPartner/0100000004*). The detail changes, but the selection on the master list stays the same.

This is awkward for the user, but luckily we can fix this problem in the Master.controller.js. Here, we'll attach to the `patternMatched` event of the detail route and handle it in a function we'll call `onDetailRouteHit`. Because we'll have to handle different cases now and some exceptions, let's build our example up step by step. First, we'll create the functions described previously, (`patternMatched` and `onDetailRouteHit`) plus one additional function that we'll use to search items based on the key to review all the items the list (see Listing 8.20). The idea is now to call `selectAnItem` once the detail route is hit in order to support a deep link.

```
onInit : function() {
  // reuse variables
  this.oList = this.byId("list");
  this.oRouter = this.getOwnerComponent().getRouter();

  this.oRouter.getRoute("detail").attachEvent("patternMatched",
    this.onDetailRouteHit.bind(this));
},
onDetailRouteHit : function(oEvent) {
  var sBusinessPartnerID =
 oEvent.getParameter("arguments").BusinessPartnerID;
  this.selectAnItem(sBusinessPartnerID);
}
selectAnItem : function(sBusinessPartnerID) {
  var sKey = this.getView().getModel().
    createKey("BusinessPartnerSet", {
      BusinessPartnerID : sBusinessPartnerID
  });
  var oItems = this.oList.getItems();
    oItems.some(function(oItem) {
    if (oItem.getBindingContext() && oItem.getBindingContext().
      getPath() === "/" + sKey) {
      this.oList.setSelectedItem(oItem);
      return;
    }
  }, this);
},
```

Listing 8.20 Handling Simple Deep Links: First Try

We should now expect that the deep links should work. However, when we start the application to test it with a deep link (e.g., *#/BusinessPartner/0100000000*), the deep link doesn't work. An analysis with the F12 tools in your browser and adding a breakpoint to the `selectAnItem` function uncovers that when we call this function, there are no items in the list yet (see Figure 8.12). This is rather interesting and offers more insight into the lifecycle of routing itself. When the event

triggered, the list binding had not yet been resolved. Therefore, the list had no items to select from.

Figure 8.12 Analysis of Item-Selection Failure

We'll need to ensure that `sap.m.List` resolves its binding and that items are available to select from before the event is thrown. The easiest way to do this is to hook into an event called `updateFinished` that we can attach to. This event is thrown once the list binding update has completed. Therefore, we can be sure that there are items in the list by that point:

```
this.oList.attachEventOnce("updateFinished", function() {
this.selectAnItem(sBusinessPartnerID);
}.bind(this));
```

With this change, the deep links should work. However, we'll still run into issues later when we want to handle errors or "not found" cases, because we do not have this error as a status we can request at any time. We can solve this issue using a *JavaScript promise* (see Listing 8.21). This becomes a little complex, because we have to ensure two things now: First, that the view already has its binding, for which we'll use `eventDelegate` functionality to attach to an event of the parent control; and second, that the `dataRequest` event can be used to identify error cases. Now, we also can react when no data could be loaded for any reason. We'll implement this later in Section 8.3.2. Add the code in Listing 8.21 to the `onInit` method of Master.controller.js now.


```
var that = this;
this.oListBindingPromise = new Promise(
  function(resolve, reject) {
    that.getView().addEventDelegate({
      onBeforeFirstShow: function() {
      that.oList.getBinding("items").attachEventOnce("dataReceived",
        function(oEvent) {
        if(oEvent.getParameter("data")){
          resolve();
        } else {
          reject();
        }
      }, this);
    }.bind(that)
  });
}
);
```

Listing 8.21 Promise to Decouple Navigation from Events

We now simply select an item programmatically once `oListBindingPromise` has resolved. However, changing the hash manually does not change the selection. We'll need to add some more logic to the `onDetailRouteHit` function to get this right.

We'll now handle these three cases individually. First, we'll handle the case in which a user selects an item manually. In this case, we simply do nothing. In the second case, the classical deep link scenario, we select an item once the binding has resolved. For all other cases, mainly the manual hash change is handled here, and we can simply select the item straight away (see Listing 8.22).

```
onDetailRouteHit : function(oEvent) {
  var sBusinessPartnerID =
 oEvent.getParameter("arguments").BusinessPartnerID;
  var oSelectedItem = this.oList.getSelectedItem();
  if (oSelectedItem && oSelectedItem.getBindingContext().
    getProperty("BusinessPartnerID") === sBusinessPartnerID) {
    return;
  } else if (!oSelectedItem) {
    this.oListBindingPromise.then(function() {
      this.selectAnItem(sBusinessPartnerID);
    }.bind(this));
  } else {
    this.selectAnItem(sBusinessPartnerID);
  }
},
```

Listing 8.22 Optimized Detail Route Handling

Handling the Default Route: Empty Pattern

As a last step, we want to cover the *empty pattern route*. An empty pattern route is hit whenever an application starts without a hash. In such a case, the current application doesn't display anything, which is not preferred; the preferred option is to display the first list item details. In addition, we'll also show that the first item is selected. Most of the code in Listing 8.23 should make sense by now and the function will be called once the master route was hit. Again, we have to ensure that the promise is resolved before we can determine the first item and trigger the navigation for the detail.

```
onMasterRouteHit : function() {
  this.oListBindingPromise.then(function() {
    var oItems = this.oList.getItems();
    this.oList.setSelectedItem(oItems[0]);
    this.oRouter.navTo("detail", {
      BusinessPartnerID : oItems[0].getBindingContext().
        getProperty("BusinessPartnerID")
    });
  }.bind(this));
},
```

Listing 8.23 Empty Pattern Route Handling

Support for Mobile Devices

For desktop devices and tablets in landscape mode, our application should work fine. Still, we also have to plan for devices that do not offer enough real estate to fit an entire master-detail layout on one screen. To do so, we'll make use of dynamic expressions in XML and the `sap.ui.Device` API that identifies device type, touch support, and much more on application startup.

If you now run this application in device emulation mode in Google Chrome with an empty hash, you'll see that it instantly jumps to the detail screen for the first item, which is not our intent. We want it to stay on the master list if the main route is hit. The following simple return statement that only comes into play on mobile devices in the function that handles the main route will fix this problem:

```
if(sap.ui.Device.system.phone){ return;}
```

If you rerun the application in Google Chrome now, you'll land on the master list. If you select an item in the master list, the navigation brings you to the detail screen.

Everything seems to work, but from the detail screen there is no easy way to get back to the master list page. We have to add a Back button to Detail.view.xml and ensure that it will only be displayed on phones. Again, we'll use the `sap.ui.Device` API, this time as a dynamic expression directly in XML and based on the same path we used previously in the `return` statement:

```
showNavButton="{= ${device>/system/phone}}"
navButtonPress="onNavButtonPressed"
```

We've also added the name of a handler function that will navigate back to the master list; we'll will implement this function in Detail.controller.js like this:

```
onNavButtonPressed : function(){
this.oRouter.navTo("master");
}
```

If you click on the Back button in the top left of the detail view now, you're returned to the master list. However, one slightly unfortunate detail is that the last item selected is still selected in the master list. This makes no sense, because we don't have something that reflects the selection on the detail side of the screen. We can suppress this selection in the list by using a different `listMode`. We now have `listMode` set to `SingleSelectMaster`. We'll also use another expression to set `listMode` to `None` on mobile devices, like so:

```
mode="{= ${device>/system/phone} ? 'None' : 'SingleSelectMaster'}"
```

This change will make some more changes necessary, because the `listMode` set to `None` will also result in the `selectionChange` event no longer being thrown. So far, we've used this event to handle clicks on list items. Now, we'll have to add a `press` handler for individual list items instead. It's possible to handle clicks on mobile devices differently from clicks on desktop devices by simply defining two handler functions. However, in the case, the same function will work for both types of devices. We also have to dynamically set the type of the list items to `Active` on mobile devices to make the items clickable, like so:

```
type="{= ${device>/system/phone} ? 'Active' : 'Inactive'}"
press="onItemPressed"
```

We need to make one more adaption to the handler function. Because the `selectionChange` event returns the list and the item as a parameter and the `press` event on an individual item returns itself as the source of the event, we'll have to cover both cases in the handler:

```
var oItem = oEvent.getParameter("listItem") || oEvent.getSource();
```

One last feature we want to handle differently on mobile devices is the way the user can refresh the master list. For desktop devices, we already display a Refresh button in the Search field, but for touch-supported devices, we should to use a pull-to-refresh feature to handle refreshing the master list. This feature is fairly simple to add. On `sap.m.SearchField`, we can add a dynamic expression that will set the `showRefreshButton` property for us:

```
showRefreshButton="{= !${device>/support/touch} }"
```

We'll also add a `sap.m.PullToRefresh` control to the content aggregation of the semantic page. Again, we'll let a dynamic expression handle the visibility for us:

```
<PullToRefresh id="pullToRefresh"
refresh="onRefresh"
visible="{device>/support/touch}"/>
```

To make `sap.m.PullToRefresh` work, we have to do two more things:

1. Hide the control once the refresh is over (ideally in the `updateFinished` event on the list), like so:

   ```
   this.byId("pullToRefresh").hide();
   ```

2. Perform the actual refresh on the list binding (in the refresh event handler), like so:

   ```
   this.oList.getBinding("items").refresh();
   ```

Note

`sap.m.PullToRefresh` has to be used as the first element in the content aggregation of the first `sap.m.ScrollContainer` on the page. Otherwise, you might experience severe rendering issues that might break the usability of your application completely.

The control also could be used on nontouch devices, resulting in the display of a clickable Refresh area.

8.3 Additional Application Features

Independent from any floorplan, the apps within certain layouts generally have some qualities or features that are always needed. Application users take most of these features for granted. We have to confess that if we put ourselves into the

position of application users—which we are, in fact, on a daily basis—we would expect, for example, to be notified if something goes wrong in an app. Many of us would associate this with some technical error—for example, when writing data to the backend—whereas others may first think about some deep link that could not be resolved as expected. Error handling and "not found" handling only form the tip of the iceberg.

However, there is much more to be considered in application development in general—not only ensuring that applications work as expected from a technical perspective, but also ensuring that they provide the user with the best possible support to fulfill daily routines.

In the following sections, we provide a quick rundown of technical and user experience-related shared application qualities and how SAPUI5 offers support in their implementation.

8.3.1 Not Found Handling

Error code 404 may be the only status code that even casual users understand. Many websites and web apps tend to spit out this technical information on the screen whenever the page a user wants to access is not available. Although there has been a trend in recent years to enrich these "not found" pages with funny designs, the numeric code seems never to disappear.

In this section, you'll learn about "not found" handling within the master-detail floorplan. Since version 1.28, SAPUI5 has provided a page to be displayed in not found cases, which is `sap.m.MessagePage`; it should be used as a single control in a view, like so:

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

Figure 8.13 shows the default display of this page.

Admittedly, its design is very business-like, but it's fit for its purpose. We'll learn how to tweak it a bit later, but first, let's look at some use cases.

On websites, you will typically have only one `notFound` page that handles all links that can't be resolved. When using business applications, more precise feedback for the user is desirable, and with a well-defined, single-purpose application, it's

easy to narrow down cases to be handled. We'll look at how to do so in the following subsections.

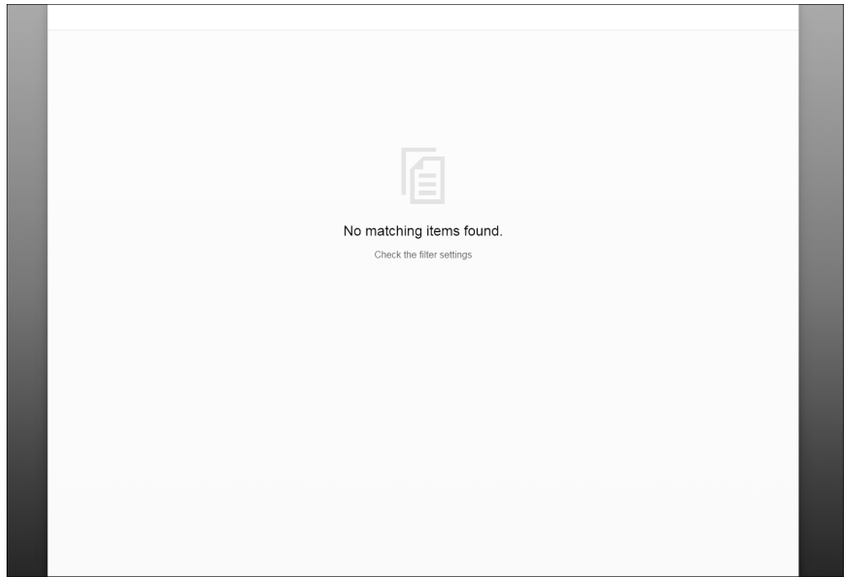


Figure 8.13 sap.m.MessagePage with Default Settings

BusinessPartnerNotFound Scenario

In routing with SAPUI5, you can define routes that have specific patterns that should be reflected in the URL. That's what we call a *deep link*.

Because these patterns often hold the ID that matches a specific data set that could later be used to bind it to a view (for a master-detail example, see Listing 8.24), we have to handle all those cases in which individual IDs can't be found in the database. The aim is to show a not found page that gives some details about what went wrong and offers a link back to the application in a valid state whenever the user enters the application with a deep link to a business partner that does not exist.

```
{
  "name": "detail",
  "pattern": "BusinessPartner/{BusinessPartnerID}",
  "greedy": false,
  "target": ["master", "detail"]}
```

Listing 8.24 Route with ID in Pattern

Now, let's define a target that should be displayed when a specific business partner can't be found. We'll need a new view to handle these cases (see Listing 8.25). We'll reference a new view in the route and call it BusinessPartnerNotFound.view.xml.

```
"businessPartnerNotFound": {
  "viewName": "BusinessPartnerNotFound",
  "controlId": "rootControl",
  "controlAggregation": "detailPages"
},
```

Listing 8.25 Target for ObjectNotFound Scenarios

We'll create this view accordingly and also customize `sap.m.MessagePage` a little to create a nice display (see Listing 8.26). Because the view will be displayed in the `detailPages` aggregation, we again have to make sure that navigation back is possible on mobile devices and must use an expression to show a button for such navigation.

```
<mvc:View
  controllerName="my.app.controller.Main"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m">
  <MessagePage
    icon="sap-icon://doctor"
    navButtonPress="backToHome"
    showNavButton="{device>/system/phone}"
    text="BusinessPartner not found"
    title="Something went wrong">
    <customDescription>
        <Link text="click here to get back to main page"
          press="backToHome"/>
    </customDescription>
  </MessagePage>
</mvc:View>
```

Listing 8.26 BusinessPartnerNotFound.view.xml

We'll simplify a bit by using the existing Main.controller.js file to implement the handler functions for not found cases. In a real application, it might make sense to have an shared controller for these cases. For the back navigation, we'll use the same logic as in Section 8.2.2. To achieve this behavior and the resulting display, shown in Figure 8.14, we have to add some logic to Detail.controller.js, in the `DetailRouteHit` function. We'll use events of the binding to implement the back navigation and extend the call of `bindElement` with the `change` event. We'll then

display the `BusinessPartnerNotFound` target whenever no `bindingContext` is set on the view, indicating some error, as in Listing 8.27.

```
this.getView().bindElement({
    path: "/" + sObjectPath,
    events: {
        change: function(){
          var oView = this.getView();
          if(!oView.getElementBinding().getBoundContext())){
            this.oRouter.getTargets().display("businessPartnerNotFound");
            }
        }.bind(this)
    }
});
```

Listing 8.27 Handling Business Partner Not Found Scenario

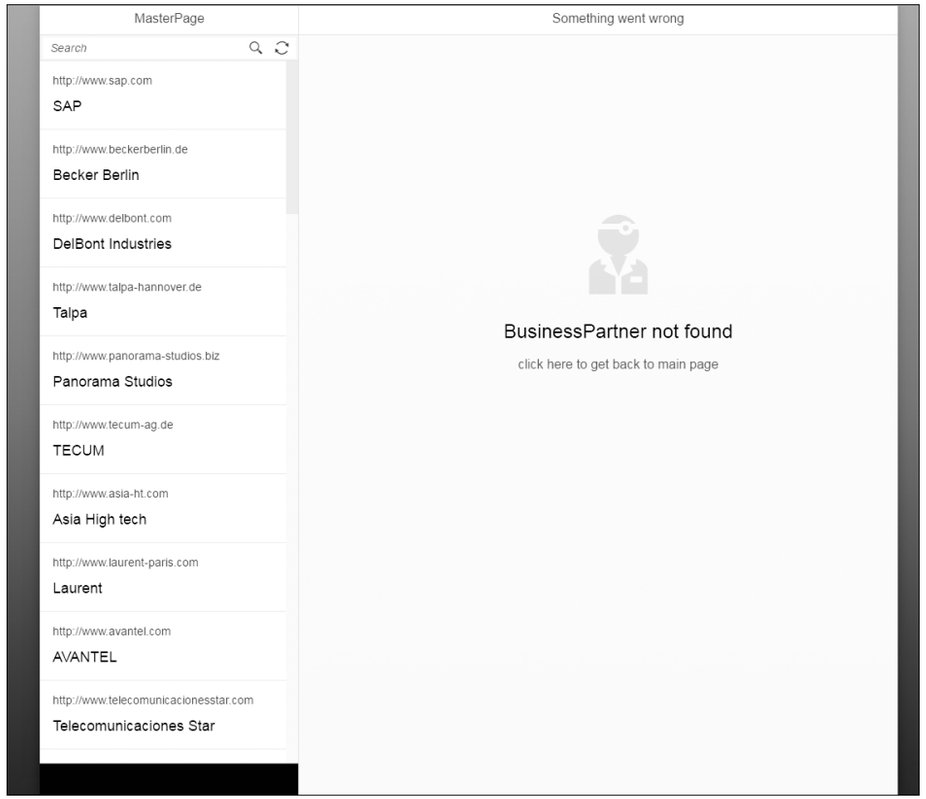


Figure 8.14 Business Partner Not Found Display

catchAll Scenario

The second situation we have to handle is generic not found cases, also referred to as `catchAll` cases. These are cases in which the user has tried to enter the application with a URL that does not match any pattern defined in the routing configuration. Luckily, this scenario is easier to implement than the previous scenario because it's not dependent on the application data from the backend. SAPUI5 routing provides a generic `bypassed` route for `catchAll`. The target(s) that should be displayed in all these cases can simply be handed over to the `config.bypassed` property in the routing configuration, as in Listing 8.28. The target that will be declared for bypassed has to be created as well. This can be done as in Listing 8.25, but we recommend using a different target and view for generic cases. This will help the user differentiate between the two situations.

```
"routing": {
  "config": {
    "bypassed": {
      "target": ["notFound", "master"]
    }
  }
}
```

Listing 8.28 Bypassed Configuration for Generic notFound Cases

Finally, we need to handle existing selections on the master list. Imagine someone manipulating the hash manually to something that is not defined in any pattern. The correct not found page will be displayed, but the selected item remains the same. To handle this deselection, attach to the `bypassed` event routing provided, and release the selection on the master list (see Listing 8.29). The result should then be as shown in Figure 8.15.

```
this.oRouter.attachEvent("bypassed", function() {
  this.oList.removeSelections(true);
}.bind(this));
```

Listing 8.29 Handling of List Selections for Bypassed Cases

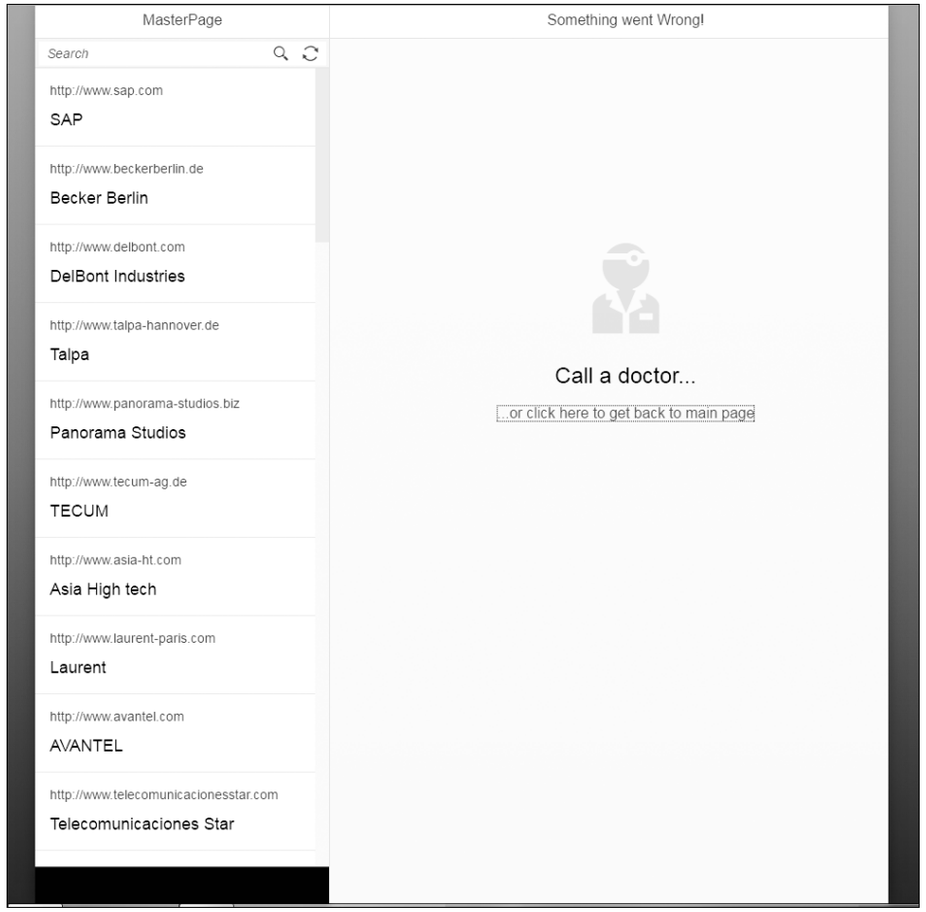


Figure 8.15 Customized sap.m.MessagePage for catchAll Cases

8.3.2 Error Handling

You've seen that not found cases and error cases have to be differentiated, and now we'll draw a clear line between them. In *error cases*, a technical error occurs that leads to the application no longer being usable. This also means that the notification for the user should occur in a more disruptive way. Best practice would be to make sure there is a clear indication that they should reload the application. Therefore, we will use a modal dialog for notifying the user. SAPUI5 provides `sap.m.MessageBox` as a convenient API that wraps `sap.m.Dialog` and additional

controls. We'll handle two cases in the following subsections, but both will be displayed in the same `sap.m.MessageBox`. Therefore, we'll use a function to bring up the notification and reuse it for both cases (see Listing 8.30). We'll implement this function in the Component.js file of our application. Please note that there may be more than one error raised by the application. To ensure that there will be only one `sap.m.MessageBox` displayed, we'll use a simple flag that indicates that a notification is already present.

```
_showServiceError: function(sDetails) {
  if (this._bMessageOpen) {
    return;
  }
  this._bMessageOpen = true;
  MessageBox.error("An Error Occurred",
    {
      details: sDetails,
      actions: [MessageBox.Action.CLOSE],
      onClose: function() {
        this._bMessageOpen = false;
      }.bind(this)
    }
  );
}
```

Listing 8.30 Generic Error Notification Function

Error Response Specification

In SAP NetWeaver and SAP Gateway OData services, there is an SAP-specific response protocol that ensures that all server messages are returned with predictable formatting. This function is handled by `sap.ui.model.odata.ODataMessageParser`, and all messages can then be accessed through `sap.ui.core.message.MessageManager`. Because these functions currently cover validation-related messages only, generic handling for error messages cannot be described at this point in time; such handling might vary based on your service implementation and other factors.

We'll also disable the automatic closing of dialogs on the router target handler, which could be controversial. However, because we will display dialogs and specific pages, and because it's not possible to synchronize the two events (routing and data requests), disabling automatic dialog closing is a valid option for most of use cases. Disable automatic dialog closing as follows:

```
this.getRouter().getTargetHandler().setCloseDialogs(false);
```

Handling Metadata Errors

For SAPUI5 applications built on top of OData services, there will always be cases in which a metadata call did not result in a success. Let's handle such cases now. We can simply attach to the `metadataFailed` event provided by `sap.ui.model.V2.ODataModel` and display a message box showing details. Let's also display the generic not found page:

```
this.getModel().attachEvent("metadataFailed", function(oEvent) {
this._showServiceError(oEvent.getParameters().getResponse);
this.getRouter().getTargets().display("notFound");
}.bind(this));
```

Handling Service Errors

For service errors, we'll need some more logic, although the overall pattern remains the same. We'll again attach to a model event—in this case, the `requestFailed` event. However, because this event is thrown for cases we already handled using the not found implementation, we'll have to exclude such cases. Therefore, we'll make the following assumptions based on the error code the event provides as part of the parameters: All 404 cases (not found) and all 400 cases (parsing error on the server) will not be handled by the error handling, because they're already covered by the not found handling, resulting in the handler function in Listing 8.31.

```
this.getModel().attachRequestFailed(function(oEvent) {
  var oParams = oEvent.getParameters();
  if (oParams.response.statusCode !== "400" &&
 oParams.response.statusCode !== "404") {
    this.getRouter().getTargetHandler().setCloseDialogs(false);
    this.getRouter().getTargets().display("notFound");
    this._showServiceError(oParams.response);
  }
}, this);
```

Listing 8.31 Handling Request Errors

8.3.3 Busy Handling

As a user, *busy handling* gives you the feeling that the hard work is done for you by showing a busy indicator. Busy handling is not only the real work an application is doing but also the responses you get in general regarding the state your application is currently in. We bet there are more apps out there that fake actual

busy time just to display nice busy animations than you might imagine. The reason for this may be that the screen flickers if the actual request only takes milliseconds, and the busy indicator will be shown and hidden again instantly. In SAPUI5, there is a default `busyIndicatorDelay` property on all controls that defaults to 500 milliseconds; we should keep it that way instead of delaying the response artificially.

Busy handling is important for the perceived performance of an application, especially at startup. It's good to assume your user has a slow Internet connection. A busy indicator showing the user that there is some work being done behind the scenes will keep him patient.

Handling the Metadata Call

As for error handling, we can differentiate two cases or, more precisely for this section, two phases of loading data. First, the OData metadata.xml file is requested. During this time, the application is not ready to work at all. We'll therefore set the outer view (Main.view.xml) as busy during this phase. The easiest way to do so is to set busy as the default behavior in the XML (`busy="true"`) for our root control, and later, when the metadata is loaded or loading failed, simply call `setBusy(false)`, as in Listing 8.32.

```
// handling the good case
this.getOwnerComponent().getModel().metadataLoaded()
  .then(function() {
    oRootControl.setBusy(false);
  });
// handling the bad case
this.getOwnerComponent().getModel().attachMetadataFailed(
  function() {
    oRootControl.setBusy(false);
});
```

Listing 8.32 Metadata Request: Busy Handling

Handling Calls on Individual Controls

While the metadata call is happening, individual requests are triggered only once at application startup, but binding refresh will occur multiple. Therefore, we should ensure that busy handling for these cases is in place.

For `sap.m.List` and `sap.m.Table`, busy handling is already implemented as a default, so we don't have to do anything for these controls. For all other controls,

we should use the appropriate events to manage busy handling. However, individual requests are triggered only once at application startup, but binding refresh will occur multiple. One rule of thumb for determining this information is to have all controls bound against the same entity. If we look at the master-detail example, this entity would be the entire detail page in the current state. However, in real applications, you would most likely fill up the detail screen and may even expand the displayed data to a related entity in the service. Possibilities for the service we've been using in this example are shown in Figure 8.16. Now, let's assume we want to display a list of sales orders next to the business partner details on the details page. In this case, we would handle the busy state for this area within the screen separately from the `sap.m.ObjectHeader` element in which we're displaying the business partner details currently.

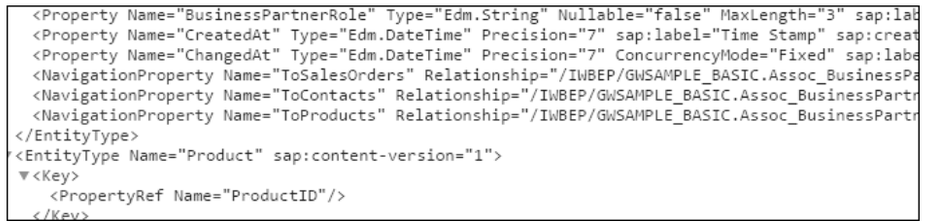


Figure 8.16 Related Entities to Business Partners

For handling the busy state, technically, we'd use binding events, as previously stated. The most appropriate choice would be to set the control to busy once data is requested and release the busy state once data is received by the control. An example implementation for this setup can be found in Listing 8.33. Here, we've implement the functions in the controller for the view that declares the controls and added the handler functions to the controls at declaration in XML. The actual implementation for this minimal example in master-detail is in Listing 8.34.

```
onDataRequested: function(oEvent) {
  oEvent.getSource().setBusy(true);
},
onDataReceived: function(oEvent) {
  oEvent.getSource().setBusy(false);
}
```

Listing 8.33 Generic Busy Handling

```
this.getView().bindElement({
  path: "/" + sObjectPath,
  events: {
    change: function(){
            if(!this.getView().getElementBinding().getBoundContext()){
                this.oRouter.getTargets().display("businessPartnerNotFo
und");
            }
        }.bind(this),
        dataRequested: function() {
          this.getView().setBusy(true);
        },
        dataReceived: function() {
            this.getView().setBusy(false);
        }
    }
});
```

Listing 8.34 Master Detail Busy Handling: Minimal Example

8.3.4 Letterboxing

Letterboxing is a term often associated with filming to ensure the original aspect ratio when transferring video material across screens with different ratios. This is achieved by using black bars, mostly displayed on the top and bottom of the screen to narrow the actual screen and fill the spaces that aren't covered by the film itself.

In application development, letterboxing has become a good practice for all cases when content is limited. Think of a simple master-detail application that only displays some details for a selected item. In such a case, it's much easier to ensure good design for the content on the screen if you can rely on a fixed content area, even on big screens. In addition, your application users will gain a more focused view.

The application examples we've presented so far have always run in `sap.m.Shell`, which uses a letterboxed display for applications to center the content by default. However, letterboxing can be disabled, because it's reflected in a property called `appWidthLimited` in the shell. This letterbox option provides a width of 1,280 px reserved for the content, and the rest of the screen displays the default application background. This background can be customized based on either the theme used or settings in the shell itself.

In order to change the behavior as shown in Figure 8.17, configure the properties on instantiation in `sap.m.Shell`, as shown in Listing 8.35.

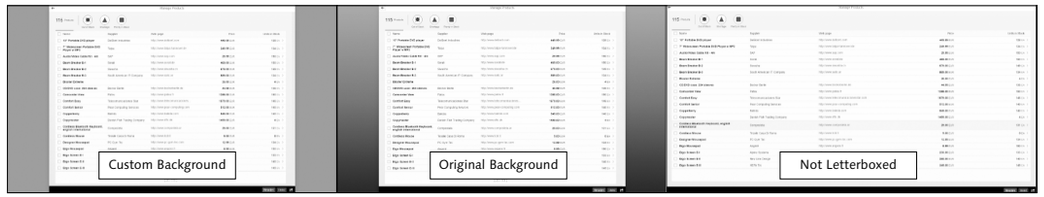


Figure 8.17 Letterboxing and Custom Background on sap.m.Shell

Unfortunately, there is no Explored app example for the control. Additional options include `backgroundImage`, `backgroundRepeat`, and `backgroundOpacity`.

```
// not letterboxed
new sap.m.Shell({
  appWidthLimited : false,
  app: new sap.ui.core.ComponentContainer({
    height : "100%",
    name : "myCompany.myApp"
  })
}).placeAt("content");

// custom background
new sap.m.Shell({
  backgroundColor : "rgb(0,153,204)",
  app: new sap.ui.core.ComponentContainer({
    height : "100%",
    name : "myCompany.myApp"
  })
}).placeAt("content");
```

Listing 8.35 Appearance Configuration for sap.m.Shell

8.3.5 Headers and Footers

Headers and footers generally provide access to certain functionality for applications users. This functionality can impact entire content areas. For example, if your application view contains a form, the button to save the form content to the backend should be displayed in the footer. This also ensures applications have focused content areas, so you're not tempted to overload your screens.

In SAPUI5, headers and footers are mainly part of `sap.m.Page` or other related controls. In this chapter, we've mainly used pages from `sap.m.semantic`, but we haven't yet looked into the main benefit they deliver: predefined buttons, so-called actions that implement design guidelines like predefined icons, texts, tooltips, and even overflow handling (also clustering on the screen according to their

distinct usage). For example, the BACK navigation button is left in the header, and all actions related to collaboration (e.g., `SendEmailAction`) are hidden in an overflow and display upon clicking the OVERFLOW INDICATOR button in a popover. Certain actions like `PositiveAction` are displayed in the app, making use of semantic colors. This highly improves development routines when designing new screens and ensures minimal distraction from implementing underlying functionality. Each action has a `press` handler that points to the controller for the view and fires the matching function there, just like it's always handled in SAPUI5.

In addition, you can add custom content in the headers and footers of the semantic pages, because they offer the `customHeaderContent` and `customFooterContent` aggregations and a `subHeader` aggregation for ambitious projects. There is even a `customShareMenuContent` aggregation that allows you to add custom actions into the popover described previously.

Let's now look at a sample `sap.m.semantic.FullscreenPage` installation with some actions and custom content; this page could be used for a simple shopping cart checkout page (Figure 8.18).

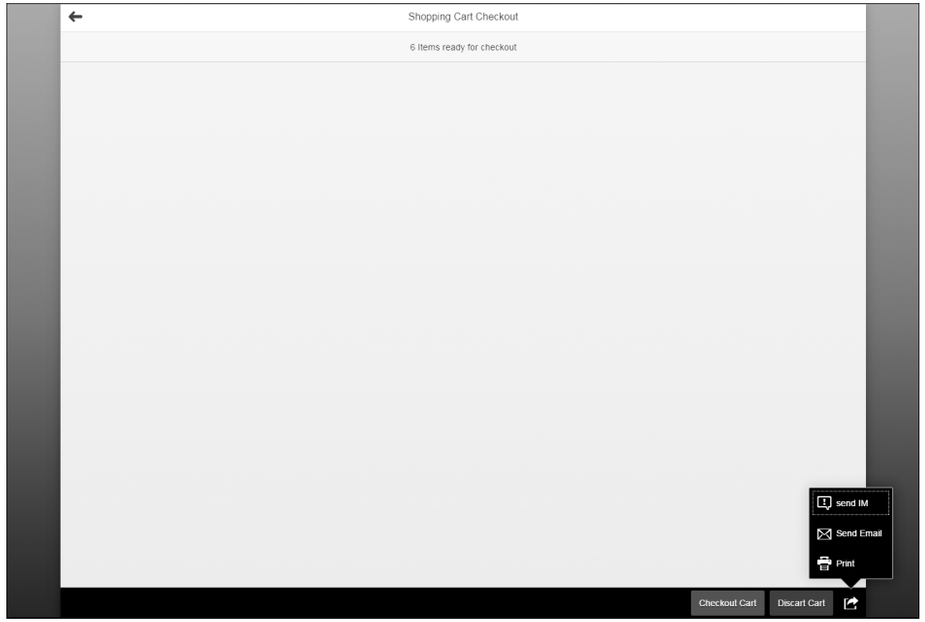


Figure 8.18 Header and Footer Options with Semantic Page

The XML to achieve this setup is pretty simple. You might have to get used to the high number of aggregations that are used here, making the API a little superfluous overall, but once you've adjusted to it, it works very well (see Listing 8.36).

```
<mvc:View
  xmlns:html=http://www.w3.org/1999/xhtml
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
  xmlns:semantic="sap.m.semantic">
  <semantic:FullscreenPage
    title="Shopping Cart Checkout"
    showNavButton="true">
    <semantic:subHeader>
      <Toolbar>
        <ToolbarSpacer/>
          <Text text="6 Items ready for checkout"/>
          <ToolbarSpacer/>
      </Toolbar>
    </semantic:subHeader>
    <semantic:sendEmailAction>
      <semantic:SendEmailAction press="onSendMailPressed"/>
    </semantic:sendEmailAction>
    <semantic:printAction>
      <semantic:PrintAction press="onPrintPressed"/>
    </semantic:printAction>
    <semantic:positiveAction>
      <semantic:PositiveAction text="Checkout Cart" press=
"onCheckoutPressed"/>
    </semantic:positiveAction>
    <semantic:negativeAction>
      <semantic:NegativeAction text="Discart Cart" press=
"onDiscartPressed"/>
    </semantic:negativeAction>
    <semantic:customShareMenuContent>
      <OverflowToolbarButton icon="sap-icon://message-popup" text=
"send IM" press="onPress"/>
    </semantic:customShareMenuContent>
  </semantic:FullscreenPage>
</mvc:View>
```

Listing 8.36 XML Declaration for Header and Footer Options with Semantic Page

The additional application features described in the current section allow you to control errors and wait times and to implement letterboxing and adjust headers and footers.

In the next section, we'll look at how to run apps in SAP Fiori Launchpad.

8.4 Running Apps in SAP Fiori Launchpad

In this chapter, you've learned that building application families that distribute functions across single-purpose-based apps is better than building one large, monolithic, multipurpose app. We have successfully built at least the outline of two small, single-purpose applications.

SAP Fiori Launchpad offers user management, application provisioning, navigation and integration of new applications, and maybe even third-party technologies. The launchpad's main purpose is to provide access to several applications and application types via one simple user interface. What sounds like a link list at first is actually a challenge not only from a technological standpoint but also from a user experience perspective. Just think of the challenge to support older technologies like Web GUI transactions as well as modern web applications like the ones we build with SAPUI5.

For our SAPUI5 applications, SAP Fiori Launchpad offers tight integration, because SAP Fiori Launchpad is itself based on SAPUI5 technology. SAP Fiori Launchpad offers a lot of functionality, not only for application users but also for application developers. Cross-application navigation from one application to another and the ability to programmatically create bookmarks that reflect a certain application state are just two features that come to mind. It soon becomes obvious that integration of at least a sandboxed SAP Fiori Launchpad early on during implementation will pay off later.

In this section, we'll start with the implementation of a simple standalone SAP Fiori Launchpad sandbox demo application and then extend it to include more than one application together in one sandboxed SAP Fiori Launchpad. From there, we'll add simple cross-application navigation using SAP Fiori Launchpad's API. Finally, we'll try out productive usage when we deploy our app to SAP HCP via the SAP Web IDE.

8.4.1 SAP Fiori Launchpad Sandbox

SAP Fiori Launchpad, when used productively, has several backend dependencies that can't be simulated in the context of single app development easily. That's why an SAP Fiori Launchpad sandbox is available that offers the most widely used features with a minimal footprint but still allows for testing during development. The display can be seen in Figure 8.19.

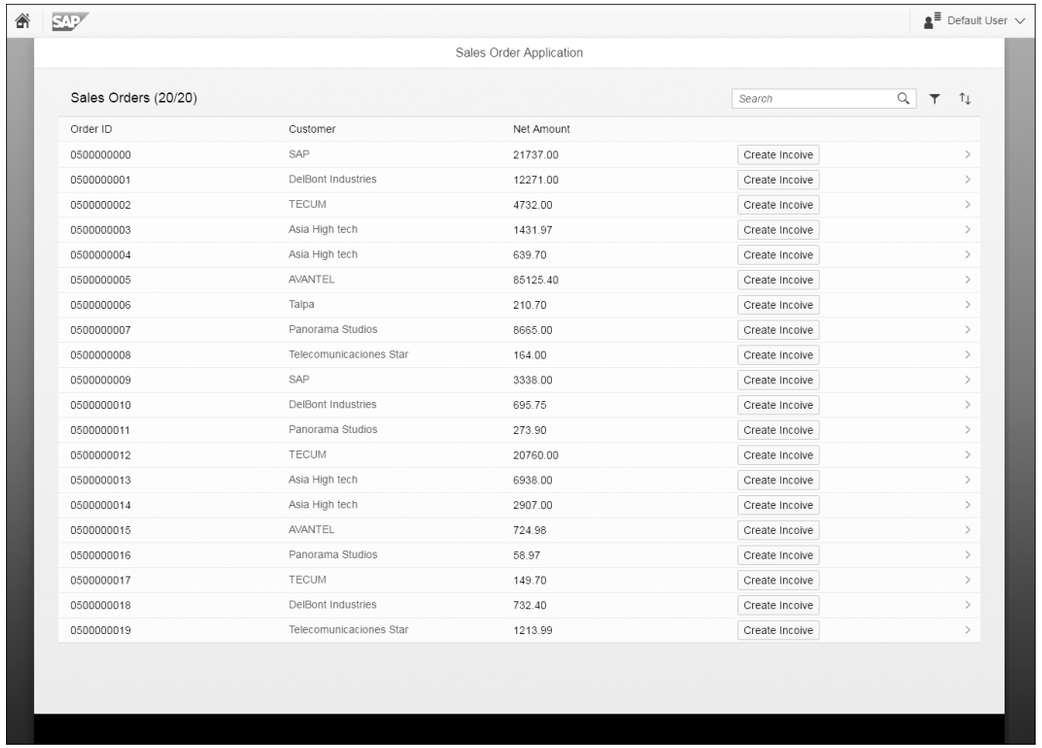


Figure 8.19 Generic SAP Fiori Launchpad Sandbox UI

There are several options to run your application in an SAP Fiori Launchpad sandbox. It's important to understand that within SAP Fiori Launchpad, you will not need a dedicated HTML file per application anymore; you'll simply register your application to the sandbox itself.

In this section, we'll look at running an application in a sandbox SAP Fiori Launchpad via the SAP Web IDE and in a custom-built sandbox SAP Fiori Launchpad.

SAP Fiori Launchpad Sandbox Runner in SAP Web IDE

The most convenient option to run your application in the SAP Fiori Launchpad sandbox is to use the built-in component runner provided by the SAP Web IDE. This feature offers a simple SAP Fiori Launchpad sandbox in which you can

currently run one application component at a time. To do so, right-click on Component.js, and the context menu opens as shown in Figure 8.20; then select Run • Run as • SAP Fiori Component on Sandbox, and your component will be launched in a minimal sandbox, as shown Figure 8.19. Here, you can test that the application still runs within SAP Fiori Launchpad. The scope of this option is still limited, and integrations such as cross-application navigation are not supported.

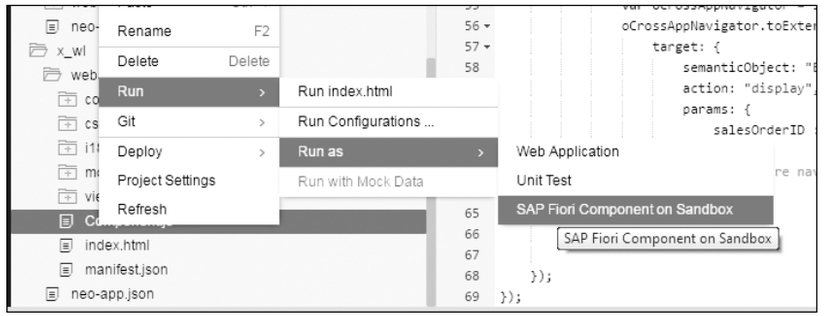


Figure 8.20 Usage of SAP Fiori Launchpad Sandbox Component Runner

Custom-Built SAP Fiori Launchpad Sandbox: Experimental

To test cross-application features in the SAP Web IDE, you can bootstrap your own sandbox. Note that this approach should be used for testing purposes only; any productive usage is not encouraged. However, this is a simple way to avoid deploying every change. Be sure to test early in the process directly in the workspace.

To do create a custom sandbox, we'll first create a new folder in our workspace. In this folder, we'll add a new HTLM file called FLPSandbox.html. In this file, we'll place a script block that will handle the SAPUI5 bootstrap as we did for all runnable files before (see Listing 8.37). In addition, we'll add some configuration for the SAP Fiori Launchpad sandbox and load an additional bootstrap script. The only important point to note here is to pay attention to the application's property block within the configuration. We'll add more here to register the application components to the sandbox in the next step.

```
<!DOCTYPE HTML>
<html>
  <head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

```
  <meta charset="UTF-8">
  <title>FLP Sandbox</title>
  <script>
    window["sap-ushell-config"] = {
      defaultRenderer : "fiori2",
      renderers: {
        fiori2: {
        componentData: {
          config: {
          search: "hidden"
        }
      }
    }
  },
  applications: {};
</script>

<script src="../test-resources/sap/ushell/bootstrap/sandbox.js" id=
"sap-ushell-bootstrap"></script>
<!-- Bootstrap the UI5 core library -->
<script id="sap-ui-bootstrap"
  src="../../resources/sap-ui-core.js"
  data-sap-ui-libs="sap.m, sap.ushell, sap.collaboration"
  data-sap-ui-theme="sap_bluecrystal"
  data-sap-ui-compatVersion="edge">
</script>

<script>
  sap.ui.getCore().attachInit(function() {
  // initialize the ushell sandbox component
  sap.ushell.Container.createRenderer().placeAt("content");
  });
</script>
</head>

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

Listing 8.37 SAP Fiori Launchpad Sandbox Initialization

In the same folder, we'll create a new folder for every app we want to run within the custom SAP Fiori Launchpad sandbox and give each one a meaningful name. Into these folders, we'll copy the respective *webapp* folders of the applications we want to run—for example, for the *Sales Orders* and *Business Partners* applications we created in Section 8.2.

Let's now register our applications to the sandbox. To do so, we'll add a new key referencing an object for every application to the applications settings block

you've seen before. This key serves as the hash to be resolved by the SAP Fiori Launchpad on navigation later. The settings for each application should be easy to understand: We need to give the component a namespace, a type, and a relative URL for where to find the component. The title can be chosen freely and will later be displayed on a tile. The coding to add the Sales Orders and Business Partners applications is found in Listing 8.38.

```
"SalesOrder-display": {
  additionalInformation: "SAPUI5.Component=sales.order.app",
  applicationType: "URL",
  url: "./SalesOrders/webapp/",
  title: "Sales Orders"
},
"BusinessPartner-display": {
  additionalInformation: "SAPUI5.Component=business.partner.app",
  applicationType: "URL",
  url: "./BusinessPartners/webapp/",
  title: "Business Partners"
}
```

Listing 8.38 Registering Applications to SAP Fiori Launchpad Sandbox

If you run the registered applications now, they should look like Figure 8.21. You can test the application, and if you click on the individual tiles, the applications should open and be displayed as we left them.

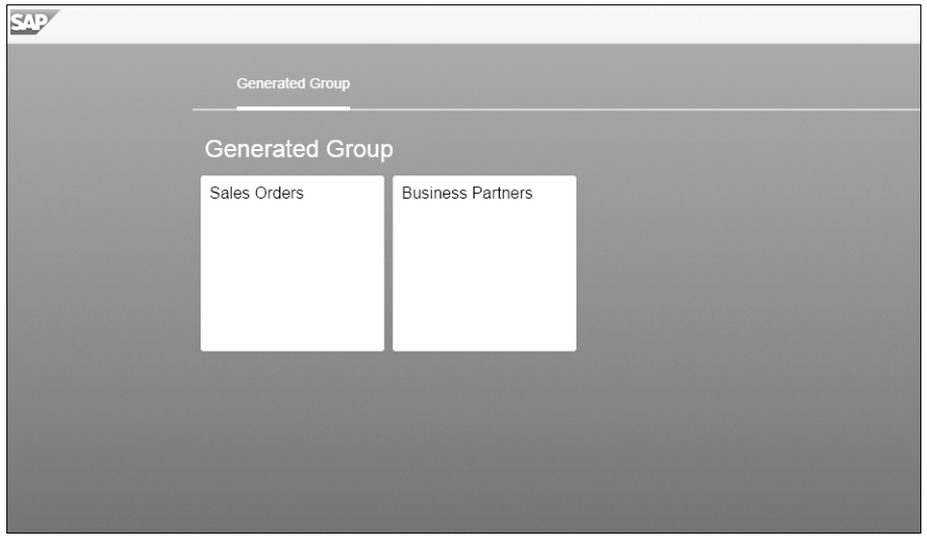


Figure 8.21 Custom SAP Fiori Launchpad Sandbox

8.4.2 Cross-Application Navigation

With the custom-built SAP Fiori Launchpad sandbox, we can now test the cross-application navigation function as a simple feature example for what the SAP Fiori Launchpad generally provides. When someone clicks on the link to one business partner in the Sales Orders application worklist screen, the Business Partner application should open with the chosen business partner selected.

Let's first look at the Sales Order application that triggers the navigation. For this, we'll add a click handler to the link on the worklist table like so:

```
<Link text="{CustomerName}" press="onCustomerPressed"/>
```

The matching event handler (see Listing 8.39) in Worklist.controller.js should then make use of the navigation service provided by SAP Fiori Launchpad and call the `toExternal` function with some parameters. We'll use the settings we just made in Listing 8.38 (`BusinessPartner-display`) to identify the application during navigation. We'll also hand over the ID of the business partner we want to navigate to as an additional parameter.

To retrieve the ID, we have to think outside the box a little. Because we can't use the ID of the `SalesOrder` we're currently using, we have to retrieve the `BusinessPartnerID` to properly handle the navigation on the other app. Therefore, we'll add an `expand` parameter to the binding of the table—`parameters : { expand : 'ToBusinessPartner'}`—and add the ID of the `BusinessPartner` as custom data in the link itself—`data:id="{ToBusinessPartner/BusinessPartnerID}`. Then, we can retrieve the ID in the handler directly from the element itself (see Listing 8.39).

```
onCustomerPressed: function(oEvent) {
  var BusinessParnterId = oEvent.getSource().data().id;
  var oCrossAppNavigator =
 sap.ushell.Container.getService("CrossApplicationNavigation");
  oCrossAppNavigator.toExternal({
    target: {
      semanticObject: "BusinessPartner",
      action: "display",
      params: {
        BusinessPartner : BusinessParnterId
      }
    }
  });
}
```

Listing 8.39 Cross-Application Navigation Handler

If you click on the Business Partner link in the Sales Order application now, navigation to the Business Partners application is triggered and it hits the master route. In the URL, the ID is visible as a parameter. We now only have to handle the selection based on the parameter we handed over on the other side. To do so, we'll retrieve the ID as `startupParamters` on the instance of the application component.

We'll add additional logic to the resolving promise in the handler of the master route that then checks for the existence of startup parameters and navigates to the matching detail if startup parameters are available, as in Listing 8.40.

```
var aBusinessPartner = this.getOwnerComponent().getComponentData().
  startupParameters.BusinessPartner;
var sId;
if (aBusinessPartner) {
    sId = aBusinessPartner[0];
} else {
    sId = this.oList.getItems().getBindingContext().
      getProperty("BusinessPartnerID");
}
this.selectAnItem(sId);
this.oRouter.navTo("detail", {
 BusinessPartnerID: sId
});
```

Listing 8.40 Handling Cross-Application Navigation in Target

8.4.3 Register and Run in Production

With all the pieces in place and working in the sandbox, we can deploy the two applications straight out of the SAP Web IDE into SAP HCP. However, first we have to locate each application in the root of our workspace again. Use the context menu triggered by right-clicking on the application root folder and select Deploy • Deploy to HANA Cloud Platform. First, we'll deploy the Business Partner application. In the popup (see Figure 8.22), you can set some details and also see the application status and whether it's already deployed. We do not need to make any changes here; simply click Deploy. For more details on deploying and managing application versions, see Appendix D.

After a while, you'll see a success notification with a prominent button marked Register to SAP Fiori Launchpad. Click on this button, and a dialog opens in which you can perform all the needed steps to set up your application in SAP Fiori Launchpad.

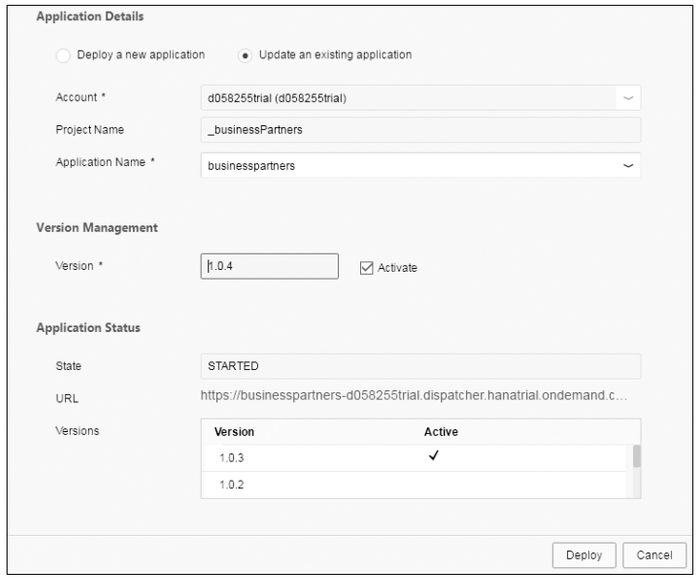


Figure 8.22 Application Deployment on SAP HCP

On the first screen (see Figure 8.23), we will mainly set up the navigation to this application within the SAP Fiori Launchpad in the Intent settings; we defined these settings in Listing 8.39 when we set up the cross application. We have to use the same settings now again for the Semantic Object and the Action. Then, click on Next.

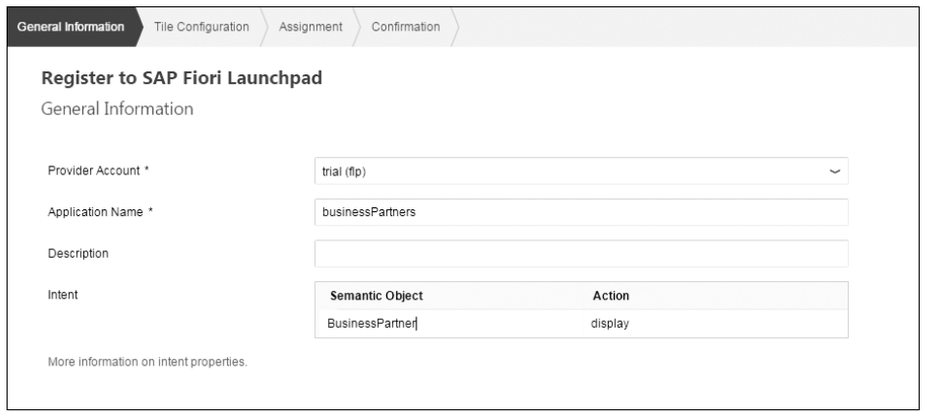


Figure 8.23 Set Up Navigation within SAP Fiori Launchpad

On the next screen (see Figure 8.24), we can set up the appearance of the tile for this application in the SAP Fiori Launchpad. You can choose between two different tile types (static and dynamic), define an icon, and set a title and subtitle to be displayed. We'll just change the title here and delete the placeholder for the subtitle. Again, proceed by clicking on Next.

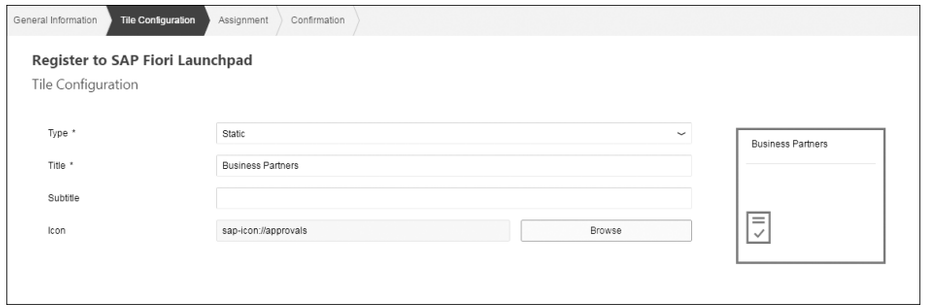


Figure 8.24 Set Up Tile Appearance within SAP Fiori Launchpad

In the last step (see Figure 8.25), we'll finally assign this new tile that represents the application and allows the user to open it to our launchpad. We can only cover some basics here, so we don't want to get into the details of these settings. SAP Fiori Launchpad as an entry point to applications uses roles to provide access, while the allocation of applications is done via catalogs. This means that a business role like—for example, for procurement—has one or many catalogues assigned. Each of these catalogues consists of a set of applications. An administrator can then assign catalogues to a role, and each employee is assigned to a role as well. This then defines what applications are to be part of their SAP Fiori Launchpad. Let's keep the default settings and click on Next.

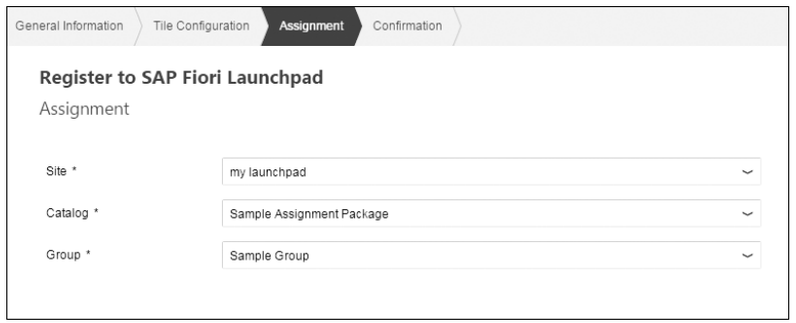


Figure 8.25 Assign Application to the SAP Fiori Launchpad

We'll now perform the same tasks for the Sales Orders application. Based on your settings, the final result should look like Figure 8.26. If all intents are set to match our cross-application navigation settings, it should now be possible to perform the navigation between the two apps as implemented in Section 8.4.2.

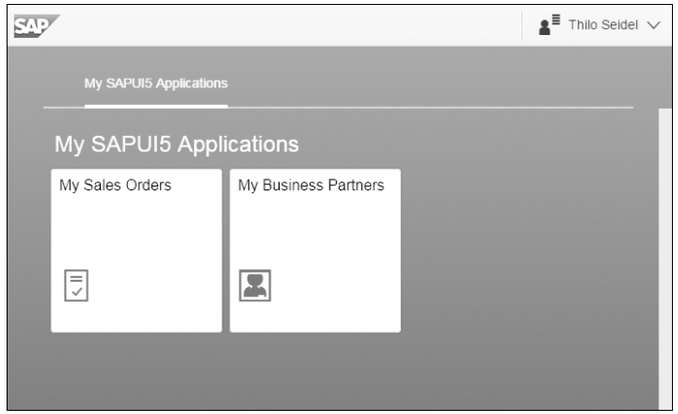


Figure 8.26 Application in SAP Fiori Launchpad at SAP HCP

8.5 SAP Fiori Reference Apps

So far, we've explored application development from different angles: In Section 8.2, we created application skeletons, and we refined them in Section 8.3. However, thus far we have not built full-blown applications but have only gained an understanding of the different building blocks that matter in application development.

In this section, we want to look at the SAP Fiori reference applications that can be evaluated directly in the SAP Web IDE. Specifically, we'll look at the Manage Products and Shop apps.

All applications are built using best practices for SAP Fiori development. This means they are component-based, come with an SAP Fiori sandbox set up, and make use of controls from the `sap.m` library. However, note that these application are built on SAPUI5 version 1.28, and therefore, for example, manifest.json isn't used. For the full list of applications available, see Figure 8.27. You can open this

wizard by clicking File in the menu bar, then New, and then Project from Sample Application.

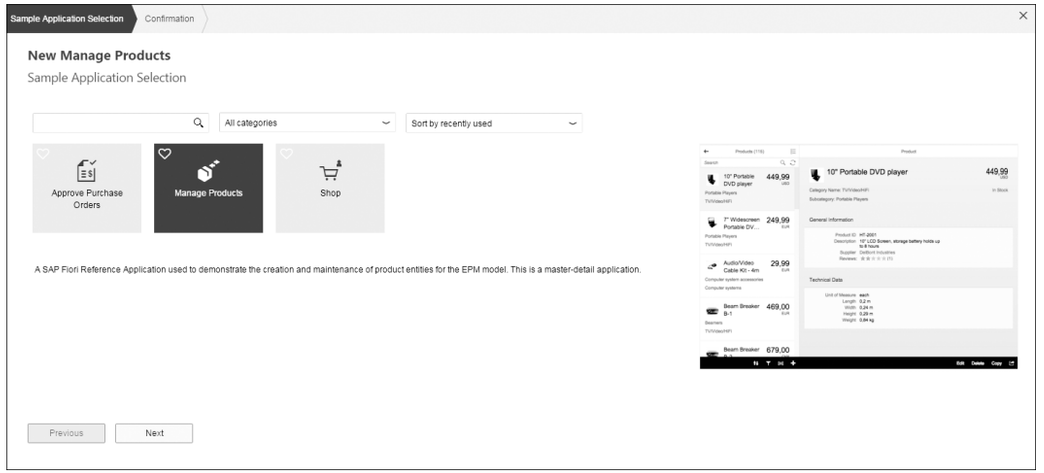


Figure 8.27 SAP Fiori Reference Apps in SAP Web IDE

8.5.1 Manage Products App

The Manage Products application (see Figure 8.28) uses the master-detail floorplan. The master list is implemented as in Section 8.2.2, but comes with additional filtering, sorting, and grouping functionality. In the detail content area are `sap.m.ObjectHeader` and two `sap.m.Panel` controls displaying different types of data within static forms related to the selected item. Footer buttons are added that allow you to switch from detail view to edit mode. You also can delete or copy any selected item.

The Manage Products application is a good example of how to build an application designed to change, add, or delete business objects. It makes good use of the master-detail floorplan; you can quickly navigate between the different products. In addition, the display on the details side is highly sorted and not overloaded.

From a coding perspective, you can see that several helper files are needed, most of them dealing with CRUD operations. Because the coding is extensively documented using inline comments, we won't go into details here.

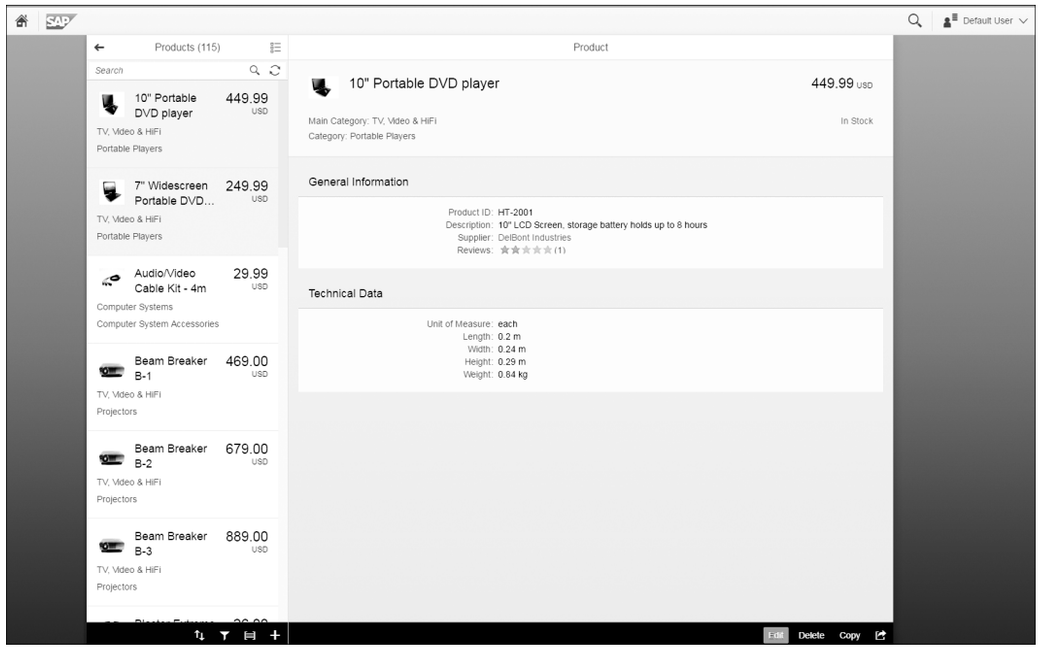


Figure 8.28 Manage Products Reference Application

8.5.2 Shop App

The SAP Fiori reference Shop application is built on top of the worklist floorplan. The scenario that is implemented here is a simple Shop application where a user can browse different items and add them to a shopping cart. In Figure 8.29, you can see that it looks familiar to what we created in Section 8.2.1.

In the worklist, an action is implemented where the user can add items to their shopping cart straight out of the item list without browsing any details. One additional control that is used here to refine the items to be displayed is a `sap.ui.comp.smartfilterbar.SmartFilterBar`. We will provide more details about the capabilities of smart controls in SAPUI5 in Chapter 9.

The navigation in the Shop application has two additional views displaying the items in the shopping cart as well a view that is used to place an order once the user is ready to checkout.

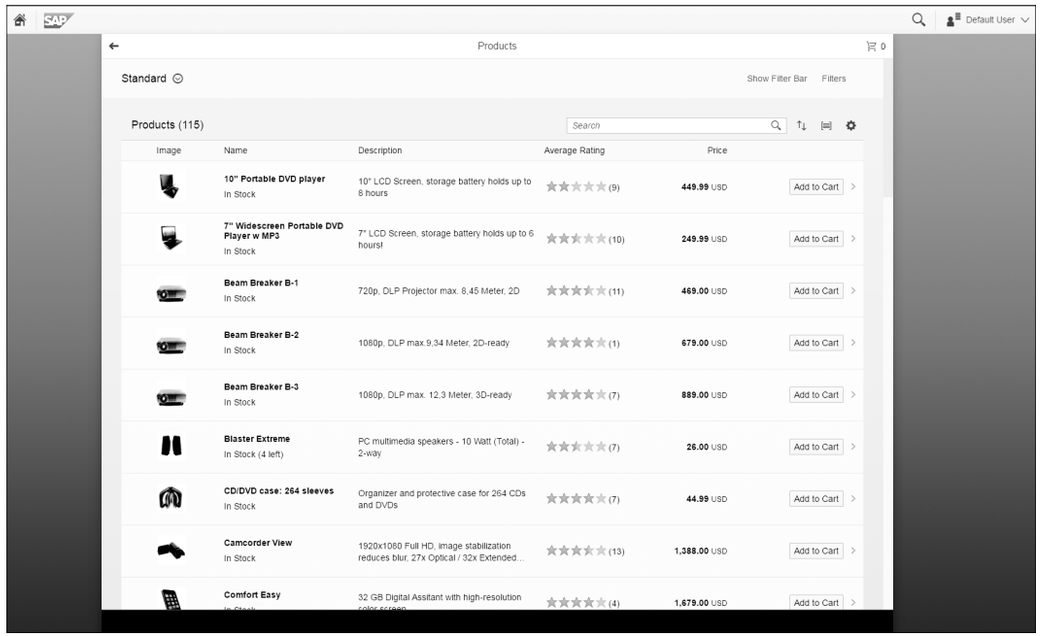


Figure 8.29 Shop Reference Application

It is interesting to see how items are to be added to this cart. One might assume that there is an additional model involved here. However, there is actually a function import on the OData model being used in this scenario. We have already learned about this OData feature in Chapter 7. Here, OData can be used for adding items to a shopping cart as well as for placing an order.

8.6 Summary

In this chapter, you've seen the complexity of application development firsthand. SAPUI5 helps to build applications, and SAP Fiori Launchpad launches applications into an environment that eases their orchestration and provisioning. Beyond just looking at the technical aspects of application patterns, it's important not to forget the needs of your application users. Therefore, within this chapter we spent some time on the general layout of applications, explained how to make use of existing application floorplans, reviewed user experience best practices, and outlined some of the most important nonfunctional application capabilities

every user expects. During this process, you built two application skeletons and learned how all the technical concepts explained in the previous chapters can be used in combination.

With this chapter, we've concluded the chapters on the pure basics in SAPUI5; in the next chapter, we'll look into more advanced concepts that build on top of what you've learned so far.

Contents

PART II SAPUI5 In Action—Building Apps

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

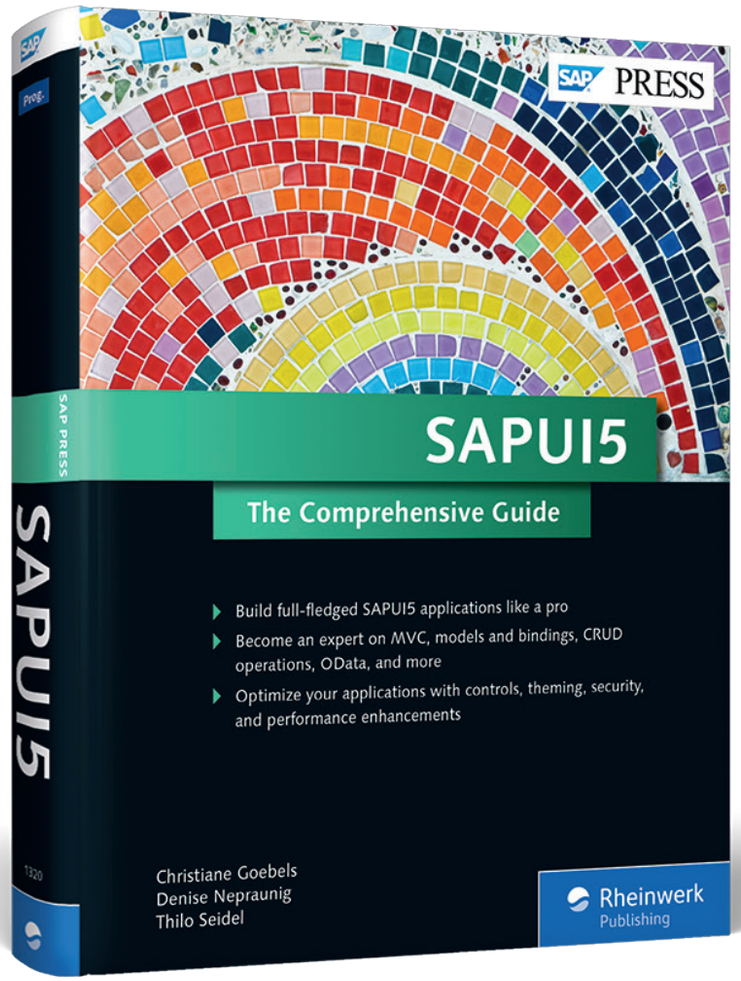
T

U

V

W

X



Christiane Goebels, Denise Nepraunig, Thilo Seidel

SAPUI5: The Comprehensive Guide

672 Pages, 2016, $79.95
ISBN 978-1-4932-1320-7

www.sap-press.com/3980

Christiane Goebels has been in web development ever since starting her career at SAP in 2000. She led her own internet agency from 2005 to 2010, and re-joined SAP in 2012 as part of the central SAPUI5 development team. She is an experienced speaker and has been giving numerous trainings and talks on JavaScript and SAPUI5 at SAP and at international conferences.

Denise Nepraunig is a software developer at SAP in Walldorf, Germany, where she creates SAPUI5 applications and was involved in the development of the SAP Web IDE. She is an experienced speaker, SAPUI5 coach, and SAP Mentor. She loves to explore new technologies, and in her free time tinkers around with SAP HCP and SAP HANA.

Thilo Seidel is the product owner of SAP Fiori Launchpad on the weekdays and an occasional hacker on the weekends. He built his first web page back in 2002 and instantly fell in love with the browser. He has taken on various roles since then, including sales, designer thinker, traveler, student, and project manager.