Chapter 2. Building a basic responsive Web Application

You may expect to find some introductory "Hello World" example here, but there is already one in the UI5 Developer Guide "Get Started" section. The point is that any serious UI5 application needs preliminary work for setup and objects which we don't always want to repeat.

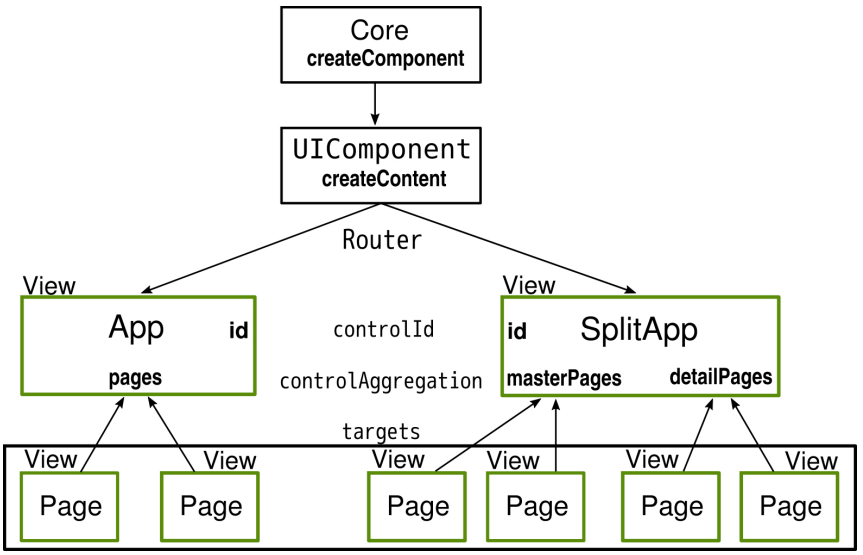
In this chapter, we will therefore build a basic web application without business logic. Step by step, the reader is introduced to concepts and flow of developing with UI5. The resulting application will be used as a template in the following chapters. Don't worry if questions remain open and you don't fully understand what is going on. As far as necessary, we will dive into the details of particular aspects in the following chapters.

The code presented in this chapter is nearly complete. If you are new to UI5, you will get the most out of it by reproducing the steps and using the browser console and UI5 Inspector to constantly inspect the state of the application along the way.

The following figure shows some basic objects composing a UI5 web application. The `Core` creates a `UIComponent`. The UIComponent creates content, which usually is a `View` providing a root element with aggregations for views. There are two such root elements for responsive applications. A full page root element is a `sap.m.App` with an aggregation named `pages`, while a `sap.m.SplitApp` provides two aggregations named `masterPages` and `detailPages`.

We use a `Router` to have `View` objects, mostly providing some kind of `Page`, placed into an aggregation specified by `controlId` and `controlAggregation`.

Figure 2.1. Basic structure of a UI5 Application



2.1. Bootstrap the Core

> Final Class sap.ui.core.Core
>
> Core Class of the SAP UI Library
>
> This class boots the Core framework and makes it available for the application via method `sap.ui.getCore()`.
>
> —openUI5 API

We begin our application by creating a project folder with a document root folder for a web server to serve files from. According to UI5 conventions, the document root folder is named `webapp`. Into that folder we either link or copy the UI5 SDK `resources` folder. All files and folders created in this chapter go into the `webapp` folder.

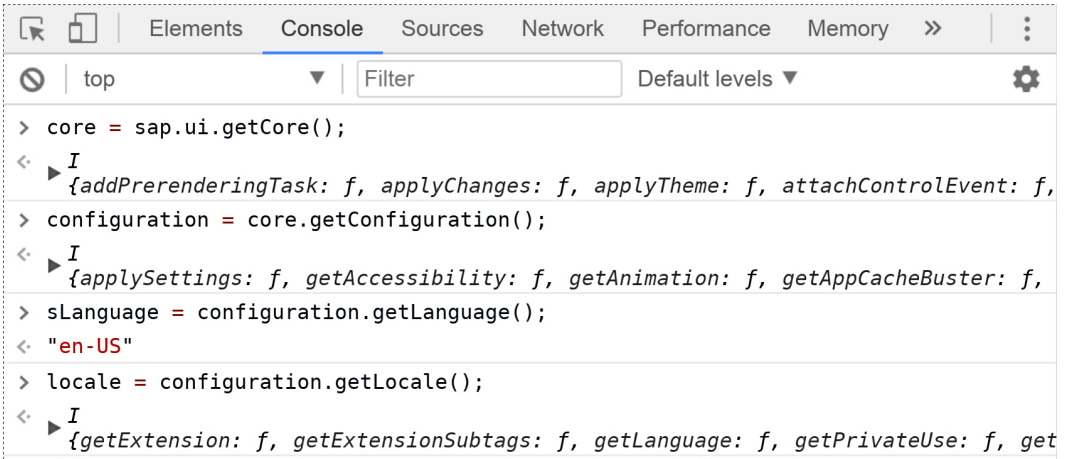
To use UI5, we first have to load the main file of the toolkit, which is called `sap-ui-core.js`. To do that, we create an `index.html` file with the bootstrap `script` element.

The `sap-ui-core.js` file is expected to be located in the `resources` subfolder relative to the `index.html`.

Listing 2.1. sap-ui-bootstrap

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>UI5 User Guide</title>
    <meta http-equiv="content-type" content="text/html;charset=UTF-8"/>
    <script id="sap-ui-bootstrap"
            type="text/javascript"
            src="resources/sap-ui-core.js"
            data-sap-ui-theme="sap_belize"                    Stylesheet theme name
            data-sap-ui-libs="sap.m"
            data-sap-ui-preload="async"
            data-sap-ui-bindingSyntax="complex"></script>
  </head>
  <body>
  </body>
<html>
```

As a first step, we just need to ensure that the UI5 `Core` is loaded and ready. After loading the `index.html` into a browser, we should now be able to get the `sap.ui.core.Core` object using the browser console.



2.2. Application manifest

> This specification defines a JSON-based manifest file that provides developers with a centralized place to put metadata associated with a web application.
>
> —Web App Manifest - W3C Working Draft 22 September 2017

Our application needs a manifest. The general format of a web application manifest is specified as a W3C Working Draft [https://www.w3.org/TR/appmanifest/].

The specifics of the UI5 manifest are listed in the UI5 SDK "Developer Guide" section "Essentials > Structuring: Components and Descriptor > Descriptor for Applications, Components, and Libraries".

Now, let us create such a `manifest.json` file. We keep it minimal.

Listing 2.2. Descriptor Application Manifest

```
{
   "start_url": "index.html",                    The application start URL
   "sap.app": {
      "id": "ui5guide",                          Mandatory attribute
      "type": "application",                     Application type: application, component, library
      "title": "{{ui5guide.title}}", ❶
      "applicationVersion": {                    Mandatory attribute
         "version": "0.1"
      }
   },
   "sap.ui": {
      "deviceTypes": { ❷
         "desktop": true,
         "tablet": true,
         "phone": false
      },
      "supportedThemes": [
         "sap_belize",
         "sap_belize_hcb",                       High Contrast Themes
         "sap_belize_hcw"
      ]
   },
   "sap.ui5": {
      "minUI5Version": "1.38.37",
      "contentDensities": { ❸
         "compact": true,
         "cozy": true
      },
      "dependencies": {
         "libs": {
            "sap.m": {}
         },
         "components": { }
      }
   }
}
```

❶ Mandatory. The `sap.app.title` property, and likewise the `sap.app.subtitle` and `sap.app.description` properties are defined as i18n keys in double curly brackets. The i18n key for the title is

`ui5guide.title`. We will keep that in mind for the time when we want to implement multi-lingual support.

❷ Mandatory. Specify, on which devices the application is expected to run. There are three categories of devices: "desktop", "tablet", and "phone". Our sample application does not claim to be suited for mobile phones, at least in portrait orientation.

❸ Mandatory. Specify, which content density modes are supported. Available options are "compact" and "cozy". Density mode 'compact' features reduced dimensions of controls. Density mode 'cozy' optimizes for touch devices.

2.3. Create a Component

> You can use a ComponentContainer to wrap a UIComponent and reuse it at any place within the openUI5 control tree.
>
> To render UI components, you must wrap them in a ComponentContainer. It is **not** possible to use the placeAt method to place UI components directly in a page.
>
> —openUI5 Developer Guide

A component may be an application, a supporting component or a library. Here, we want to build an application. In UI5, an application is called a component. There are two types of components:

- faceless components extending `sap.ui.core.Component`,
- components with a user interface extending `sap.ui.core.UIComponent`.

A `UIComponent` can be understood as an engine to create visible content for a web page. Just a few more steps and we will have something to show.

With the `Core` loaded and the `manifest.json` in place, we are ready to create a `Component`.

To be able to add content to the HTML page, our `sap.ui.core.UIComponent` needs to be wrapped in a `sap.ui.core.ComponentContainer`. Therefore, in the `head` element of the `index.html`, after the sap-ui-bootstrap `script`, we add the following code:

Listing 2.3. Create a Component

```
<script type="text/javascript">

sap.ui.getCore().attachInit(function () { ❶
   sap.ui.require([
      "sap/ui/core/ComponentContainer"
   ], function (ComponentContainer) {
      const oumComponent = sap.ui.getCore().createComponent({
         id: "oumComponent", ❷
         name: "oum"                                  Namespace of our custom Component
      });
      new ComponentContainer({ ❸
         component: oumComponent                      Associate the component
      }).placeAt("oumContent");                       Insert the ComponentContainer into the HTML
   });
});

</script>
```

❶ With asynchronous loading in mind, we want to make sure that the `Core` is ready, before we let it create a `Component`. For this, we use the `attachInit` function. The OpenUI5 API explains: “The given function will either be called as soon as the framework has been initialized or, if it has been initialized already, it will be called immediately.”

❷ The component id is required to get the `Component` from the `Core` using `sap.ui.getCore().getComponent(componentId);`

❸ The `Shell` control can be used as a wrapper for an App or SplitApp, allowing to set a background style and also providing properties for an application `title` and `logo`, among others. It mostly affects the user interface of wide browser windows.

```
new sap.m.Shell({
   title: "OpenUI5 User Guide",
   app: new ComponentContainer({
           component: oumComponent
        })
}).placeAt("oumContent");
```

In lines 4 to 6, we see the `sap.ui.require` syntax, which is the same for `sap.ui.define` (see Listing 2.4, “Initialize the Component”). This is the UI5 way of handling the loading of dependencies. For now, it is sufficient to note the path syntax used to load the `sap.ui.core.ComponentContainer` (line 5), which we then add as parameter to the function (line 6). This is used to instantiate a new ComponentContainer (line 11 to 13).

In lines 7 to 10, we use the `createComponent` function of the `Core` to import the `Component` with `id` "oumComponent" and `name` "oum". This means, that UI5 will first request `oum/Component-preload.js` and next `oum/Component.js`. Where does UI5 look for these resources?

> By default, all files have to be located in a subfolder of the resources folder of the Web application. If this location is not appropriate, deviations can be configured ...
>
> —openUI5 Developer Guide

The default resource path is `resources`. But we want our `Component.js` file to be in the `webapp` folder along with the `index.html` and the `manifest.json`. For this situation, UI5 has the concept of a resource path mapping. We add such an entry to the Listing 2.1, “sap-ui-bootstrap”:

```
data-sap-ui-resourceroots='{
   "oum": "."
}'
```

With the above resource root entry added, UI5 resolves "oum" to ".", so that `oum/Component.js` is expected to be found at `./Component.js`.

We create such a file `Component.js` and define our `oum.Component`, which extends `sap.ui.core.UIComponent`. Next, we create an empty `Component-preload.js` (we show how to generate the 'real' file in Section 19.4, “Building the distribution package”). This will be requested before the Component is initialized. We get an HTTP 404 error, if it can't be found.

Listing 2.4. Initialize the Component

```
sap.ui.define([
   "sap/ui/core/UIComponent"
], function (UIComponent) {
   const Component = UIComponent.extend("oum.Component", {
      metadata: {
         manifest: "json"                                    Read the metadata from the manifest.json
      }
   });

   oum.Component.prototype.init = function() {               The Component init function
      UIComponent.prototype.init.apply(this, arguments);
   };
   return Component;
});
```

In line 13 of our Listing 2.3, “Create a Component”, we tell the `ComponentContainer` to be placed at "oumContent". We add this 'place' with a few lines of HTML into the `body` container of the `index.html`.

```
<section class="sapUiBody">
   <div id="oumContent"></div>
</section>
```

While UI5 is often used to construct complete pages, we can place content from a UI5 UI-Component anywhere in the HTML DOM. This makes it possible to use UI5 just for some parts of an application or website.

At this stage, excluding the `resources` folder, we have the following files in our `webapp` folder:

```
webapp
├── Component.js
├── Component-preload.js
├── index.html
└── manifest.json
```

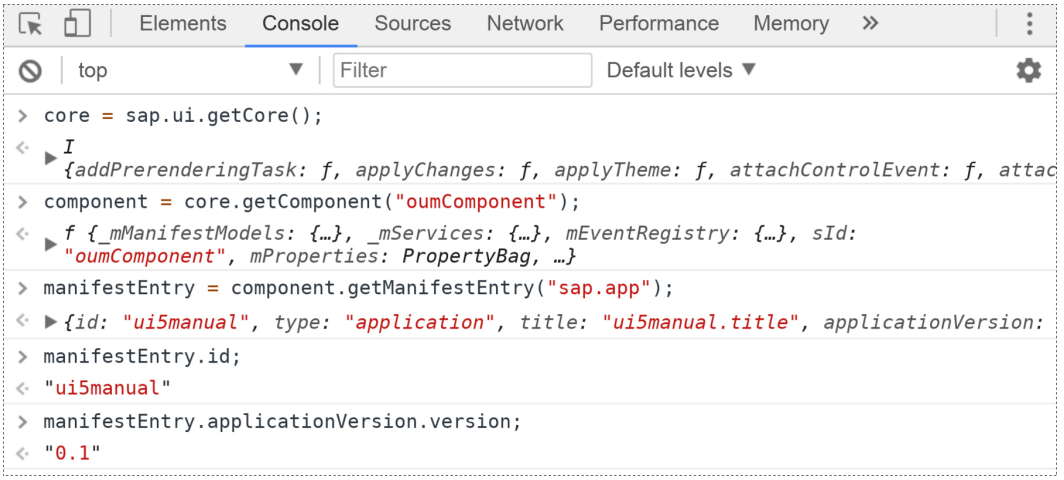
Let's take a break at this point and load the application in a browser. The HTML source looks like this:

```
•••<!DOCTYPE html> == $0
  <html class="sap-desktop  sapUiTheme-sap_belize sapUiMedia-Std-Tablet sapUiMedia-
StdExt-Tablet" dir="ltr" data-sap-ui-browser="cr61" data-sap-ui-os="linux" lang="en-US"
data-sap-ui-animation="on">
  ▶<head>…</head>
  ▼<body role="application">
    ▼<section class="sapUiBody">
      ▼<div id="oumContent" data-sap-ui-area="oumContent">
        ▶<div id="__container0" data-sap-ui="__container0" class=
        "sapUiComponentContainer">…</div>
        </div>
      </section>
      <div aria-hidden="true" id="sap-ui-preserve" class="sapUiHidden
      sapUiForcedHidden" style="width: 0px; height: 0px; overflow: hidden;"></div>
    </body>
  </html>
```

UI5 generates and modifies the HTML. Different versions of UI5 do it differently. It should be clear that we don't want to directly manipulate the generated Document Object Model (DOM). That could only lead to trouble and debugging nightmares. Instead, we are going to use UI5 to do it for us.

Back in the browser console, we see some HTTP 404 errors, because the component tries to load language properties from the default URI `i18n/i18n.properties`. We can ignore them for now. But we should now be able to access the Component:

Figure 2.2. Browser console: Access Component



2.4. Router

So far, our component shows a blank screen. For now, we leave it at that and move on to prepare the routing instead. The routing configuration is part of the application manifest. We begin by adding a `routing` section to the Listing 2.2, “Descriptor Application Manifest”:

Listing 2.5. Routing section of the manifest

```
"sap.ui5": {
  ...
  "routing": {
    "config": { ❶
      "routerClass": "oum.Router",
      "viewType": "JS",
      "viewPath": "oum.view",
      "controlId": "oumApp",                    See Listing 2.7, “XML view with sap.m.App”
      "controlAggregation": "pages"
    },
    "routes": [                                 The routes array
      {
        "pattern": "",
        "name": "home",
        "target": "entry"
      }
    ],
    "targets": {                                The named targets
```

```
19          "app": {
20             "viewName": "app",                ./view/app.view.xml
21             "viewType": "XML"
22          },
23          "entry": {
24             "viewName": "entry",              ./view/entry.view.js
25             "title": "{ui5guide.title}"
26          }
27       }
28    }
29 }
```

❶ The `config` section contains the global router configuration and default values that apply for all routes and targets.

In line 5, we specify a custom router class `oum.Router`. A custom Router is easily set up and gives us more flexibility during the development process. It extends the `sap.m.routing.Router`.

Listing 2.6. Initial custom Router

```
sap.ui.define([
   "sap/m/routing/Router"
], function (mRouter) {
   mRouter.extend("oum.Router", {
      constructor: function() {
         mRouter.apply(this, arguments);
      },
      destroy: function() {
         mRouter.prototype.destroy.apply(this, arguments);
      }
   });
});
```

To make it work, we have to make sure that the `oum.Router` is loaded before the `UIComponent` is constructed, which is why we add it to to the dependencies. Then we can initialize the `oum.Router` in the `Component init` function (see Listing 2.4, “Initialize the Component”):

```
sap.ui.define([
   "sap/ui/core/UIComponent",
   "oum/Router"
], function (UIComponent) {
   ...

   oum.Component.prototype.init = function() {
      UIComponent.prototype.init.apply(this, arguments);

      this.getRouter().initialize();
   };
   return Component;
});
```

Now, let's get back to Listing 2.5, “Routing section of the manifest”. In the `routes` section (lines 11 to 17), we add a route with a `pattern` "" and a `name` "home", which will load the `target` "entry" (lines 23 to 26) into the `controlAggregation` "pages" (line 9) of the root control with `controlId` "oumApp" (line 8).

When we load the `index.html` without any pattern, our `oum.Router` is configured to navigate to the route named "home" with a `target` "entry".

In the `targets` section, we specify two targets, which we have to provide. We defined our `viewPath` as "oum.view" (line 7). Because we defined a resource root "oum" as ".", our views go into the folder `./view`.

For the `App` view, we use XML, which is not the default `viewType` (line 6). The default values set in the `routing/config` section may be overwritten by each target entry. Therefore, we have to specify it explicitly in our target configuration (line 21).

2.5. First views

A view is a file in the `view` folder. The name is composed of the `viewName` and the `viewType`. According to the UI5 naming conventions the name of our "app" view of type XML is `app.view.xml`.

Listing 2.7. XML view with sap.m.App

```
<mvc:View xmlns="sap.m"
          xmlns:mvc="sap.ui.core.mvc"
          displayBlock="true">
  <App id="oumApp"
       defaultTransitionName="slide"
       />
</mvc:View>
```

The `<App>` `id` attribute value "oumApp" is the `controlId` default value (Listing 2.5, “Routing section of the manifest” line 8).

To load the app view, we use the `createContent` function of the `UIComponent` by adding some lines to our `Component.js`:

```
oum.Component.prototype.createContent = function() {
   return sap.ui.view({
      viewName: "oum.view.app",
      type: "XML"
   });
};
```

So far, we still haven't created any visible content. It is about time for our first view to show at least something.

With the default `routing.config.viewPath` specified as "oum.view", "entry" is expected to resolve as "oum.view.entry" and requested as `./view/entry.view.js`. Therefore, we create a file `entry.view.js` in the `view` folder.

Listing 2.8. Basic JS View returning Page

```
sap.ui.jsview("oum.view.entry", {
   createContent: function(oController) {
      const entryPage = new sap.m.Page();
      return entryPage;
   }
});
```

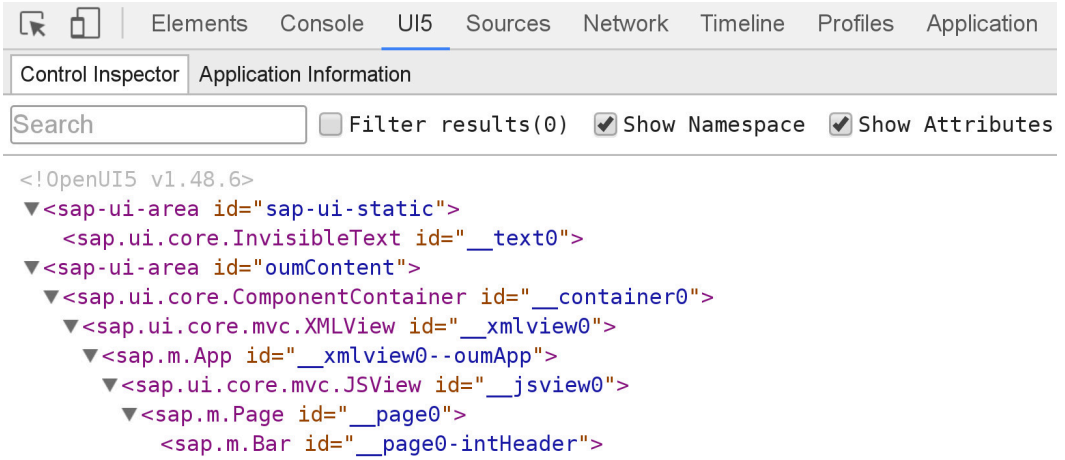
Since version 1.56, the `sap.ui.jsview` function has been marked as deprecated. Yet the API Reference for the OpenUI5 (v1.56) states, that there is no replacement for JavaScript views

and the function still has to be used. This is not clever, but don't worry, because UI5 doesn't carelessly remove deprecated functions. Therefore, until things clear up, we leave this issue aside.

With our first two views in place, we have the following tree in our `webapp` folder:

```
.
├── view
│   ├── app.view.xml
│   └── entry.view.js
├── Component.js
├── Component-preload.js
├── index.html
├── manifest.json
└── Router.js
```

Let us load the `index.html` into the browser. We still see an empty page, but the Page header now has a background color. The Chrome web browser extension UI5 Inspector shows us more:



The `id` of our `<App>` is not "oumApp", like we specified in Listing 2.7, “XML view with sap.m.App”. We don't get it from the `Core` either: `sap.ui.getCore().byId("oumApp")` returns `undefined`. Nevertheless, when we remove the id we get an error, because it is the default `controlId` configured in our manifest (see Listing 2.5, “Routing section of the manifest”).

The generated HTML should not concern us too much, because we never want to directly manipulate the UI5-generated DOM. We are writing JavaScript and XML to build the application, not HTML.

2.6. Internationalization

In Listing 2.2, “Descriptor Application Manifest” we specified the `sap.app.title` of our application as `ui5guide.title`, but we haven't provided the related resources yet. UI5 uses property files as resource for the language specific texts, one for each language and an additional default language file.

By default, these property files go into the `i18n` folder. We begin with two languages, English and German ("en" and "de"), and need to create an `i18n_en.properties` and `i18n_de.properties` file. The default language properties file should further be copied to `i18n.properties`.

Initially, the language properties contain only the language specific string for key `ui5guide.title`.

```
ui5guide.title=UI5 User Guide
```

```
ui5guide.title=UI5 Benutzerhandbuch
```

To start with, UI5 determines the current language from the browser `window.navigator` object. Various language code formats are accepted. We don't want to deal with the complexities of language codes, so we can simplify the issue by just looking at the first two characters of the language code. In order to achieve this, we can add some code to our `Component` `init` function (see Listing 2.4, “Initialize the Component”).

Listing 2.9. Loading the i18n Resource

```
 1 const configuration = sap.ui.getCore().getConfiguration();
 2 let languageCode = configuration.getLanguage();
 3 if (typeof languageCode === "string" && languageCode.length > 2) {
 4    languageCode = languageCode.substring(0, 2).toLowerCase();
 5 }
 6 const availableLanguages = ["en", "de"];                    Array of available languages
 7 if (availableLanguages.indexOf(languageCode) === -1) {
 8    languageCode = "en";                                     Default language
 9 }
10 configuration.setLanguage(languageCode);
11
12 const i18nModel = new sap.ui.model.resource.ResourceModel({
13    bundleUrl: "i18n/i18n.properties",
14    bundleLocale: languageCode
15 });
16 this.setModel(i18nModel, "i18n");                           Set the i18n model for the Component
```

First, we extract the current language string from the UI5 configuration (line 2). If it is longer than 2 characters, we cut off all characters after the second and change the remainder of the string to lower case (lines 3 to 5). If the language string is not one of the available languages, the default language is used (lines 6 to 9). To settle the issue, one of the available language strings is set as UI5 current language (line 10).

The resources are handled by a `ResourceModel`. To load the proper language file, the two parameters `bundleUrl` and `bundelLocale` have to be set (lines 12 to 15). The model is then set to the `Component` with the model name `i18n` (line 16).

After providing language properties and making them available for our component, we are now ready to add a title to our entry page (Listing 2.8, “Basic JS View returning Page”). We do that by setting the value for the `title` property of the `Page`.

Listing 2.10. Page with title

```
const entryPage = new sap.m.Page({
   title: "{i18n>ui5guide.title}"                    Model property notation: modelName>propertyPath
});
```

2.7. Adding a LanguageSwitcher

The language setting can be overwritten with a URL parameter `sap-language` (`index.html?sap-language=de`). That's good, but not enough. To allow the user to change the language derived from the `Navigator` object, the user interface needs a language switcher.

A language switcher requires an array of available languages, as well as a property to store the currently selected language. In Listing 2.9, “Loading the i18n Resource” we already specified available languages (line 6) and the default language (line 8).

At this point, the application will benefit from custom configuration data and custom library functions. Let us begin by creating a `config.json` in the `webapp` folder.

Listing 2.11. Custom configuration

```
{
   "componentId": "oumComponent",
   "availableLanguages": [ "en", "de" ],
   "defaultLanguage": "en"
}
```

As custom library namespace we use `oui5lib`. To prepare the namespace, create a folder `oui5lib` and in that folder a file `init.js` with the following code:

Listing 2.12. Initialize custom namespace

```
if (typeof oui5lib === "undefined") {
   var oui5lib = {};
}
oui5lib.namespace = function(string) {                    Function to get or create namespace
   let object = this;
   const levels = string.split(".");
   for (let i = 0, l = levels.length; i < l; i++) {
      if (typeof object[levels[i]] === "undefined") {
         object[levels[i]] = {};
      }
      object = object[levels[i]];
   }
   return object;
};

const xhr = new XMLHttpRequest();
xhr.open("GET", "config.json", false);                    Request config.json synchronously
xhr.onload = function() {
   if (xhr.status === 200 || xhr.status === 0) { ❶
      try {
         const configData = JSON.parse(xhr.responseText);    Parse incoming JSON
         oui5lib.config = configData;                        Store config data
      } catch (e) {
         throw new Error("Not valid JSON");
      }
   }
};
xhr.send();
```

❶ The HTTP response status 200 means 'OK'. A status 0 is not a valid HTTP status code and therefore can not come from a web-server. Instead, it is set by the XMLHttpRequest object and applies when the request URL uses the 'file' protocol. We add it here to be able to run the application from the file system, without a web-server.

We need to ensure that the `oui5lib/init.js` is loaded before the `Component init` function is executed. It could be done through a `script` tag in the `index.html`. But because the custom library belongs to the component, it is better to require it in the `Component.js`.

```
sap.ui.define([
   "sap/ui/core/UIComponent",
   "oum/Router",
   "oui5lib/init"
], function (UIComponent) {
   ...
});
```

Finally, in order for UI5 to find the resource, we also have to add another resource root to Listing 2.1, “sap-ui-bootstrap”:

```
data-sap-ui-resourceroots='{
   "oum": ".",
   "oui5lib": "./oui5lib"
}'
```

On top of the configuration data, the language switcher will also need a function to set the current language and update the Component i18n `ResourceModel`. Let us use the namespace `oui5lib.configura-`

tion and therefore create a file `configuration.js` in the `oui5lib` folder for this. The following code listing is quite long, but it merely consolidates the language-related code snippets we already looked at.

Listing 2.13. Custom namespace for configuration issues

```
(function () {
   const configuration = oui5lib.namespace("configuration");          Namespace oui5lib.configuration

   function getConfigData(key) {
      if (typeof oui5lib.config === "undefined" ||
          typeof oui5lib.config[key] === "undefined") {
         return undefined;
      }
      return oui5lib.config[key];
   }

   function getComponent() {                                          Function to get the Component
      const componentId = getConfigData("componentId");
      if (typeof componentId === "string") {
         return sap.ui.getCore().getComponent(componentId);
      }
      return null;
   }

   function setLanguageModel(languageCode) {
      const i18nModel = new sap.ui.model.resource.ResourceModel({
         bundleUrl: "i18n/i18n.properties",
         bundleLocale: languageCode
      });
      const component = getComponent();
      component.setModel(i18nModel, "i18n");                          Set i18n model for the Component
   }

   function getAvailableLanguages() {
      return getConfigData("availableLanguages");
   }

   function setCurrentLanguage(languageCode) {
      const availableLanguages = getAvailableLanguages();
      if (!(availableLanguages instanceof Array)) {                   No available languages configured?
         return;
      }

      if (availableLanguages.indexOf(languageCode) === -1) {          Not one of the available languages?
         languageCode = getConfigData("defaultLanguage");             Use default language instead
      }
      const ui5configuration = sap.ui.getCore().getConfiguration();
      ui5configuration.setLanguage(languageCode);                     Set language of Core configuration
      setLanguageModel(languageCode);
      oui5lib.config.currentLanguage = languageCode;                  Used by the following function
   }

   function getCurrentLanguage() {
      return getConfigData("currentLanguage");
   }


   configuration.getComponent = getComponent;                         Add functions to the namespace
```

```
   configuration.getAvailableLanguages = getAvailableLanguages;      Used by the LanguageSwitcher
   configuration.getCurrentLanguage = getCurrentLanguage;
   configuration.setCurrentLanguage = setCurrentLanguage;
}());
```

Only the `getComponent`, `setCurrentLanguage`, `getCurrentLanguage` and `getAvailableLanguages` functions are added to the namespace. The `getConfigData` and `setLanguageModel` functions will be "private".

We use similar code in the `Component.js`. We should refactor it to avoid duplicating code for the same task. To be sure that the `setCurrentLanguage` function is loaded, we add it as a requirement to the `oui5lib/init.js` with

```
sap.ui.require([
   "oui5lib/configuration"                                  Loading happens synchronously
]);
```

Next, we replace the code from Listing 2.9, “Loading the i18n Resource” (lines 6 to 16) with

```
oui5lib.configuration.setCurrentLanguage(languageCode);
```

Everything needed for a language switcher is now ready and loaded. We will implement it as a fragment to be able to reuse it for different views. The name is `oum.fragment.LanguageSwitcher`, which resolves to path `./fragment/LanguageSwitcher.fragment.js`.

To define a fragment, the `sap.ui.jsfragment` function is used. The first parameter is the fragment name and the second an object with a `createContent` function returning a content control. Here, it is a `Select` control.

Listing 2.14. LanguageSwitcher fragment

```
sap.ui.jsfragment("oum.fragment.LanguageSwitcher", {
   createContent: function () {
      const languageSelect = new sap.m.Select({                 Construct the Select control
         tooltip: "{i18n>language.select.tooltip}"              i18n
      });

      const availableLanguages = oui5lib.configuration.getAvailableLanguages();
      if (availableLanguages !== undefined) {
         let item;
         availableLanguages.forEach(function(languageKey) {
            item = new sap.ui.core.Item({                       Construct Item for each language
               text: "{i18n>language." + languageKey + "}",     i18n keys: language.de and language.en
               key: languageKey
            });
            languageSelect.addItem(item);                       Add the Item to the Select
         });
      }
      return languageSelect;                                    Return the Select control
   }
});
```

We want to add the LanguageSwitcher to the header of the entry page. In the UI5 documentation we find the Bar control, which can be used in this case.

> The Bar control can be used as a header, sub-header and a footer in a page. It has the capability to center a content like a title, while having other controls on the left and right side.

—openUI5 API

Let's change our entry view to use a `sap.m.Bar` control with the title in the middle and the LanguageSwitcher fragment to the right. To achieve this, the `Page` is modified to use the `customHeader` aggregation. We learn more about using this aggregation from the UI5 documentation.

> If this aggregation is set, the simple properties "title", "showNavButton", "NavButtonText" and "icon" are not used.
>
> —openUI5 API

By using a `Bar` for the page header the `title` property we use in Listing 2.10, “Page with title” is ignored. Instead, we now use the `sap.m.Title` control for the title. Its `text` comes from our i18n properties. The `level` is set for semantic purposes and the `titleStyle` to style the title text.

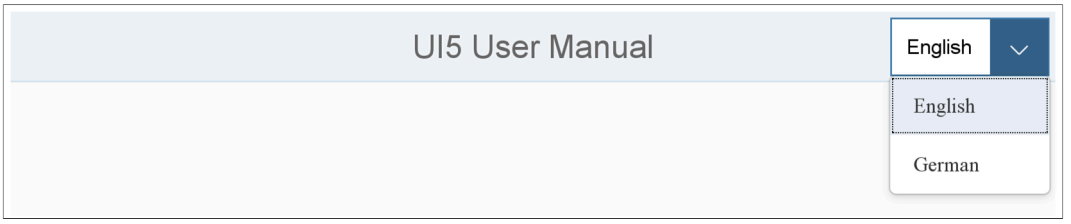
Listing 2.15. Add LanguageSwitcher fragment to Page

```
const headerTitle = new sap.m.Title({
   text: "{i18n>ui5guide.title}",                          i18n property
   level: "H2", titleStyle: "H2"                           Enumeration sap.ui.core.TitleLevel
});

const headerBar = new sap.m.Bar({
   design: "Header",                                       Enumeration sap.m.BarDesign
   contentMiddle: [ headerTitle ],
   contentRight: [
      sap.ui.jsfragment("oum.fragment.LanguageSwitcher")   Add fragment to the page header
   ]
});

const entryPage = new sap.m.Page({
   customHeader: headerBar                                 Set customHeader aggregation
});
```

We introduced a few new i18n keys in Listing 2.14, “LanguageSwitcher fragment” ("language.select.tooltip", "language.en" and "language.de"), which we have to add to the i18n properties. After reloading the application, we should see a title and a drop-down list in the header. Please note: You may have to clean the browser cache to see the i18n model refreshed.



The language select control looks good, but unfortunately, it doesn't do anything yet. If the user selects a different language, we want our code to get busy. To achieve this we attach a function to the LanguageSwitcher `Select change` event. The UI5 API Reference explains when the event occurs:

> This event is fired when the value in the selection field is changed in combination with one of the following actions:
>
> - The focus leaves the selection field
> - The Enter key is pressed

- The item is pressed

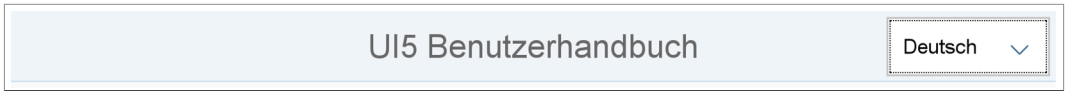
—openUI5 API

On top of handling a change of the selection, we would like to display the currently selected language. For this, we need to set our current language as the `selectedKey` of the control.

Listing 2.16. Attach handler to the change event of the language Select control

```
const languageSelect = new sap.m.Select({
   tooltip: "{i18n>language.select.tooltip}",
   selectedKey: oui5lib.configuration.getCurrentLanguage(),
   change: function (oEvent) {
      const selectedLanguage = oEvent.getParameter("selectedItem").getKey();
      if (oui5lib.configuration.getCurrentLanguage() !== selectedLanguage) {
         oui5lib.configuration.setCurrentLanguage(selectedLanguage);
      }
   }
});
```

Switching back to the browser, we clear the cache as a routine precaution, then reload the application and should now see the LanguageSwitcher switching the language.



Let us look again at the filesystem tree at this stage:

```
.
├── fragment
│   └── LanguageSwitcher.fragment.js
├── i18n
│   ├── i18n_de.properties
│   └── i18n_en.properties
├── oui5lib
│   ├── configuration.js
│   └── init.js
├── view
│   ├── app.view.xml
│   └── entry.view.js
├── Component.js
├── Component-preload.js
├── config.json
├── index.html
├── manifest.json
└── Router.js
```

2.8. Logging

UI5 can be set into debugging mode by setting the URL parameter `sap-ui-debug` to `true` (`index.html?sap-ui-debug=true`). The generated console messages provide valuable insights about how UI5 works, but they are not really tailored to our needs. Let's change that.

Any serious logging implementation needs to output messages depending upon a logging level setting. It should be enough to use four logging levels: DEBUG, INFO, WARN, ERROR. Error mes-

sages will always be logged. To set an initial log level, we add a property to the `config.json`. During development we set `"logLevel": "DEBUG"`.

To get the log level, we add the related function to the `oui5lib.configuration` namespace:

```
function getLogLevel() {
   const logLevel = getConfigData("logLevel");
   if (logLevel === undefined) {
      return "WARN";                                    Defaults to "WARN", if not configured
   }
   return logLevel;
}
configuration.getLogLevel = getLogLevel;                Add function to namespace
```

UI5 includes a Logging API under namespace `jQuery.sap.log`. It seems unnecessarily complex for our custom logger, which should simply log messages to the browser console. Therefore, let us next create a `logger.js` in our custom library folder (`webapp/oui5lib`).

Listing 2.17. Namespace oui5lib.logger

```
/* eslint no-console: "off" */                          Overwrite ESLint default configuration
/* eslint no-fallthrough: "off" */
(function () {
   const logger = oui5lib.namespace("logger");

   if (!window.console) {                               Without console object
      logger.debug = function(){};                      no logging
      logger.info = function(){};
      logger.warn = function(){};
      logger.error = function(){};
      return;
   }

   logger.debug = function(msg) {
      console.log("oui5lib - DEBUG " + msg);
   };
   logger.info = function(msg) {
      console.info("oui5lib - INFO " + msg);
   };
   logger.warn = function(msg) {
      console.warn("oui5lib - WARN " +  msg);
   };
   logger.error = function(msg) {
      console.error("oui5lib - ERROR " + msg);
   };

   const logLevel =  oui5lib.configuration.getLogLevel();

   switch (logLevel) {                                  Disable logging depending upon the log level
   case "ERROR":
      logger.warn = function(){};
   case "WARN":
      logger.info = function(){};
   case "INFO":
      logger.debug = function(){};
   }
}());
```

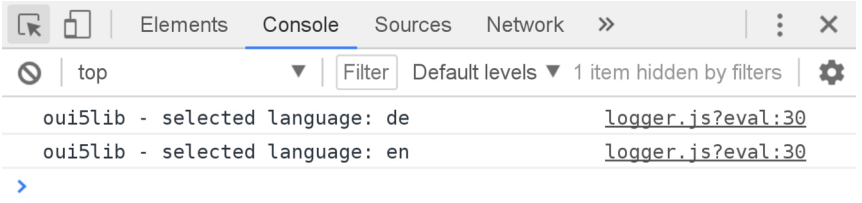
We make loading mandatory by adding the logger as a dependency in the `oui5lib/init.js`:

```
sap.ui.require([
   "oui5lib/configuration",
   "oui5lib/logger"
], function() {
    oui5lib.logger.info("oui5lib successfully loaded");
});
```

To use the logger, let us add a logging entry in the LanguageSwitcher `change` event handler function of Listing 2.16, “Attach handler to the change event of the language Select control ”.

```
const selectedLanguage = oEvent.getParameter("selectedItem").getKey();
oui5lib.logger.debug("selected language: " + selectedLanguage);
```

After clearing the browser cache and reloading the application, we should now see a log message in the console when we change the language selection.



2.9. More Routes and Pages

So far, we only have the entry view and don't fail gracefully, if a requested route doesn't exist. To change that, we add a `bypassed` property to the `routing.config` properties of Listing 2.5, “Routing section of the manifest”.

Listing 2.18. Configuring target for non-existing routes

```
"bypassed": {
   "target": "noRoute"
}
```

The related target is added to the `routing.targets` section:

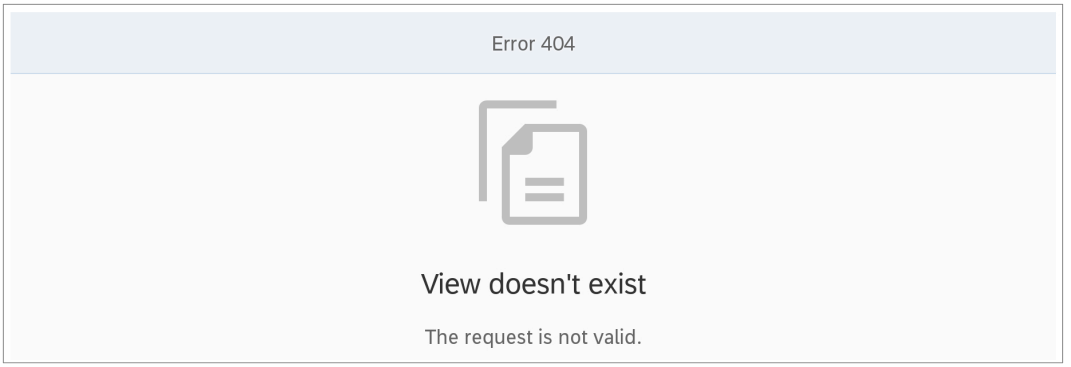
```
"noRoute": {
   "viewName": "noRoute",
   "viewType": "XML"
}
```

The view is of type XML.

Listing 2.19. No such route Page

```
<mvc:View xmlns="sap.m"
          xmlns:mvc="sap.ui.core.mvc">
  <MessagePage
      title="{i18n>http.404error}"
      text="{i18n>view.noRoute.text}"
      description="{i18n>view.noRoute.description}"/>
</mvc:View>
```

After adding the new i18n properties, we can see the page by requesting a non-existing route: `index.html#/help`.



To assist the user with questions, our application should have a help section with a table of contents and the help content in two separate containers. This is the typical `SplitApp` scenario. The SplitApp has two aggregations named `masterPages` and `detailPages` to which we can add views.

Listing 2.20. XML view with SplitApp

```
<mvc:View xmlns="sap.m"
          xmlns:mvc="sap.ui.core.mvc"
          displayBlock="true">
  <SplitApp id="oumSplitApp"                                    ID of the SplitApp
            defaultTransitionNameDetail="slide"
            defaultTransitionNameMaster="slide"
            masterButtonTooltip="{i18n>help.master.button.tooltip}"
            />
</mvc:View>
```

We add a route to the `routes` section of the application manifest.

```
{
   "pattern": "help",
   "name": "help",
   "target": ["helpIndex", "helpIntro"]                         Two targets
}
```

And the related targets to the `targets`.

Listing 2.21. SplitApp target entries

```
"splitApp": {
   "viewName": "splitApp",
   "viewType": "XML"
},
"helpIndex": {
   "parent": "splitApp",                                        Parent target name
   "viewName": "help.index",
   "title": "{i18n>view.help.index.title}",
   "controlId": "oumSplitApp",                                  SplitApp ID
   "controlAggregation": "masterPages"                          SplitApp aggregation
},
"helpIntro": {
```

```
   "parent": "splitApp",
   "viewName": "help.intro",
   "title": "{i18n>view.help.intro.title}",
   "controlId": "oumSplitApp",
   "controlAggregation": "detailPages"
}
```

The target configuration tells the router to load the `View` "help.index" into the "splitApp" `parent` with `controlId` "oumSplitApp" into the `controlAggregation` "masterPages". Meanwhile, the "help.intro" is being loaded into the `controlAggregation` "detailPages".

If you have followed the excercises up to this point, you should now be able to create the help index and help intro views (see Listing 2.8, “Basic JS View returning Page”). If it should be an XML view (see Listing 2.7, “XML view with sap.m.App”), the `viewType` of the manifest targets would have to be specified, as the default is "JS". After that is done, the above request `index.html#/help` will now be serviceable.

To get to the help section, we want to add a button to the entry page right next to the language select. So far, our `oum.Router` (Listing 2.6, “Initial custom Router”) hasn't been called directly to navigate. It extends the `navTo` function from the `sap.m.routing.Router`, which has a default behavior we want to change. We read about that in the documentation:

> IF the given route name can't be found, an error message is logged to the console and the hash will be changed to empty string.
>
> —openUI5 API

Instead of navigating to the home page, we want to navigate to the target named "noRoute". This is the same behavior we configured through the manifest `bypassed` property (see Listing 2.18, “Configuring target for non-existing routes”).

Listing 2.22. Router function to navigate to a named target

```
vNavTo: function(routeName, routeParameters, replace) {
   const route = this.getRoute(routeName);
   if (typeof route === "undefined") {                          There is no route with the given name
      this.navTo("noRoute");
   } else {
      if (routeParameters === undefined || routeParameters === null) {
         routeParameters = {};
      }
      if (typeof replace !== "boolean") { ❶
         replace = false;
      }
      this.navTo(routeName, routeParameters, replace);          Navigate to the route
   }
}
```

❶ Parameter `replace` concerns the browser history. Here, the default is set to `false`, which means that browser history entries are set.

Now, we are ready to implement our help button. Because we want to be able to use the help button for other views as well, we implement it as a fragment with the path `fragment/HelpButton.fragment.js`. It will return a `sap.m.Button` with the `icon` and `tooltip` properties set and a handler function attached to the `press` event.

Fragments are not meant to do anything, but, in this case, it is clear what the help button is supposed to do, isn't it? Why put the function somewhere else?

Listing 2.23. HelpButton fragment

```
 1 sap.ui.jsfragment("oum.fragment.HelpButton", {
 2    createContent: function () {
 3       const btn = new sap.m.Button({
 4          icon: "sap-icon://sys-help",
 5          tooltip: "{i18n>button.help.tooltip}",
 6          press: function() {
 7             const component = sap.ui.getCore().getComponent("oumComponent");
 8             const router = component.getRouter();
 9             router.vNavTo("help");
10          }
11       });
12       return btn;
13    }
14 });
```

The icon URI (line 4) could also be specified with `sap.ui.core.IconPool.getIconURI("sys-help")`. The default `IconPool` of UI5 contains hundreds of names. Be aware, that these icons are no images but embedded font, which can be easily styled. The OpenUI5 SDK includes the Icon Explorer application to explore the icons. Start it by loading `test-resources/sap/m/demokit/iconExplorer/webapp/index.html` into your browser. We can add or overwrite icons using the `IconPool.addIcon` function.

To add the button to the user interface, we add it as a fragment to the `contentRight` aggregation of the `Bar` control used for the `Page` `customHeader` aggregation (see Listing 2.15, “Add LanguageSwitcher fragment to Page”).

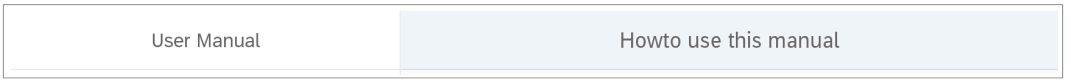
```
const headerBar = new sap.m.Bar({
   design: "Header",
   contentMiddle: [ headerTitle ],
   contentRight: [
      sap.ui.jsfragment("oum.fragment.LanguageSwitcher"),
      sap.ui.jsfragment("oum.fragment.HelpButton")
   ]
});
```

Figure 2.3. Entry page with custom header

2.10. Adding Home and Back Buttons

When we navigate to the SplitApp with the two help views, there is no way back or home.

Figure 2.4. SplitApp Header



Nearly every application with a user interface needs 'back' and 'home' buttons. And because they are needed for most of the views, we need `BackButton` and `HomeButton` fragments.

But before we get to that, our router needs to be able to navigate back. In contrast to standard browser behavior, we don't ever want to navigate "away" from our application. If there is no prior application history, the back button will just navigate to "home".

Listing 2.24. Router function to navigate back

```
navBack: function() {
   const history = sap.ui.core.routing.History.getInstance();          Routing history singleton
   const previousHash = history.getPreviousHash();

   if (typeof previousHash === "undefined") {
      this.navTo("home", {}, true);
   } else {
      window.history.go(-1);
   }
}
```

Both the `BackButton` and `HomeButton` fragments need to get the `Router`, like the `HelpButton` (see Listing 2.23, “HelpButton fragment” lines 7 and 8). Instead of repeating the code, we move it into a utility namespace `oui5lib.util` and add it to the requirements in the `oui5lib/init.js`.

Listing 2.25. Setup custom utility namespace

```
(function (configuration) {
   const util = oui5lib.namespace("util");

   function getComponentRouter() {
      const component = configuration.getComponent()
      return component.getRouter();
   }

   util.getRouter = getComponentRouter;                          Add function getRouter to namespace
}(oui5lib.configuration));
```

After refactoring the HelpButton fragment, we get to the BackButton and HomeButton fragments. They are nearly identical, except for the `icon`, `tooltip` and routing target.

Listing 2.26. BackButton fragment

```
sap.ui.jsfragment("oum.fragment.BackButton", {
   createContent: function () {
      const btn = new sap.m.Button({                             Construct the button control
         icon: "sap-icon://nav-back",
         tooltip: "{i18n>button.back.tooltip}",
         press: function () {
            const router = oui5lib.util.getRouter();             Get the Router
            router.navBack();                                    Navigate back in recorded history
         }
      });
      return btn;                                                Return the button control
   }
});
```

The only tasks left are to add new i18n entries and the new buttons to the help content page. We place the back button on the left side of the header and the home button on the right.

Listing 2.27. JavaScript view creating Page with custom header

```
sap.ui.jsview("oum.view.help.intro", {
   createContent: function(oController) {
      const pageTitle = new sap.m.Title({                          Construct the page title
         text: "{i18n>view.help.intro.title}",
         level: "H2",                                              Define the semantic level
         titleStyle: "H4"                                          Define the style
      });

      const page = new sap.m.Page({                                Construct the page layout
         customHeader: new sap.m.Bar({                             Use customHeader aggregation
            contentLeft: [
               sap.ui.jsfragment("oum.fragment.BackButton")        Add the 'back' button
            ],
            contentMiddle:[ pageTitle ],                           Add the page title
            contentRight: [
               sap.ui.jsfragment("oum.fragment.HomeButton")        Add the 'home' button
            ]
         }),
         content: [ ]
      });
      return page;
   }
});
```

After reloading the application and navigating to the 'help' route, we will see the SplitApp detail view header with the 'back' and 'home' buttons.

Figure 2.5. Help view with navigation buttons

2.11. Adding tiles to the entry page

The typical UI5 entry page has tiles to navigate to the various sections of the application. The definition of these entry points should not be hidden in the view and controller code. Instead, we add it to our `config.json`.

```
{
   availableLanguages: [ "en", "de" ],
   defaultLanguage: "en",
   logLevel: "DEBUG",
   entryPoints: [                                                Array of entry point objects
      {
         "header": "{i18n>tiles.help.header}",
         "tooltip": "{i18n>tiles.help.tooltip}",
         "icon": "sap-icon://sys-help",
         "iconText": "{i18n>tiles.help.iconText}",
         "footer": "{i18n>tiles.help.footer}",
         "routeName": "help"
      }
   ]
};
```

To access the entry points definition, we add a function to the `oui5lib.configuration` namespace.

```
function getTilesDef() {
   const entryPoints = getConfigData("entryPoints");
   if (entryPoints !== undefined && entryPoints instanceof Array) {
      return entryPoints;
   }
   return false;
}
configuration.getEntryPoints = getTilesDef;                    Add function to namespace
```

After this, it is time to add the tiles to the entry view (`view/entry.view.js`). We will use a `sap.m.GenericTile` control to display each entry point.

Listing 2.28. Add Tiles to Page

```
 1 const tiles = [];
 2 const entryPoints = oui5lib.configuration.getEntryPoints();
 3 if (entryPoints) {
 4    let tile;
 5    entryPoints.forEach(function(tileDef) {
 6       tile = new sap.m.GenericTile({                          Construct a GenericTile
 7          header: tileDef.header,
 8          tooltip: tileDef.tooltip,
 9          tileContent: [
10             new sap.m.TileContent({                           Construct TileContent
11                content: new sap.m.ImageContent({
12                   src: tileDef.icon,
13                   description: tileDef.iconText
14                }),
15                footer: tileDef.footer
16             })
17          ],
18          press: function(oEvent) {
19             oController.routeTo(oEvent);                     Attach function to 'press' event
20          }
21       });
22       tile.data("routeName", tileDef.routeName);            Add the routeName as custom data
23
24       tile.addStyleClass("sapUiTinyMarginBegin");
25       tile.addStyleClass("sapUiTinyMarginTop");
26       tile.addStyleClass("tileLayout");
27
28       tiles.push(tile);                                     Add GenericTile to tiles array
29    });
30 }
```

At this point, the entry `Page` is constructed like this:

```
const landmarks = new sap.m.PageAccessibleLandmarkInfo({
   headerRole: "Region",
   headerLabel: "{i18n>view.entry.headerLabel}",
   contentRole: "Main",
   contentLabel: "{i18n>view.entry.contentLabel}"
});
const entryPage = new sap.m.Page({
   landmarkInfo: landmarks,
   customHeader: headerBar,
   content: [ tiles ]
});
```

The landmark labels inform the user about the purpose of the correspondent page section. An english example text for the `headerLabel` is "select language, navigate to help section" and for the `contentLabel` "application entry points".

If we reload the application and click on or touch the tile, nothing happens. Instead, we see an error in the browser console saying "Uncaught TypeError: oController.routeTo is not a function". To correct the error, we have to connect the view with a controller providing the attached `routeTo` function.

But before we come to the actual controller code, we introduce a custom controller implementation to be used as a base class for all our controllers. This will allow us to easily provide common functions for all controllers extending this base class.

2.11.1. Use a BaseController

The default `sap.ui.core.mvc.Controller` has some missing functions, which are needed for most controllers. Extending controllers is the best way to share controller functions used by multiple views. Common functions are imported, while view specific controller functions remain separated.

Listing 2.29. A custom BaseController

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

   const BaseController = oController.extend("oui5lib.controller.BaseController", {
      getRouter: function () {                                          Get the component Router
         return sap.ui.core.UIComponent.getRouterFor(this);
      },
      getEventBus: function () {                                        Get the component EventBus
         return this.getOwnerComponent().getEventBus();
      },
      getResourceBundle: function () {                                  Get the language ResourceBundle
         return this.getOwnerComponent().getModel("i18n").getResourceBundle();
      },

      debug: function (msg) {                                            Logging
         oui5lib.logger.debug(this.addControllerName(msg));
      },
      info: function (msg) {
         oui5lib.logger.info(this.addControllerName(msg));
      },
      warn: function (msg) {
         oui5lib.logger.warn(this.addControllerName(msg));
      },
      error: function (msg) {
         oui5lib.logger.error(this.addControllerName(msg));
      },

      addControllerName: function(msg) {
         const metadata = this.getMetadata();
         return metadata.getName() + " > " + msg;
      }
   });
   return BaseController;
});
```

If a controller extends the `BaseController`, logging can be both simplified and improved. Within these controllers, calling the logging functions is most convenient:

```
this.info("just an example");
```

Additionally, the log messages will now include the controller name for easier debugging.

2.11.2. Adding an entry controller

Now, let us return to the missing event handler function of Listing 2.28, “Add Tiles to Page”. To connect a controller with a view, we add the function `getControllerName` to the entry view which returns the controller name to be connected to the view.

```
getControllerName: function() {
   return "oum.controller.entry";
}
```

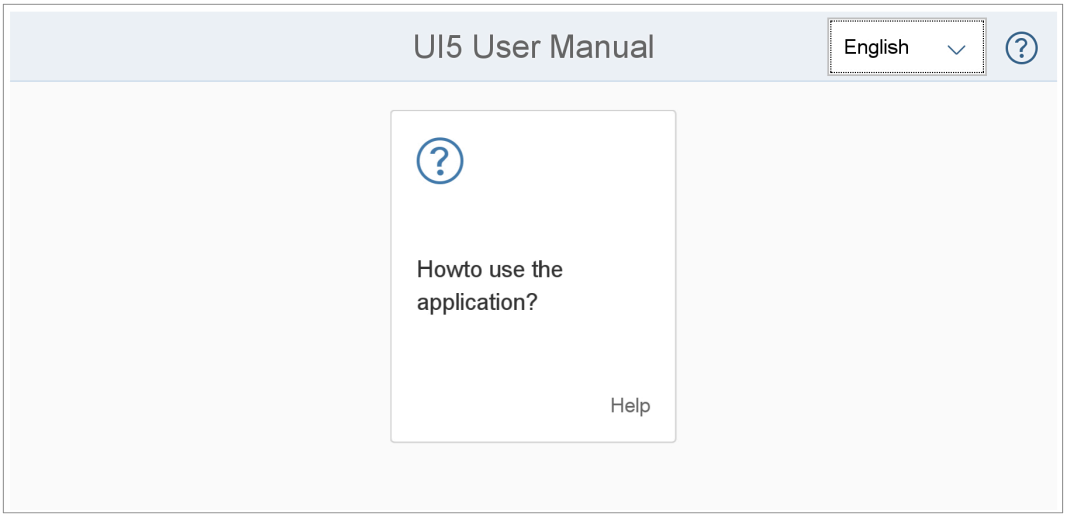
According to the UI5 naming scheme and the `sap-ui-resourceroots` setting, the controller goes into the folder `controller` and has the filename `entry.controller.js`.

Listing 2.30. The entry controller extends the BaseController

```
 1 sap.ui.define([
 2    "oui5lib/controller/BaseController"
 3 ], function(BaseController) {
 4    "use strict";
 5
 6    const entryController = BaseController.extend("oum.controller.entry", {
 7       onInit: function () { ❶
 8          this.getRouter().getRoute("home")
 9                          .attachMatched(this._onRouteMatched, this);
10       },
11       _onRouteMatched: function(oEvent) {
12          this.debug("coming home");                          Use BaseController debug function
13       },
14
15       routeTo: function(oEvent) {
16          const tile = oEvent.getSource();
17          const routeName = tile.data("routeName"); ❷
18
19          this.info("navTo: " + routeName);                   Use BaseController info function
20          this.getRouter().vNavTo(routeName);
21       }
22    });
23
24    return entryController;
25 });
```

❶ The `onInit` function (lines 7 to 10) is used for nearly every controller to set models required for the view. Additionally, the controller usually needs to do something when the route of the view was requested. First, we get the route with a given name ("home") from the `Router` (line 8) and then use the `attachMatched` function of the `Route` to attach a handler function to the `matched` event (line 9).

❷ The `routeName` was set in Listing 2.28, “Add Tiles to Page” line 21.



If we now reload the application with the browser console open, we will notice that it is beginning to give us feedback. This will be crucial for debugging our application. Having drawn near to the completion of the template component, we have created a lot of files:

```
.
├── controller
│   └── entry.controller.js
├── fragment
│   ├── BackButton.fragment.js
│   ├── HelpButton.fragment.js
│   ├── HomeButton.fragment.js
│   └── LanguageSwitcher.fragment.js
├── i18n
│   ├── i18n_de.properties
│   └── i18n_en.properties
├── oui5lib
│   ├── controller
│   │   └── BaseController.js
│   ├── configuration.js
│   ├── init.js
│   ├── logger.js
│   └── util.js
├── view
│   ├── help
│   │   ├── index.view.js
│   │   └── intro.view.js
│   ├── app.view.xml
│   ├── entry.view.js
│   ├── noRoute.view.xml
│   └── splitApp.view.xml
├── Component.js
├── Component-preload.js
├── config.json
├── index.html
├── manifest.json
└── Router.js
```

The template component doesn't do anything, but has everything in place to focus on content development. We will use it as starting point in the following chapters.

2.12. Adding a Footer

For most pages, a footer will hold various action controls. But the home page has no such actions. Instead, we want to show some basic information about the running application, like the application title and version.

The `sap.m.Page` has a `footer` aggregation accepting controls implementing `sap.m.IBar`. We have already used the `sap.m.Bar` for the header. For the footer we use the `sap.m.Toolbar`. This fragment is of type XML and its location `fragment/AppInfoToolbar.fragment.xml`.

Listing 2.31. XML Toolbar Fragment showing basic application information

```
<core:FragmentDefinition xmlns="sap.m"
                         xmlns:core="sap.ui.core">
   <Toolbar>
      <Text text="{i18n>application}"/>
      <Text text="{/appTitle}"/>
      <Text text="{i18n>version}"/>
      <Text text="{/appVersion}"/>
      <ToolbarSpacer/>
      <Text text="OpenUI5 Version: "/>
      <Text text="{/openui5Version}"/>
   </Toolbar>
</core:FragmentDefinition>
```

The fragment adds some i18n properties ("application", "version"). There are also data we need to get from the manifest ("appTitle", "appVersion") and UI5 ("openui5Version"). Implementing the Model-View-Controller (MVC) pattern, UI5 uses models to separate the data from the view. To provide the required model for the footer, we add a `getAppInfoModel` function to the `oui5lib.configuration` namespace.

```
function getAppModel() {
   const component = getComponent();
   const appConfig = component.getManifestEntry("sap.app");

   const appModel = new sap.ui.model.json.JSONModel({
      appTitle: appConfig.title,
      appVersion: appConfig.applicationVersion.version,
      openui5Version: sap.ui.version
   });
   return appModel;
}
configuration.getAppInfoModel = getAppModel;                    Add function to namespace
```

Where do we set the model? As a general rule, it is best to choose the smallest scope possible. In our case, it is the `Toolbar` of the entry page `footer` aggregation. We, therefore, set the model in the `controller/entry.controller.js` `onInit` function.

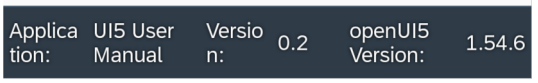
```
const page = this.getView().getContent()[0]; ❶
page.getFooter().setModel(oui5lib.configuration.getAppInfoModel());
```

❶ Here, the `sap.m.Page` is the first and only item of the `sap.ui.core.mvc.View` `content` aggregation, which holds an array of content controls.

Finally, we add the fragment to the `footer` aggregation of the `Page` of our entry view.

```
const entryPage = new sap.m.Page({
   landmarkInfo: landmarks,
   customHeader: headerBar,
   content: [ tiles ],
   footer: sap.ui.xmlfragment("oum.fragment.AppInfoToolbar")
});
```

The `sap.m.Toolbar` will shrink its content with the available space. This causes a problem with mobile phones in portrait orientation. We can ignore this for now because our application manifest doesn't claim to be fully usable with mobile phones (see Listing 2.2, “Descriptor Application Manifest” under `sap.ui.deviceTypes`).



We will learn how to construct a more responsive toolbar in the following Chapter 3, *Views and Pages*.