**<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>

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