* Go to the development perspective by clicking the “</>”icon in the left toolbar.
* On the left side you will see a folder list with just one entry: “Workspace”.
* Add a new project folder MyApp by right-clicking it and selecting “New > Folder”.
* Inside the project folder, create a new folder webapp which will contain all sources of the app.
* Now create a new root HTML file called index.html in your app folder with the contents:

*<!DOCTYPE html >*

*<html>*

*<head>*

*<meta http-equiv="X-UA-Compatible" content="IE=edge"/>*

*<meta charset="UTF-8">*

*<title>My Web Apps with SAPUI5</title>*

*</head>*

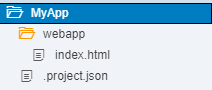
*<body>*

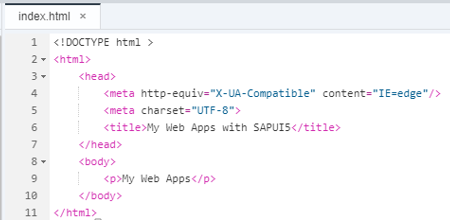
*<p>My Web Apps</p>*

*</body>*

*</html>*

* At end should look like this:





* Save your changes and Run 
* Before we can do anything with SAPUI5, we need to load and initialize it. This process of loading and initializing SAPUI5 is called bootstrapping. Once this bootstrapping is finished, we simply display an alert.
* Add code to webapp/index.html header:

*<script*

*id="sap-ui-bootstrap"*

*src="https://sapui5.hana.ondemand.com/resources/sap-ui-core.js"*

*data-sap-ui-theme="sap\_bluecrystal"*

*data-sap-ui-libs="sap.m"*

*data-sap-ui-compatVersion="edge"*

*data-sap-ui-preload="async">*

*</script>*

*<script>*

*sap.ui.getCore().attachInit(function () {*

*alert("SAPUI5 is ready");*

*});*

*</script>*

* Add code to webapp/index.html body:

*<body class="sapUiBody" id="content">*

*</body>*



* Save your changes and Run 

We initialize the UI5 core with the following configuration options:

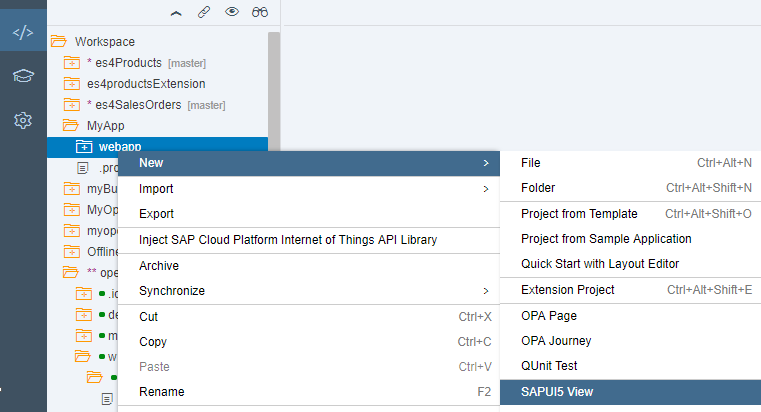
* The src attribute of the first <script> tag tells the browser where to find the SAPUI5 core library – it initializes the SAPUI5 runtime and loads additional resources, such as the libraries specified in the data-sap-ui-libs attribute.
* The SAPUI5 controls support different themes, we choose sap\_bluecrystal as our default theme.
* We specify the required UI library sap.m containing the UI controls we need for this tutorial.
* To make use of the most recent functionality of SAPUI5 we define the compatibility version as edge.
* We configure the process of “bootstrapping” to run asynchronously.  
  This means that the SAPUI5 resources can be loaded simultaneously in the background for performance reasons.

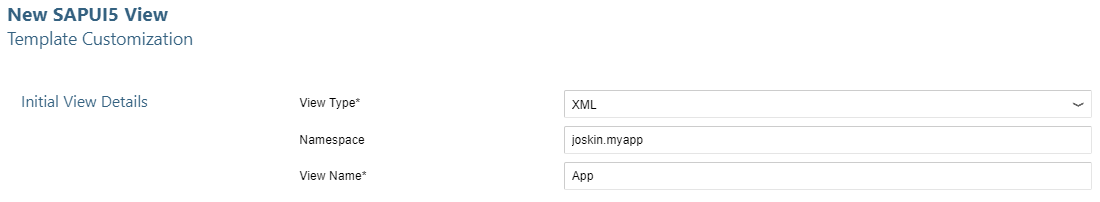
When all resources and libraries are loaded, the SAPUI5 runtime fires the global init event to signal that the library is ready. When loading the libraries asynchronously, you have to listen for this event in order to trigger your application logic only after the required resources are loaded.

In the example above, we get a reference to the SAPUI5 core by calling sap.ui.getCore() and register an anonymous callback function for the init event by calling attachInit(…) on the core. In SAPUI5 these kinds of callback functions are often referred to as event handlers, event listener functions, or simply listeners. The core is a singleton and can be accessed from anywhere in the code.

Our anonymous callback function is executed when the bootstrap of SAPUI5 is finished and displays a native JavaScript alert.

* Create a new view:





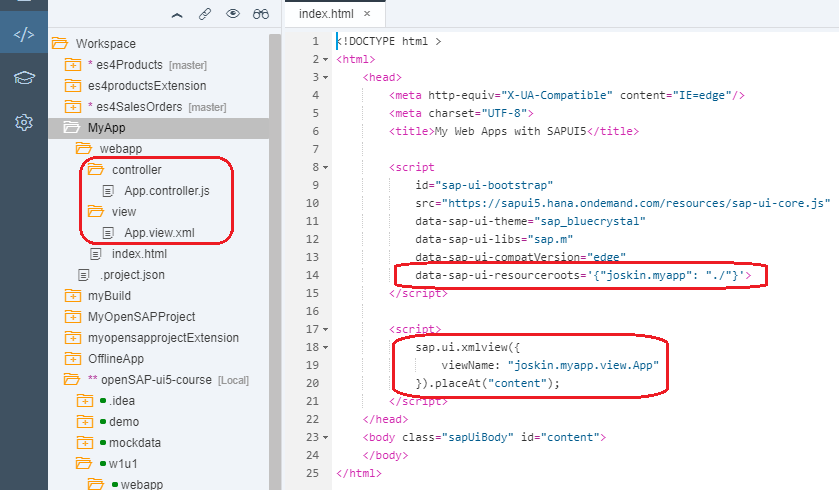
* Add code to webapp/index.html header:

*data-sap-ui-resourceroots='{"joskin.myapp": "./"}'>*

*sap.ui.xmlview({*

*viewName: "joskin.myapp.view.App"*

*}).placeAt("content");*



* Save your changes and Run 

**Views Conventions**

* View names are capitalized.
* All views are stored in the view folder.
* Names of XML views always end with \*.view.xml.
* The default XML namespace is sap.m.
* Other XML namespaces are referenced by an alias.

**Controllers Conventions**

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

In this step, we will encapsulate all UI assets in a component and introduce the app descriptor.

Components are independent and reusable parts used in SAPUI5 applications. For example, SAP Fiori apps usually run in an SAP Fiori launchpad which acts as a so-called component container. It manages and loads the apps by loading the corresponding component of the app. Whenever we address resources of the app, we will now do this relatively to the component (instead of relatively to the index.html). This architectural change allows our app to be used in more flexible environments than our static index.html page

First, in the webapp folder, we create an initial Component.js file that will hold our application setup. The init function of the component is automatically invoked by SAPUI5 when the component is instantiated. Our component inherits from base class sap.ui.core.UIComponent and must call the init function of the base class in the overridden init method. The Component.js file consists of two parts: The new metadata section that refers to the app descriptor which we still have to implement, and the previously introduced init function that is called when the component is initialized. Instead of instantiating the root view directly from the index.html file as we did previously, the component will now manage the display of the app view.

* Create a new webapp/Component.js and add code:

*sap.ui.define([*

*"sap/ui/core/UIComponent"*

*], function (UIComponent) {*

*"use strict";*

*return UIComponent.extend("joskin.myapp.Component", {*

*metadata : {*

*manifest: "json"*

*},*

*init : function () {*

*// call the init function of the parent*

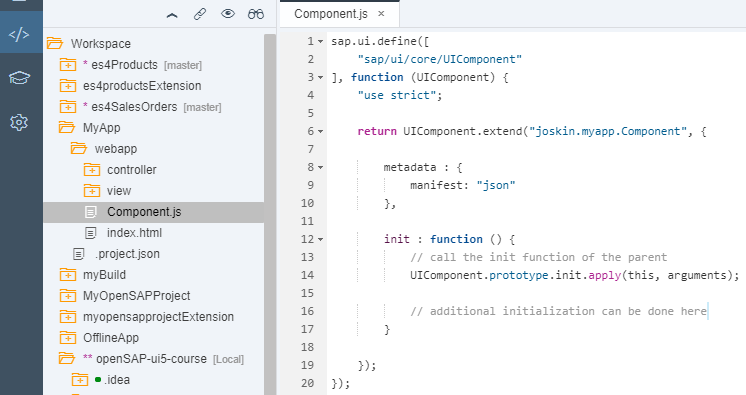
*UIComponent.prototype.init.apply(this, arguments);*

*// additional initialization can be done here*

*}*

*});*

*});*



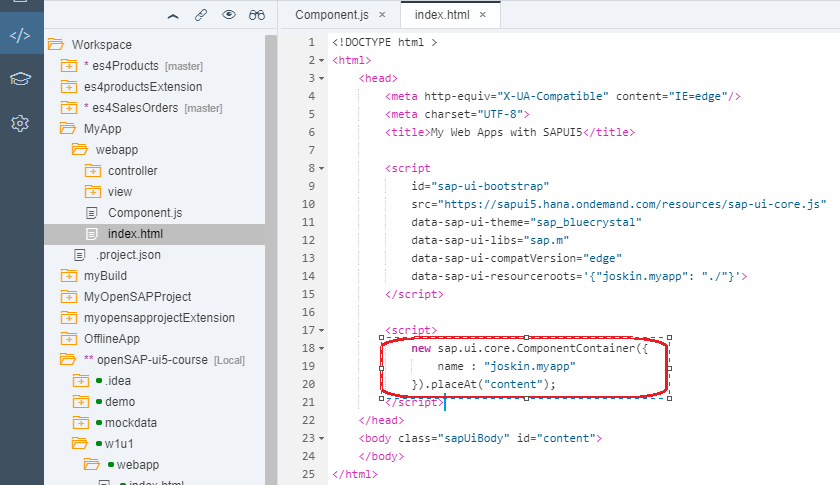
On the index page, we now instantiate the component instead of the app view. The factory method sap.ui.core.ComponentContainer instantiates the component by searching for a Component.js file in the namespace that is passed as an argument. The component automatically displays the root view that is defined within the app descriptor.

* Add code to webapp/index.html:

*new sap.ui.core.ComponentContainer({*

*name : "joskin.myapp"*

*}).placeAt("content");*



All application-specific configuration settings will now be put into a separate descriptor file called manifest.json. This clearly separates the application code from the configuration settings and makes our app even more flexible. The SAP Fiori launchpad acts as an application container and instantiates the app without having an application-specific HTML file for the bootstrap. Instead, the descriptor file is parsed and the component is loaded into the current HTML page. This allows several apps to be displayed in the same context. Each app can define local settings, such as dependent libraries, supported themes and device types, and more.

The content of the manifest.json file is a configuration object in JSON format that contains all application settings and parameters. The manifest file is also called the “descriptor for applications, components, and libraries”, and is sometimes abbreviated as “descriptor” or “app descriptor” when used for applications. It is stored in the webapp folder and read by SAPUI5 to instantiate the component that we will create in this step. For SAPUI5 applications, there are three important sections defined by namespaces in the manifest.json file: sap.app, sap.ui and sap.ui5.

* Create a new webapp/manifest.json and add code:

*{*

*"\_version": "1.3.0",*

*"sap.app": {*

*"\_version": "1.3.0",*

*"id": "joskin.myapp",*

*"type": "application",*

*"title": "{{appTitle}}",*

*"description": "{{appDescription}}",*

*"applicationVersion": {*

*"version": "1.0.0"*

*}*

*},*

*"sap.ui": {*

*"\_version": "1.3.0",*

*"technology": "UI5",*

*"deviceTypes": {*

*"desktop": true,*

*"tablet": true,*

*"phone": true*

*},*

*"supportedThemes": [*

*"sap\_bluecrystal"*

*]*

*},*

*"sap.ui5": {*

*"\_version": "1.2.0",*

*"rootView": {*

*"viewName": "joskin.myapp.view.App",*

*"type": "XML",*

*"id": "app"*

*},*

*"autoPrefixId": true,*

*"dependencies": {*

*"minUI5Version": "1.34",*

*"libs": {*

*"sap.ui.core": {*

*"minVersion": "1.34.0"*

*},*

*"sap.m": {*

*"minVersion": "1.34.0"*

*},*

*"sap.ui.layout": {*

*"minVersion": "1.34.0"*

*}*

*}*

*},*

*"contentDensities": {*

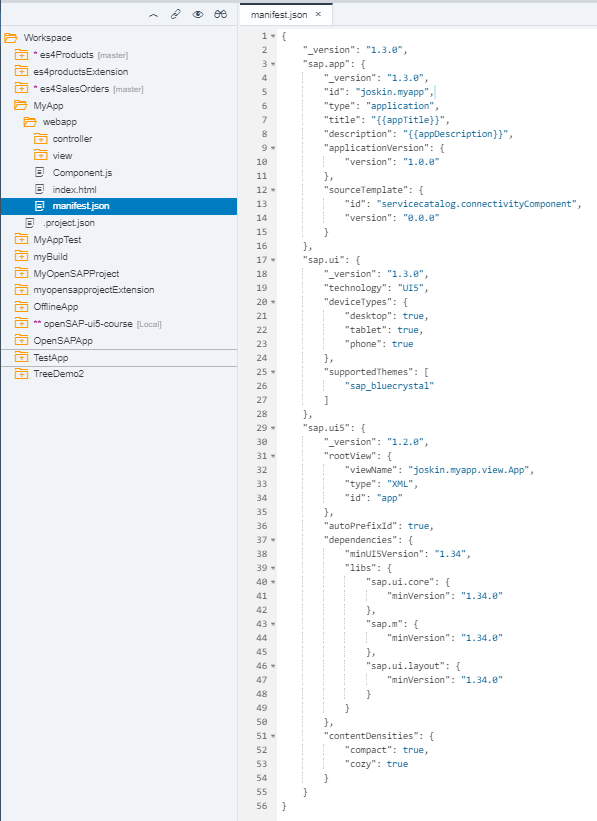
*"compact": true,*

*"cozy": true*

*}*

*}*

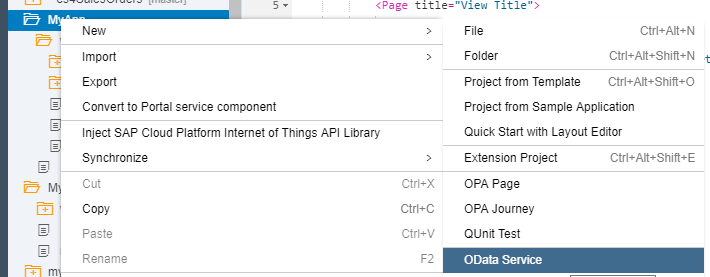
*}*



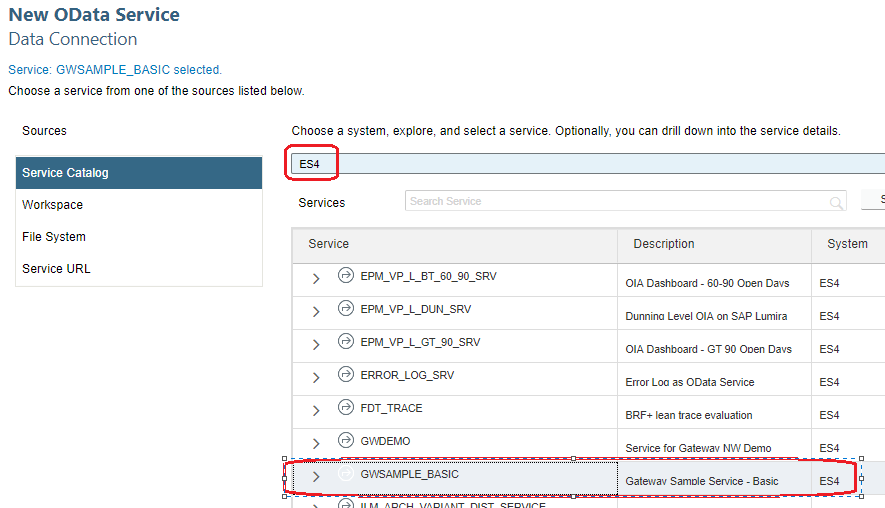
* Save your changes and Run 

**Conventions**

* The component file is named Component.js.
* Together with all UI assets of the app, the component is located in the webapp folder.
* If the index.html file is used productively, it is located in the webapp folder.
* The descriptor file is named manifest.json and located in the webapp folder.
* Use translatable strings for the title and the description of the app.
* Add a new OData service:



* I am using SAP Demo Gateway, you should select your gateway and service:



The file neo-app.json is automatically generated.

The folder localService with file metadata.xml are automatically generated.

The file .project.json is updated.

* The file manifest.json is updated with the New OData service. In your case you should see your Service. Now we need to register the data model in sap.ui5:

*"models": {*

*"ES4": {*

*"dataSource": "ES4"*

*}*

*},*

* At end should look like this:



* Add a List control to App.view.xml to display data, in your case you should use your data model from your gateway with the filed from your Odata:

*<List headerText="Products" items="{ES4>/ProductSet}">*

*<items>*

*<ObjectListItem*

*title="{ES4>Name}"*

*number="{ES4>Price}"*

*intro="{ES4>ProductID}" />*

*</items>*

*</List>*



* Add the sap.m.SearchField to the list header in webapp/view/App.view.xml.

We will put the SearchField into the list’s ‘HeaderToolbar’ where we can also place the ‘headerText’. The ‘headerText’ can therefore be removed from the list properties. For the ‘search’ event we provide the handler function ‘onFilterProducts’ which we need to implement in the controller.

*<List headerText="Products" items="{ES4>/ProductSet}" id="productsList">*

*<headerToolbar>*

*<Toolbar>*

*<Title text="Product List Title"/>*

*<ToolbarSpacer/>*

*<SearchField width="50%" search="onFilterProducts"/>*

*</Toolbar>*

*</headerToolbar>*

*<items>*

*<ObjectListItem title="{ES4>Name}" number="{ES4>Price}" intro="{ES4>ProductID}"/>*

*</items>*

*</List>*



* Add method onFilterProducts in webapp/controller/App.controller.js to search in the Odata:

Applying filters (and sorters) can be done using the list’s ‘items’ binding. So we first retrieve the list control using its id suffix ‘productList’. The control object gives us access to the list binding object. Now we can create our filter object for field ‘ProductId’. We choose a filter operator and fetch the search field value from the handler function’s event parameter. Finally, we can call the binding’s filter method with our new filter.

Please note: With the above code we have implemented a search which is case-sensitive. When you try searching for a product ID, take care to write the same letters in upper or lower case as they are in the actual product ID.

*,*

*"sap/ui/model/Filter",*

*"sap/ui/model/FilterOperator"*

And

*, Filter, FilterOperator*

And

*onFilterProducts : function (oEvent) {*

*// build filter array*

*var aFilter = [], sQuery = oEvent.getParameter("query"),*

*// retrieve list control*

*oList = this.getView().byId("productsList"),*

*// get binding for aggregation 'items'*

*oBinding = oList.getBinding("items");*

*if (sQuery) {*

*aFilter.push(new Filter("ProductID", FilterOperator.Contains, sQuery));*

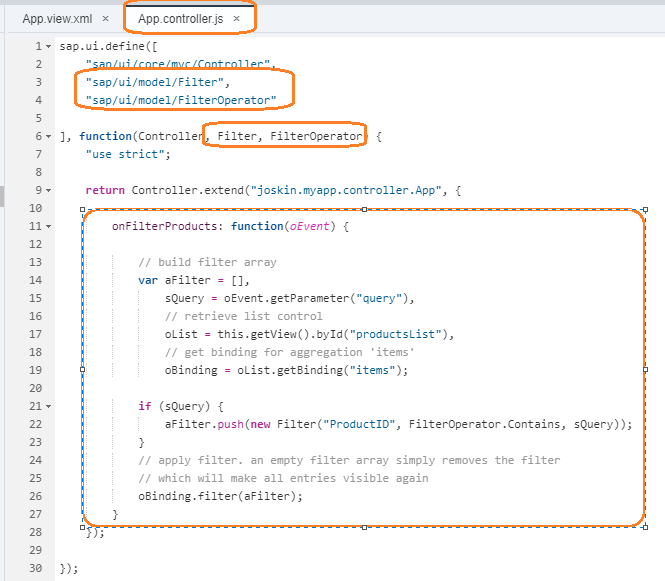
*}*

*// apply filter. an empty filter array simply removes the filter*

*// which will make all entries visible again*

*oBinding.filter(aFilter);*

*}*



* Save your changes and Run 