**<script**

**src=""**

-> core library path

**theme**

-> supports themes

**libs**

-> UI library "sap.m"

**compatVersion**

-> "edge" for most recent funcs

**async "true"**

-> process of "bootstrapping" to be async

**onInit**

-> module loaded initially, in a declarative way, to avoid directly executable JS code in HTML file (more secure)

**resourceroots sap.ui.demo.walkthrough": "./"**

-> we tell that resources are located in the same folder index.html

**> </script>**

**index.js contains the app logic**, which will be called by index.html, where we define it as a module in a declarative way to avoid having executable code directly in the HTML file. (security)

/\* ----------------------------- \*/

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

The name of the control is prefixed by the namespace of its control library **sap/m/** and the options are passed to the constructor **with a JavaScript object**.

**placeAt** that is used to place SAPUI5 controls inside a node of the document object model (DOM) or any other SAPUI5 control instance. We pass the ID of a DOM node as an argument.

**! Note**

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

/\* ----------------------------- \*/

/\* XML View \*/

Using XML to force to separate the view declaration from the controller logic.

The root node of the XML structure is the view. Here, we reference the default namespace sap.m where the majority of our UI assets are located.

We define an additional sap.ui.core.mvc namespace with alias mvc, where the SAPUI5 views and all other Model-View-Controller (MVC) assets are located.

**Note**

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

/\* ----------------------------- \*/

*/\* CONTROLLER \*/*

**"sap/ui/core/mvc/Controller"**

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

We create the folder webapp/controller and a new file App.controller.js inside.

We define the app controller in its own file by extending the Controller object of the SAPUI5 core.

**Conventions**

1. Controller names are capitalized
2. Controllers carry the same name as the related view (if there is a 1:1 Relationship)
3. Event handlers are prefixed with *on*
4. Controller names always end with “.controller.js”

/\* ----------------------------- \*/

**Conventions**

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

/\* ----------------------------- \*/

**"sap/ui/model/json/JSONModel"**

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

The JSON Model only contains a property “recipient” which contains one property: name

var oData { recipient { name: “Hello World” } }

We bind it’s value to a SAPUI5 model by using the declarative binding syntax for XML views:

* The curly brackers {...} indicate that data is taken from the value of the recipient’s object name property. This is called **data binding**.
* /recipient/name declares the path in the model.
* **<Input**
* **value="{/recipient/name}"**
* **description="Hello {/recipient/name}"**
* **valueLiveUpdate="true"**
* **width="60%"/>**

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

If the setting above is not **“edge”** or is omitted, only standard binding syntax is allowed “{/recipient/name}”, and not “Hello {/recipient/name}”

**Note**

You can either use data-sap-ui-compatVersion="edge" or data-sap-ui-bindingSyntax="complex" in the script. By setting the "edge" compatibility mode, the complex binding syntax is automatically enabled. The edge mode automatically enables compatibility features that otherwise would have to be enabled manually. For more information, see [Compatibility Version Information](https://sapui5.hana.ondemand.com/#/topic/9feb96da02c2429bb1afcf6534d77c79.html).

/\* ----------------------------- \*/

i18n is the process of internationalization => translatable texts of our UI are moved to a separate resource file to be all in a central place and be easily translated into other languages.

The getProperty method can be called in any model and takes the data path as an argument. In addition, the resource bundle has a specific getText method that takes an array of strings as second argument.

* Never concatenate strings that are translated, always use placeholders.
* Use Unicode escape sequences for special characters.

/\* ----------------------------- \*/

majestic.json => descriptor for applications

/\* ----------------------------- \*/

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

/\* ----------------------------- \*/

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

/\* ----------------------------- \*/

If the dialog in the fragment does not exist yet, the fragment is instantiated by calling the loadFragment API.

As you can see in the snippet above, we store the loading Promise of the dialog fragment on the controller instance. This allows us to handle the opening of the dialog asynchronously on each click of the helloDialogButton button.

To reuse the dialog opening and closing functionality in other controllers, you can create a new file sap.ui.demo.walkthrough.controller.BaseController, which extends sap.ui.core.mvc.Controller, and put all your dialog-related coding into this controller. Now, all the other controllers can extend from sap.ui.demo.walkthrough.controller.BaseController instead of sap.ui.core.mvc.Controller.

Private functions and variables should always start with an underscore.

/\* ----------------------------- \*/

/\* ----------------------------- \*/

/\* Extra Controls inside a control \*/

In a **.view.xml**, inside a control, <Dialog> </Dialog> for e.g., we can add content such as buttons, icons.

Buttons by <beginButton> <Button ….> </beginButton>

(for a better fitting? As it can be added without the beginButton control too)

Icons by <content> <core:Icon /> </content>

/\* ----------------------------- \*/

**"invoice": {**

**"type": "sap.ui.model.json.JSONModel",**

**"uri": "Invoices.json"**

**}**

This time we want a JSONModel, so we set the type to sap.ui.model.json.JSONModel. The uri key is the path to our test data relative to the component. With this little configuration our component will automatically instantiate a new JSONModel which loads the invoice data from the Invoices.json file.

/\* ----------------------------- \*/

number="{

parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/currency'}],

type: 'sap.ui.model.type.Currency',

formatOptions: {

showMeasure: false

}

}"

numberUnit="{view>/currency}"

**numberState="{= ${invoice>ExtendedPrice} > 50 ? 'Error' : 'Success' }"**/>

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

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

An expression binding is specified in an XML view by one of the following two options:

* {=**expression**}

This variant uses one-way binding. This allows the automatic recalculation if the model values change.

* {:=**expression**}

This variant uses one-time binding, meaning that the value is calculated only once. This variant needs less resources because no change listeners to the model have to be maintained.

The syntax of the expression is similar to JavaScript syntax, but you can only use a subset of the JavaScript expression syntax as defined in the table below. Additionally, you can embed values from the model layer into an expression as additional bindings by using one of the following syntaxes:

* ${**binding**}
* %{**binding**}

More about it here: https://sapui5.hana.ondemand.com/#/topic/daf6852a04b44d118963968a1239d2c0.html

/\* ----------------------------- \*/

/\* ----------------------------- \*/

**"../model/formatter"**

], function( formatter ),

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

**formatter: formatter,**

When we create files for webapp/model/ , most probably we won’t need to extend from any base project, as we gonna just return JavaScript content (object?) “with our formatter functions inside the sap.ui.define call.” (step 21 example)”

In order to load the functions from /model/ , we have to add it in our controller.

When we want to add more content to an item object from a list, we can just close the tag for now, and add the ending tag after.

Like, instead of <ObjectsListItem /> , we can do <ObjectsListItem> our new content </ObjectsListItem>

/\* ----------------------------- \*/

/\* FILTERING \*/

We load two new dependencies for the filtering. The filter object will hold our configuration for the filter action and the FilterOperator is a helper type that we need in order to specify the filter.

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

**"sap/ui/model/FilterOperator"**

Event handlers always receive an event argument that can be used to access the parameters that the event provides. The search field defines a parameter query that we access by calling getParameter(“query”) on the oEvent parameter.

! The filter operator FilterOperator.Contains is **not** case-sensitive.

**var aFilter = [];**

**var sQuery = oEvent.getParameter("query");**

**if (sQuery) {**

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

**}**

To sort the items in an <List> control, we can just add a sorter to the “items” attribute, like this:

We switch from simple binding syntax “items=”{invoice>/Invoices}” to the object notation:

**items="{**

**path : 'invoice>/Invoices',**

**sorter : {**

**path : 'ProductName'**

**}**

**}"**

/\* ----------------------------- \*/

/\* DEBUGGER \*/

Just add the following query info to your link: ?sap-ui-debug=true

Like: [http://localhost:8080/index.html**?sap-ui-debug=true**](http://localhost:8080/index.html?sap-ui-debug=true)

/\* ----------------------------- \*/

/\* ROUTING \*/

There are three subsections that define the routing and navigation structure of the app:

* config

This section contains the global router configuration and default values that apply for all routes and targets. We define the router class that we want to use and where our views are located in the app. To load and display views automatically, we also specify which control is used to display the pages and what aggregation should be filled when a new page is displayed.

* routes

Each route defines a name, a pattern, and one or more targets to navigate to when the route has been hit. The pattern is basically the URL part that matches to the route, we define two routes for our app. The first one is a default route that will show the overview page with the content from the previous steps, and the second is the detail route with the URL pattern detail that will show a new page.

* targets

A target defines a view that is displayed, it is associated with one or more routes and it can also be displayed manually from within the app. Whenever a target is displayed, the corresponding view is loaded and shown in the app. In our app we simply define two targets with a view name that corresponds to the target name.

To assign multiple targets to one route, we put them in an array, like:

"routes": [{

"pattern": "employees/overview",

"name": "employeeOverview",

"target": [**"employeeOverviewTop", "employeeOverviewContent"**]

}],

Then, when we do targets settings, we declare the parent like:

"targets": {

**"employeeOverview"**: {

"type": "View",

"name": "EmployeeOverview",

"level": 2,

"controlId": "app",

"controlAggregation": "content"

},

"employeeOverviewTop": {

**"parent": "employeeOverview",**

"type": "View",

"name": "EmployeeOverviewTop",

"controlId": "EmployeeOverviewParent",

"controlAggregation": "content"

},

"employeeOverviewContent": {

**"parent": "employeeOverview",**

"type": "View",

"name": "EmployeeOverviewContent",

"controlId": "EmployeeOverviewParent",

"controlAggregation": "content"

}

}

! One instance of a controller is instantiated for each view that references the controller.

**webapp/view/App.view.xml**

<mvc:View

controllerName="sap.ui.demo.walkthrough.controller.App"

xmlns="sap.m"

xmlns:mvc="sap.ui.core.mvc"

displayBlock="true">

**<Shell>**

**<App class="myAppDemoWT" id="app"/>**

**</Shell>**

</mvc:View>

The router will automatically add the view that corresponds to the current URL into the app control. The router identifies the app control with the ID that corresponds to the property controlId: “app” in the AppDescriptor.

**type="Navigation"**

**press="onPress"**>

We add a press event to the list item and set the item type to Navigation so that the item can actually be clicked.

Then, in the controller:

**onPress: function (oEvent) {**

**var oRouter = this.getOwnerComponent().getRouter();**

**oRouter.navTo("detail");**

**}**

We add the event handler function of our invoices list. Now it is time to navigate to the detail page by clicking an item in the invoice list. We access the router instance for our app by calling the helper method getOwnerComponent().getRouter(). On the router we call the navTo method to navigate to the detail route that we specified in the routing configuration.

/\* ROUTING WITH PARAMETERS \*/

"pattern": "detail**/{invoicePath}**",

We now add a navigation parameter invoicePath to the detail route so that we can hand over the information for the selected item to the detail page. Mandatory navigation parameters are defined with curly brackets.

The binding context can be accessed by calling the getBindingContext method with the model name on any bound SAPUI5 control. We need to remove the first / from the binding path by calling .substr(1) on the string because this is a special character in URLs and is not allowed, we will add it again on the detail page.

onPress: function (oEvent) {

**var oItem = oEvent.getSource();**

var oRouter = this.getOwnerComponent().getRouter();

oRouter.navTo("detail"**, {**

**invoicePath: window.encodeURIComponent(oItem.getBindingContext("invoice").getPath().substr(1))**

**}**);

}

## webapp/controller/Detail.controller.js (New)

**sap.ui.define([**

**"sap/ui/core/mvc/Controller"**

**], function (Controller) {**

**"use strict";**

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

**onInit: function () {**

**var oRouter = this.getOwnerComponent().getRouter();**

**oRouter.getRoute("detail").attachPatternMatched(this.\_onObjectMatched, this);**

**},**

**\_onObjectMatched: function (oEvent) {**

**this.getView().bindElement({**

**path: "/" + window.decodeURIComponent(oEvent.getParameter("arguments").invoicePath),**

**model: "invoice"**

**});**

**}**

**});**

**});**

In the onInit method of the controller we fetch the instance of our app router and attach to the detail route by calling the method attachPatternMatched on the route that we accessed by its name. We register an internal callback function \_onObjectMatched that will be executed when the route is hit, either by clicking on the item or by calling the app with a URL for the detail page.

In the \_onObjectMatched method that is triggered by the router we receive an event that we can use to access the URL and navigation parameters. The arguments parameter will return an object that corresponds to our navigation parameters from the route pattern. We access the invoicePath that we set in the invoice list controller and call the bindElement function on the view to set the context. We have to add the root / in front of the path again that was removed for passing on the path as a URL parameter.

The bindElement function is creating a binding context for a SAPUI5 control and receives the model name as well as the path to an item in a configuration object. This will trigger an update of the UI controls that we connected with fields of the invoice model. You should now see the invoice details on a separate page when you click on an item in the list of invoices.

/\* ON NAV BACK \*/

**"sap/ui/core/routing/History"**

function (Controller**, History**) {

**onNavBack: function () {**

**var oHistory = History.getInstance();**

**var sPreviousHash = oHistory.getPreviousHash();**

**if (sPreviousHash !== undefined) {**

**window.history.go(-1);**

**} else {**

**var oRouter = this.getOwnerComponent().getRouter();**

**oRouter.navTo("overview", {}, true);**

**}**

**}**

/\* ------------------------ \*/

/\* CUSTOM CONTROL \*/

## webapp/control/ProductRating.js (New)

**sap.ui.define([**

**"sap/ui/core/Control"**

**], function (Control) {**

**"use strict";**

**return Control.extend("sap.ui.demo.walkthrough.control.ProductRating", {**

**metadata : {**

**},**

**init : function () {**

**},**

**renderer : function (oRM, oControl) {**

**}**

**});**

**});**

Due to their nature, they are sometimes also referred to as "notepad” or “on the fly” controls.

The metadata section defines the data structure and thus the API of the control. With this meta information on the properties, events, and aggregations of the control SAPUI5 automatically creates setter and getter methods and other convenience functions that can be called within the app.

The renderer defines the HTML structure that will be added to the DOM tree of your app whenever the control is instantiated in a view. It is usually called initially by the core of SAPUI5 and whenever a property of the control is changed. The parameter oRM of the render function is the SAPUI5 render manager that can be used to write strings and control properties to the HTML page.

The init method is a special function that is called by the SAPUI5 core whenever the control is instantiated. It can be used to set up the control and prepare its content for display.

**Note**

Controls always extend sap.ui.core.Control and render themselves. You could also extend sap.ui.core.Element or sap.ui.base.ManagedObject directly if you want to reuse life cycle features of SAPUI5 including data binding for objects that are not rendered. Please refer to the API reference to learn more about the inheritance hierarchy of controls.

MORE about it: <https://sapui5.hana.ondemand.com/#/topic/d12d2ee6a5454d799358d425f9e7c4db>

/\* --------------------------- \*/

/\* RESPONSIVENESS \*/

A <List> Control is similar to <Table> as for properties, but <Table> is more responsive.

So we can switch from list to table to gain use of the responsiveness. Also, we can apply certain properties that will set the importance of a column to be shown or not when screen is smaller.

We set the property minScreenWidth to Small to indicate that this column is not so important on phones. We will tell the table to display this column below the main column by setting the property demandPopin to true.

/\* DEVICE ADAPTATION \*/

class="**sapUiVisibleOnlyOnDesktop**"/>

^ this class will make the control available only on Desktop. Will disappear on other devices.

(the opposite is sapUiHideOnDesktop, to show it on any other device but desktop)

On parent control:

**expandable="{device>/system/phone}"**

**expanded="{= !${device>/system/phone} }"**>

We add two new properties expandable and expanded to the HelloPanel.

The property expandable is bound to a model named device and the path /system/phone. So the panel can be expanded on phone devices only.

The expanded property controls the state of the panel and we use expression binding syntax to close it on phone devices and have the panel expanded on all other devices.

**Note**

The sap.ui.Device API detects the device type (Phone, Tablet, Desktop) based on the user agent and many other properties of the device. **Therefore simply reducing the screen size will not change the device type.** To test this feature, you will have to enable device emulation in your browser or open it on a real device.

**Note**

We have to set the binding mode to OneWay as the device model is read-only and we want to avoid changing the model accidentally when we bind properties of a control to it. By default, models in SAPUI5 are bidirectional (TwoWay). When the property changes, the bound model value is updated as well.

Some controls already have built-in responsive features that can be configured. The ObjectHeader control can be put in a more flexible mode by setting the attribute responsive to true and fullScreenOptimized to true as well. This will show the data that we add to the view now at different positions on the screen based on the device size.

/\* ----------------------------- \*/

/\* CONTENT DENSITY \*/

We use certain classes which make the buttons and some other controls more compact/cozy, as when using a mouse, we don’t need big buttons like we do when touchscreen is available.

The "Hello World" dialog is not part of the HelloPanel view but opened in a special part of the DOM called "static area". The content density class defined on the app view is not known to the dialog so we sync the style class of the app with the dialog manually.

/\* ----------------------------- \*/

/\* ACCESSIBILITY \*/

**Tip**

ARIA is short for **Accessible Rich Internet Applications**. It is a set of attributes that enable us to make apps more accessible by assigning semantic characteristics to certain elements. For more information, see [Accessible Rich Internet Applications (ARIA) – Part 1: Introduction[](http://help.sap.com/disclaimer?site=https://blogs.sap.com/2015/06/01/accessible-rich-internet-applications-aria-part-1-introduction/)](http://help.sap.com/disclaimer?site=https://blogs.sap.com/2015/06/01/accessible-rich-internet-applications-aria-part-1-introduction/).

We child our content to a parent Panel, with `accessibleRole=true` attribute