SAP UI5 - 4

Coding Basics

It's Basic HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>SAPUI5 Walkthrough</title>
  </head>
  <body>
    <div>Hello World</div>
  </body>
</html>
```

Make it a UI5 Bootstrap

This script tag makes it an SAP Bootstrap.

```
        <script
                id="sap-ui-bootstrap"
                src="https://sdk.openui5.org/resources/sap-ui-core.js"
                data-sap-ui-theme="sap_belize"
                data-sap-ui-libs="sap.m"
                data-sap-ui-compatVersion="edge"
                data-sap-ui-async="true"
                data-sap-ui-onInit="module:sap/ui/demo/walkthrough/index"
                data-sap-ui-resourceroots='{
                        "sap.ui.demo.walkthrough": "./"
                }'>
        </script>
```

For every filename.html, we are going to have an filename.js

```
  data-sap-ui-onInit="module:sap/ui/demo/walkthrough/index"
  data-sap-ui-resourceroots='{
  "sap.ui.demo.walkthrough": "./"
}'
```

Namespace is current folder and the javascript file to pick is index in the same

folder.
The index.js file is based around sap.ui.define.

```
sap.ui.define([
    'require',
    'dependency'
], function(require, factory) {
    'use strict';
    alert("UI5 is ready")
});
```

It contains two parts - dependencies and a function to execute. All dependencies are referenced to in the function. All those dependencies should exist.
Conventions:

- Use `sap.ui.define` for controllers and all other JavaScript modules to define a global namespace. With the namespace, the object can be addressed throughout the application.
- Use `sap.ui.require` for asynchronously 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).

Refer the HTML Control in JS

index.html is given an id tag.
`<body class="sapUiBody" id="content"></body>`
This ID will be referred to in JS by adding
`.placeAt("content");`
to the function.

```
sap.ui.define(["sap/m/Text"], function (Text) {
  "use strict";

  new Text({
    text: "Hello World",
  }).placeAt("content");
});
```

Notice, the dependency sap/m/text. This is referred to in the function and is placed at content section in index.html
All controls of SAPUI5 have a fixed set of properties, aggregations, and associations for configuration. API Reference - Demo Kit - SAPUI5 SDK (ondemand.com)

[Step 4: XML Views - Documentation - Demo Kit - SAPUI5 SDK (ondemand.com)]

Split into Views

Rather than dumping all code into a single index.js, it makes more sense to split the code into independent views. Two files are created for this - `viewname.view.xml` and `viewname.controller.js`
The view is an MVC view

```
<mvc:View xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Text text="Hello World"/>
</mvc:View>
```

while the controller file is not mandatory. You can use the main index.js itself - the file is identical to old index.js with the additional activity of creating an XMLView and the view is passed as a function.
The `"use strict";` literal expression was introduced by ECMAScript 5. It tells the browser to execute the code in a so called “strict mode”. The strict mode helps to detect potential coding issues at an early state at development time, that means, for example, it makes sure that variables are declared before they are used. Thus, it helps to prevent common JavaScript pitfalls and it’s therefore a good practice to use strict mode.

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

  XMLView.create({
    viewName: "sap.ui.demo.walkthrough.view.App",
  }).then(function (oView) {
    oView.placeAt("content");
  });
});
```

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

        XMLView.create({
                viewName: "sap.ui.demo.walkthrough.view.App"
        }).then(function (oView) {
                oView.placeAt("content");
```

```
        });
});
```

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 use the last part of the SAP namespace as alias (for example, `mvc` for `sap.ui.core.mvc`)

The View has it's Own Controller

In case you are going to use a controller for the view separately(which is the standard practice)

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Button
        text="Say Hello"
        press=".onShowHello"/>
</mvc:View>
```

You have a controllerName which links it to the view and some kind of action(say) where the view is looking for in the controller.
There are two aspects of it. Controller is passed as a dependency and extend is used to pass the functionality.

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

```
sap.ui.define(["sap/ui/core/mvc/Controller"], function (Controller) {
  "use strict";
  return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
    onShowHello: function () {
      // show a native JavaScript alert
      alert("Hello World");
    },
  });
});
```

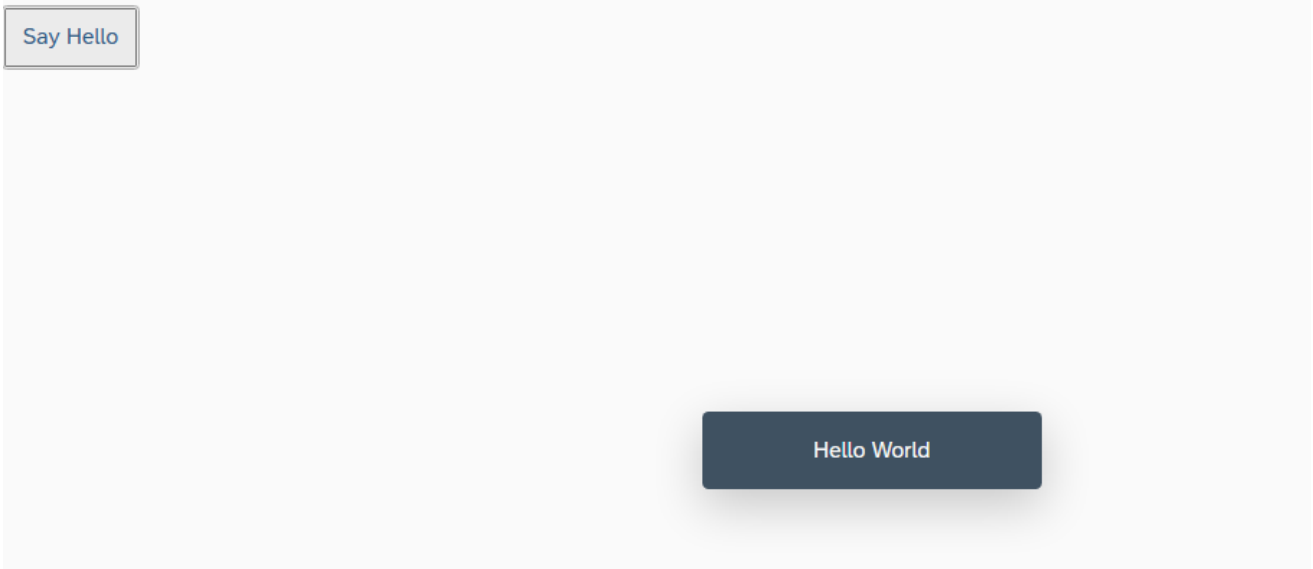
Conventions

- 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`

Adding Custom Modules

The dependencies defined can be either custom modules or standard system entities. These standard system entities are called as modules. THey are used to extend the functionality and enrich the experience. For example, in this case, MessageToast is one such module. It is used to display a system message in UI5.

```
sap.ui.define(
  ["sap/ui/core/mvc/Controller", "sap/m/MessageToast"],
  function (Controller, MessageToast) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
      onShowHello: function () {
        // show a native JavaScript alert
        alert("Hello World");
        MessageToast.show("Hello World");
      },
    });
  }
);
```



We extend the array of required modules with the fully qualified path to `sap.m.MessageToast`. Once both modules, `Controller` and `MessageToast`, are loaded, the callback function is called and we can make use of both objects by accessing the parameters passed on to the function.

This Asynchronous Module Definition (AMD) syntax allows to clearly separate the module loading from the code execution and greatly improves the performance of the application. The browser can decide when and how the resources are loaded prior to code execution.

JSON Model

OnInit is invoked whenever a new controller is activated.

```
function (Controller, MessageToast, JSONModel) {
  "use strict";
  return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
    onInit: function () {
      // set data model on view
      var oData = {
        recipient: {
          name: "World",
        },
      };
      var oModel = new JSONModel(oData);
      this.getView().setModel(oModel);
    },
    onShowHello: function () {
      MessageToast.show("Hello World");
    },
  });
}
```

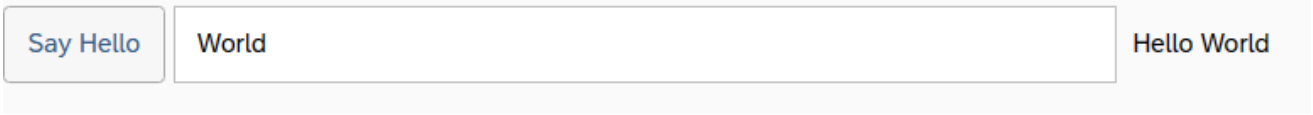
setModel is used to pass the model to the view.

```
    var oData = {
      recipient: {
        name: "World",
      },
    };
```

The above JSON is referenced as

```
<Input value="{/recipient/name}" description="Hello {/recipient/name}"
```

curly brackets do the databinding to the object.
Complex Binding syntax should be enabled globally by setting the bootstrap parameter `data-sap-ui-compatVersion` to `edge`. If this setting is omitted, then only standard binding syntax is allowed, meaning "Hello `{/recipient/name}`" would not work anymore while "`{/recipient/name}`" would work just fine.



Conventions

- Use Hungarian notation for variable names.

Internationalization

Internationalization is used to separate the core website with language specific data. This is done through the Resource Model i18n.
Here, all parameters are loaded into a special i18n.properties. This contains a set of key value pairs.

```
showHelloButtonText=Say Hello
helloMsg=Hello {0}
```

For the languages to be used, all you need to do is to have a language specific file with the naming convention `i18n_<language name>.properties`. Eg: `i18n_de.properties`. This is invoked by the hook `?sap-language=DE`
As like in other cases, this model is also instantiated. Bundlename gives the actual properties file.

```
  var i18nModel = new ResourceModel({
    bundleName: "sap.ui.demo.walkthrough.i18n.i18n",
  });
  this.getView().setModel(i18nModel, "i18n");
},
```

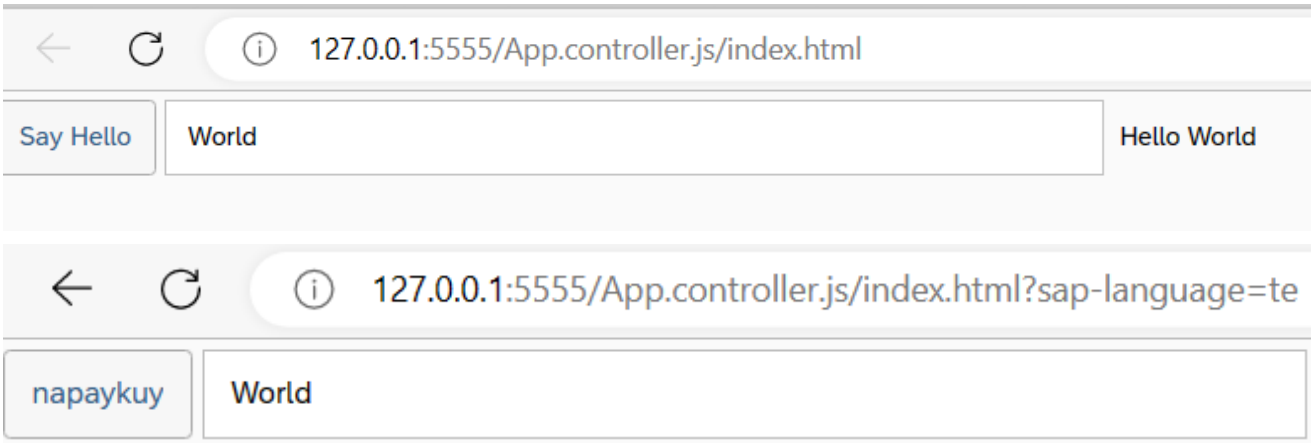
This is referenced slightly differently in the view:

```
<Button text="{i18n>showHelloButtonText}" press=".onShowHello"/>
```

In controller.js, it is referred to as like any other mode - first, the resourceBundle is read and that is passed as input to getModel.

```
  onShowHello: function () {
    // read msg from i18n model
    var oBundle = this.getView().getModel("i18n").getResourceBundle();
    var sRecipient = this.getView()
      .getModel()
      .getProperty("/recipient/name");
    var sMsg = oBundle.getText("helloMsg", [sRecipient]);
    // show message
    MessageToast.show(sMsg);
  },
```

Internationalization:



Conventions

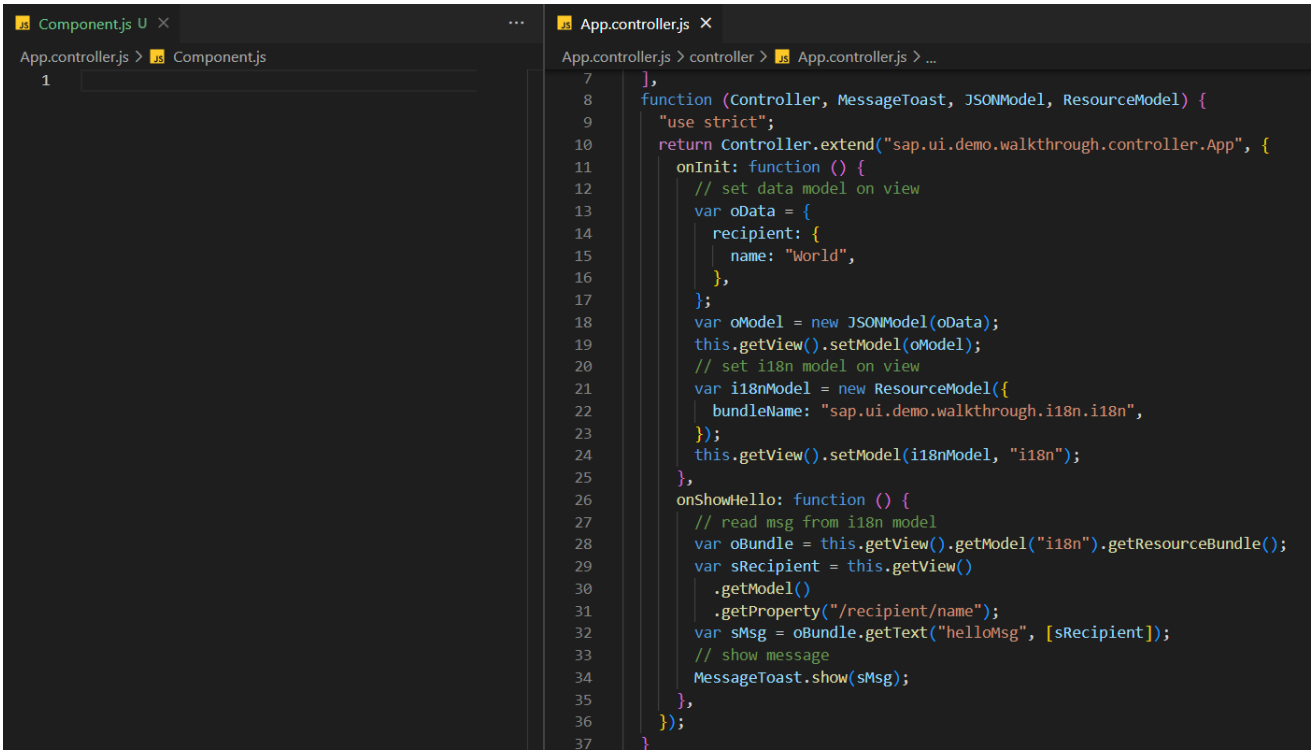
- The resource model for internationalization is called the `i18n` model.
- The default filename is `i18n.properties`.
- Resource bundle keys are written in (lower) camelCase.
- Resource bundle values can contain parameters like `{0}`, `{1}`, `{2}`, ...
- Never concatenate strings that are translated, always use placeholders.
- Use Unicode escape sequences for special characters.

https://github.com/SAP-samples/teched2021-DEV160

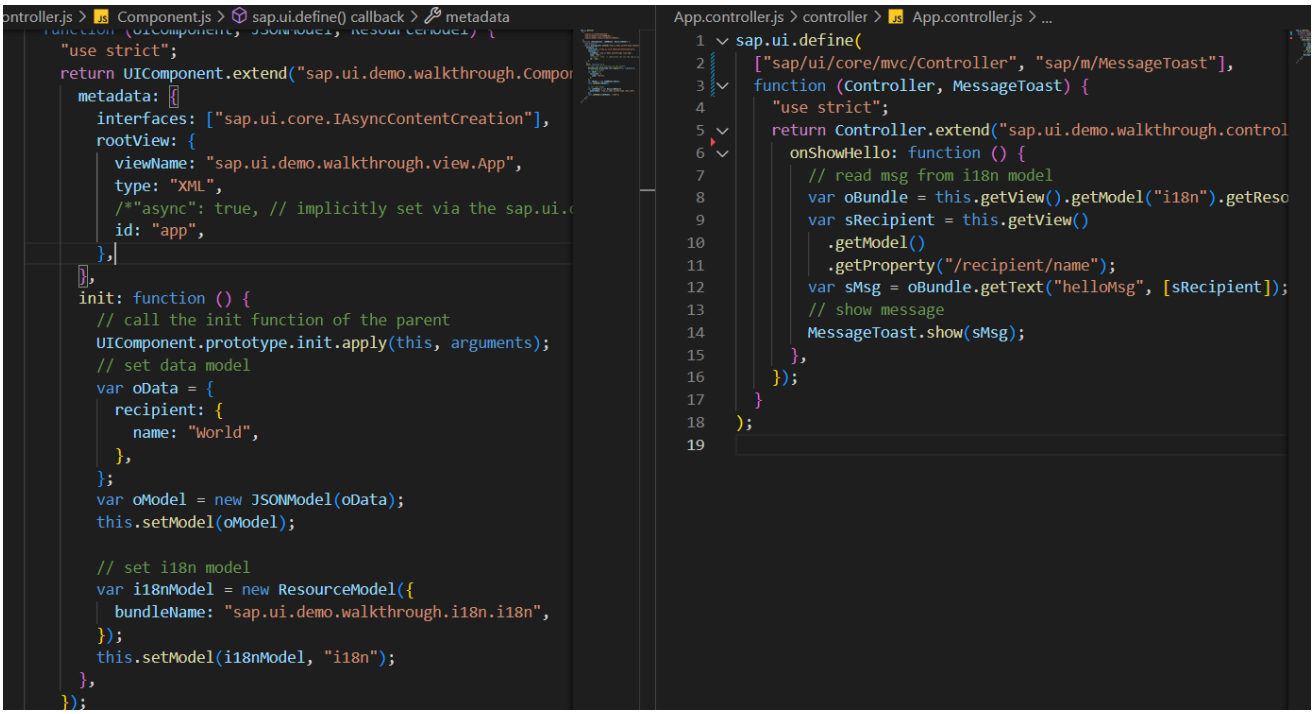
Components

Component dissociates UI assets from index.html and place them in a reusable way

Before



After. The two front end entities - JSONModel and i18nModel are moved to Component Model. The whole onInit moves into Component and under init.



This is how a basic component file looks like.

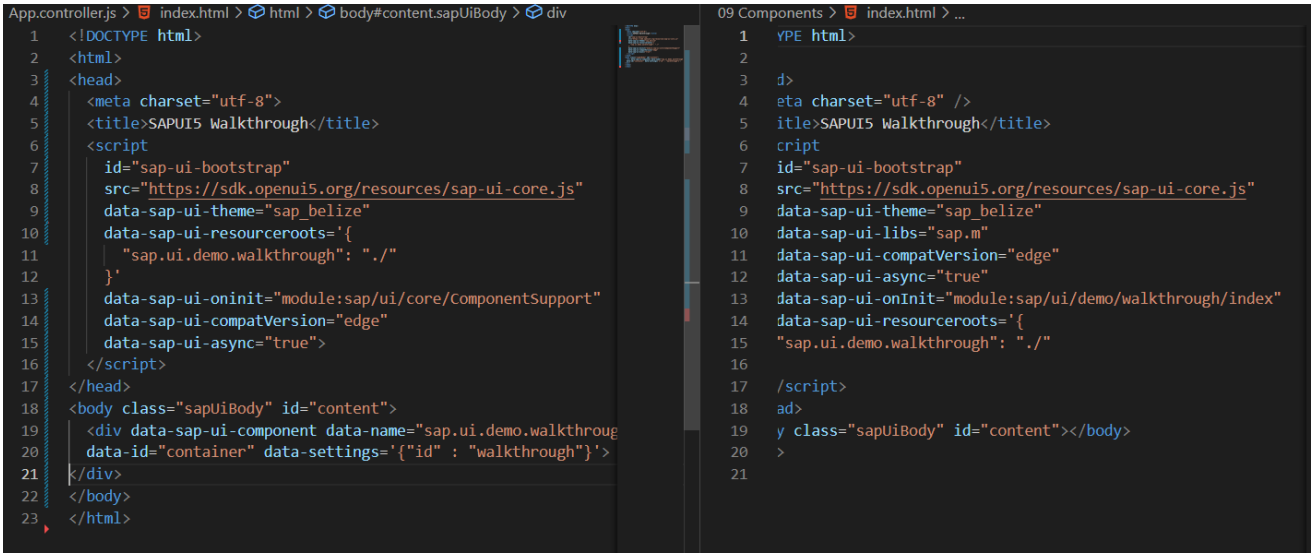
```
sap.ui.define(["sap/ui/core/UIComponent"], function (UIComponent) {
  "use strict";
  return UIComponent.extend("", {
    init: function () {
      // call the init function of the parent
      UIComponent.prototype.init.apply(this, arguments);
    },
  });
});
```

rootView is mandatory.
The flow is index.html → index.js → Component.js → view/App.xml → controller/App.controller.js
Conventions

- The component is named `Component.js`.
- Together with all UI assets of the app, the component is located in the `webapp` folder.
- The `index.html` file is located in the `webapp` folder if it is used productively.

Descriptors

manifest.json acts as the descriptor - the common location to store all application specific configurations.
The SAP Fiori launchpad acts as an application container and instantiates the app without having a local HTML file for the bootstrap. Instead, the descriptor file will be 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 language properties, supported devices, and more. And we can also use the descriptor file to load additional resources and instantiate models like our `i18n` resource bundle.
While the manifest.json will have the actual data, index.html will refer to the data in the manifest and not to any specific view.



In the older case, oninit invokes index.js directly. But here, we get component support.

There is an additional div tag in body which displays a container.

Notice that the missing dependency sap.m is present in manifest.json

Because of the div used, index.js is no more needed.

App.controller.js > index.js > ...

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

        new ComponentContainer({
            name: "sap.ui.demo.walkthrough",
            settings: {
                id: "walkthrough",
            },
            async: true,
        }).placeAt("content");
    }
);

```

There are three important sections defined by namespaces in the `manifest.json` file:

- `sap.app`
  The `sap.app` namespace contains the following application-specific attributes:
  - `id` (mandatory): The namespace of our application component
    The ID must not exceed 70 characters. It must be unique and must correspond to the component ID/namespace.
  - `type`: Defines what we want to configure, here: an application
  - `i18n`: Defines the path to the resource bundle file
  - `title`: Title of the application in handlebars syntax referenced from the app's resource bundle
  - `description`: Short description text what the application does in handlebars syntax referenced from the app's resource bundle
  - `applicationVersion`: The version of the application to be able to easily update the application later on
- `sap.ui`
  The `sap.ui namespace` contributes the following UI-specific attributes:
  - `technology`: This value specifies the UI technology; in our case we use SAPUI5

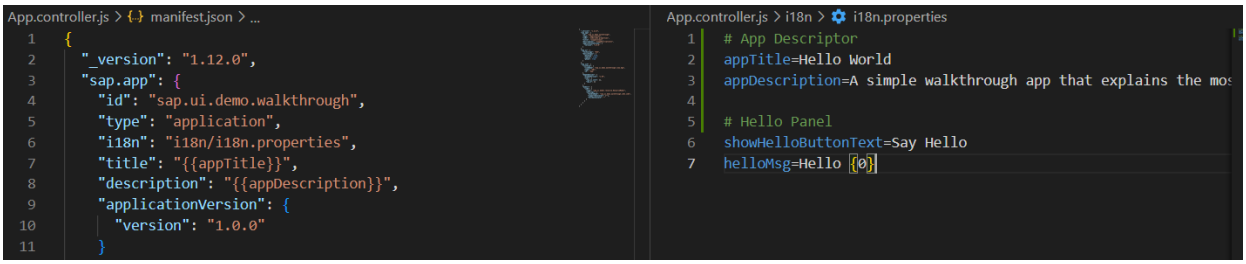
- `deviceTypes`: Tells what devices are supported by the app: desktop, tablet, phone (all true by default)

- `sap.ui5`

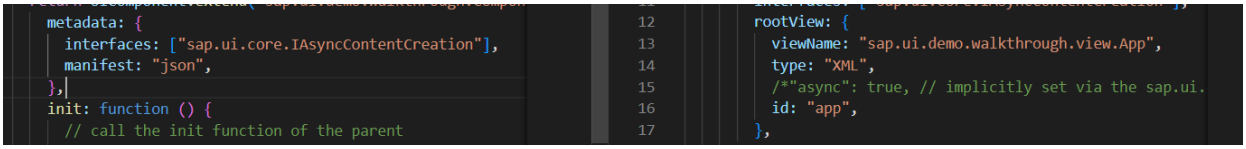
  The `sap.ui5` namespace adds SAPUI5-specific configuration parameters that are automatically processed by SAPUI5. The most important parameters are:
  - `rootView`: If you specify this parameter, the component will automatically instantiate the view and use it as the root for this component
  - `dependencies`: Here we declare the UI libraries used in the application
  - `models`: In this section of the descriptor we can define models that will be automatically instantiated by SAPUI5 when the app starts. Here we can now define the local resource bundle. We define the name of the model "i18n" as key and specify the bundle file by namespace. As in the previous steps, the file with our translated texts is stored in the `i18n` folder and named `i18n.properties`. We simply prefix the path to the file with the namespace of our app. The manual instantiation in the app component's init method will be removed later in this step.

  The `supportedLocales` and `fallbackLocale` properties are set to empty strings, as in this tutorial our demo app uses only one `i18n.properties` file for simplicity, and we'd like to prevent the browser from trying to load additional `i18n_*.properties` files based on your browser settings and your locale.

  apptitle and appdescription ideally comes from i18n



  Root View in Component.js is replaced



  Even resource model(i18n) is not needed.

  In the component's `metadata` section, we now replace the `rootView` property with the property key `manifest` and the value `json`. This defines a reference to the descriptor that will be loaded and parsed automatically when the component is instantiated. We can now completely remove the lines of code containing the model instantiation for our resource bundle. It is done automatically by SAPUI5 with the help of the configuration entries in the descriptor. We can also remove the dependency

to `sap/ui/model/resource/ResourceModel` and the corresponding formal parameter `ResourceModel` because we will not use this inside our anonymous callback function.

Conventions

- 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.

```
## Each View has it's Own Panel
```

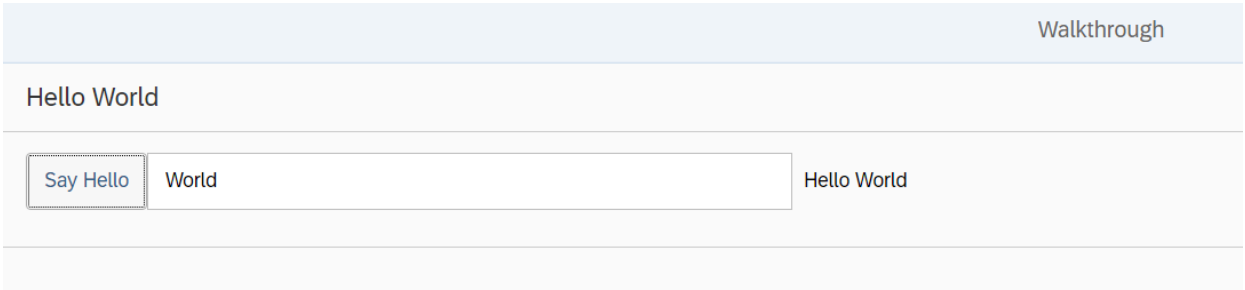
You can put it in a Panel and source the comments from i18n directly.

App.controller.js > view > App.view.xml > mvc:View > App > pages > Page > content > Panel > content

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc" displayBlock="true">
    <App>
        <pages>
            <Page title="{i18n>homePageTitle}">
                <content>
                    <Panel headerText="{i18n>helloPanelTitle}">
                        <content>
                            <Button text="{i18n>showHelloButtonText}" press=".onShowHello"/>
                            <Input value="{/recipient/name}" description="Hello {/recipient/name}"
                        </content>
                    </Panel>
                </content>
            </Page>
        </pages>
    </App>
</mvc:View>
```

App.controller.js > i18n > i18n.properties

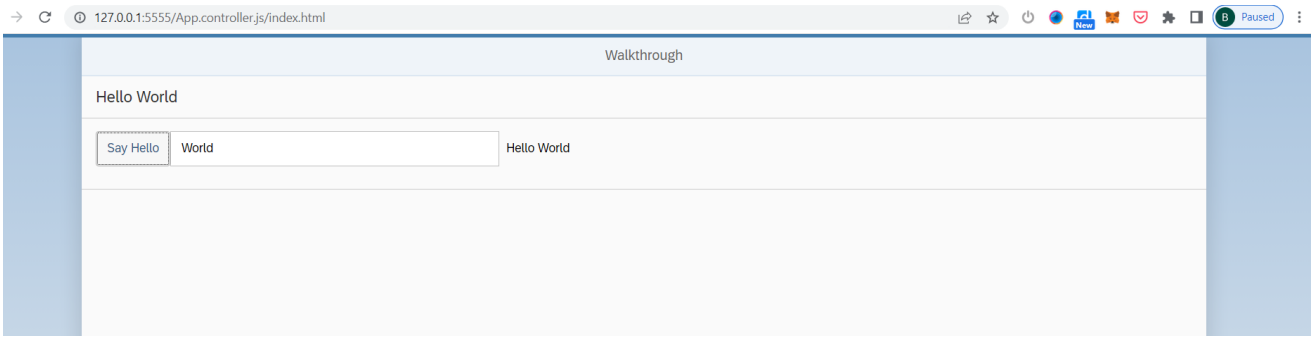
```
# App Descriptor
appTitle=Hello World
appDescription=A simple walkthrough app that explains the most important concepts of SAPUI5

# Hello Panel
showHelloButtonText=Say Hello
helloMsg=Hello {0}
homePageTitle=Walkthrough
helloPanelTitle=Hello World
```



Shell as Container

Now, wrap this in a shell. It's a very good way to enforce Responsive Design
The shell takes care of visual adaptation of the application to the device's screen size by introducing a so-called letterbox on desktop screens.

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc" displayBlock="true">
    <Shell >
        <App>
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <Panel headerText="{i18n>helloPanelTitle}">
                            <content>
                                <Button text="{i18n>showHelloButtonText}" press=".onShow
                                <Input value="{/recipient/name}" description="Hello {/re
                            </content>
                        </Panel>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```



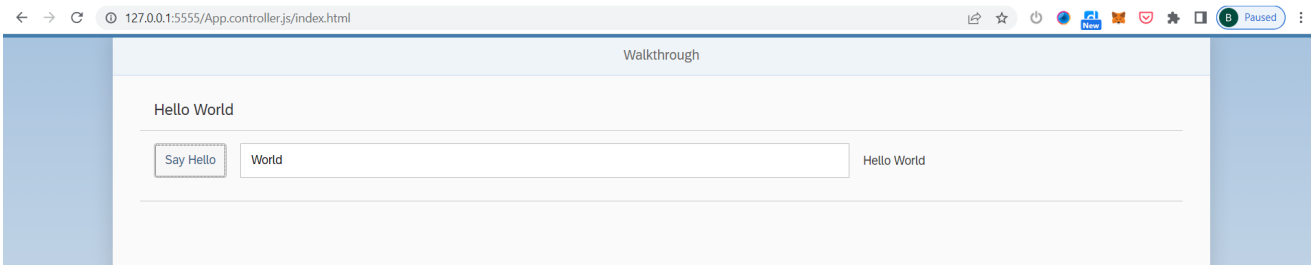
There are further options to customize the shell, like setting a custom background image or color and setting a custom logo.
[sap.m.Shell - API Reference - Demo Kit - SAPUI5 SDK (ondemand.com)]

Margins and Paddings

UI5 has it's own margin and padding classes. There is no need to go for CSS per se.

App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page > content > Panel

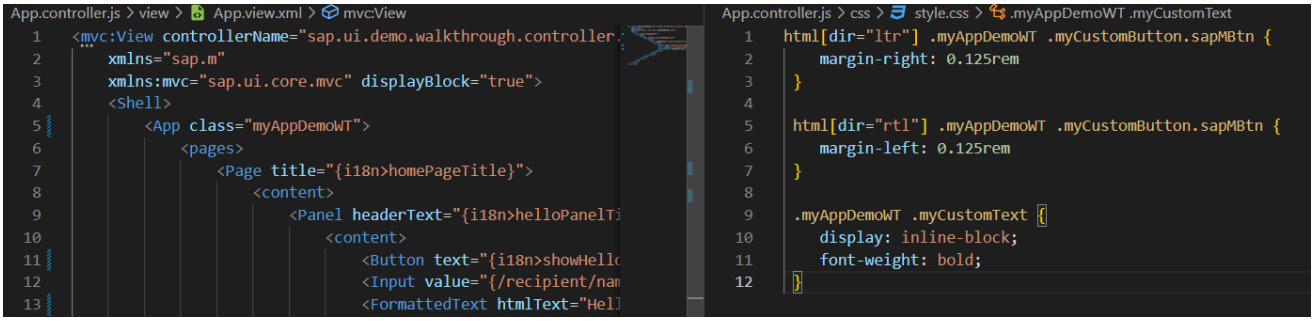
```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc" displayBlock="true">
    <Shell>
        <App>
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <Panel headerText="{i18n>helloPanelTitle}" class="sapUiResponsiveMargin" width="auto">
                            <content>
                                <Button text="{i18n>showHelloButtonText}" press=".onShowHello" class="sapUiSmallMarginEnd"/>
                                <Input value="{/recipient/name}" valueLiveUpdate="true" width="60%"/>
                                <Text text="Hello {/recipient/name}" class="sapUiSmallMargin"/>
                            </content>
                        </Panel>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

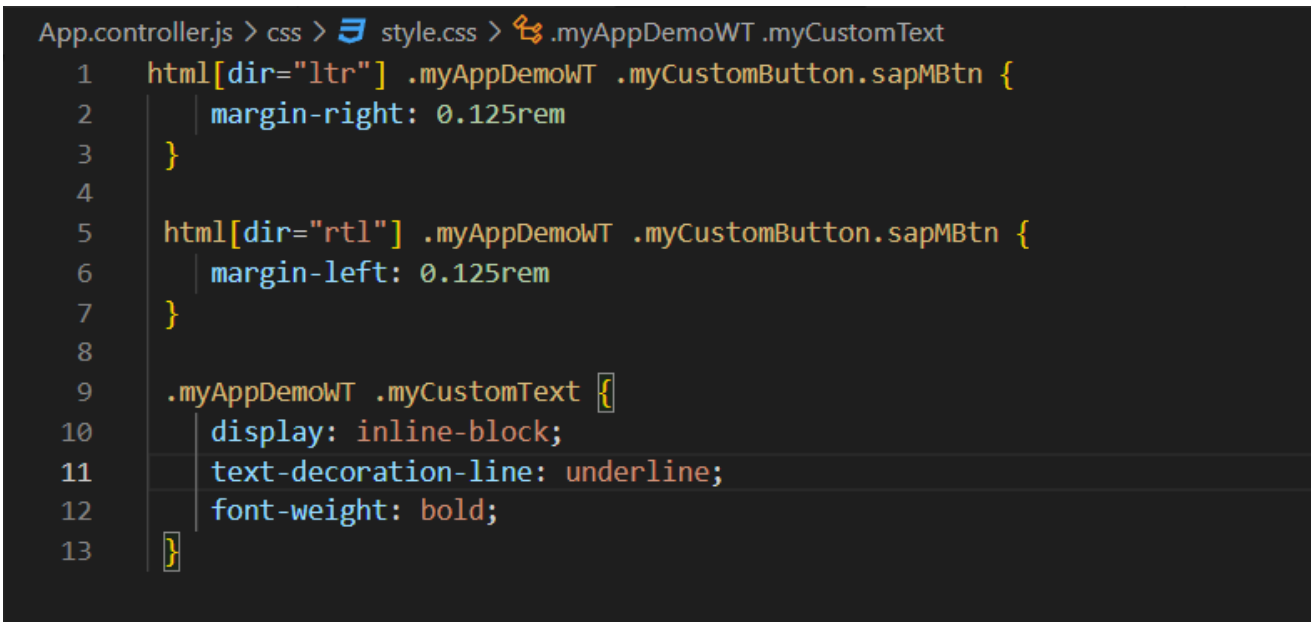


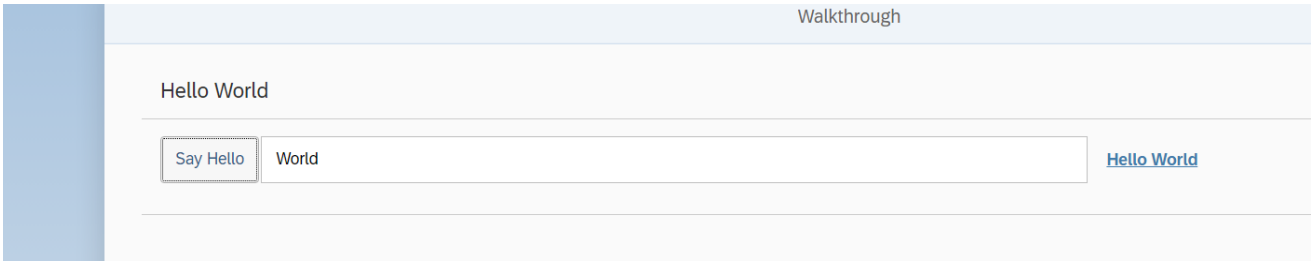
Custom CSS

Using standard classes doesn't mean you can't use custom CSS!!

```
"models": {
  "i18n": {
    "type": "sap.ui.model.resource.ResourceModel",
    "settings": {
      "bundleName": "sap.ui.demo.walkthrough.i18n.i18n",
      "supportedLocales": [""],
      "fallbackLocale": ""
    }
  }
},
"resources": {
  "css": [
    {
      "uri": "css/style.css"
    }
  ]
}
}
```







[List of Supported CSS Classes - Documentation - Demo Kit - SAPUI5 SDK (ondemand.com)]

Nested Views

Well, a view can also call a view. Why does that matter? Better refactoring.

App.controller.js just contains a reference to the new controller and the code logic is shifted there

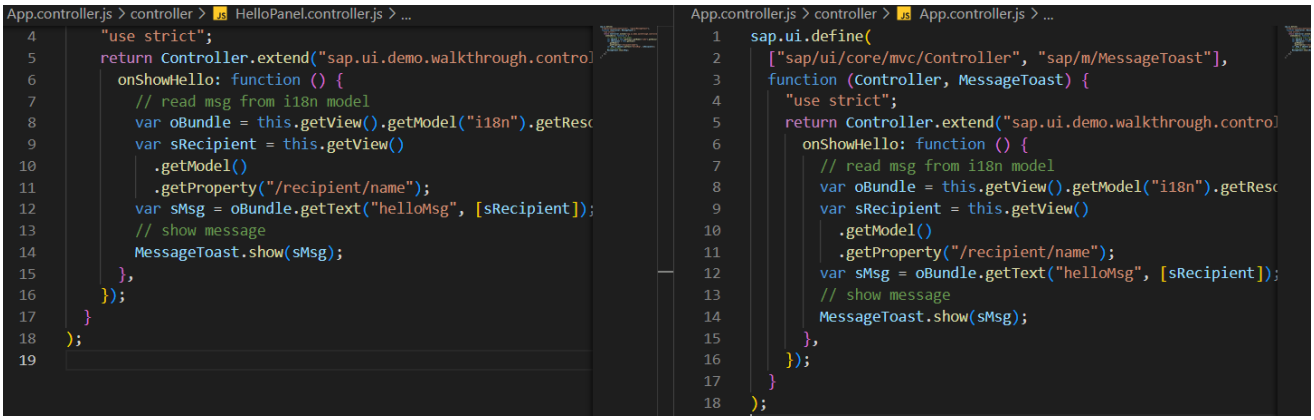
App.controller.js > controller > App.controller.js > ...

```
sap.ui.define(["sap/ui/core/mvc/Controller"], function (Controller) {
  "use strict";
  return Controller.extend("sap.ui.demo.walkthrough.controller.App", {});
});
```

App.controller.js > controller > HelloPanel.controller.js > ...

```
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.HelloPanel", {
      onShowHello: function () {
        // read msg from i18n model
        var oBundle = this.getView().getModel("i18n").getResourceBundle();
        var sRecipient = this.getView()
          .getModel()
          .getProperty("/recipient/name");
        var sMsg = oBundle.getText("helloMsg", [sRecipient]);
        // show message
        MessageToast.show(sMsg);
      },
    });
  }
);
```

App.controller.js → before



```
"use strict";
return Controller.extend("sap.ui.demo.walkthrough.contro
  onShowHello: function () {
    // read msg from i18n model
    var oBundle = this.getView().getModel("i18n").getResc
    var sRecipient = this.getView()
      .getModel()
      .getProperty("/recipient/name");
    var sMsg = oBundle.getText("helloMsg", [sRecipient]);
    // show message
    MessageToast.show(sMsg);
  },
});
}
);
```

```
sap.ui.define(
  ["sap/ui/core/mvc/Controller", "sap/m/MessageToast"],
  function (Controller, MessageToast) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.contro
      onShowHello: function () {
        // read msg from i18n model
        var oBundle = this.getView().getModel("i18n").getResc
        var sRecipient = this.getView()
          .getModel()
          .getProperty("/recipient/name");
        var sMsg = oBundle.getText("helloMsg", [sRecipient]);
        // show message
        MessageToast.show(sMsg);
      },
    });
  }
);
```

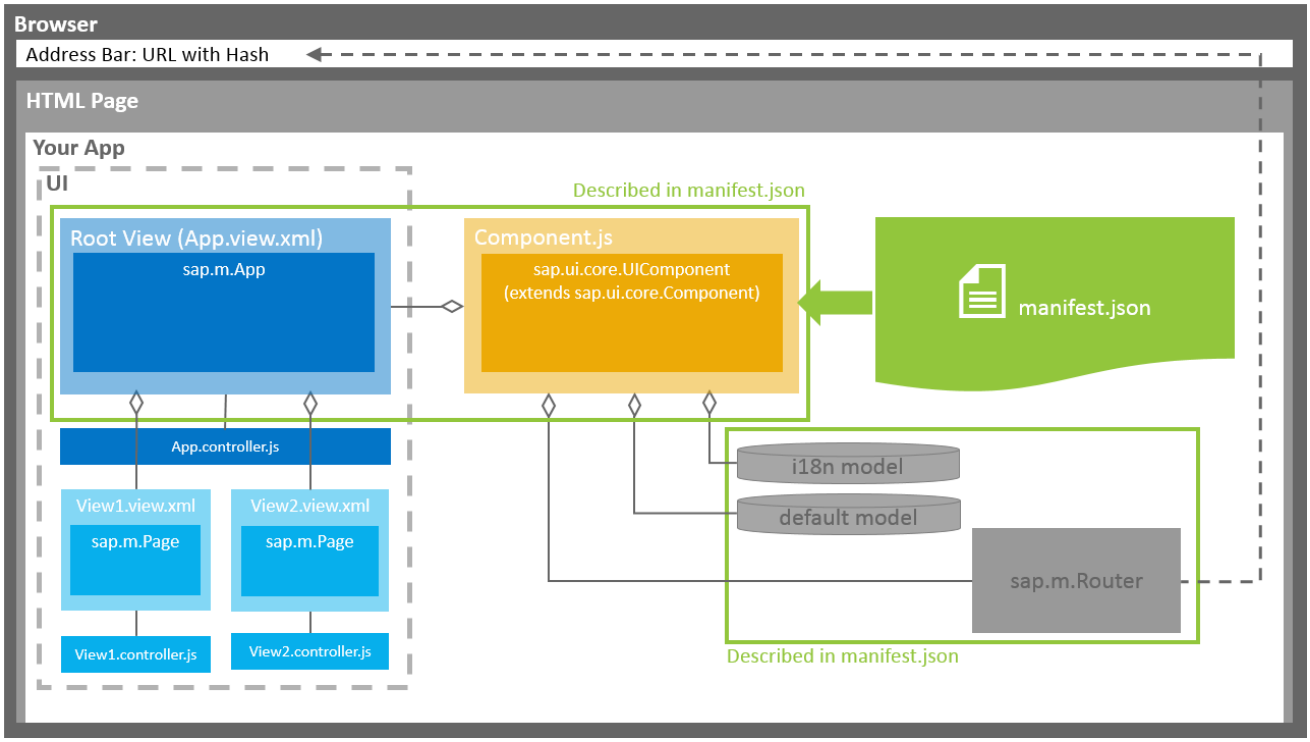
After

App.controller.js > controller > HelloPanel.controller.js > ...

```
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.contro
      onShowHello: function () {
        // read msg from i18n model
        var oBundle = this.getView().getModel("i18n").getResc
        var sRecipient = this.getView()
          .getModel()
          .getProperty("/recipient/name");
        var sMsg = oBundle.getText("helloMsg", [sRecipient]);
        // show message
        MessageToast.show(sMsg);
      },
    });
  }
);
```

App.controller.js > controller > App.controller.js > ...

```
sap.ui.define(["sap/ui/core/mvc/Controller"], function (Contr
  "use strict";
  return Controller.extend("sap.ui.demo.walkthrough.controlle
});
```



Fragments and Dialogs

Fragments are small UI components which can be reused. They don't have any controls associated and can be used across multiple views. Dialogs on the other hand, are simply display popups.

App.controller.js > view > HelloPanel.view.xml > mvc:View > Panel > content > Button

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{i18n>helloPanelTitle}" class="sapUiResponsiveMargin" width="auto">
        <content>
            <Button id="helloDialogButton" text="{i18n>openDialogButtonText}" press=".onOpenDialog" class="sapUiSmallMarginEnd"/>
            <Button text="{i18n>showHelloButtonText}" press=".onShowHello" class="myCustomButton"/>
            <Input value="{/recipient/name}" valueLiveUpdate="true" width="60%"/>
            <FormattedText htmlText="Hello {/recipient/name}" class="sapUiSmallMargin sapThemeHighlight-asColor myCustomText"/>
        </content>
    </Panel>
</mvc:View>
```

HelloDialog.controller.js. loadFragment is used to select the dialog and is opened with the keyword open. The logic goes like, if dialog is not open, open the dialog. (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.)

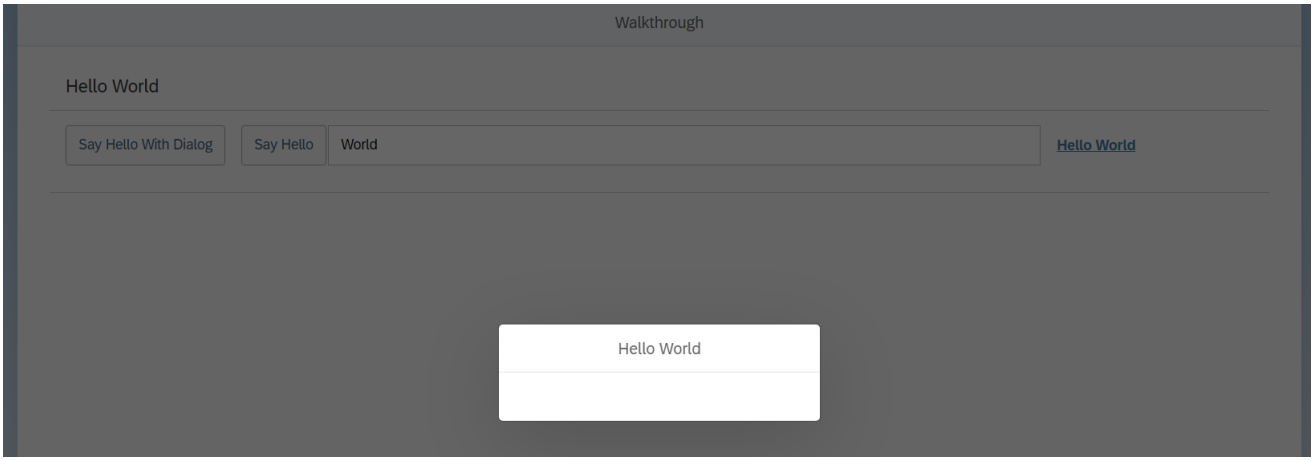
```
    onOpenDialog: function () {
      // create dialog lazily
      if (!this.pDialog) {
        this.pDialog = this.loadFragment({
          name: "sap.ui.demo.walkthrough.view.HelloDialog",
        });
      }
      this.pDialog.then(function (oDialog) {
        oDialog.open();
      });
    },
  });
```

Notice, this is just a piece of raw code. No controller, nothing.

App.controller.js > view > HelloDialog.fragment.xml > core:FragmentDefinition

```
1  <core:FragmentDefinition xmlns="sap.m"
2      xmlns:core="sap.ui.core">
3      <Dialog id="helloDialog" title="Hello {/recipient/name}">
4      </Dialog>
5  </core:FragmentDefinition>
```

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`



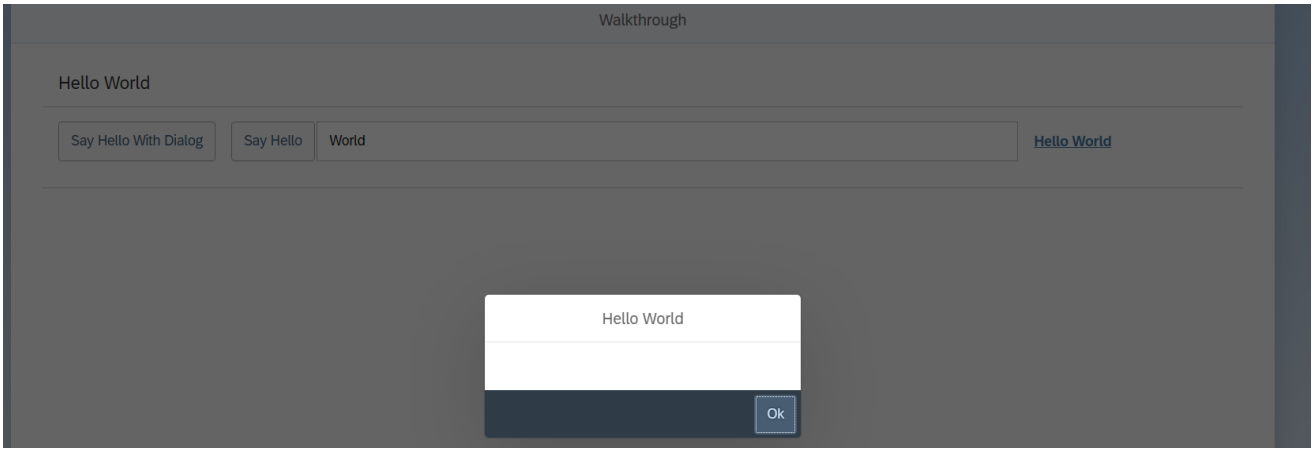
Fragment Callbacks

Now, we have created a dialog. You need to have a way to close it or perform other actions on it!!
It's straightforward enough -
Begin Button in the fragment does the trick.

```
<core:FragmentDefinition xmlns="sap.m"
    xmlns:core="sap.ui.core">
    <Dialog id="helloDialog"
      title ="Hello {/recipient/name}">
        <beginButton>
            <Button text="{i18n>dialogCloseButtonText}" press=".onCloseDialog"/>
        </beginButton>
    </Dialog>
</core:FragmentDefinition>
```

This is done against a closeDialog in the controller by simply invoking a close.

```
    onCloseDialog: function () {
      // note: We don't need to chain to the pDialog promise, since this event-handler
      // is only called from within the loaded dialog itself.
      this.byId("helloDialog").close();
    },
```



Placing buttons in both of these aggregations makes sure that the `beginButton` is placed before the `endButton` on the UI. What `before` means, however, depends on the text direction of the current language. We therefore use the terms `begin` and `end` as a synonym to “left” and “right". In languages with left-to-right direction, the `beginButton` will be rendered left, the `endButton` on the right side of the dialog footer; in right-to-left mode for specific languages the order is switched.

Icons on Dialogs

Now, let's add an icon to the dialog we created. You just need to add an icon

```
App.controller.js > view > HelloPanel.view.xml > mvc:View > Panel > content > Button
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{i18n>helloPanelTitle}" class="sapUiResponsiveMargin"
        <content>
            <Button id="helloDialogButton" icon="sap-icon://world" text="{i
            <Button text="{i18n>showHelloButtonText}" press=".onShowHello" c
            <Input value="{/recipient/name}" valueLiveUpdate="true" width="6
            <FormattedText htmlText="Hello {/recipient/name}" class="sapUiSm
        </content>
    </Panel>
</mvc:View>
```

and define it as content in the dialog fragment.

```
title ="Hello {/recipient/name}">
  <content>
    <core:Icon src="sap-icon://hello-world" size="8rem" class="sapUiMediumMargin"/>
  </content>
  <beginButton>
    <Button text="{i18n>dialogCloseButtonText}" press=".onCloseDialog"/>
  </beginButton>
```

The `sap-icon://` protocol is indicating that an icon from the icon font should be loaded. The identifier `world` is the readable name of the icon in the icon font. To call any icon, use its name as listed in the Icon Explorer in `sap-icon://<iconname>`.
Icon Explorer (ondemand.com)
Always use icon fonts rather than images wherever possible, as they are scalable without quality loss (vector graphics) and do not need to be loaded separately.

Aggregation Binding - Multiple Views in a single page

Let's say the new view is a JSON table. Add it to manifest.json and give the actual file as reference.

```
    "invoice": {
      "type": "sap.ui.model.json.JSONModel",
      "uri": "Invoices.json"
    }
  },
  "resources": {
```

Add a second view

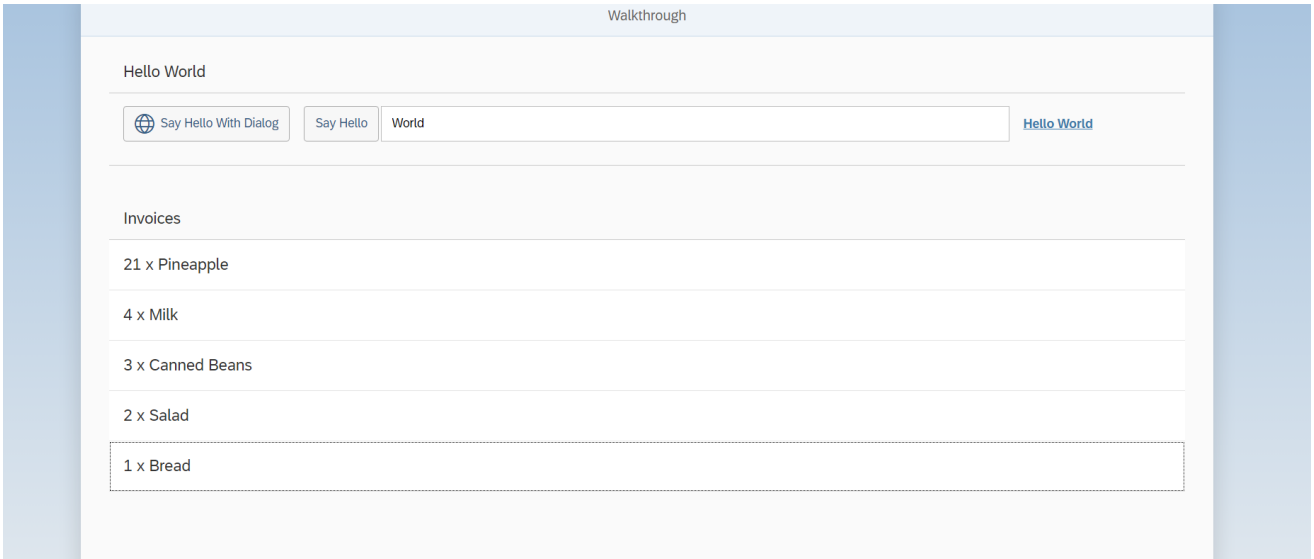
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page > content > mvc:XMLView

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc" displayBlock="true">
    <Shell>
        <App class="myAppDemoWT">
            <pages>
                <Page title="{i18n>homePageTitle}">
                    <content>
                        <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
                        <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.InvoiceList"/>
                    </content>
                </Page>
            </pages>
        </App>
    </Shell>
</mvc:View>
```

Rest all is drab work -

App.controller.js > view > InvoiceList.view.xml > mvc:View

```
<mvc:View xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <List headerText="{i18n>invoiceListTitle}" class="sapUiResponsiveMargin" width="auto" items="{invoice>/Invoices}">
        <items>
            <ObjectListItem title="{invoice>Quantity} x {invoice>ProductName}"/>
        </items>
    </List>
</mvc:View>
```



Datatypes

Let's make the table more legible.

The JSON gives the values as numbers but you need a currency against it!

```
{
  "ProductName": "Salad",
  "Quantity": 2,
  "ExtendedPrice": 8.8,
  "ShipperName": "ACME",
  "ShippedDate": "2015-04-12T00:00:00",
  "Status": "C"
},
```

sap.ui.model.type.Currency gives the currency under use.

App.controller.js > view > InvoiceList.view.xml > mvc:View

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <List headerText="{i18n>invoiceListTitle}" class="sapUiResponsiveMargin" width="auto" items="{invoice>/Invoices}">
        <items>
            <ObjectListItem title="{invoice>Quantity} x {invoice>ProductName}" number="{
            parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/currency'}],
            type: 'sap.ui.model.type.Currency',
            formatOptions: {
                showMeasure: false
            }
        }" numberUnit="{view>/currency}"/>
        </items>
    </List>
</mvc:View>
```

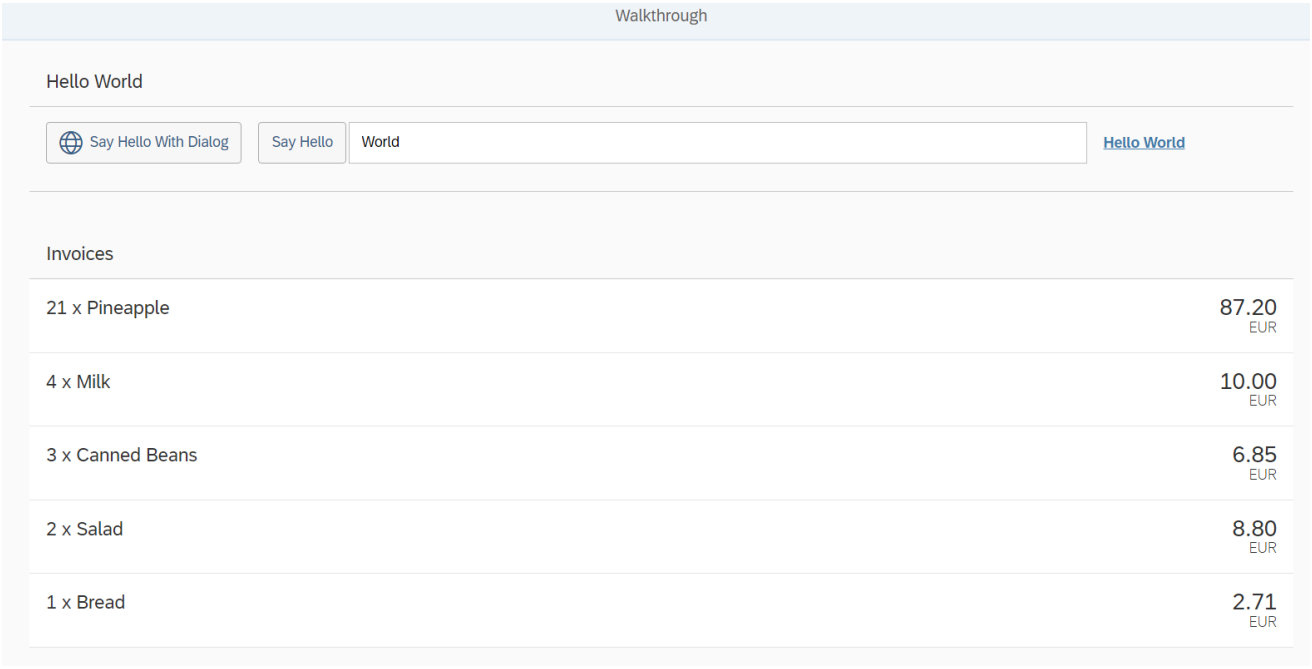
and give the currency in the controller

App.controller.js > controller > InvoiceList.controller.js > ...

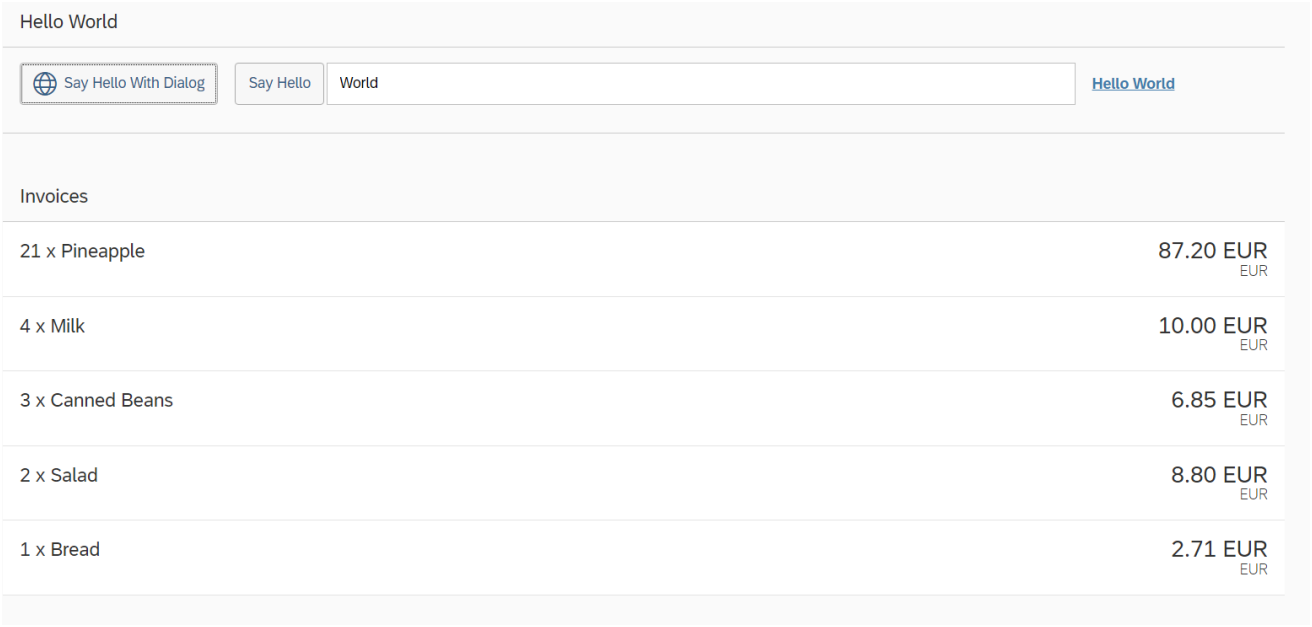
```
sap.ui.define(
  ["sap/ui/core/mvc/Controller", "sap/ui/model/json/JSONModel"],
  function (Controller, JSONModel) {
    "use strict";

    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {
      onInit: function () {
        var oViewModel = new JSONModel({
          currency: "EUR",
        });
        this.getView().setModel(oViewModel, "view");
      },
    });
  }
);
```

Now, we are just formatting the value as a currency, that's all. Whatever currency you give there will be visible.

we are using a special binding syntax for the `number` property of the `ObjectListItem`. This binding syntax makes use of so-called "Calculated Fields", which allows the binding of multiple properties from different models to a single property of a control. The properties bound from different models are called “parts”. In the example above, the property of the control is `number` and the bound properties (“parts”) retrieved from two different models are `invoice>ExtendedPrice` and `view>/currency`.



Additionally, we set the formatting option `showMeasure` to `false`. This hides the currency code in the property `number`, because it is passed on to the `ObjectListItem` control as a separate property `numberUnit`.
If you set it to true →

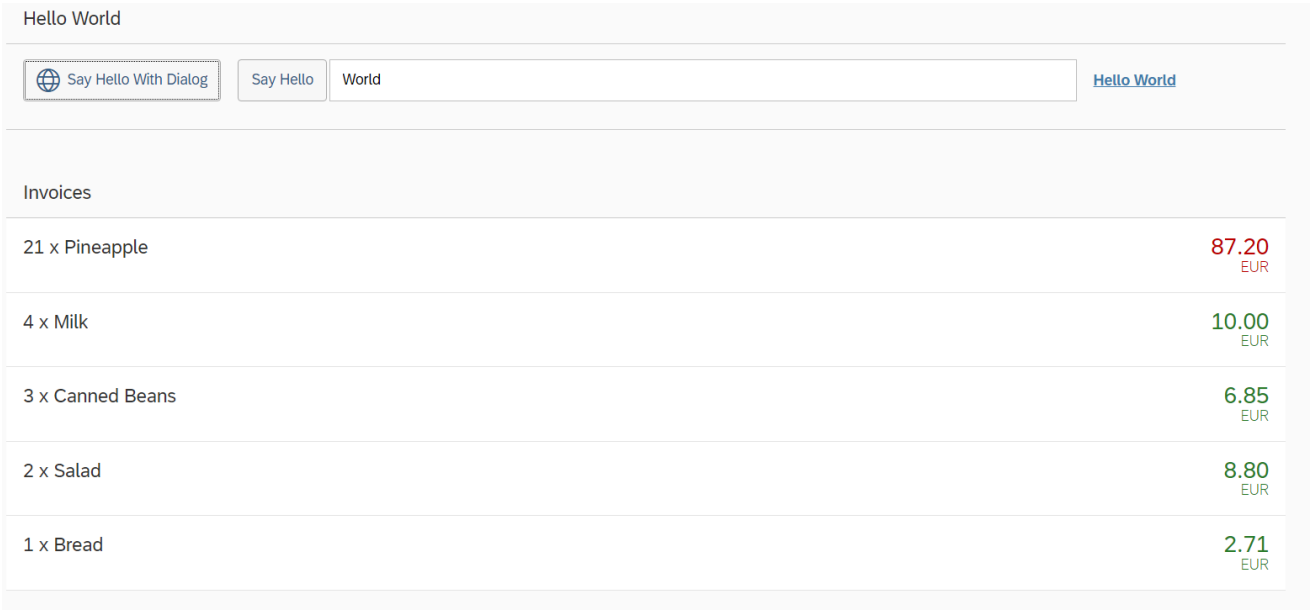


Expression Binding

Is it possible to put a condition on these numbers?

```
<List headerText="{i18n>invoiceListTitle}" class="sapUiResponsiveMargin" width="auto" items="{invoice>/Invoices}">
    <items>
        <ObjectListItem title="{invoice>Quantity} x {invoice>ProductName}" 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' }"/>
    </items>
```

numberState passes the condition on number.



A model binding inside an expression binding has to be escaped with the `$` sign as you can see in the code.

```
numberState="{= ${invoice>Quantity} > 3 ? 'Error' : 'Success' }"/>
```

The query can be used on any field as possible to change the State of number. - Only use expression binding for trivial calculations.

Custom Formatters

But, what's above is just basic. Now, if you want to make it a complex calculation?
Consider a basic switch function.

App.controller.js > model > formatter.js > ...

```
sap.ui.define([], function () {
    "use strict";
    return {
        statusText: function (sStatus) {
            var resourceBundle = this.getView().getModel("i18n").getResourceBundle();
            switch (sStatus) {
                case "A":
                    return resourceBundle.getText("invoiceStatusA");
                case "B":
                    return resourceBundle.getText("invoiceStatusB");
                case "C":
                    return resourceBundle.getText("invoiceStatusC");
                default:
                    return sStatus;
            }
        }
    };
});
```

Refer to the formatter in controller.js

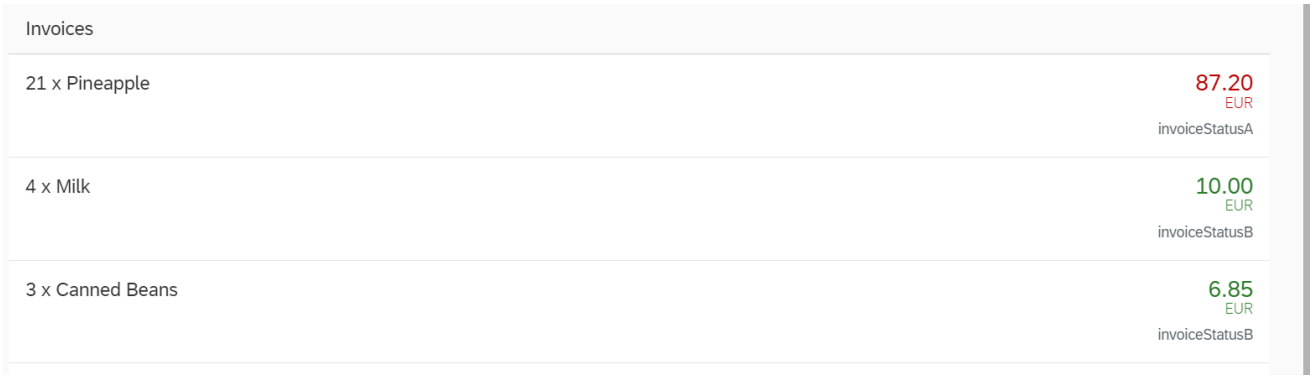
```
controller.js > controller > InvoiceList.controller.js > sap.ui.define() callback
sap.ui.define(
  [
    "sap/ui/core/mvc/Controller",
    "sap/ui/model/json/JSONModel",
    "../model/formatter",
  ],
  function (Controller, JSONModel, formatter) {
    "use strict";
    return Controller.extend("sap.ui.demo.walkthrough.controller.InvoiceList", {
      formatter: formatter,
      onInit: function () {
        var oViewModel = new JSONModel({
          currency: "EUR",
        });
        this.getView().setModel(oViewModel, "view");
      },
    });
  }
);
```

And call it in view.xml

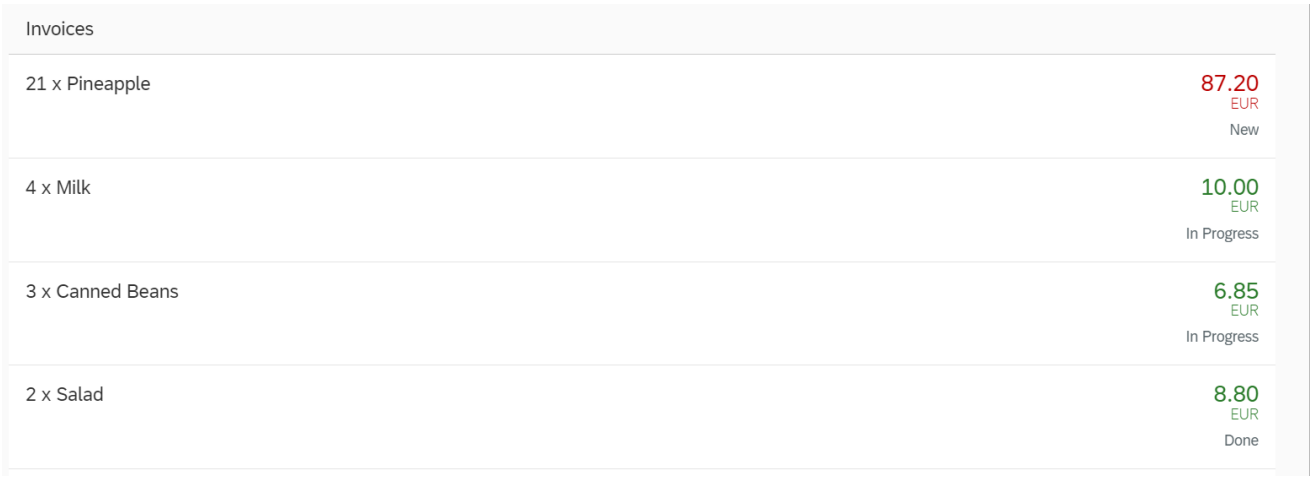
```
                <firstStatus>
                    <ObjectStatus text="{
                path: 'invoice>Status',
                formatter: '.formatter.statusText'
            }"/>
                </firstStatus>
            </ObjectListItem>
        </items>
```

This status is based on the JSON field value

```
"Invoices": [
  {
    "ProductName": "Pineapple",
    "Quantity": 21,
    "ExtendedPrice": 87.2,
    "ShipperName": "Fun Inc.",
    "ShippedDate": "2015-04-01T00:00:00",
    "Status": "A"
  },
```



Add the values to i18n



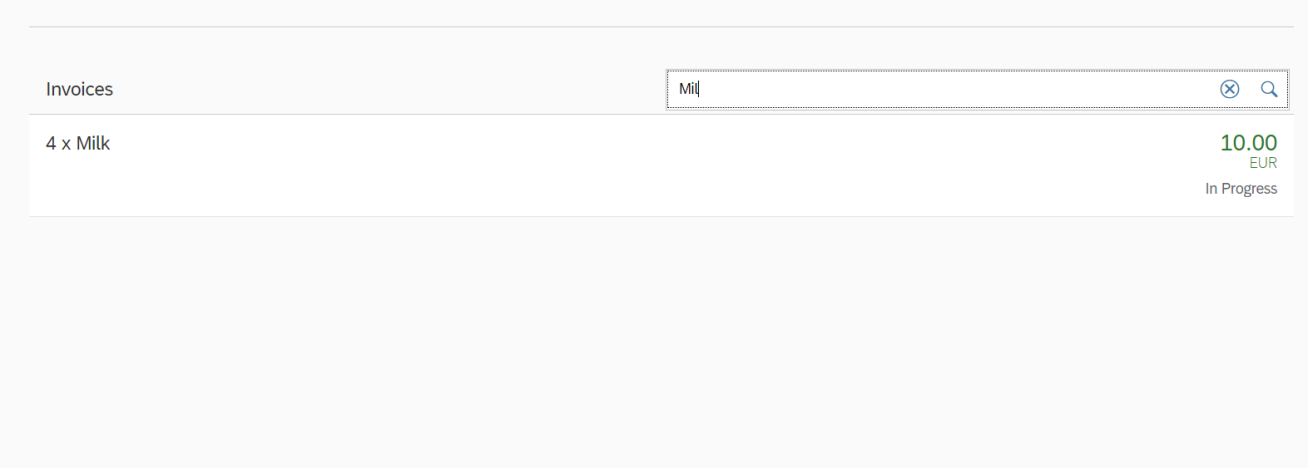
Filtering

Now, let's filter some records.
Add a search bar.

```
<List id="invoiceList" headerText="{i18n>invoiceListTitle}" class="sapUiResponsiveMargin" width="auto" it
    <headerToolbar>
        <Toolbar>
            <Title text="{i18n>invoiceListTitle}"/>
            <ToolbarSpacer/>
            <SearchField width="50%" search=".onFilterInvoices"/>
        </Toolbar>
    </headerToolbar>
    <items>
        <ObjectListItem title="{invoice>Quantity} x {invoice>ProductName}" number="{
        parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/currency'}],
        type: 'sap.ui.model.type.Currency',
        formatOptions: {
            showMeasure: false
        }
```

and add the query in the controller

```
    },
    onFilterInvoices: function (oEvent) {
      // build filter array
      var aFilter = [];
      var sQuery = oEvent.getParameter("query");
      if (sQuery) {
        aFilter.push(
          new Filter("ProductName", FilterOperator.Contains, sQuery)
        );
      }

      // filter binding
      var oList = this.byId("invoiceList");
      var oBinding = oList.getBinding("items");
      oBinding.filter(aFilter);
    },
  });
}
```

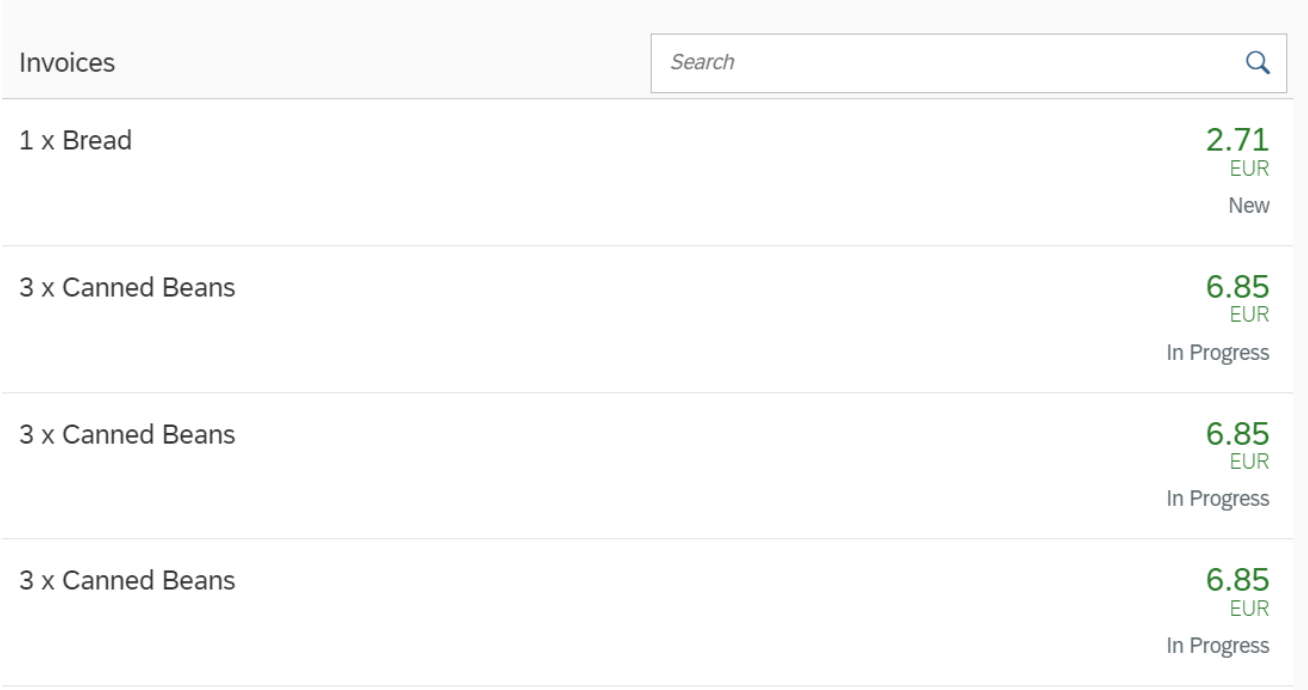


In the `onFilterInvoices` function we construct a filter object from the search string that the user has typed in the search field. Event handlers always receive an event argument that can be used to access the parameters that the event provides. In our case the search field defines a parameter `query` that we access by calling `getParameter("query")` on the `oEvent` parameter.
If the query is not empty, we add a new filter object to the still empty array of filters. However, if the query is empty, we filter the binding with an empty array. This makes sure that we see all list elements again.
If the query is not empty, we add a new filter object to the still empty array of filters. However, if the query is empty, we filter the binding with an empty array. This makes sure that we see all list elements again.
sap.ui.model.Filter - API Reference - Demo Kit - SAPUI5 SDK (ondemand.com)
sap.ui.model.FilterOperator - API Reference - Demo Kit - SAPUI5 SDK (ondemand.com)
sap.m.SearchField - API Reference - Demo Kit - SAPUI5 SDK (ondemand.com)

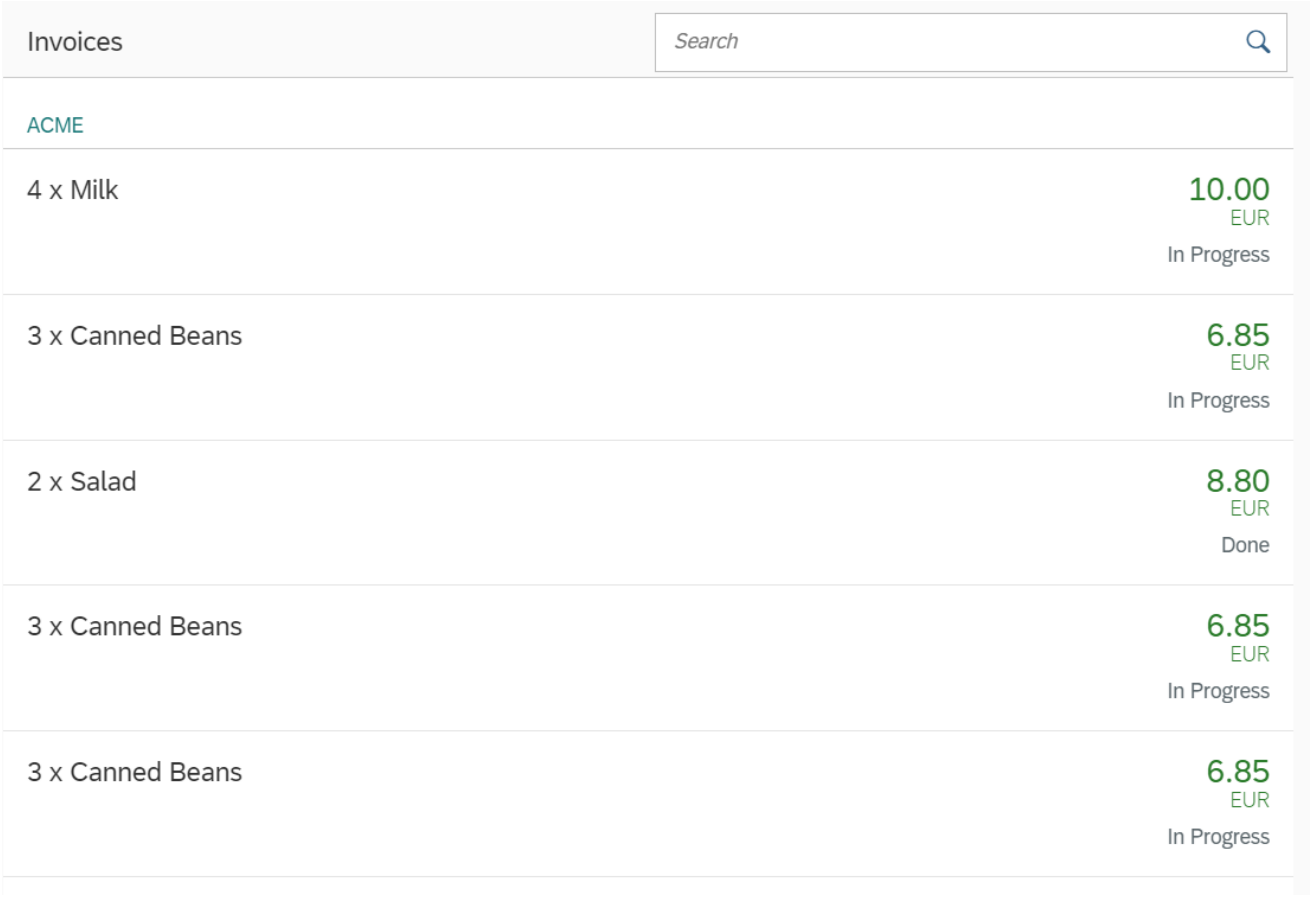
Sorting and Grouping

As like before, you can sort and group based on any field. A basic change in the view will work.

```
<List id="invoiceList" headerText="{i18n>invoiceListTitle}" class="sapUiResponsiveMargin" width="auto" items="{
    path : 'invoice>/Invoices',
    sorter : {
       path : 'ProductName'
    }
}">
```




```
<List id="invoiceList" headerText="{i18n>invoiceListTitle}" class="sapUiResponsiveMargin" width="auto" items="{
    path : 'invoice>/Invoices',
    sorter : {
       path : 'ShipperName',
       group: true
    }
}">
```



Remote OData as Source

Rather than try to build our own JSON, why not use a standard Remote OData? In fact, this is how SAP and other sources connect to UI5.

Add an OData source

```
    "applicationVersion": {
      "version": "1.0.0"
    },
    "dataSources": {
      "invoiceRemote": {
        "uri": "https://services.odata.org/V2/Northwind/Northwind.svc/",
        "type": "OData",
        "settings": {
          "odataVersion": "2.0"
        }
      }
    }
  },
  "sap.ui": {
```

and define it as the data source.

```
    "invoice": {
      "dataSource": "invoiceRemote"
    }
  },
  "resources": {
```

This is more than enough. But, it will not work because of CORS.

```
Access to XMLHttpRequest at 'https://services.odata.org/V2/Northwind/N  index.html:1
orthwind.svc/$metadata?sap-language=EN' from origin 'http://127.0.0.1:5555' has been
blocked by CORS policy: Response to preflight request doesn't pass access control
check: No 'Access-Control-Allow-Origin' header is present on the requested resource.
▶2023-02-19 23:35:22.375500 [ODataMetadata] initial loading of        Log-dbg.js:493
metadata failed -
▶2023-02-19 23:35:22.375500 Error: HTTP request failed -              Log-dbg.js:493
Failed to load resource:                  services.odata.org/V…a?sap-language=EN:1
net::ERR_FAILED
```

Due to the so called same-origin policy, browsers deny AJAX requests to service endpoints in case the service endpoint has a different domain/subdomain, protocol, or port than the app. The browser refuses to connect to a remote URL directly for security reasons.

Mock Server

Copy index.html into a test folder and create a new entry point(oninit)

App.controller.js > test > mockServer.html > html > head

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>SAPUI5 Walkthrough - Test Page</title>
    <script
      id="sap-ui-bootstrap"
      src="https://sdk.openui5.org/resources/sap-ui-core.js"
      data-sap-ui-theme="sap_belize"
      data-sap-ui-resourceroots='{
          "sap.ui.demo.walkthrough": "../"
        }'
      data-sap-ui-oninit=
      "module:sap/ui/demo/walkthrough/test/initMockServer"
      data-sap-ui-compatVersion="edge"
      data-sap-ui-async="true"
    ></script>
  </head>
  <body class="sapUiBody" id="content">
    <div
      data-sap-ui-component
      data-name="sap.ui.demo.walkthrough"
      data-id="container"
      data-settings='{"id" : "walkthrough"}'
    ></div>
  </body>
</html>
```

App.controller.js > index.html > html > head > script#sap-ui-bootstrap

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>SAPUI5 Walkthrough</title>
    <script
      id="sap-ui-bootstrap"
      src="https://sdk.openui5.org/resources/sap-ui-core.js"
      data-sap-ui-theme="sap_belize"
      data-sap-ui-resourceroots='{
      "sap.ui.demo.walkthrough": "./"
    }'
      data-sap-ui-oninit=
      "module:sap/ui/core/ComponentSupport"
      data-sap-ui-compatVersion="edge"
      data-sap-ui-async="true"
    ></script>
  </head>
  <body class="sapUiBody" id="content">
    <div
      data-sap-ui-component
      data-name="sap.ui.demo.walkthrough"
      data-id="container"
      data-settings='{"id" : "walkthrough"}'
    ></div>
  </body>
</html>
```

Initialize a mock server.

App.controller.js > test > initMockServer.js > ...

```
sap.ui.define(["../localService/mockserver"], function (mockserver) {
    "use strict";

    // initialize the mock server
    mockserver.init();

    // initialize the embedded component on the HTML page
    sap.ui.require(["sap/ui/core/ComponentSupport"]);
});
```

The `mockserver` depencency that we are about to implement is our local test server. Its `init` method is immediately called before we load the component. This way we can catch all requests that would go to the "real" service and process them locally by our test server when launching the app with the `mockServer.html` file. The component itself does not "know" that it will now run in test mode.
And then, you need to create the localService.
Define the metadata in the standard format

App.controller.js > localService > metadata.xml > edmx:Edmx

```
<edmx:Edmx Version="1.0"
    xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
    <edmx:DataServices m:DataServiceVersion="1.0" m:MaxDataServiceVersion="3.0"
        xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
        <Schema Namespace="NorthwindModel"
            xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
            <EntityType Name="Invoice">
                <Key>
                    <PropertyRef Name="ProductName"/>
                    <PropertyRef Name="Quantity"/>
                    <PropertyRef Name="ShipperName"/>
                </Key>
                <Property Name="ShipperName" Type="Edm.String" Nullable="false" MaxLength="40" FixedLength="false" Unicode="true"/>
                <Property Name="ProductName" Type="Edm.String" Nullable="false" MaxLength="40" FixedLength="false" Unicode="true"/>
                <Property Name="Quantity" Type="Edm.Int16" Nullable="false"/>
                <Property Name="ExtendedPrice" Type="Edm.Decimal" Precision="19" Scale="4"/>
                <Property Name="Status" Type="Edm.String" Nullable="false" MaxLength="1" FixedLength="false" Unicode="true"/>
            </EntityType>
        </Schema>
        <Schema Namespace="ODataWebV2.Northwind.Model"
            xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
            <EntityContainer Name="NorthwindEntities" m:IsDefaultEntityContainer="true" p6:LazyLoadingEnabled="true"
                xmlns:p6="http://schemas.microsoft.com/ado/2009/02/edm/annotation">
                <EntitySet Name="Invoices" EntityType="NorthwindModel.Invoice"/>
            </EntityContainer>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

The metadata file contains information about the service interface and does not need to be written manually. It can be accessed directly from the “real” service by calling the service URL and adding `$metadata` at the end (e.g. in our case `http://services.odata.org/V2/Northwind/Northwind.svc/$metadata`). The mock server will read this file to simulate the real OData service, and will return the results from our local source files in the proper format so that it can be consumed by the app (either in XML or in JSON format).
And finally, create the mockserver.

App.controller.js > localService > mockserver.js > ...

```
sap.ui.define(
  ["sap/ui/core/util/MockServer", "sap/base/util/UriParameters"],
  function (MockServer, UriParameters) {
    "use strict";

    return {
      init: function () {
        // create
        var oMockServer = new MockServer({
          rootUri: "https://services.odata.org/V2/Northwind/Northwind.svc/",
        });

        var oUriParameters = new UriParameters(window.location.href);

        // configure mock server with a delay
        MockServer.config({
          autoRespond: true,
          autoRespondAfter: oUriParameters.get("serverDelay") || 500,
        });

        // simulate
        var sPath = sap.ui.require.toUrl(
          "sap/ui/demo/walkthrough/localService"
        );
        oMockServer.simulate(sPath + "/metadata.xml", sPath + "/mockdata");

        // start
        oMockServer.start();
      },
    };
  }
```

Create the same Invoice.JSON but without the header

App.controller.js > localService > mockdata > {} Invoices.json > ...

```
[
  {
    "ProductName": "Pineapple",
    "Quantity": 21,
    "ExtendedPrice": 87.2,
    "ShipperName": "Fun Inc.",
    "ShippedDate": "2015-04-01T00:00:00",
    "Status": "A"
  },
  {
    "ProductName": "Milk",
    "Quantity": 4,
    "ExtendedPrice": 10,
    "ShipperName": "ACME",
    "ShippedDate": "2015-02-18T00:00:00",
    "Status": "B"
  },
```

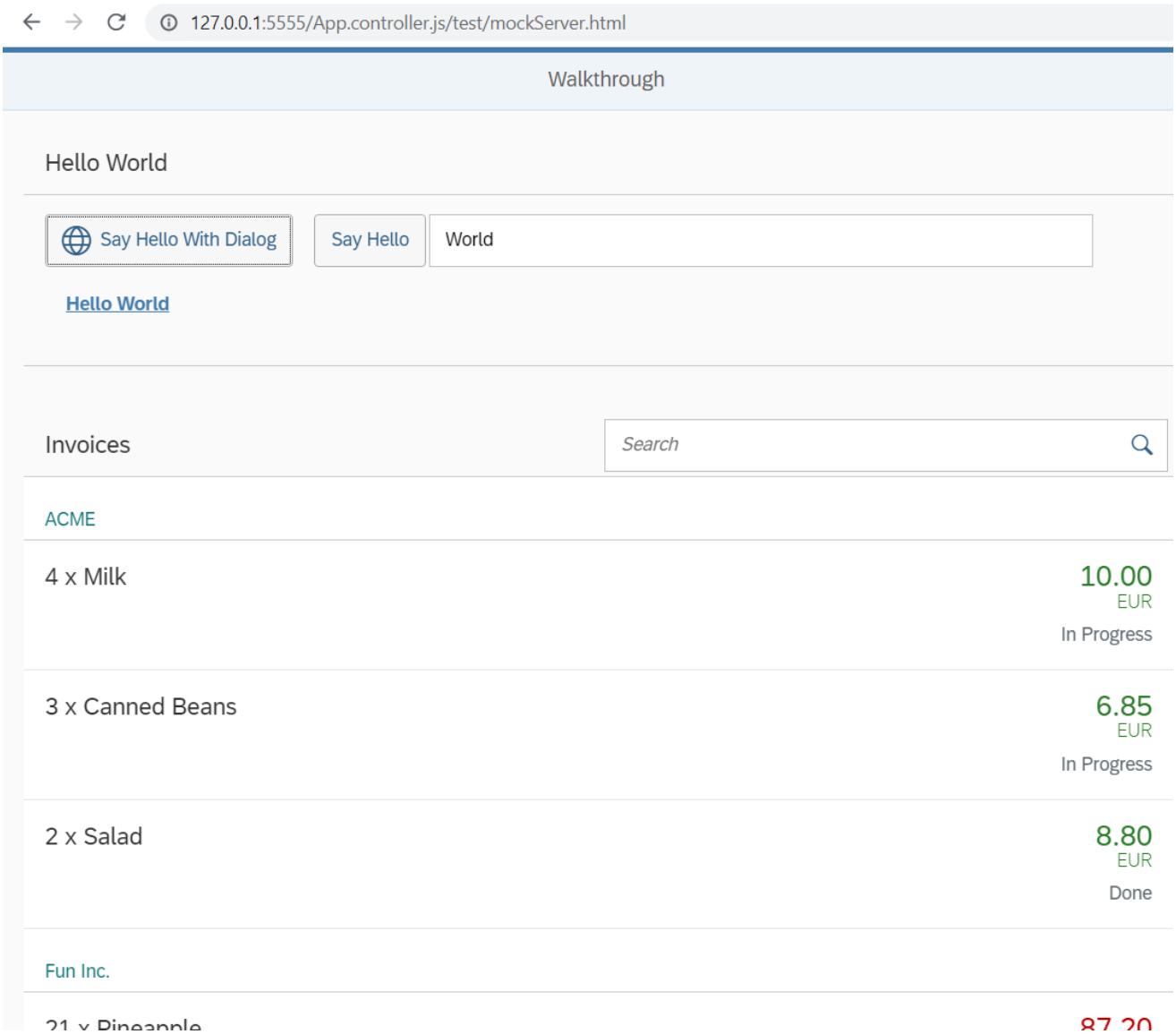
App.controller.js > {} Invoices.json > [ ] Invoices > {} 5 > ShipperName

```
{
  "Invoices": [
    {
      "ProductName": "Pineapple",
      "Quantity": 21,
      "ExtendedPrice": 87.2,
      "ShipperName": "Fun Inc.",
      "ShippedDate": "2015-04-01T00:00:00",
      "Status": "A"
    },
    {
      "ProductName": "Milk",
      "Quantity": 4,
      "ExtendedPrice": 10,
      "ShipperName": "ACME",
      "ShippedDate": "2015-02-18T00:00:00",
      "Status": "B"
```

This is straightforward configuration. Now, because we are simulating

```
oMockServer.simulate(sPath + "/metadata.xml", sPath + "/mockdata");
```

we will see the data in the file displayed.



The URL in configuration parameter `rootUri` has to be exactly the same as the `uri` that is defined for the data source in the `manifest.json` descriptor file. This can be an absolute or, for example in SAP Web IDE, a relative URL to a destination. The URL will now be served by our test server instead of the real service. Next, we set two global configuration settings that tell the server to respond automatically and introduce a delay of one second to imitate a typical server response time. Otherwise, we would have to call the respond method on the `MockServer` manually to simulate the call.

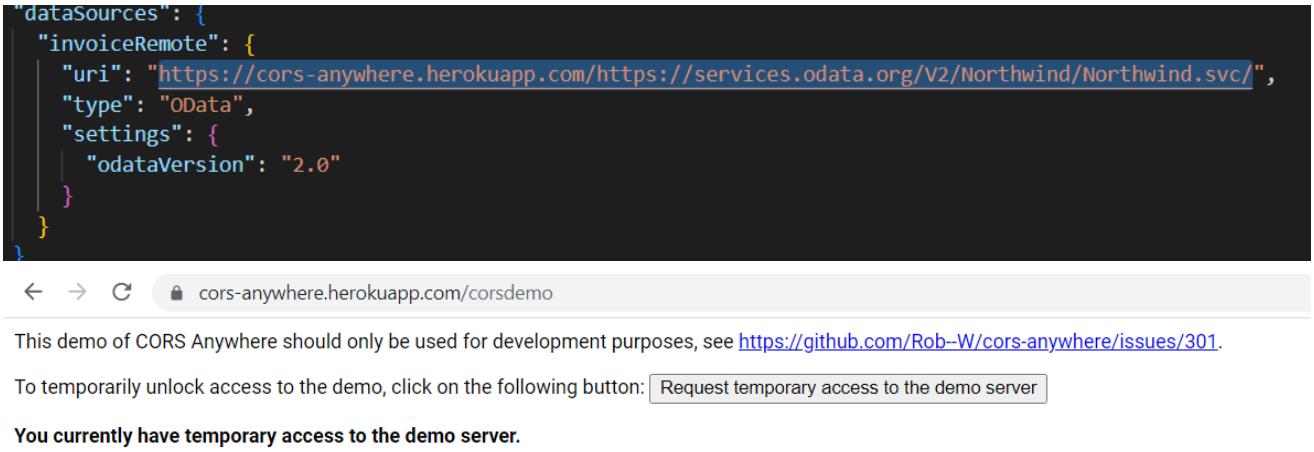
To simulate a service, we can simply call the `simulate` method on the `MockServer` instance with the path to our newly created `metadata.xml`. This will read the test data from our local file system and set up the URL patterns that will mimic the real service.

Finally, we call start on `oMockServer`. From this point, each request to the URL pattern `rootUri` will be processed by the `MockServer`. If you switch from the `index.html` file to the `mockServer.html` file in the browser, you can now see that the test data is displayed from the local sources again, but with a short delay. The delay can be specified with the URI parameter `serverDelay`, the default value is one second.

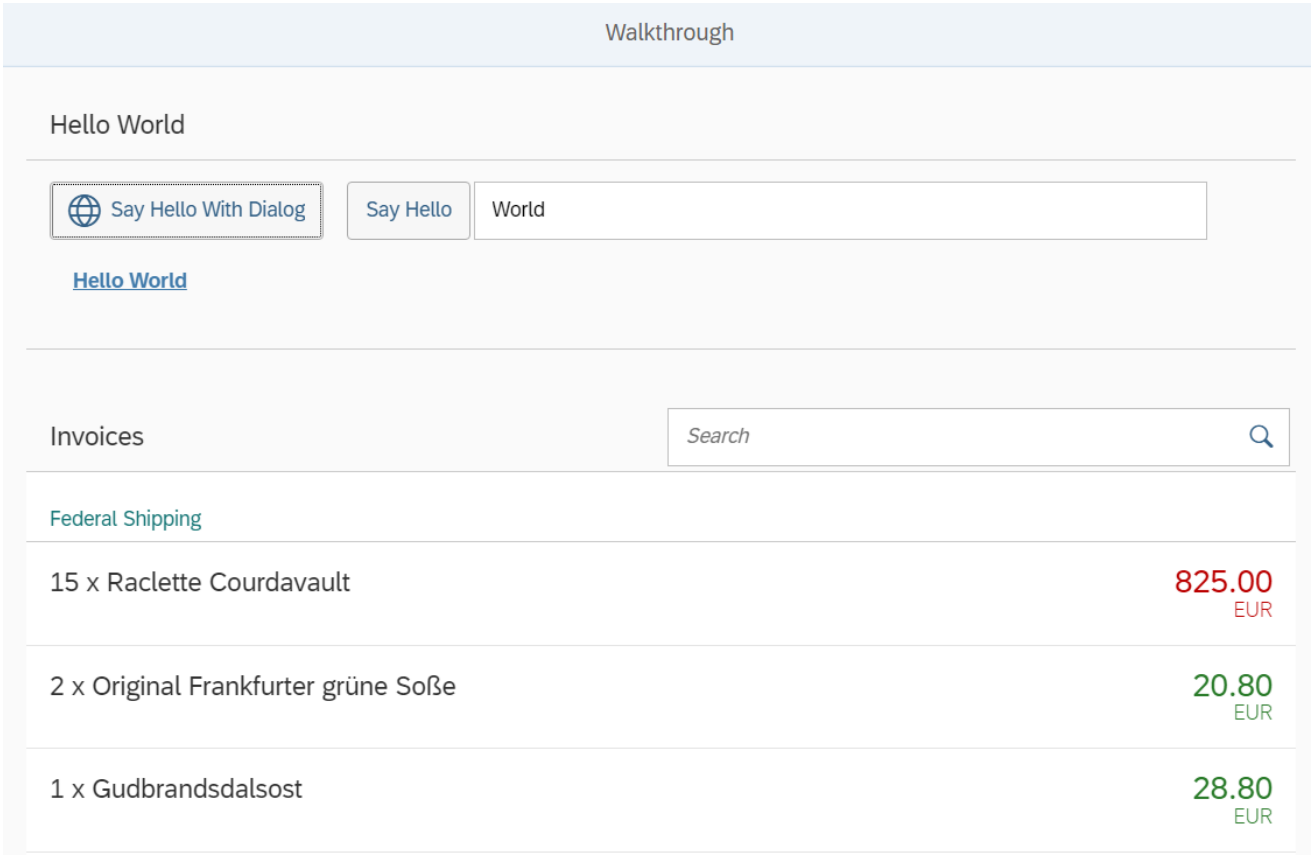
This approach is perfect for local testing, even without any network connection. This way your development does not depend on the availability of a remote server, i.e. to run your tests.
Try calling the app with the `index.html` file and the `mockServer.html` file to see the difference. If the real service connection cannot be made, for example when there is no network connection, you can always fall back to the local test page.

CORS

But, how will you bypass CORS? This method for local development.
For testing, the quickest way is to add `https://cors-anywhere.herokuapp.com/` before the path, open it and request access.



This removes the need to create a local proxy server.



Testing

Refer to Testing Section

Debugging

Refer to Debugging Section

Routing and Navigation

Routing output to a different page.
Routing contains routes and targets along with a configuration header.

App.controller.js > {..} manifest.json > {} sap.ui5

```
"routing": {
  "config": {
  "routerClass": "sap.m.routing.Router",
  "type": "View",
  "viewType": "XML",
  "path": "sap.ui.demo.walkthrough.view",
  "controlId": "app",
  "controlAggregation": "pages"
  },
  "routes": [
  {
    "pattern": "",
    "name": "overview",
    "target": "overview"
  },
  {
    "pattern": "detail",
    "name": "detail",
    "target": "detail"
  }
  ],
  "targets": {
  "overview": {
    "id": "overview",
    "name": "Overview"
  },
  "detail": {
    "id": "detail",
    "name": "Detail"
  }
  }
},
```

- `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.

Add the router.

App.controller.js > Component.js > sap.ui.define() callback > init

```
19            var oModel = new JSONModel(oData);
20            this.setModel(oModel);
21
22            // create the views based on the url/hash
23            this.getRouter().initialize();
24        },
25    });
26  }
27);
28
```

Create a view overview

App.controller.js > view > Overview.view.xml > mvc:View

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}">
        <headerContent>
            <Button icon="sap-icon://hello-world" press=".onOpenDialog"/>
        </headerContent>
        <content>
            <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
            <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.InvoiceList"/>
        </content>
    </Page>
</mvc:View>
```

Move app.view.xml to Overview.view.xml

App.controller.js > view > Overview.view.xml

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}">
        <headerContent>
            <Button
                icon="sap-icon://hello-world"
                press=".onOpenDialog"/>
        </headerContent>
        <content>
            <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.HelloPanel"/>
            <mvc:XMLView viewName="sap.ui.demo.walkthrough.view.InvoiceList"/>
        </content>
    </Page>
</mvc:View>

```

App.view will now contain only the id app

App.controller.js > view > App.view.xml > mvc:View

```
<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>
```

Create a Detail.view.xml

This will display a second view which will be displayed when clicked

App.controller.js > view > Detail.view.xml > mvc:View

```
<mvc:View xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>detailPageTitle}">
        <ObjectHeader title="Invoice"/>
    </Page>
</mvc:View>
```

This is invoked by adding Navigation .onClick in the Object list.

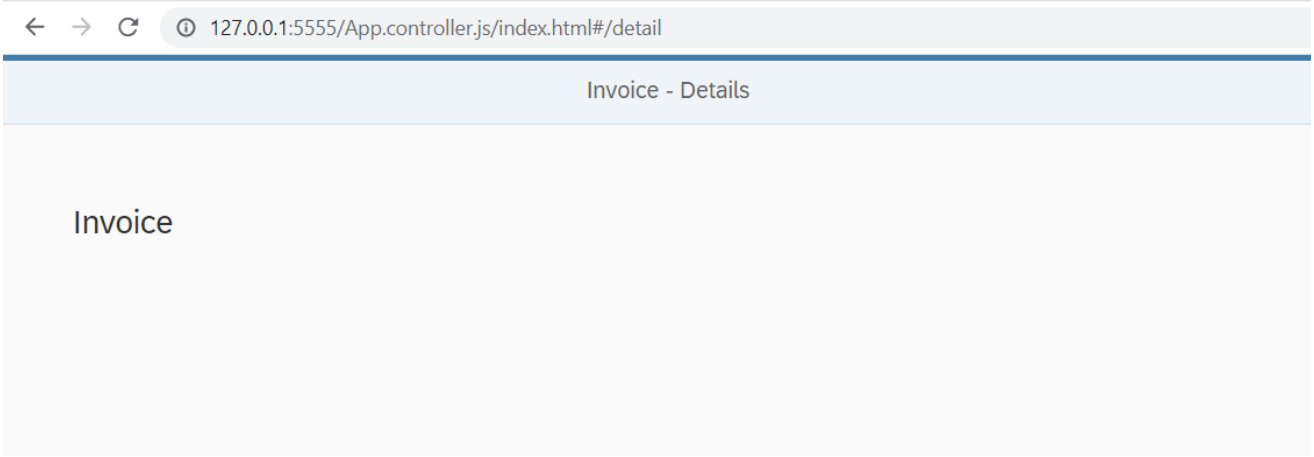
App.controller.js > view > InvoiceList.view.xml > mvc:View > List > items > ObjectListItem

```
<ObjectListItem
    title="{invoice>Quantity} x {invoice>ProductName}"
    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' }"
    type="Navigation"
    press=".onPress">
```

Controller will say, route to where.

App.controller.js > controller > InvoiceList.controller.js > sap.ui.define() callback > onInit > oViewModel

```
34
35          onPress: function (oEvent) {
36              var oRouter = this.getOwnerComponent().getRouter();
37              oRouter.navTo("detail");
38          }
39      });
40
41  });
```



Routing with Parameters

Now, rather than routing to a template page, let's route it to that particular invoice.

```
"pattern": "detail",
```

The pattern in manifest.json's routing pattern should be updated to refer to that specific invoice

```
{
    "pattern": "detail/{invoicePath}",
  "name": "detail",
```

This means that the Detail View should have it's own controller and the data should contain specific information as against the basic title.

App.controller.js > view > Detail.view.xml > mvc:View

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>detailPageTitle}">
        <ObjectHeader intro="{invoice>ShipperName}" title="{invoice>ProductName}"/>
    </Page>
</mvc:View>
```

On the other hand, invoice List's controller's getSource will return the Object List and getBindingContext will fetch the relevant record. As the path contains slash, fetch only the second part from it.

```
  onPress: function (oEvent) {
    var oItem = oEvent.getSource();
    var oRouter = this.getOwnerComponent().getRouter();
    oRouter.navTo("detail", {
      invoicePath: window.encodeURIComponent(
        oItem.getBindingContext("invoice").getPath().substr(1)
      ),
    });
  },
});
```

Detailer.controller.js

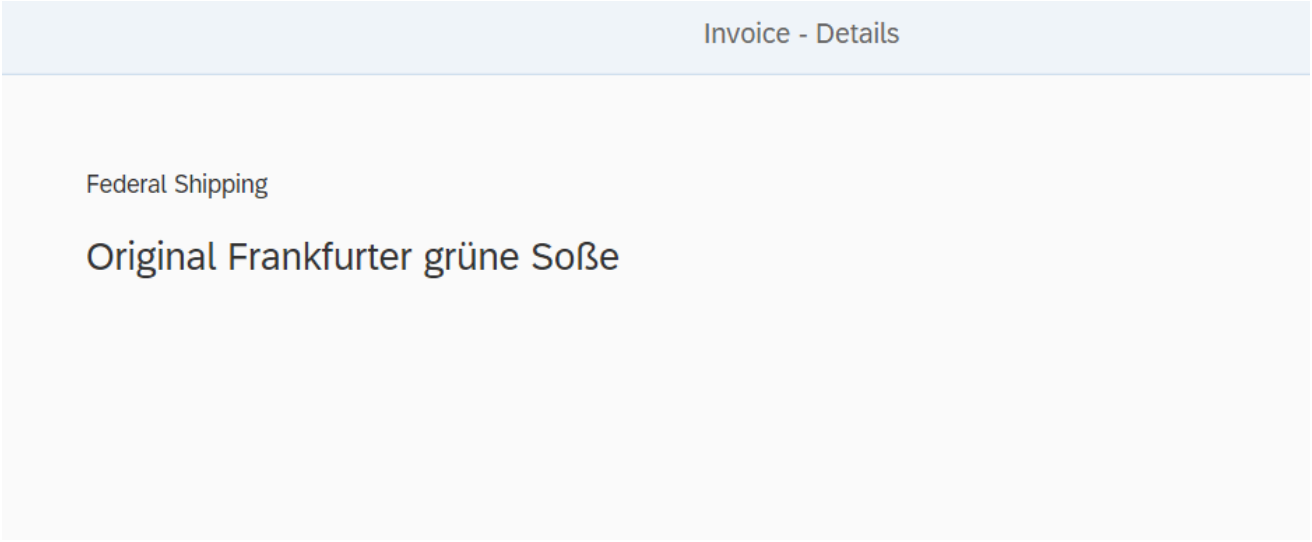
App.controller.js > controller > Detail.controller.js > ...

```
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.



Routing Back and History

Now then, how will you go back to main page from here?
Include Navigation button and have a function

App.controller.js > view > Detail.view.xml > mvc:View > Page

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>detailPageTitle}" showNavButton="true" navButtonPress=".onNavBack">
```

onNavBack works on History → previous hash of the event in the app.

App.controller.js > controller > Detail.controller.js > sap.ui.define() callback

```
    },
    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 Controls for Routing

Create a new control which uses SAP's Control functionality.

App.controller.js > control > ProductRating.js > ...

```
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) {},
  });
});
```

Custom controls are small reuse components that can be created within the app very easily. Due to their nature, they are sometimes also referred to as "notepad" or “on the fly” controls. A custom control is a JavaScript object that has two special sections (`metadata` and `renderer`) and a number of methods that implement the functionality of the control.
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.

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.

The completed file uses RatingIndicator, Label and Button and

```
sap.ui.define(
  [
    "sap/ui/core/Control",
    "sap/m/RatingIndicator",
    "sap/m/Label",
    "sap/m/Button",
  ],
  function (Control, RatingIndicator, Label, Button) {
    "use strict";
    return Control.extend("sap.ui.demo.walkthrough.control.ProductRating", {
      metadata: {
        properties: {
          value: { type: "float", defaultValue: 0 },
        },
        aggregations: {
          _rating: {
            type: "sap.m.RatingIndicator",
            multiple: false,
            visibility: "hidden",
          },
```

renders the same.

```
    renderer: function (oRm, oControl) {
      oRm.openStart("div", oControl);
      oRm.class("myAppDemoWTProductRating");
      oRm.openEnd();
      oRm.renderControl(oControl.getAggregation("_rating"));
      oRm.renderControl(oControl.getAggregation("_label"));
      oRm.renderControl(oControl.getAggregation("_button"));
      oRm.close("div");
    },
  });
}
```

Init module contains how to manipulate the ratings

```
reset: function () {
  var oResourceBundle = this.getModel("i18n").getResourceBundle();

  this.setValue(0);
  this.getAggregation("_label").setDesign("Standard");
  this.getAggregation("_rating").setEnabled(true);
  this.getAggregation("_label").setText(
    oResourceBundle.getText("productRatingLabelInitial")
  );
  this.getAggregation("_button").setEnabled(true);
},

_onRate: function (oEvent) {
  var oRessourceBundle = this.getModel("i18n").getResourceBundle();
  var fValue = oEvent.getParameter("value");

  this.setProperty("value", fValue, true);

  this.getAggregation("_label").setText(
    oRessourceBundle.getText("productRatingLabelIndicator", [
      fValue,
      oEvent.getSource().getMaxValue(),
    ])
```

We now enhance our new custom control with the custom functionality that we need. In our case we want to create an interactive product rating, so we define a value and use three internal controls that are displayed updated by our control automatically. A `RatingIndicator` control is used to collect user input on the product, a label is displaying further information, and a button submits the rating to the app to store it.

In the `metadata` section we therefore define several properties that we make use in the implementation:

- Properties
  - Value
    We define a control property `value` that will hold the value that the user selected in the rating. Getter and setter function for this property will automatically be created and we can also bind it to a field of the data model in the XML view if we like.
- Aggregations
  As described in the first paragraph, we need three internal controls to realize our rating functionality. We therefore create three “hidden aggregations” by setting the `visibility` attribute to `hidden`. This way, we can use the models that are set on the view also in the inner controls and SAPUI5 will take care

of the lifecycle management and destroy the controls when they are not needed anymore. Aggregations can also be used to hold arrays of controls but we just want a single control in each of the aggregations so we need to adjust the cardinality by setting the attribute `multiple` to `false`.

- `_rating`: A `sap.m.RatingIndicator` control for user input
- `_label`: A `sap.m.Label` to display additional information
- `_button`: A `sap.m.Button` to submit the rating
  You can define `aggregations` and `associations` for controls. The difference is in the relation between the parent and the related control:
- An aggregation is a strong relation that also manages the lifecycle of the related control, for example, when the parent is destroyed, the related control is also destroyed. Also, a control can only be assigned to one single aggregation, if it is assigned to a second aggregation, it is removed from the previous aggregation automatically.
- An association is a weak relation that does not manage the lifecycle and can be defined multiple times. To have a clear distinction, an association only stores the ID, whereas an aggregation stores the direct reference to the control. We do not specify associations in this example, as we want to have our internal controls managed by the parent.

- Events
  - Change
    We specify a `change` event that the control will fire when the rating is submitted. It contains the current value as an event parameter. Applications can register to this event and process the result similar to “regular” SAPUI5 controls, which are in fact built similar to custom controls.

In the `init` function that is called by SAPUI5 automatically whenever a new instance of the control is instantiated, we set up our internal controls. We instantiate the three controls and store them in the internal aggregation by calling the framework method `setAggregation` that has been inherited from `sap.ui.core.Control`. We pass on the name of the internal aggregations that we specified above and the new control instances. We specify some control properties to make our custom control look nicer and register a `liveChange` event to the rating and a press event to the button. The initial texts for the label and the button are referenced from our `i18n` model.
Let’s ignore the other internal helper functions and event handlers for now and define our renderer. With the help of the SAPUI5 render manager and the control instance that are passed on as a reference, we can now render the HTML structure of our control. We render the start of the outer `<div>` tag as `<div` and

call the helper method `writeControlData` to render the ID and other basic attributes of the control inside the `div` tag. Next, we add a custom CSS class so that we can define styling rules for the custom control in our CSS file later. This CSS class and others that have been added in the view are then rendered by calling `writeClasses` on the renderer instance. Then we close the surrounding `div` tag and render three internal controls by passing the content of the internal aggregation to the render managers `renderControl` function. This will call the renderer of the controls and add their HTML to the page. Finally, we close our surrounding `<div>` tag.
The `setValue` is an overridden setter. SAPUI5 will generate a setter that updates the property value when called in a controller or defined in the XML view, but we also need to update the internal rating control in the hidden aggregation to reflect the state properly. Also, we can skip the rerendering of SAPUI5 that is usually triggered when a property is changed on a control by calling the `setProperty` method to update the control property with true as the third parameter.
Now we define the event handler for the internal rating control. It is called every time the user changes the rating. The current value of the rating control can be read from the event parameter value of the `sap.m.RatingIndicator` control. With the value we call our overridden setter to update the control state, then we update the `label` next to the rating to show the user which value he has selected currently and also displays the maximum value. The string with the placeholder values is read from the `i18n` model that is assigned to the control automatically.
Next, we have the `press` handler for the rating button that submits our rating. We assume that rating a product is a one-time action and first disable the rating and the button so that the user is not allowed to submit another rating. We also update the label to show a "Thank you for your rating!" message, then we fire the change event of the control and pass in the current value as a parameter so that applications that are listening to this event can react on the rating interaction.
We define the `reset` method to be able to revert the state of the control on the UI to its initial state so that the user can again submit a rating.

```
sap.ui.define([
        "sap/ui/core/Control",
        "sap/m/RatingIndicator",
        "sap/m/Label",
        "sap/m/Button"

], function (Control, RatingIndicator, Label, Button) {
        "use strict";
```

```
        return
Control.extend("sap.ui.demo.walkthrough.control.ProductRating", {
                metadata : {
                        properties : {
                                value:  {type : "float", defaultValue : 0}
                        },
                        aggregations : {
                                _rating : {type : "sap.m.RatingIndicator",
multiple: false, visibility : "hidden"},
                                _label : {type : "sap.m.Label", multiple:
false, visibility : "hidden"},
                                _button : {type : "sap.m.Button", multiple:
false, visibility : "hidden"}
                        },
                        events : {
                                change : {
                                        parameters : {
                                                value : {type : "int"}
                                        }
                                }
                        }
                },
                init : function () {
                        this.setAggregation("_rating", new
RatingIndicator({
                                value: this.getValue(),
                                iconSize: "2rem",
                                visualMode: "Half",
                                liveChange: this._onRate.bind(this)
                        }));
                        this.setAggregation("_label", new Label({
                                text: "{i18n>productRatingLabelInitial}"
                        }).addStyleClass("sapUiSmallMargin"));
                        this.setAggregation("_button", new Button({
                                text: "{i18n>productRatingButton}",
                                press: this._onSubmit.bind(this)
                        }).addStyleClass("sapUiTinyMarginTopBottom"));
                },

                setValue: function (fValue) {
                        this.setProperty("value", fValue, true);
```

```
                        this.getAggregation("_rating").setValue(fValue);
                },

                reset: function () {
                        var oResourceBundle =
this.getModel("i18n").getResourceBundle();

                        this.setValue(0);

this.getAggregation("_label").setDesign("Standard");
                        this.getAggregation("_rating").setEnabled(true);

this.getAggregation("_label").setText(oResourceBundle.getText("productRatin
gLabelInitial"));
                        this.getAggregation("_button").setEnabled(true);
                },

                _onRate : function (oEvent) {
                        var oRessourceBundle =
this.getModel("i18n").getResourceBundle();
                        var fValue = oEvent.getParameter("value");

                        this.setProperty("value", fValue, true);


this.getAggregation("_label").setText(oRessourceBundle.getText("productRati
ngLabelIndicator", [fValue, oEvent.getSource().getMaxValue()]));
                        this.getAggregation("_label").setDesign("Bold");
                },

                _onSubmit : function (oEvent) {
                        var oResourceBundle =
this.getModel("i18n").getResourceBundle();

                        this.getAggregation("_rating").setEnabled(false);

this.getAggregation("_label").setText(oResourceBundle.getText("productRatin
gLabelFinal"));
                        this.getAggregation("_button").setEnabled(false);
                        this.fireEvent("change", {
                                value: this.getValue()
```

```
                        });
                },
                renderer : function (oRm, oControl) {
                        oRm.openStart("div", oControl);
                        oRm.class("myAppDemoWTProductRating");
                        oRm.openEnd();

oRm.renderControl(oControl.getAggregation("_rating"));

oRm.renderControl(oControl.getAggregation("_label"));

oRm.renderControl(oControl.getAggregation("_button"));
                        oRm.close("div");
                }
        });
});
```

INclude the same in View.xml

App.controller.js > view > Detail.view.xml > mvc:View

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.Detail"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:wt="App.controller.js.control">
    <Page title="{i18n>detailPageTitle}" showNavButton="true" navButtonPress=".onNavBack">
        <ObjectHeader intro="{invoice>ShipperName}" title="{invoice>ProductName}"/>
        <wt:ProductRating id="rating" class="sapUiSmallMarginBeginEnd" change=".onRatingChange"/>
    </Page>
</mvc:View>
```

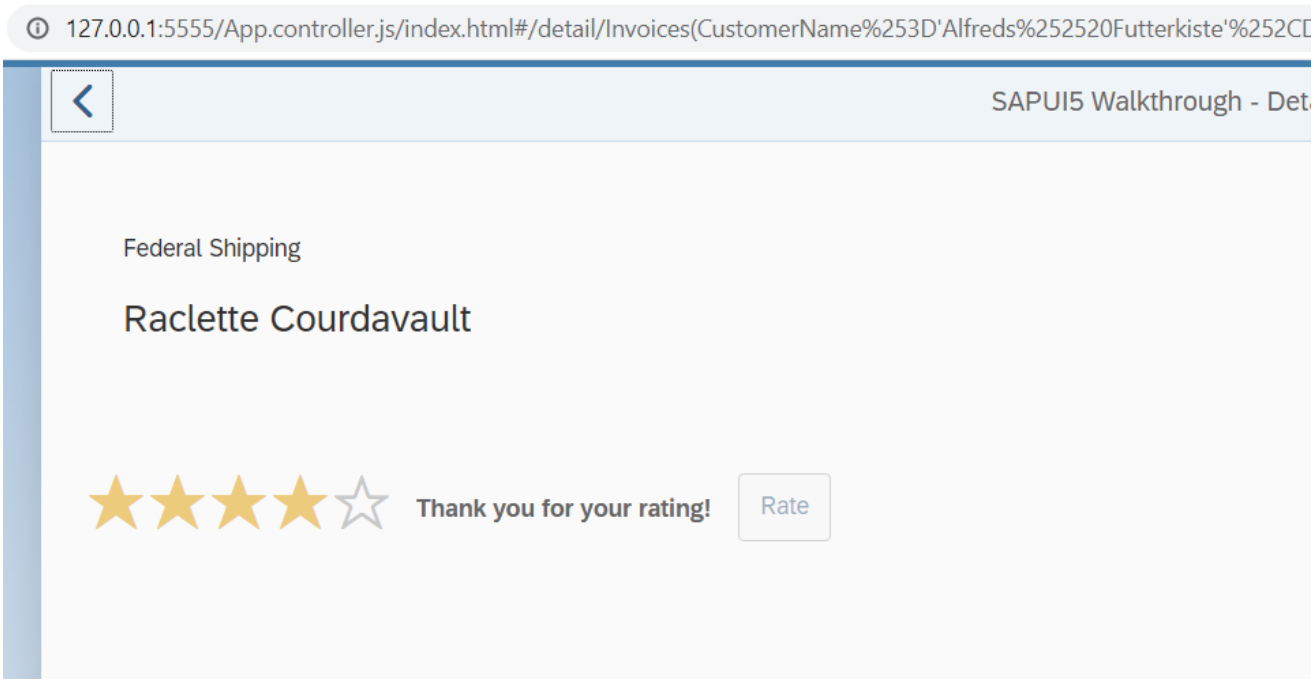
and display a note when rating is changed.

App.controller.js > controller > Detail.controller.js > sap.ui.define() callback

```
        onRatingChange: function (oEvent) {
          var fValue = oEvent.getParameter("value");
          var oResourceBundle = this.getView()
            .getModel("i18n")
            .getResourceBundle();

          MessageToast.show(
            oResourceBundle.getText("ratingConfirmation", [fValue])
          );
        },
      });
    }
  );
```

Reset is used to change the value

```
_onObjectMatched: function (oEvent) {
    this.byId("rating").reset();
    this.getView().bindElement({
```



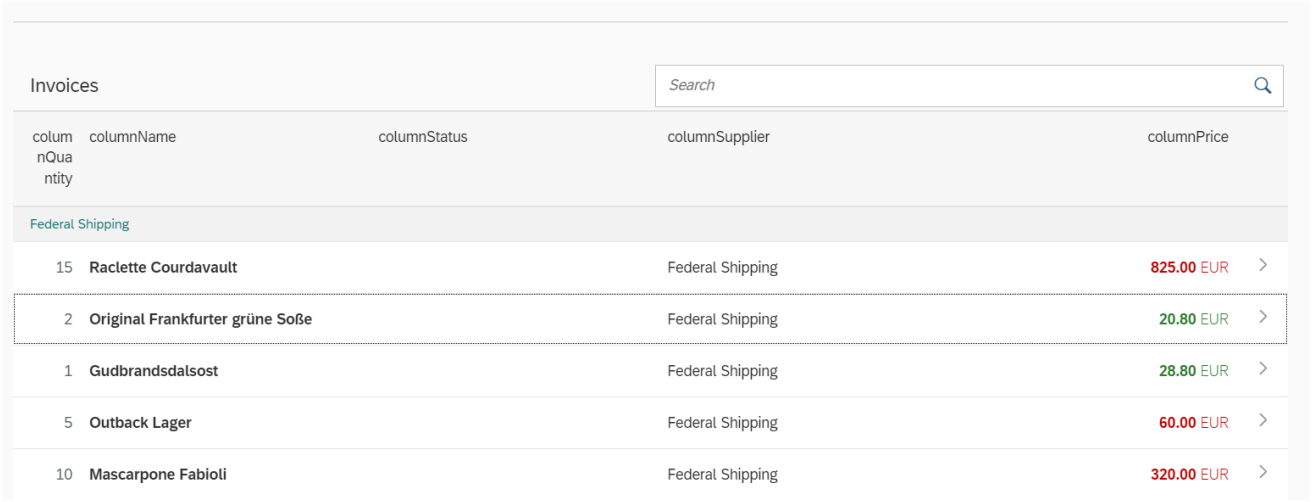
Responsiveness

While a list is not responsive, a table is responsive!! It's as simple as that.

```
<List
    id="invoiceList"
    class="sapUiResponsiveMargin"
    width="auto"
    items="{
        path : 'invoice>/Invoices',
        sorter : {
            path : 'ShipperName',
            group : true
        }
    }">
    <headerToolbar>
        <Toolbar>
            <Title text="{i18n>invoiceListTitle}"/>
            <ToolbarSpacer/>
            <SearchField width="50%" search=".onFilterInvoices"/>
        </Toolbar>
    </headerToolbar>
```

Make the requisite changes to build the structure from scratch again

```
<Table id="invoiceList" class="sapUiResponsiveMargin" width="auto" items="{
        path : 'invoice>/Invoices',
        sorter : {
            path : 'ShipperName',
            group : true
        }
    }">
    <headerToolbar>
        <Toolbar>
            <Title text="{i18n>invoiceListTitle}"/>
            <ToolbarSpacer/>
            <SearchField width="50%" search=".onFilterInvoices"/>
        </Toolbar>
    </headerToolbar>
    <columns>
        <Column hAlign="End" minScreenWidth="Small" demandPopin="true" width="4em">
            <Text text="{i18n>columnQuantity}"/>
        </Column>
```



Device Specific

Now that we have made it responsive, it makes more sense to make the code device adaptable.

```
ontroller.js > view > HelloPanel.view.xml > mvc:View > Panel > content > Button
  :View
  controllerName="sap.ui.demo.walkthrough.controller.HelloPanel"
  xmlns="sap.m"
  xmlns:mvc="sap.ui.core.mvc">
  <Panel
      headerText="{i18n>helloPanelTitle}"
      class="sapUiResponsiveMargin"
      width="auto"
      expandable="{device>/system/phone}"
      expanded="{= !${device>/system/phone} }">
```

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 device model is filled with the sap.ui.Device API of SAPUI5.

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. The device API of SAPUI5 offers more functionality to detect various device-specific settings, please have a look at the documentation for more details.
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.
The Device dependency is added to Component.js

```
// set device model
var oDeviceModel = new JSONModel(Device);
oDeviceModel.setDefaultBindingMode("OneWay");
this.setModel(oDeviceModel, "device");
```

In the `app` component we add a dependency to `sap.ui.Device` and initialize the device model in the `init` method. We can simply pass the loaded dependency `Device` to the constructor function of the JSONModel. This will make most properties of the SAPUI5 device API available as a JSON model which can be referred in data binding.
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`).

App.controller.js > view > Detail.view.xml > mvc:View

```
4   xmlns:wt="sap.ui.demo.walkthrough.control">
5   <Page title="{i18n>detailPageTitle}" showNavButton="true" navButtonPress=".onNavBack">
6       <ObjectHeader responsive="true" fullScreenOptimized="true" number="{
7               parts: [{path: 'invoice>ExtendedPrice'}, {path: 'view>/currency'}],
8               type: 'sap.ui.model.type.Currency',
9               formatOptions: {
10                  showMeasure: false
11              }
12          }" numberUnit="{view>/currency}" intro="{invoice>ShipperName}" title="{invoice>ProductName}">
13          <attributes>
14              <ObjectAttribute title="{i18n>quantityTitle}" text="{invoice>Quantity}"></ObjectAttribute>
15              <ObjectAttribute title="{i18n>dateTitle}" text="{
16                  path: 'invoice>ShippedDate',
17                  type: 'sap.ui.model.type.Date',
18                  formatOptions: {
19                    style: 'long',
20                    source: {
21                      pattern: 'yyyy-MM-ddTHH:mm:ss'
22                    }
23                  }
24                }"/>
25          </attributes>
26      </ObjectHeader>
```

The same is mentioned in the view → while `responsive` & `fullScreenOptimized` are set to true, a new attribute model is defined for the specific device.

And pass the same model.

App.controller.js > controller > Detail.controller.js > sap.ui.define() callback

```
    "sap/ui/core/mvc/Controller",
    "sap/ui/core/routing/History",
    "sap/m/MessageToast",
    "sap/ui/model/json/JSONModel",
  ],
  function (Controller, History, MessageToast) {
    "use strict";

    return Controller.extend("sap.ui.demo.walkthrough.controller.Detail", {
      onInit: function () {
        var oViewModel = new JSONModel({
          currency: "EUR",
        });
        this.getView().setModel(oViewModel, "view");

```

Content Density

Content density refers to the ratio of content on a page in relation to the size of that same page. In mobile devices, Context Density is lesser on mobile phones and more on desktop.

App.controller.js > Component.js > sap.ui.define() callback > init

```
      getContentDensityClass: function () {
        if (!this._sContentDensityClass) {
          if (!Device.support.touch) {
            this._sContentDensityClass = "sapUiSizeCompact";
          } else {
            this._sContentDensityClass = "sapUiSizeCozy";
          }
        }
        return this._sContentDensityClass;
      },
    });
```

This helper method queries the Device API directly for touch support of the client and returns the CSS class `sapUiSizeCompact` if touch interaction is not supported and `sapUiSizeCozy` for all other cases. We will use it throughout the application coding to set the proper content density CSS class.

App.controller.js > controller > App.controller.js > sap.ui.define() callback

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

  return Controller.extend("sap.ui.demo.walkthrough.controller.App", {
    onInit: function () {
      this.getView().addStyleClass(
        this.getOwnerComponent().getContentDensityClass()
      );
    },
  });
});
```

Content Density is controlled by applying that CSS class at init.

App.controller.js > controller > HelloPanel.controller.js > sap.ui.define() callback > onOpenDialog

```
      if (!this.pDialog) {
        this.pDialog = this.loadFragment({
          name: "sap.ui.demo.walkthrough.view.HelloDialog",
        }).then(
          function (oDialog) {
            // forward compact/cozy style into dialog
            syncStyleClass(
              this.getOwnerComponent().getContentDensityClass(),
              this.getView(),
              oDialog
            );
            return oDialog;
          }.bind(this)
        );
      }
```

App.controller.js > {..} manifest.json > {} sap.ui5

```
    "contentDensities": {
      "compact": true,
      "cozy": true
    },
```

AIRA and Accessibility

Accessible Rich Internet Applications (ARIA) – Part 1: Introduction | SAP Blogs

`sap.m.PageAccessibleLandmarkInfo` to define ARIA roles and labels for the overview page areas.

App.controller.js > view > Overview.view.xml > mvc:View > Page > landmarkInfo

```
<mvc:View
    controllerName="sap.ui.demo.walkthrough.controller.App"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Page title="{i18n>homePageTitle}">
        <landmarkInfo>
            <PageAccessibleLandmarkInfo
                rootRole="Region"
                rootLabel="{i18n>Overview_rootLabel}"
                contentRole="Main"
                contentLabel="{i18n>Overview_contentLabel}"
                headerRole="Banner"
                headerLabel="{i18n>Overview_headerLabel}"/>
        </landmarkInfo>
```

App.controller.js > view > InvoiceList.view.xml > mvc:View > Panel

```
<mvc:View controllerName="sap.ui.demo.walkthrough.controller.InvoiceList"
    xmlns="sap.m"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel accessibleRole="Region">
        <headerToolbar>
            <Toolbar>
                <Title text="{i18n>invoiceListTitle}"/>
                <ToolbarSpacer/>
                <SearchField width="50%" search=".onFilterInvoices" ariaLabelledBy="searchFieldLabel" ariaDescribedBy="searchFieldDesc
            </Toolbar>
```

The role is referred to in the relevant page

App.controller.js > view > HelloPanel.view.xml > mvc:View > Panel

```
ResponsiveMargin" width="auto" expandable="{device>/system/phone}" expanded="{= !${device>/system/phone} }" accessibleRole="Region">
```

Data Binding

Data binding is used to bind UI elements to data sources to keep the data in sync and allow data editing on the UI.
For data binding, you need a model and a binding instance: The model instance holds the data and provides methods to set the data or to retrieve the data from a server. It also provides a method for creating bindings to the data. When this method is called, a binding instance is created, which contains the binding information and provides an event, which is fired whenever the bound data changes. An element can listen to this event and update its visualization according to the new data.
The UI uses data binding to bind controls to the model which holds the application data, so that the controls are updated automatically whenever application data changes. Data binding is also used the other way round, when changes in the control cause updates in the underlying application data, for example data entered by the user. This is called two-way binding.

Basic Script - No Data Binding

Consider a basic html built in the UI5 template.

App.controller.js > index.html > ...

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Data Binding</title>
    <script
      id="sap-ui-bootstrap"
      src="https://sdk.openui5.org/resources/sap-ui-core.js"
      data-sap-ui-theme="sap_belize"
      data-sap-ui-libs="sap.m"
      data-sap-ui-resourceroots='{"sap.ui.demo.db": "./"}'
      data-sap-ui-compatVersion="edge"
      data-sap-ui-async="true"
    ></script>
    <script src="index.js"></script>
  </head>
  <body class="sapUiBody" id="content"></body>
</html>
```

App.controller.js > index.js > sap.ui.require() callback > attachInit() callback > text

```
sap.ui.require(["sap/m/Text"], function (Text) {
  "use strict";

  // Attach an anonymous function to the SAPUI5 'init' event
  sap.ui.getCore().attachInit(function () {
    // Create a text UI element that displays a hardcoded text string
    new Text({ text: "Hi, my name is I don't know" }).placeAt("content");
  });
});
```

Notice that the text is hardcoded - meaning there is no data binding.

Creating a Data Model

There are four data models which UI5 accesses -

- JSON
- XML
- OData
- Custom Data Model
  In addition, there is something called Resource Model which is a wrapper object around a resource bundle file. These files end with `.properties` and generally contain language data.
  JSON, XML and Resource Models are Client Models - whenever the data model is invoked, the whole data is loaded and will be known to the application.
  OData is a server side model. Meaning whenever application needs data, the

data should be requested from the server. Because it's real time data, sorting, aggregation etc in this case should be handled by the server. Consider a JSON Model. Initialize it and assign the model to UI5.

App.controller.js > index.js > sap.ui.require() callback

```
3   function (Text, JSONModel) {
4     "use strict";
5
6     // Attach an anonymous function to the SAPUI5 'init' event
7     sap.ui.getCore().attachInit(function () {
8       // Create a JSON model from an object literal
9       var oModel = new JSONModel({
10        greetingText: "Hi, my name is Harry Hawk",
11      });
12
13      // Assign the model object to the SAPUI5 core
14      sap.ui.getCore().setModel(oModel);
15
16      // Create a text UI element that displays a hardcoded text string
17      new Text({ text: "Hi, my name is I don't know" }).placeAt("content");
18    });
19  }
20 );
```

Though the data is created and initiated, it's still not bound - the data is still hardcoded.

Property Binding

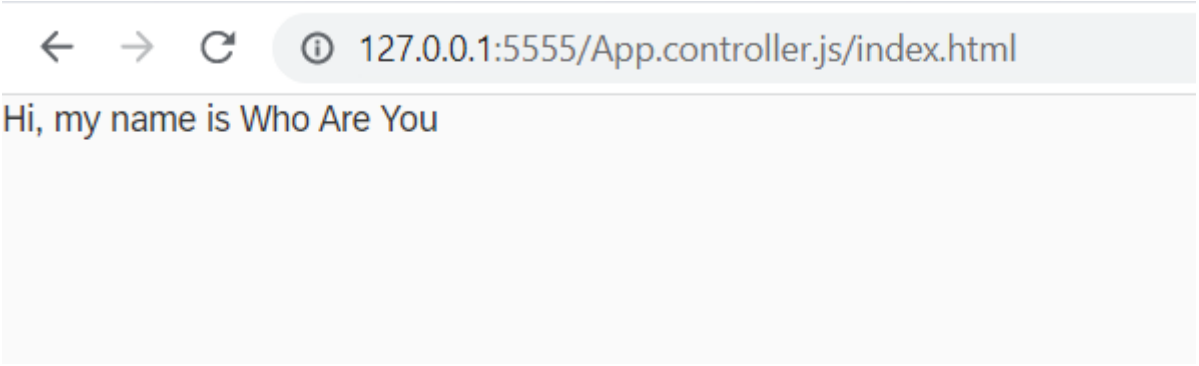
The data model is created but it needs to be invoked.

App.controller.js > index.js > ...

```
3   function (Text, JSONModel) {
4     "use strict";
5
6     // Attach an anonymous function to the SAPUI5 'init' event
7     sap.ui.getCore().attachInit(function () {
8       // Create a JSON model from an object literal
9       var oModel = new JSONModel({
10        greetingText: "Hi, my name is Who Are You",
11      });
12
13      // Assign the model object to the SAPUI5 core
14      sap.ui.getCore().setModel(oModel);
15
16      // Create a text UI element that displays a hardcoded text string
17      new Text({ text: "{/greetingText}" }).placeAt("content");
18    });
19  }
20 );
```

The property inside that model is directly called with a / before it(/ denotes root).

{} indicate that it's a data binding.



Two Way Binding

In two way binding, the model is capable of considering user inputs and adding it to the model.
Let's consider some input fields in the file and an enabled flag.

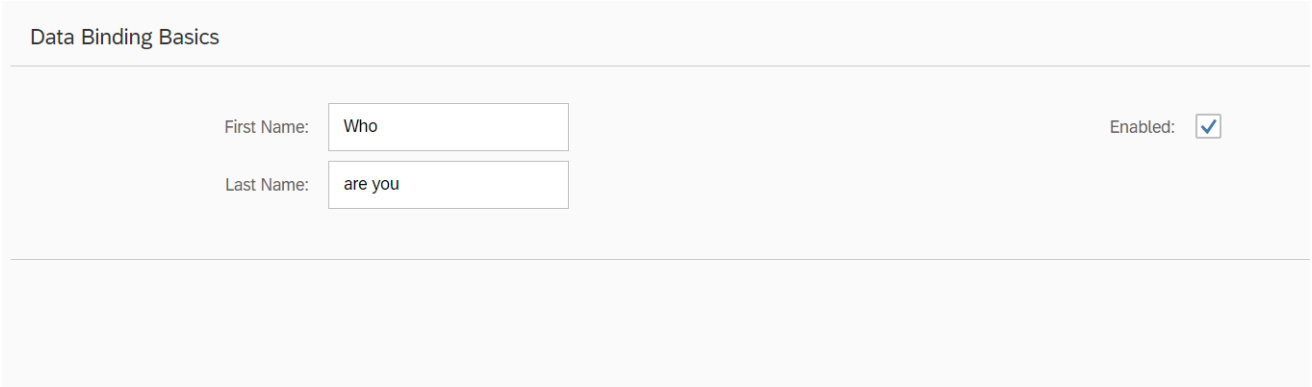
App.controller.js > view > App.view.xml > mvc:View > Panel > form:SimpleForm > CheckBox

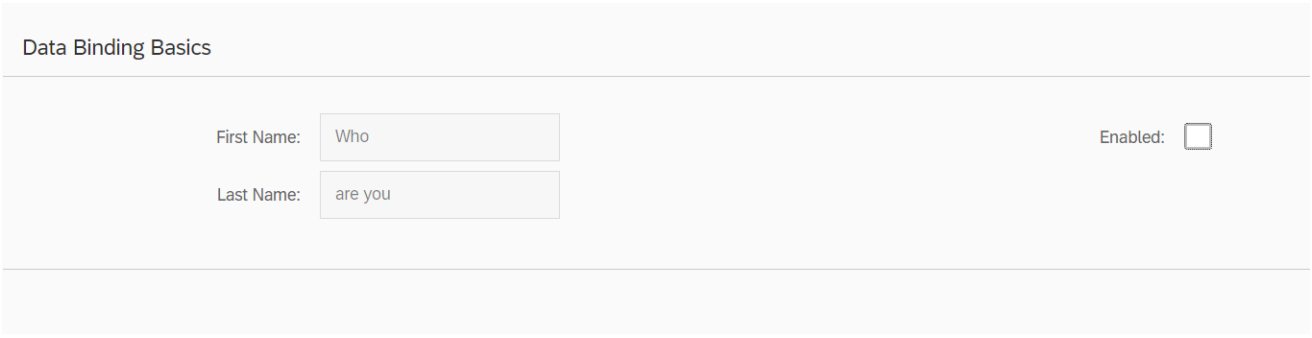
```
<mvc:View
    xmlns="sap.m"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{/panelHeaderText}" class="sapUiResponsiveMargin" width="auto">
        <form:SimpleForm editable="true" layout="ColumnLayout">
            <Label text="First Name"/>
            <Input value="{/firstName}" valueLiveUpdate="true" width="200px" enabled="{/enabled}"/>
            <Label text="Last Name"/>
            <Input value="{/lastName}" valueLiveUpdate="true" width="200px" enabled="{/enabled}"/>
            <Label text="Enabled"/>
            <CheckBox selected="{/enabled}"/>
        </form:SimpleForm>
    </Panel>
</mvc:View>
```

App.controller.js > index.js > sap.ui.require() callback > attachInit() callback > oModel > lastName

```
sap.ui.require(
  ["sap/ui/model/json/JSONModel", "sap/ui/core/mvc/XMLView"],
  function (JSONModel, XMLView) {
    "use strict";

    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
      // Create a JSON model from an object literal
      var oModel = new JSONModel({
        firstName: "Who",
        lastName: "are you",
        enabled: true,
        panelHeaderText: "Data Binding Basics",
      });
      // Assign the model object to the SAPUI5 core
      sap.ui.getCore().setModel(oModel);

      // Display the XML view called "App"
      new XMLView({
        viewName: "sap.ui.demo.db.view.App",
      }).placeAt("content");
    });
  }
```

If this is two way binding, when enabled becomes disabled, index.html should be updated(user input on one side and display standard data on the other)





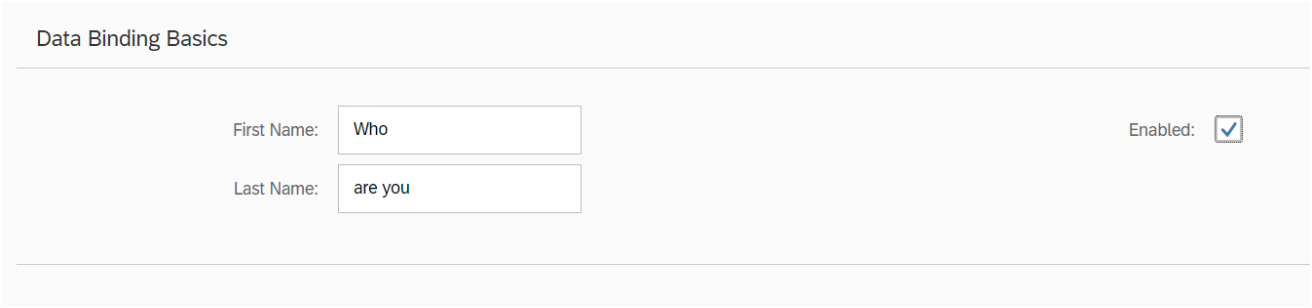
Two way binding is the default behaviour for JSON models.

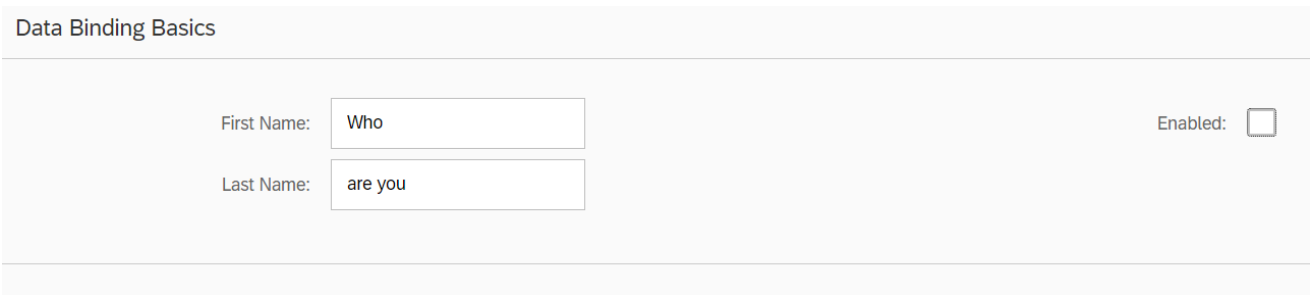
One Way Binding

But then, if it is one way binding, it will not work!! Set One Way Binding immediately after the creation of the model object.

App.controller.js > index.js > sap.ui.require() callback > attachInit() callback

```
    "sap/ui/model/BindingMode",
  ],
  function (JSONModel, XMLView, BindingMode) {
    "use strict";

    // Attach an anonymous function to the SAPUI5 'init' event
    sap.ui.getCore().attachInit(function () {
      // Create a JSON model from an object literal
      var oModel = new JSONModel({
        firstName: "Who",
        lastName: "are you",
        enabled: true,
        panelHeaderText: "Data Binding Basics",
      });

      oModel.setDefaultBindingMode(BindingMode.OneWay);

      // Assign the model object to the SAPUI5 core
      sap.ui.getCore().setModel(oModel);
```





If you alter the default binding mode of a model (as in the example above), then unless you explicitly say otherwise, all binding instances created after that point in time will use the altered binding mode.

Resource Model

Resource Model is SAP's way to handle multiple languages though it can be used to centralize parameters. All UI display parameters can be maintained in a separate file which can be centrally managed.
The resource bundle declared by default can support multiple languages. Which means that a default fallback should also be defined.

App.controller.js > index.js > sap.ui.require() callback > attachInit() callback

```
21        sap.ui.getCore().setModel(oModel);
22
23        // Create a resource bundle
24        var oResourceModel = new ResourceModel({
25          bundleName: "sap.ui.demo.db.i18n.i18n",
26          supportedLocales: ["", "de"],
27          fallbackLocale: "",
28        });
29
30        // Assign the model object to the SAPUI5 core using the name "i18n"
31        sap.ui.getCore().setModel(oResourceModel, "i18n");
32
```

This bundle will have the extension `.properties`

App.controller.js > i18n > i18n.properties

```
1   # Field labels
2   firstName=First Name
3   lastName=Last Name
4   enabled=Enabled
5
6   # Screen titles
7   panelHeaderText=Data Binding
```

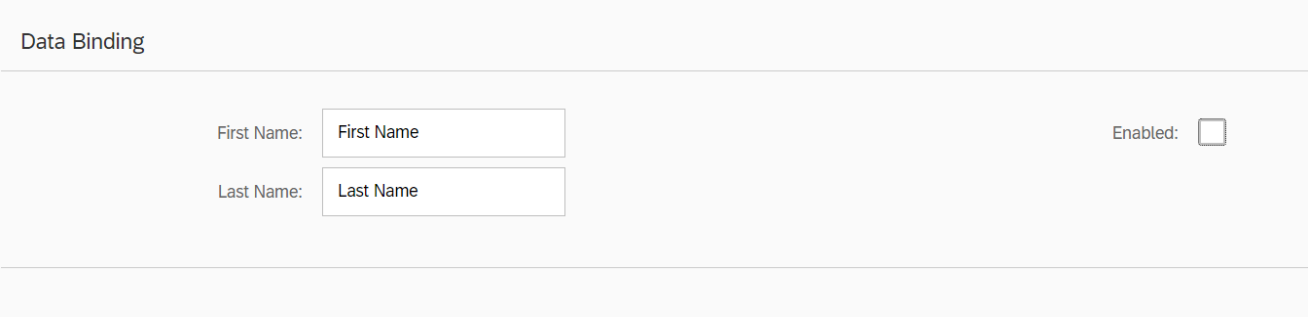
We can start with parameterizing the values.

App.controller.js > view > App.view.xml > mvc:View

```
1   <mvc:View
2       xmlns="sap.m"
3       xmlns:form="sap.ui.layout.form"
4       xmlns:mvc="sap.ui.core.mvc">
5       <Panel headerText="{/panelHeaderText}" class="sapUiResponsiveMargin" width="auto">
6           <form:SimpleForm editable="true" layout="ColumnLayout">
7               <Label text="First Name"/>
8               <Input value="{/firstName}" valueLiveUpdate="true" width="200px" enabled="{/enabled}"/>
9               <Label text="Last Name"/>
10              <Input value="{/lastName}" valueLiveUpdate="true" width="200px" enabled="{/enabled}"/>
11              <Label text="Enabled"/>
12              <CheckBox selected="{/enabled}"/>
```

App.controller.js > view > App.view.xml > ...

```
<mvc:View xmlns="sap.m"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc">
    <Panel headerText="{i18n>panelHeaderText}" class="sapUiResponsiveMargin" width="auto">
        <form:SimpleForm editable="true" layout="ColumnLayout">
            <Label text="First Name"/>
            <Input value="{i18n>firstName}" valueLiveUpdate="true" width="200px" enabled="{/enabled}"/>
            <Label text="Last Name"/>
            <Input value="{i18n>lastName}" valueLiveUpdate="true" width="200px" enabled="{/enabled}"/>
            <Label text="Enabled"/>
            <CheckBox selected="{i18n>enabled}"/>
        </form:SimpleForm>
    </Panel>
</mvc:View>
```

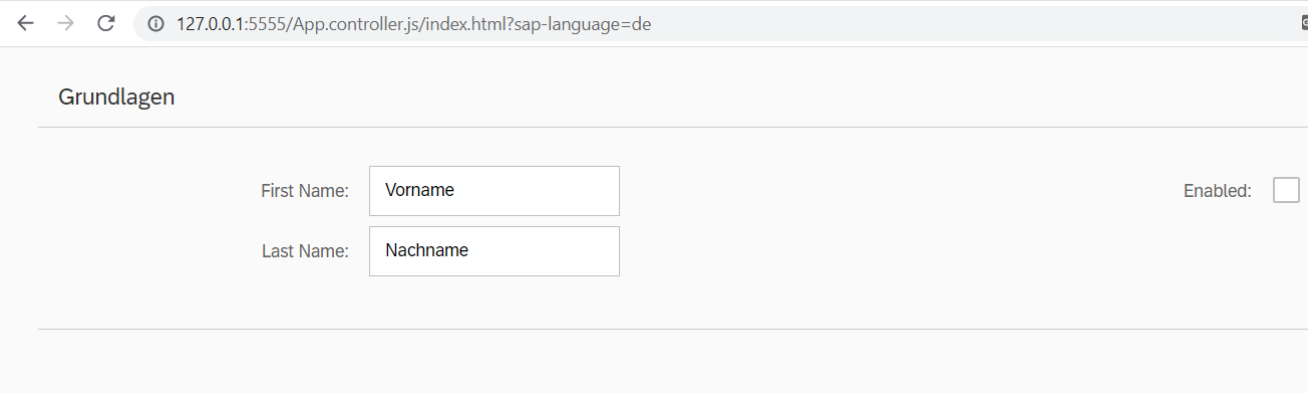


Resource Model with Multiple Languages

As DE is already defined, to extend the model to German, all you need to do is create a file for DE `i18n_de.properties` and fill it with the relevant data. Any new language you need it, define it in the resource model and use it. All you need to do is pass the URL with the hook `?sap-language=de`

App.controller.js > i18n > i18n_de.properties

```
# Field labels
firstName=Vorname
lastName=Nachname
enabled=Activiert

# Screen titles
panelHeaderText=Grundlagen
```



Alternately, change the default language of the browser to German

Hierarchical Models

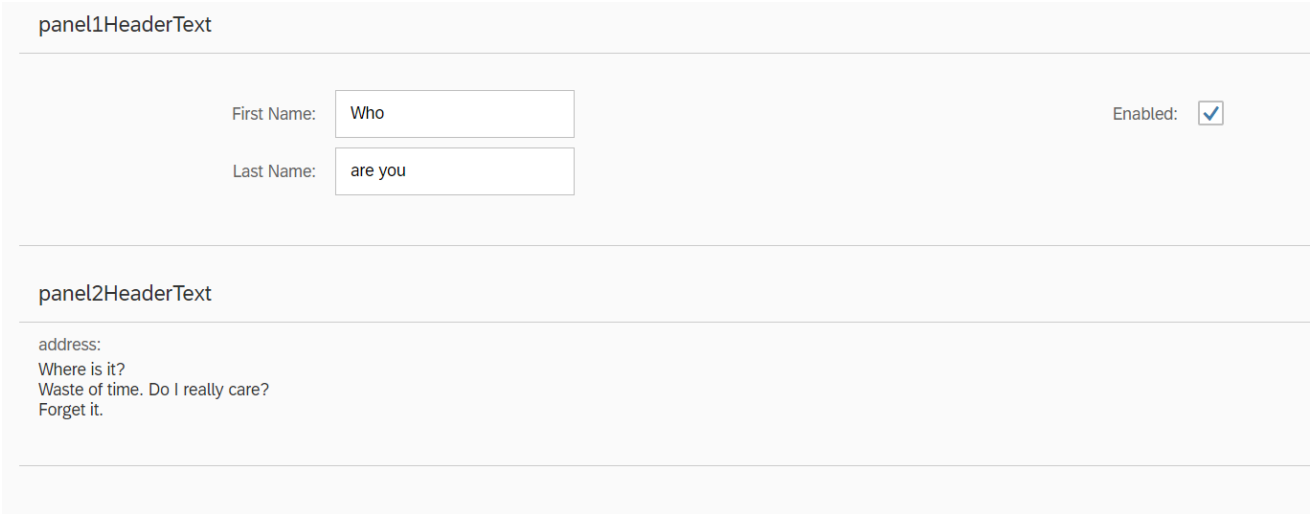
Except Resource Models, all other models are hierarchical.

```
var oModel = new JSONModel({
    firstName: "Who",
    lastName: "are you",
    enabled: true,
    panelHeaderText: "Data Binding Basics",
    address: {
        street: "Where is it?",
        city: "Do I really care?",
        country: "Forget it."

    }
});
```

App.controller.js > view > App.view.xml > mvc:View > Panel

```
<mvc:View xmlns="sap.m"
    xmlns:form="sap.ui.layout.form"
    xmlns:mvc="sap.ui.core.mvc"
    xmlns:l="sap.ui.layout">
    <Panel headerText="{i18n>panel2HeaderText}" class="sapUiResponsiveMargin" width="auto">
        <content>
            <l:VerticalLayout>
                <Label labelFor="address" text="{i18n>address}:"/>
                <FormattedText class="sapUiSmallMarginBottom"
                    htmlText="{/address/street}&lt;br>{/address/zip} {/address/city}&lt;br>{/address/country}"
                    id="address"
                    width="200px"/>
            </l:VerticalLayout>
        </content>
    </Panel>
</mvc:View>
```

If you notice the HTML text, you will notice the second level entities being called.



Data Formatting

Let's add an email to the visualization.

App.controller.js > controller > app.controller.js > ...

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

    return Controller.extend("sap.ui.demo.db.controller.App", {
      formatMail: function (sFirstName, sLastName) {
        var oBundle = this.getView().getModel("i18n").getResourceBundle();
        return mobileLibrary.URLHelper.normalizeEmail(
          sFirstName + "." + sLastName + "@example.com",
          oBundle.getText("mailSubject", [sFirstName]),
          oBundle.getText("mailBody")
        );
      },
    });
  }
);
```

Notice, multiple values are combined and reformatted.
Add the same to the view

App.controller.js > view > App.view.xml > mvc:View > Panel > content > l:VerticalLayout

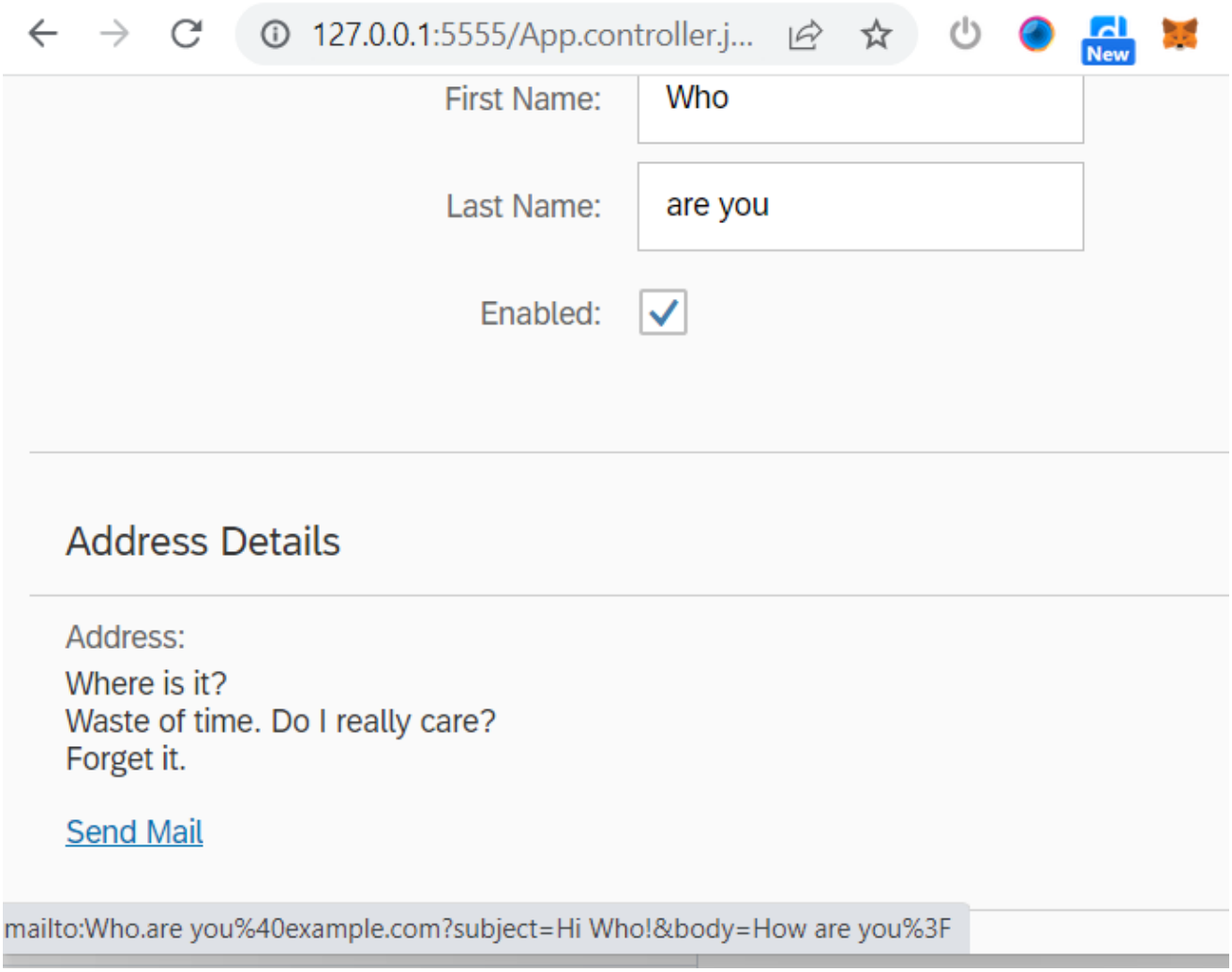
```
                <Link href="{
                        parts: [
                            '/firstName',
                            '/lastName'
                        ],
                        formatter: '.formatMail'
                    }" text="{i18n>sendEmail}"/>
            </l:VerticalLayout>
        </content>
    </Panel>
</mvc:View>
```

The `href` property of the `Link` element now contains an entire object inside the string value. In this case, the object has two properties:

- `parts`
  This is a JavaScript array in which each element is an object containing a `path` property. The number and order of the elements in this array corresponds directly to the number and order of parameters expected by the `formatMail` function.
- `formatter`
  A reference to the function that receives the parameters listed in the `parts` array. Whatever value is returned by the formatter function becomes the value set for this property. The dot (`formatMail`) at the beginning of the formatter tellsSAPUI5 to look for a `formatMail` function on

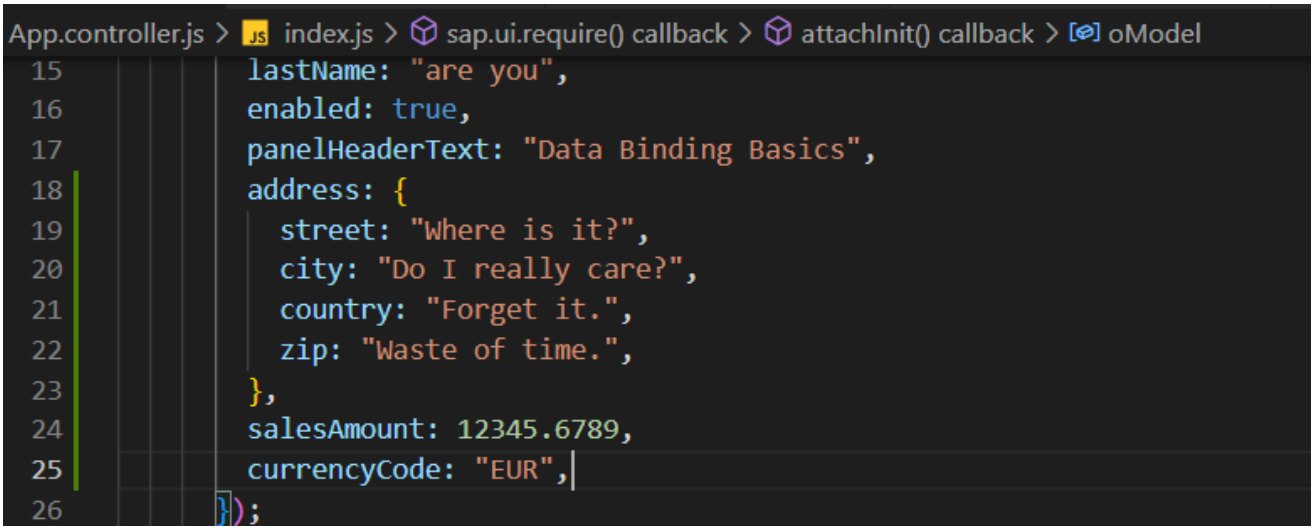
the controller instance of the view. If you do not use the dot, the function will be resolved by looking into the global namespace.

When using formatter functions, the binding is automatically switched to "one-way". So you can't use a formatter function for "two-way" scenarios, but you can use data types (which will be explained in the following steps).



Property Formatting

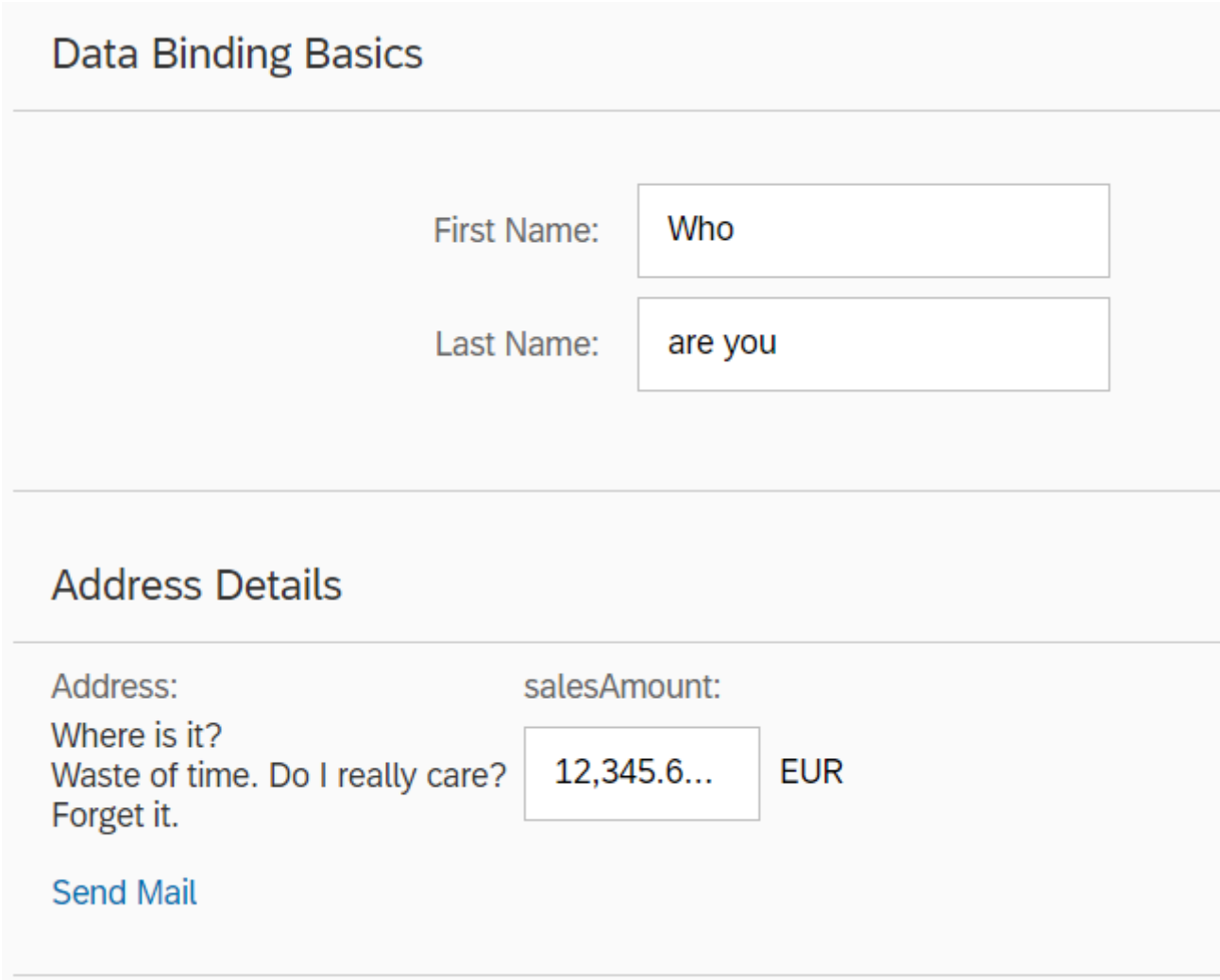
Let's add a currency field to this.



App.controller.js > view > App.view.xml > mvc:View > Panel > content > l:HorizontalLayout

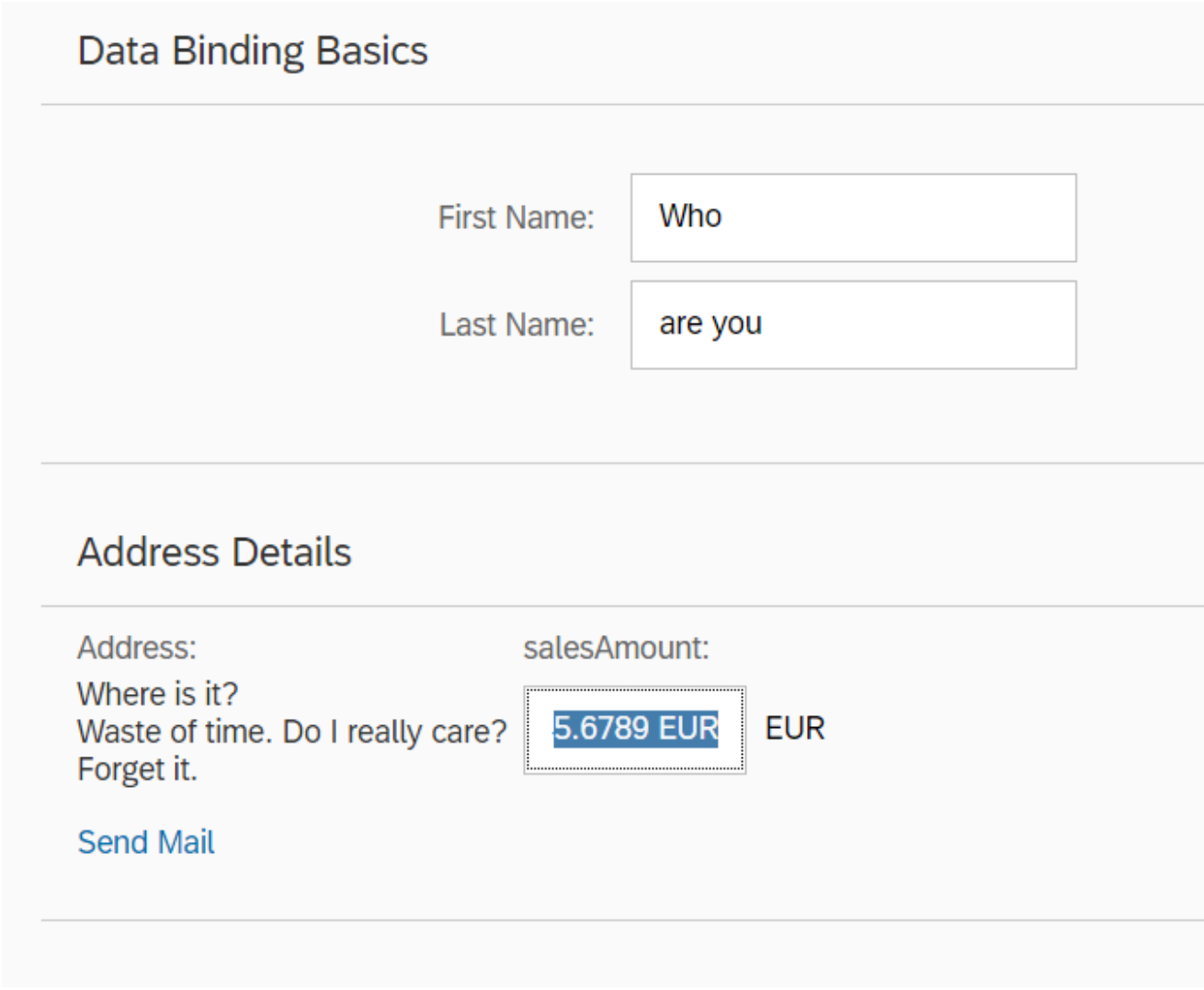
```
                    }" text="{i18n>sendEmail}"/>
                </l:VerticalLayout>
                <l:VerticalLayout>
                    <Label labelFor="salesAmount" text="{i18n>salesAmount}:"/>
                    <Input description="{/currencyCode}" enabled="{/enabled}" id="salesAmount" value="{
                            parts: [
                                {path: '/salesAmount'},
                                {path: '/currencyCode'}
                            ],
                            type: 'sap.ui.model.type.Currency',
                            formatOptions: {showMeasure: false}
                        }" width="200px"/>
                </l:VerticalLayout>
            </l:HorizontalLayout>
        </content>
    </Panel>
</mvc:View>
```

type: Currency will format it as a currency. Format Options will restrict whether currency should be shown or not.



If show measure is true

```
    <Input description="{/currencyCode}" enabled="{/enablec
            parts: [
                {path: '/salesAmount'},
                {path: '/currencyCode'}
            ],
            type: 'sap.ui.model.type.Currency',
            formatOptions: {showMeasure: true}
        }" width="200px"/>
</l:VerticalLayout>
HorizontalLayout>
```



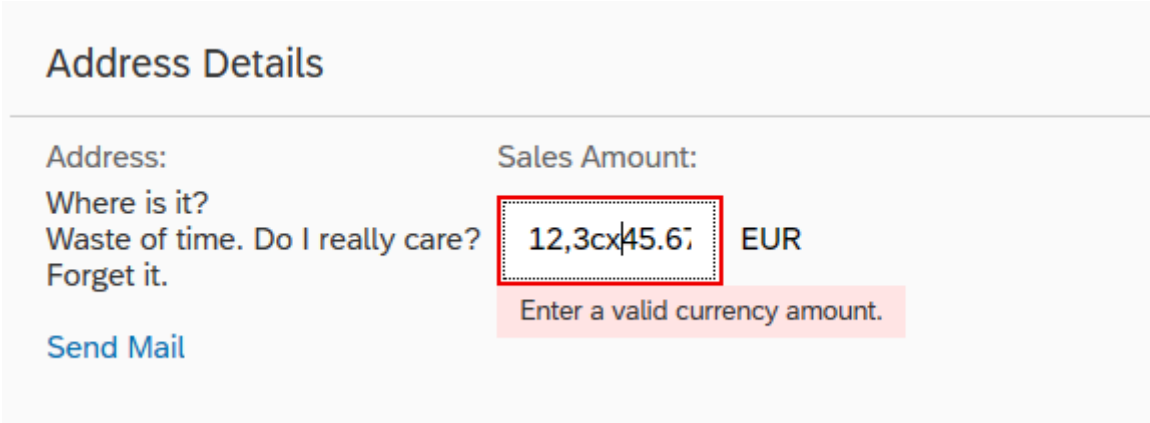
Validating the Data - Message Manager

Let's declare the XMLView against a variable name and register it with Message Manager

App.controller.js > index.js > sap.ui.require() callback

```
40
41        // Display the XML view called "App"
42        var oView = new XMLView({
43          viewName: "sap.ui.demo.db.view.App",
44        });
45
46        // Register the view with the message manager
47        sap.ui.getCore().getMessageManager().registerObject(oView, true);
48
49        // Insert the view into the DOM
50        oView.placeAt("content");
51      });
52    }
53  );
```

Enter a non-numeric value into the Sales Amount field and either press Enter or move the focus to a different UI control. This action triggers either the `onenter` or `onchange` event and then SAPUI5 executes the validation function belonging to the `sap.ui.model.type.Currency` data type.
Now that the view has been registered with the `MessageManager`, any validation error messages will be picked up by the `MessageManager`, which in turn checks its list of registered objects and then passes the error message back to the correct view for display.



Aggregation Binding with Templates

Aggregation binding (or "list binding") allows a control to be bound to a list within the model data and allows relative binding to the list entries by its child controls. It will automatically create as many child controls as are needed to display the data in the model using one of the following two approaches:

- Use template control that is cloned as many times as needed to display the data.
- Use a factory function to generate the correct control per bound list entry based on the data received at runtime.

Create a new data model based on a file.

App.controller.js > index.js > sap.ui.require() callback > attachInit() callback > oModel >

```
10      // Attach an anonymous function to the SAPUI5 'init' event
11      sap.ui.getCore().attachInit(function () {
12        // Create a JSON model from an object literal
13        var oProductModel = new JSONModel();
14        oProductModel.loadData("./model/Products.json");
15        sap.ui.getCore().setModel(oProductModel, "products");
16
```

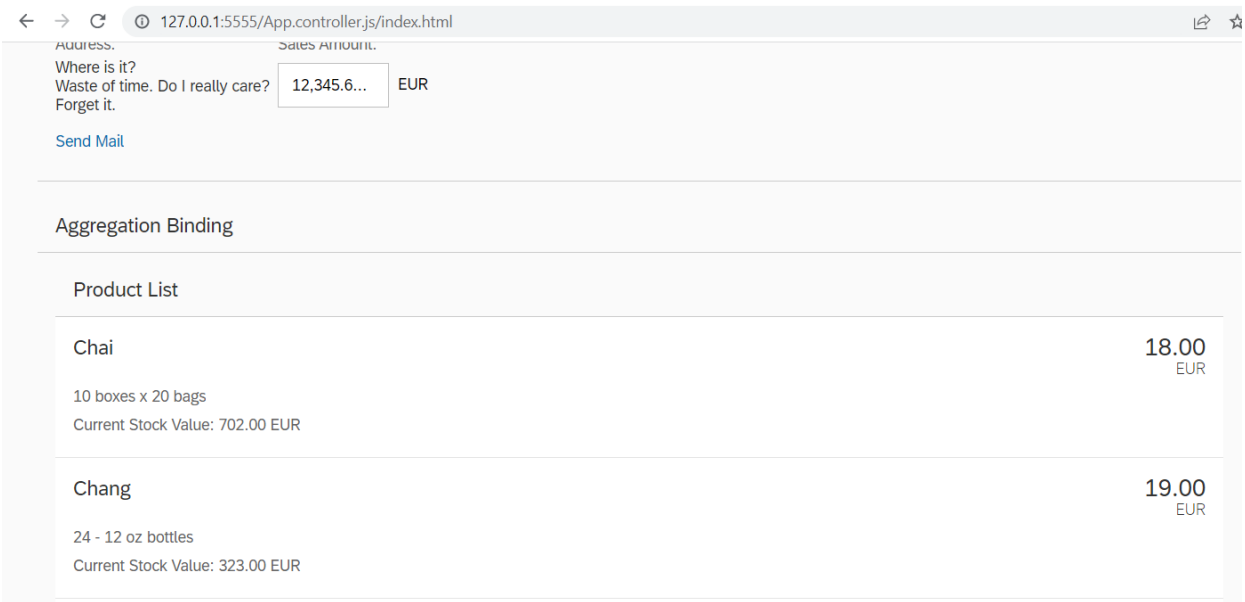
and add this as a new panel

App.controller.js > view > App.view.xml > mvc:View > Panel

```
43      <Panel headerText="{i18n>panel3HeaderText}" class="sapUiResponsiveMargin" width="auto">
44          <List headerText="{i18n>productListTitle}" items="{products>/Products}">
45              <items>
46                  <ObjectListItem title="{products>ProductName}" number="{
47                          parts: [
48                              {path: 'products>UnitPrice'},
49                              {path: '/currencyCode'}
50                          ],
51                          type: 'sap.ui.model.type.Currency',
52                          formatOptions: { showMeasure: false }
53                      }" numberUnit="{/currencyCode}">
54                      <attributes>
55                          <ObjectAttribute text="{products>QuantityPerUnit}"/>
56                          <ObjectAttribute title="{i18n>stockValue}" text="{
57                                  parts: [
58                                      {path: 'products>UnitPrice'},
59                                      {path: 'products>UnitsInStock'},
60                                      {path: '/currencyCode'}
61                                  ],
62                                  formatter: '.formatStockValue'
63                              }"/>
64                      </attributes>
65                  </ObjectListItem>
```

App.controller.js > controller > app.controller.js > ...

```
            );
          },
          formatStockValue: function (fUnitPrice, iStockLevel, sCurrCode) {
            var sBrowserLocale = sap.ui.getCore().getConfiguration().getLanguage();
            var oLocale = new Locale(sBrowserLocale);
            var oLocaleData = new LocaleData(oLocale);
            var oCurrency = new Currency(oLocaleData.mData.currencyFormat);
            return oCurrency.formatValue(
              [fUnitPrice * iStockLevel, sCurrCode],
              "string"
            );
          },
        });
      }
    );

```

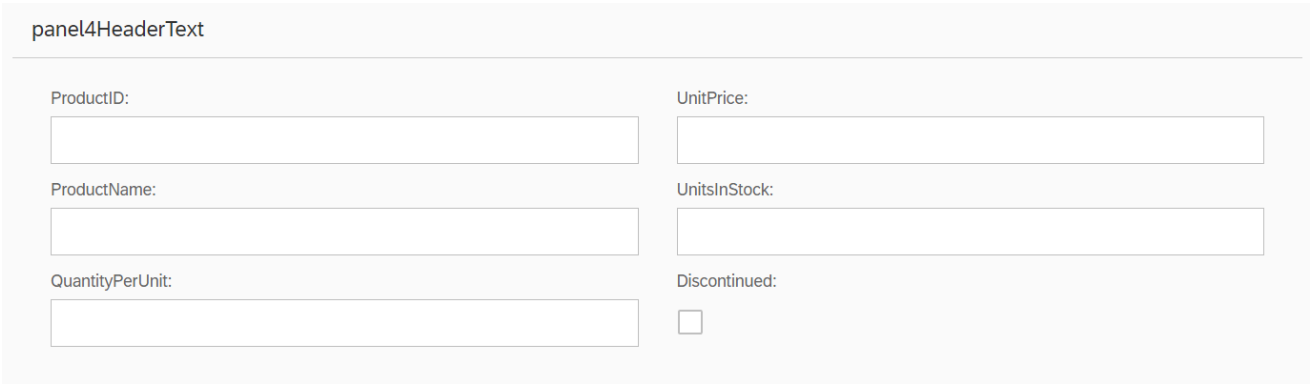


Element Binding

Let's have another view

App.controller.js > view > App.view.xml > mvc:View > Panel

```
    <Panel id="productDetailsPanel" headerText="{i18n>panel4HeaderText}" class="sapUiResponsiveMargin" width="auto">
        <form:SimpleForm editable="true" layout="ColumnLayout">
            <Label text="{i18n>ProductID}"/>
            <Input value="{products>ProductID}"/>

            <Label text="{i18n>ProductName}"/>
            <Input value="{products>ProductName}"/>

            <Label text="{i18n>QuantityPerUnit}"/>
            <Input value="{products>QuantityPerUnit}"/>

            <Label text="{i18n>UnitPrice}"/>
            <Input value="{products>UnitPrice}"/>

            <Label text="{i18n>UnitsInStock}"/>
            <Input value="{products>UnitsInStock}"/>

            <Label text="{i18n>Discontinued}"/>
            <CheckBox selected="{products>Discontinued}"/>
        </form:SimpleForm>
    </Panel>
</mvc:View>
```

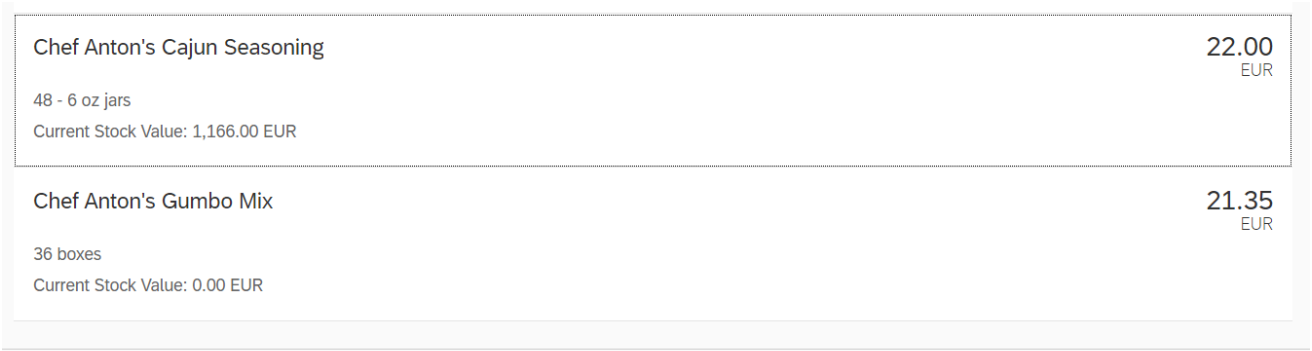


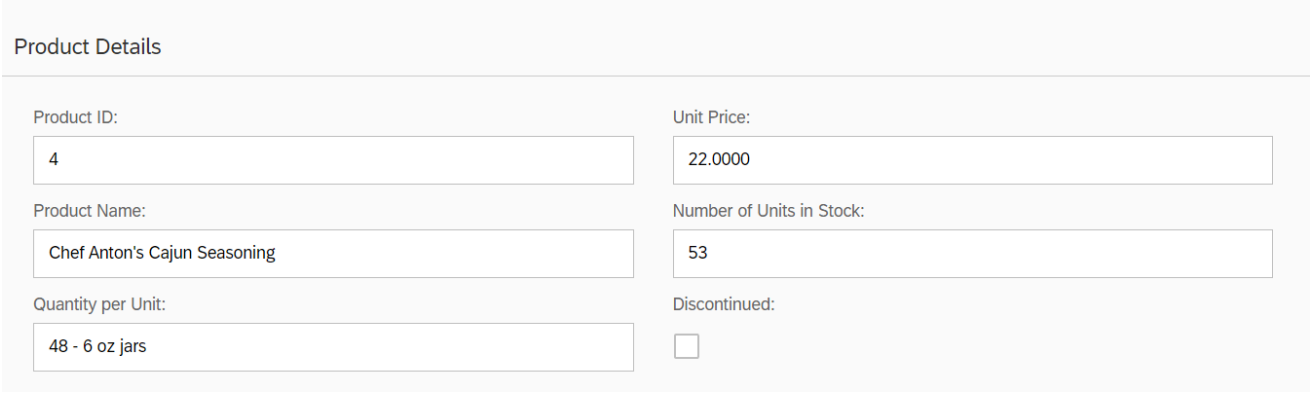
Add a trigger to the view

```
<Panel headerText="{i18n>panel3HeaderText}" class="sapUiResponsiveMargin" width="auto">
    <List headerText="{i18n>productListTitle}" items="{products>/Products}">
        <items>
            <ObjectListItem
            press=".onItemSelected"
                type="Active"
            title="{products>ProductName}" number="{
```

App.controller.js > controller > app.controller.js > ...

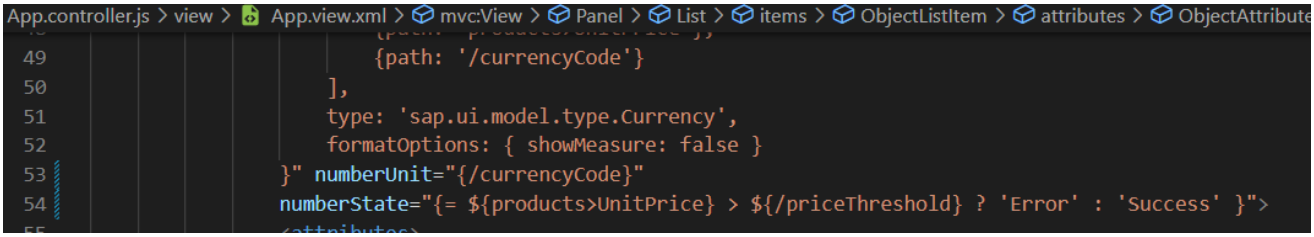
```
30          onItemSelected: function (oEvent) {
31            var oSelectedItem = oEvent.getSource();
32            var oContext = oSelectedItem.getBindingContext("products");
33            var sPath = oContext.getPath();
34            var oProductDetailPanel = this.byId("productDetailsPanel");
35            oProductDetailPanel.bindElement({ path: sPath, model: "products" });
36          },
37        });
38    }
```



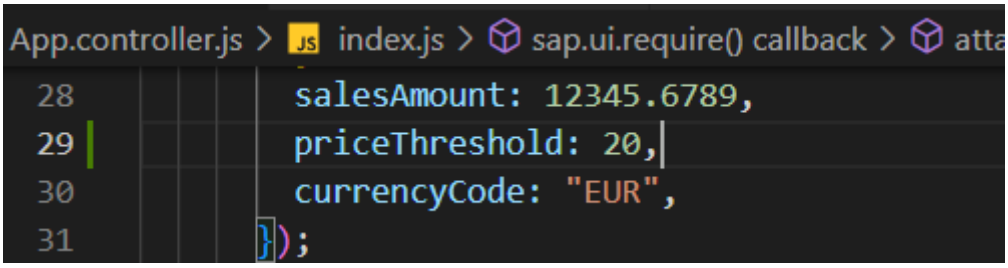


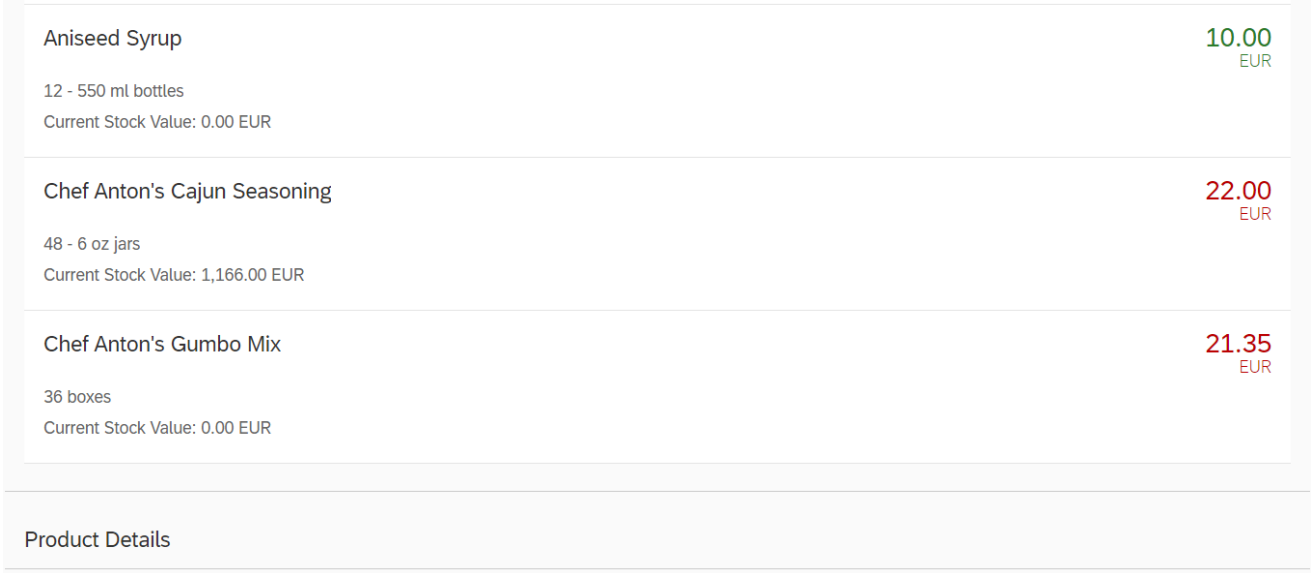
Expression Binding

Rather than displaying a value, pick the value as the output of an expression - calculation, condition etc. Let's introduce a condition



which is maintained in the controller





Look at the following two expressions:

- `numberState="{= ${products>UnitPrice} > ${/priceThreshold} ? 'Error' : 'Success' }"`
- `numberState="{= ${products>UnitPrice} <= ${/priceThreshold} ? 'Success' : 'Error' }"`

Logically, both expressions are identical; yet the first one works, and the second does not: it produces only an empty screen and an "Invalid XML" message in the browser's console. When an XML file is parsed, certain characters have a special (that is, high priority) meaning to the XML parser. When such characters are encountered, they are always interpreted to be part of the XML definition itself and not part of any other content that might exist within the XML document. As soon as the XML parser encounters one of these high-priority characters (in this case, a less-than (`<`) character), it will always be interpreted as the start of

a new XML tag – irrespective of any other meaning that character might have within the context of the expression. This is known as a syntax collision.
In this case, the collision occurs between the syntax of XML and the syntax of the JavaScript-like expression language used by SAPUI5.
Therefore, this statement fails because the less-than character is interpreted as the start of an XML tag: `numberState="{= ${products>UnitPrice} <= ${/priceThreshold} ? 'Success' : 'Error' }"`
This particular problem can be avoided in one of two ways:

- Reverse the logic of the condition (use "greater than or equal to" instead of "less than")
- Use the escaped value for the less-than character: `numberState="{= ${products>UnitPrice} &lt;= ${/priceThreshold} ? 'Success' : 'Error' }"`
  Since the use of an escaped character is not so easy to read, the preferred approach is to reverse the logic of the condition and use a greater-than character instead.
  The ampersand (`&`) character also has a high priority meaning to the XML parser. This character will always be interpreted to mean "The start of an escaped character". So if you wish to use the Boolean `AND` operator (`&&`) in a condition, you must escape both ampersand characters (`&&`).

Aggregation Binding with Factory Functions

Instead of hard-coding a single template control, we use a factory function to generate different controls based on the data received at runtime. This approach is much more flexible and allows complex or heterogeneous data to be displayed.

App.controller.js > view > App.view.xml > mvc:View > Panel

```
38                      }" width="200px"/>
39                  </l:VerticalLayout>
40              </l:HorizontalLayout>
41          </content>
42      </Panel>
43      <Panel headerText="{i18n>panel3HeaderText}" class="sapUiResponsiveMargin" width="auto">
44          <List id="ProductList" headerText="{i18n>productListTitle}" items="{
45              path: 'products>/Products',
46              factory: '.productListFactory'
47          }">
48              <dependents>
49                  <core:Fragment fragmentName="sap.ui.demo.db.view.ProductSimple" type="XML"/>
50                  <core:Fragment fragmentName="sap.ui.demo.db.view.ProductExtended" type="XML"/>
51              </dependents>
52          </List>
53      </Panel>
```

This is just an empty container and the layout is moved to two different fragments.

App.controller.js > view > ProductSimple.fragment.xml > core:FragmentDefinition

```xml
<core:FragmentDefinition
    xmlns="sap.m"
    xmlns:core="sap.ui.core">
    <StandardListItem
        id="productSimple"

        icon="sap-icon://warning"
        title="{products>ProductName} ({products>QuantityPerUnit})"
        info="{i18n>Discontinued}"
        type="Active"
        infoState="Error"
        press=".onItemSelected">
    </StandardListItem>
</core:FragmentDefinition>
```

App.controller.js > view > ProductExtended.fragment.xml > core:FragmentDefinition

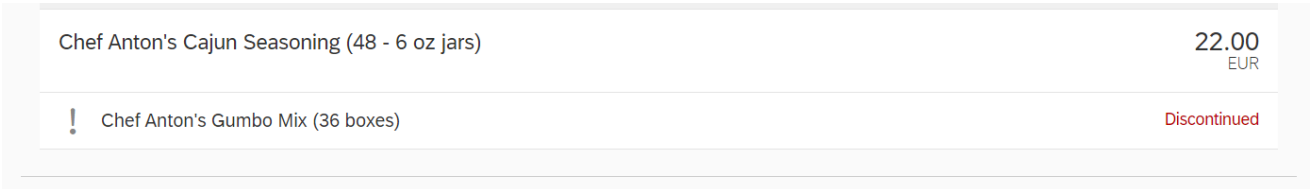
```xml
<core:FragmentDefinition xmlns="sap.m"
    xmlns:core="sap.ui.core">
    <ObjectListItem id="productExtended" title="{products>ProductName} ({products>QuantityPerUnit})" number="{
            parts: [
                {path: 'products>UnitPrice'},
                {path: '/currencyCode'}
            ],
            type: 'sap.ui.model.type.Currency',
            formatOptions : {
                showMeasure : false
            }
        }" type="Active" numberUnit="{/currencyCode}" press=".onItemSelected">
    </ObjectListItem>
</core:FragmentDefinition>
```

The controlling logic in this case is maintained through productListFactory

App.controller.js > controller > app.controller.js > sap.ui.define() callback

```js
productListFactory: function (sId, oContext) {
  var oUIControl;

  // Decide based on the data which dependent to clone
  if (
    oContext.getProperty("UnitsInStock") === 0 &&
    oContext.getProperty("Discontinued")
  ) {
    // The item is discontinued, so use a StandardListItem
    oUIControl = this.byId("productSimple").clone(sId);
  } else {
    // The item is available, so we will create an ObjectListItem
    oUIControl = this.byId("productExtended").clone(sId);

    // The item is temporarily out of stock, so we will add a status
    if (oContext.getProperty("UnitsInStock") < 1) {
      oUIControl.addAttribute(
        new ObjectAttribute({
          text: {
            path: "i18n>outOfStock",
          },
        })
```

A factory function returns a control for the associated binding context, similar to the XML templates we have defined in the previous steps. The types of controls returned by this factory function must suit the items aggregation of the `sap.m.List` object. In this case, we return either a `StandardListItem` or an `ObjectListItem` based on the data stored in the context of the item to be created.
We decide which type of control to return by checking the current stock level and whether or not the product has been discontinued. For both options, we prepare and load an XML fragment so that we can define the view logic declaratively and assign the current controller. If the stock level is zero and the product has also been discontinued, then we use the `ProductSimple` XML fragment, otherwise the `ProductExtended` XML fragment.
The XML fragments need to be loaded only once for each case, so we create a Singleton by storing a helper variable on the controller and only loading it once. For each item of the list, we clone the corresponding control stored on the controller. This method creates a fresh copy of a control that we can bind to the context of the list item. Please note: In a factory function, you are responsible for the life cycle of the control you create.
If the product is not discontinued but the stock level is zero, we are temporarily out of stock. In this case, we add a single `ObjectAttribute` that adds the Out of Stock message to the control using JavaScript. Similar to declarative definitions in the XML view or fragments, we can bind properties using data binding syntax. In this case, we bind the text to a property in the resource bundle. Since the `Attribute` is a child of the list item, it has access to all assigned models and the current binding context.
Finally, we return the control that is displayed inside the list.



OData

OData is a standard protocol for creating and consuming data by using simple HTTP and REST APIs for create, read, update, delete (CRUD) operations.
Documentation · OData - the Best Way to REST
Basic Tutorial · OData - the Best Way to REST
The OData V4 model supports the following:

- Read access

- Updating properties of OData entities (in entity sets and contained entities) via two-way-binding
- Deleting entities
- Operation (function and action) execution
- Grouping data requests in a batch request
- Server-side sorting and filtering

This time, let's use a standard template as entry point.
[OData V4 - Step 1 - Set Up the Initial App - Samples - Demo Kit - SAPUI5 SDK (ondemand.com)]
What matters is the dataSources block in manifest.json

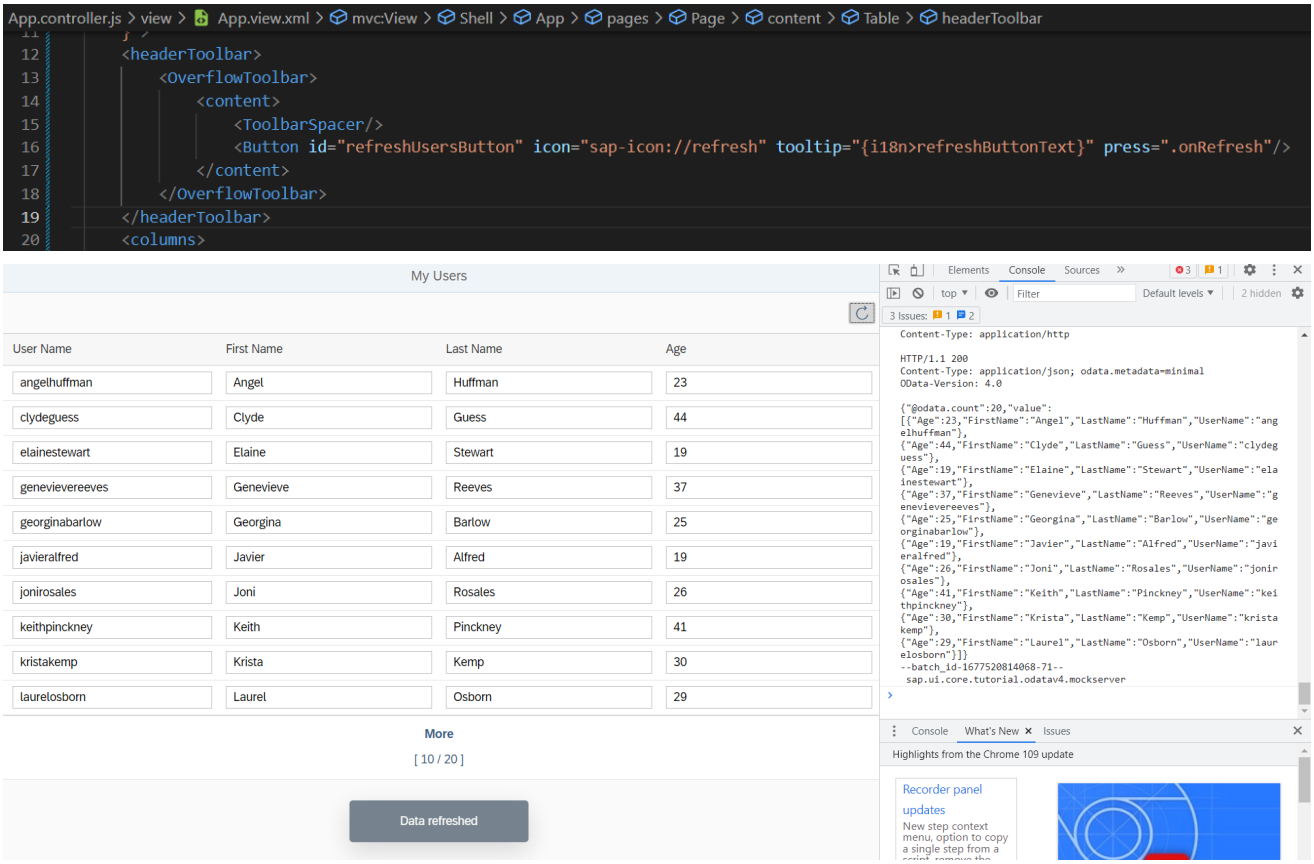
```
    "title": "{{appTitle}}",
    "description": "{{appDescription}}",
    "dataSources": {
        "default": {
            "uri": "https://services.odata.org/TripPinRESTierService/(S(id))/",
            "type": "OData",
            "settings": {
                "odataVersion": "4.0"
            }
        }
    }
},
```

Understanding how data is loaded

But, how does this communicate with the server?
Add a message box on refresh

```
    onRefresh: function () {
      var oBinding = this.byId("peopleList").getBinding("items");

      if (oBinding.hasPendingChanges()) {
        MessageBox.error(this._getText("refreshNotPossibleMessage"));
        return;
      }
      oBinding.refresh();
      MessageToast.show(this._getText("refreshSuccessMessage"));
    },

    _getText: function (sTextId, aArgs) {
      return this.getOwnerComponent()
        .getModel("i18n")
        .getResourceBundle()
        .getText(sTextId, aArgs);
    },
  });
```

and add a button to refresh.



The loaded message is displayed as console log.

Note:

`data-sap-ui-oninit="module:sap/ui/core/tutorial/odatav4/initMockServer"` is what blocks the CORS issue by invoking the mock server in index.html

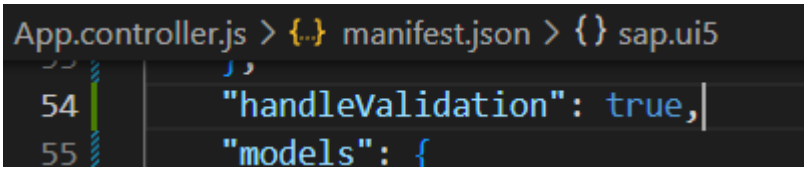
Server request is raised against

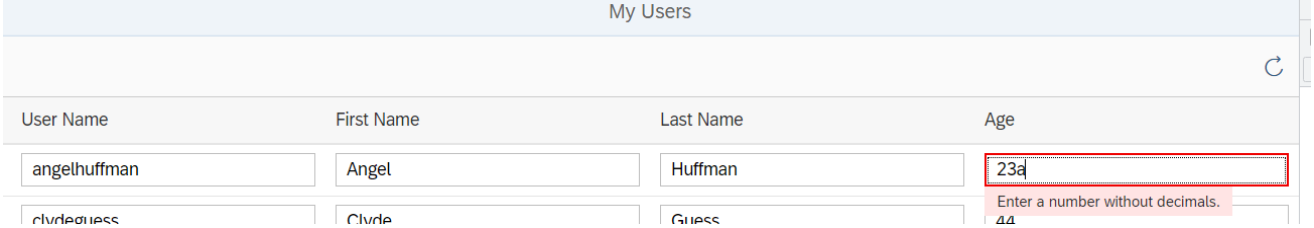
`https://services.odata.org/TripPinRESTierService/(S(id))/$metadata`

The URL contains the session ID `(S(id))`. Since the public TripPin service can be used by multiple persons at the same time, the session ID separates read and write requests from different sources. You could use a different ID or request the service without a specified session ID. In the latter case, you will get a response with a new, random session ID.

Automatic Data Type Detection

handleValidation = true in manifest.json makes sure that message manager will detect any errors.





Filtering, Sorting, and Counting

Create an onSearch and onSort function

App.controller.js > controller > App.controller.js > sap.ui.define() callback

```
onSearch : function () {
  var oView = this.getView(),
    sValue = oView.byId("searchField").getValue(),
    oFilter = new Filter("LastName", FilterOperator.Contains, sValue);

  oView.byId("peopleList").getBinding("items").filter(oFilter, FilterType.Application);
},

onSort : function () {
  var oView = this.getView(),
    aStates = [undefined, "asc", "desc"],
    aStateTextIds = ["sortNone", "sortAscending", "sortDescending"],
    sMessage,
    iOrder = oView.getModel("appView").getProperty("/order");

  iOrder = (iOrder + 1) % aStates.length;
  var sOrder = aStates[iOrder];

  oView.getModel("appView").setProperty("/order", iOrder);
  oView.byId("peopleList").getBinding("items").sort(sOrder && new Sorter("LastName", sOrder === "desc"));

  sMessage = this._getText("sortMessage", [this._getText(aStateTextIds[iOrder])]);
  MessageToast.show(sMessage);
  },
```

And initiate the counter as 0.

```
onInit: function () {
  var oJSONData = {
      busy: false,
      order: 0,
```

and set `$count` as true. We add the `$count : true` parameter to tell the OData service to send the number of entities. With this setting, we automatically get the full number of entities (20) and the number of displayed entities (10) beneath the More button.

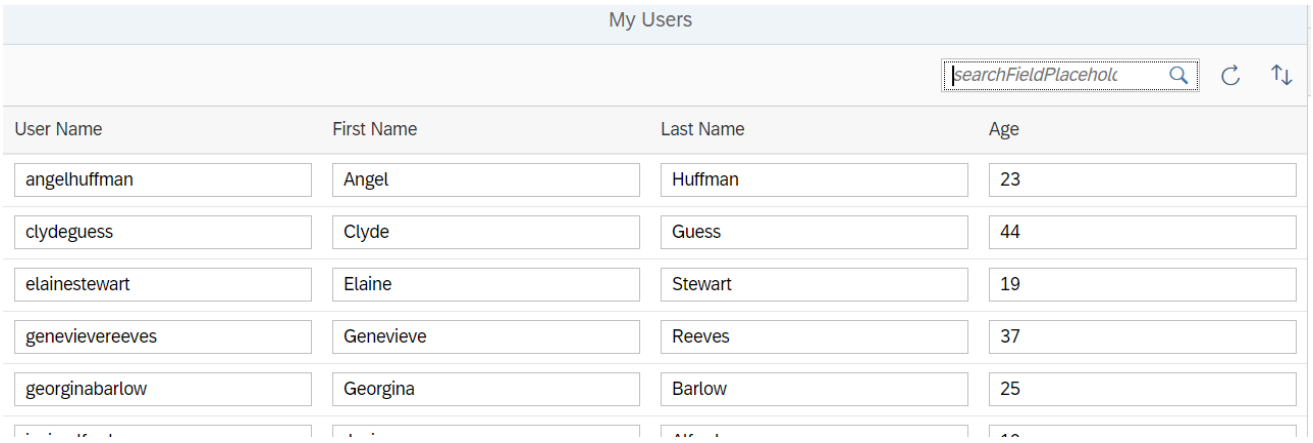
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page

```
<content>
    <Table id="peopleList" growing="true" growingThreshold="10" items="{
            path: '/People',
            parameters: {
                $count: true
            }
        }">
        <headerToolbar>
```

And add the search and sort functions

```
<headerToolbar>
    <OverflowToolbar>
        <content>
            <ToolbarSpacer/>
            <SearchField
                id="searchField"
                width="20%"
                placeholder="{i18n>searchFieldPlaceholder}"
                search=".onSearch"/>
            <Button id="refreshUsersButton" icon="sap-icon://refresh" tooltip="
        </content>
```

```
    <Button
    id="sortUsersButton"
    icon="sap-icon://sort"
    tooltip="{i18n>sortButtonText}"
    press="onSort"/>
</content>
```



Request Batch Grouping

Batch Grouping won't impact user experience when data is less but when data is large or when number of requests are huge, Batch groups are used to group multiple requests into one server request to improve the overall performance. All you need to do is set groupID to auto in the data source.


```
GET People?
$count=true&$select=Age,FirstName,LastName,UserName&$skip=0&$top=10
HTTP/1.1
Accept:application/json;odata.metadata=minimal;IEEE754Compatible=tr
ue
Accept-Language:en-IN
Content-Type:application/json;charset=UTF-8;IEEE754Compatible=true


--batch_id-1677611303586-21--
Group ID: $auto
 sap.ui.core.tutorial.odatav4.mockserver
```

```
2023-03-01 00:38:24.432699 Mockserver: Sent response with return code 200 - Response headers: {"Content-Type":"multipart/mixed;boundary=batch_id-1677611303586-21","OData-Version":"4.0"}
```

Log-dbg.js:497

```
{"@odata.count":20,"value":
[{"Age":23,"FirstName":"Angel","LastName":"Huffman","UserName":"ang
elhuffman"},
{"Age":44,"FirstName":"Clyde","LastName":"Guess","UserName":"clydeg
uess"},
{"Age":19,"FirstName":"Elaine","LastName":"Stewart","UserName":"ela
inestewart"},
{"Age":37,"FirstName":"Genevieve","LastName":"Reeves","UserName":"g
enevievereeves"},
{"Age":25,"FirstName":"Georgina","LastName":"Barlow","UserName":"ge
orginabarlow"},
{"Age":19,"FirstName":"Javier","LastName":"Alfred","UserName":"javi
eralfred"},
{"Age":26,"FirstName":"Joni","LastName":"Rosales","UserName":"jonir
osales"},
{"Age":41,"FirstName":"Keith","LastName":"Pinckney","UserName":"kei
thpinckney"},
{"Age":30,"FirstName":"Krista","LastName":"Kemp","UserName":"krista
kemp"},
{"Age":29,"FirstName":"Laurel","LastName":"Osborn","UserName":"laur
elosborn"}]}
--batch_id-1677612312031-21--
```


However, one should note that this behaviour is present in SAPUI5 by default.

Create and Edit

Replace the onInit method.

App.controller.js > controller > App.controller.js > sap.ui.define() callback > onInit

```
28        onInit: function () {
29          var oMessageManager = sap.ui.getCore().getMessageManager(),
30            oMessageModel = oMessageManager.getMessageModel(),
31            oMessageModelBinding = oMessageModel.bindList(
32              "/",
33              undefined,
34              [],
35              new Filter("technical", FilterOperator.EQ, true)
36            ),
37            oViewModel = new JSONModel({
38              busy: false,
39              hasUIChanges: false,
40              usernameEmpty: true,
41              order: 0,
42            });
43          this.getView().setModel(oViewModel, "appView");
44          this.getView().setModel(oMessageModel, "message");
45
46          oMessageModelBinding.attachChange(this.onMessageBindingChange, this);
47          this._bTechnicalErrors = false;
48        },
49
```

The `appView` model receives two additional properties, which we will use to control whether certain controls in the view are enabled or visible during user entries. We also make the `MessageModel` available to the view and add

a `ListBinding`. When the OData service reports errors while writing data, the OData Model adds them to the `MessageModel` as technical messages. Therefore we apply a filter to the `ListBinding`. We register our own handler to the `change` event of that `ListBinding` in order to capture any errors.

```
    _setUIChanges: function (bHasUIChanges) {
      if (this._bTechnicalErrors) {
        // If there is currently a technical error, then force 'true'.
        bHasUIChanges = true;
      } else if (bHasUIChanges === undefined) {
        bHasUIChanges = this.getView().getModel().hasPendingChanges();
      }
      var oModel = this.getView().getModel("appView");
      oModel.setProperty("/hasUIChanges", bHasUIChanges);
    },
  });
```

`_setUIChanges` private method that lets us set the property `hasUIChanges` of the `appView` model. Unless there are currently technical messages in the `MessageModel` or it is called with a given value for its `bHasUIChanges` parameter, the method uses `ODataModel.hasPendingChanges`. That method returns `true` if there are any changes that have not yet been written to the service.

App.controller.js > controller > App.controller.js > sap.ui.define() callback

```
48        },
49        onCreate : function () {
50          var oList = this.byId("peopleList"),
51            oBinding = oList.getBinding("items"),
52            oContext = oBinding.create({
53              "UserName" : "",
54              "FirstName" : "",
55              "LastName" : "",
56              "Age" : "18"
57            });
58
59          this._setUIChanges();
60          this.getView().getModel("appView").setProperty("/usernameEmpty", true);
61
62          oList.getItems().some(function (oItem) {
63            if (oItem.getBindingContext() === oContext) {
64              oItem.focus();
65              oItem.setSelected(true);
66              return true;
67            }
68          });
69        },
```

We add the `onCreate` event handler that responds to the `press` event of the Add User button. We use the `create` method of the `ODataListBinding` API to create a new user with some initial data and insert it at the top of the table.
The `create` method returns the binding context of the new user. That context provides a `created` method which returns a `Promise`. The `Promise` is resolved when the new user is successfully transferred to the OData service.

App.controller.js > controller > App.controller.js > sap.ui.define() callback

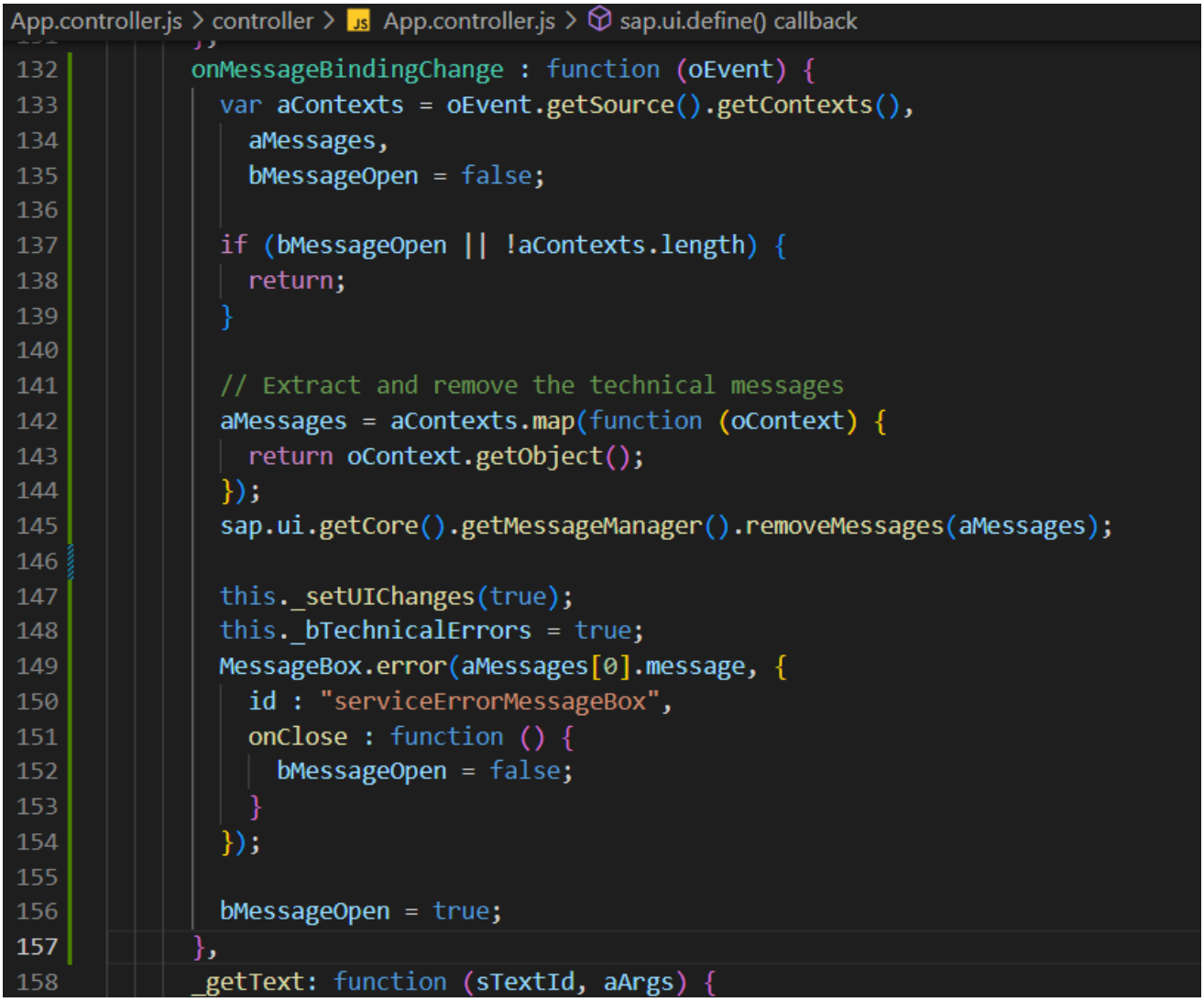
```
80        onSave: function () {
81          var fnSuccess = function () {
82            this._setBusy(false);
83            MessageToast.show(this._getText("changesSentMessage"));
84            this._setUIChanges(false);
85          }.bind(this);
86
87          var fnError = function (oError) {
88            this._setBusy(false);
89            this._setUIChanges(false);
90            MessageBox.error(oError.message);
91          }.bind(this);
92
93          this._setBusy(true); // Lock UI until submitBatch is resolved.
94          this.getView()
95            .getModel()
96            .submitBatch("peopleGroup")
97            .then(fnSuccess, fnError);
98          this._bTechnicalErrors = false; // If there were technical errors, a new save resets them.
99        },
```

`onSave` event handler, in which we call the `submitBatch` method of the `ODataModel` API to submit our changes. Because the changes we submit refer to the table, we need to pass the update group `peopleGroup` that we declared in the table binding.

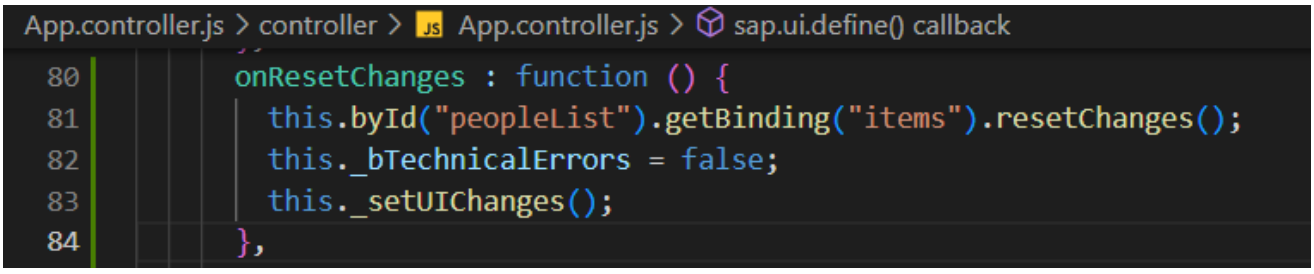
The `submitBatch` method returns a `Promise` that is rejected only if the batch request itself fails, for example, if the OData service is unavailable or if there were authorization problems. It is resolved in all other cases, also if the service returns errors for single requests that are contained in the batch request. Therefore, we have to implement the error handling for single requests differently.

```
  _setBusy: function (bIsBusy) {
    var oModel = this.getView().getModel("appView");
    oModel.setProperty("/busy", bIsBusy);
  },
});
```

a `_setBusy` private function to lock the whole UI while the data is submitted to the back end.



the event handler for the `change` event of the `ListBinding` to the `MessageModel`. We created the `ListBinding` with a filter to only include technical messages. That means that the `change` event will be fired with every change but only technical messages will have a binding context. In case of technical messages, we get the first one and display it as an error. We also make sure that the toolbar for saving or discarding changes stays visible. We delete the technical messages so that they do not accumulate.



The `onResetChanges` method handles discarding pending changes. It uses the `resetChanges` method of the `ODataListBinding` API to remove any such changes. Then it calls the `_setUIChanges` private method to enable the elements of the header toolbar again and hide the footer.

```
App.controller.js > controller > App.controller.js > sap.ui.define() callback
69        },
70        onInputChange : function (oEvt) {
71          if (oEvt.getParameter("escPressed")) {
72            this._setUIChanges();
73          } else {
74            this._setUIChanges(true);
75            if (oEvt.getSource().getParent().getBindingContext().getProperty("UserName")) {
76              this.getView().getModel("appView").setProperty("/usernameEmpty", false);
77            }
78          }
79        },
```

The `onInputChange` event handler manages entries in any of the `Input` fields and triggers updates to the `appView` model as needed. It does an extra check on the `UserName` field to make sure that users cannot be saved without a `UserName`. Otherwise the OData service would return errors because `UserName` is a mandatory field.

```
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page > content > Table
13                $$updateGroupId : 'peopleGroup'
```

add `$$updateGroupId: 'peopleGroup'` parameter to the table. This means that changes in the table are not sent to the service immediately but instead are collected until we explicitly send them.

```
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page > content > Table > headerToolbar > OverflowToolbar
20        <SearchField id="searchField" width="20%" placeholder="{i18n>searchFieldPlaceholder}"
21            enabled="{= !${appView>/hasUIChanges}}" search=".onSearch"/>
22        <Button id="refreshUsersButton" icon="sap-icon://refresh" tooltip="{i18n>refreshButtonText}" press=".onR
23        <Button id="sortUsersButton" icon="sap-icon://sort" tooltip="{i18n>sortButtonText}" press="onSort"/>
```

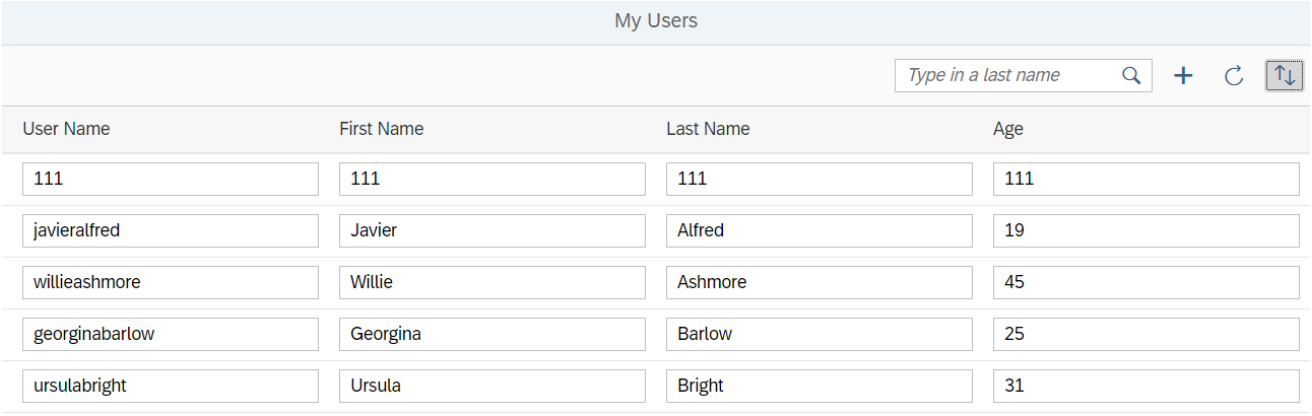
```
<Button id="addUserButton" icon="sap-icon://add" tooltip="{i18n>createButtonText}" press=".onCreate">
    <layoutData>
        <OverflowToolbarLayoutData priority="NeverOverflow"/>
    </layoutData>
</Button>
<Button id="refreshUsersButton" icon="sap-icon://refresh" tooltip="{i18n>refreshButtonText}" press=".onRefresh"/>
```

```
  <Button id="refreshUsersButton" icon="sap-icon://refresh" tooltip="{i18n>refreshButtonText}" press=".onRefresh"/>
  <Button id="sortUsersButton" icon="sap-icon://sort" enabled="{= !${appView>/hasUIChanges}}" tooltip="{i18n>sortButtonText}"
/content>
```

```
<ColumnListItem>
    <cells>
        <Input value="{UserName}" valueLiveUpdate="true" liveChange=".onInputChange"/>
    </cells>
    <cells>
        <Input value="{FirstName}" liveChange=".onInputChange"/>
    </cells>
    <cells>
        <Input value="{LastName}" liveChange=".onInputChange"/>
    </cells>
    <cells>
        <Input value="{Age}" valueLiveUpdate="true" liveChange=".onInputChange"/>
    </cells>
</ColumnListItem>
```

```
<footer>
    <Toolbar visible="{appView>/hasUIChanges}">
        <ToolbarSpacer/>
        <Button
            id="saveButton"
            type="Emphasized"
            text="{i18n>saveButtonText}"
            enabled="{= ${message>/}.length === 0 &amp;&amp; ${appView>/usernameEmpty} === false }"
            press=".onSave"/>
        <Button
            id="doneButton"
            text="{i18n>cancelButtonText}"
            press=".onResetChanges"/>
    </Toolbar>
</footer>
```

add a new Add User button to the overflow toolbar in the table header, and define a footer toolbar that contains Save and Cancel buttons that we can display or hide through the `appView` model. We can disable the Save button separately, for example when a user enters invalid data.
Finally, add the `liveChange="onInputChange"` event handler to the table cells to make it possible to react to user input. In addition, we set the `valueLiveUpdate` properties for the fields for `UserName` and `Age`. That makes sure that the SAPUI5 types validate the field content with each keystroke.



Delete

```
App.controller.js > controller > App.controller.js > sap.ui.define() callback
69      },
70      onDelete : function () {
71          var oContext,
72              oSelected = this.byId("peopleList").getSelectedItem(),
73              sUserName;
74
75          if (oSelected) {
76              oContext = oSelected.getBindingContext();
77              sUserName = oContext.getProperty("UserName");
78              oContext.delete().then(function () {
79                  MessageToast.show(this._getText("deletionSuccessMessage", sUserName));
80              }.bind(this), function (oError) {
81                  this._setUIChanges();
82                  if (oError.canceled) {
83                      MessageToast.show(this._getText("deletionRestoredMessage", sUserName));
84                      return;
85                  }
86                  MessageBox.error(oError.message + ": " + sUserName);
87              }.bind(this));
88              this._setUIChanges(true);
89          }
90      },
```

add the `onDelete` event handler to the controller. In the event handler, we check whether an item is selected in the table and if so, we retrieve the binding context of the selection and call its `delete` method. By doing this, the context is removed from the table on the client side and the deletion is stored as a pending change in the update group of the table's list binding. A call to `_setUIChanges` ensures that the `appView` model reflects the deletion as a pending change and that the Save button becomes enabled. The deletion will be submitted with all other changes related to the same update group once the Save button is pressed. If the deletion fails on the server side, or the changes are reset via API, the related entity is restored in the table automatically. To distinguish these two situations, the rejected error has `canceled` set to `true` in case of a reset.
And the view

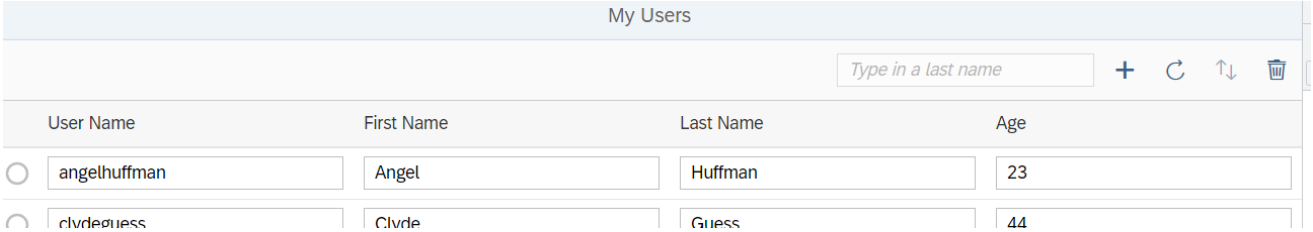
```
App.controller.js > view > App.view.xml > mvc:View > 
15              }" mode="SingleSelectLeft">
```

Change the `mode` of the table to `SingleSelectLeft` to make it possible to select a row.

```
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages >
28      <Button
29      id="deleteUserButton"
30      icon="sap-icon://delete"
31      tooltip="{i18n>deleteButtonText}"
32      press=".onDelete">
33      <layoutData>
34          <OverflowToolbarLayoutData priority="NeverOverflow"/>
35      </layoutData>
```

With the `OverflowToolbarLayoutData priority="NeverOverflow"` parameter, we

make sure that the button is always visible.



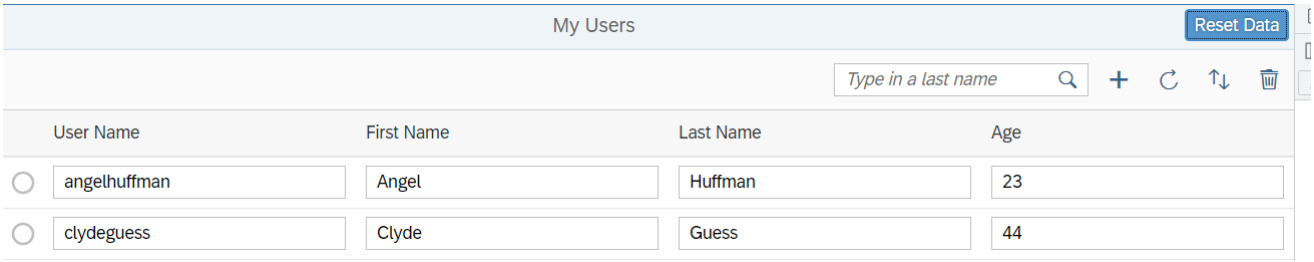
Reset Data

```
App.controller.js > controller > App.controller.js > sap.ui.define() callback
131     onResetDataSource : function () {
132       var oModel = this.getView().getModel(),
133         oOperation = oModel.bindContext("/ResetDataSource(...)");
134
135       oOperation.execute().then(function () {
136           oModel.refresh();
137           MessageToast.show(this._getText("sourceResetSuccessMessage"));
138         }.bind(this), function (oError) {
139           MessageBox.error(oError.message);
140         }
141       );
142     },
```

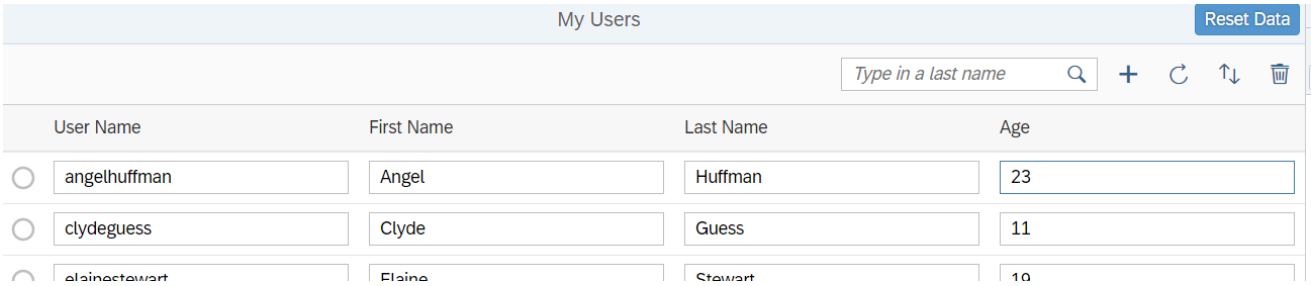
The `onResetDataSource` event handler calls the `ResetDataSource` action, which is an action of the TripPin OData service that resets the data of the service to its original state.
We call that action by first creating a deferred operation binding on the model. The `(...)` part of the binding syntax marks the binding as deferred. We use a deferred binding because we want to control when the action is executed. Since it is deferred, we need to explicitly call its `execute` method.
The execution is asynchronous, therefore the execute method returns a `Promise`. We attach simple success and error handlers to that `Promise` by calling its `then` method.

```
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page >
8                   <headerContent>
9                       <Button
10                          id="resetChangesButton"
11                          text="{i18n>resetChangesButtonText}"
12                          enabled="{= !${appView>/hasUIChanges}}"
13                          press="onResetDataSource"
14                          type="Emphasized">
15                      </Button>
16                  </headerContent>
```

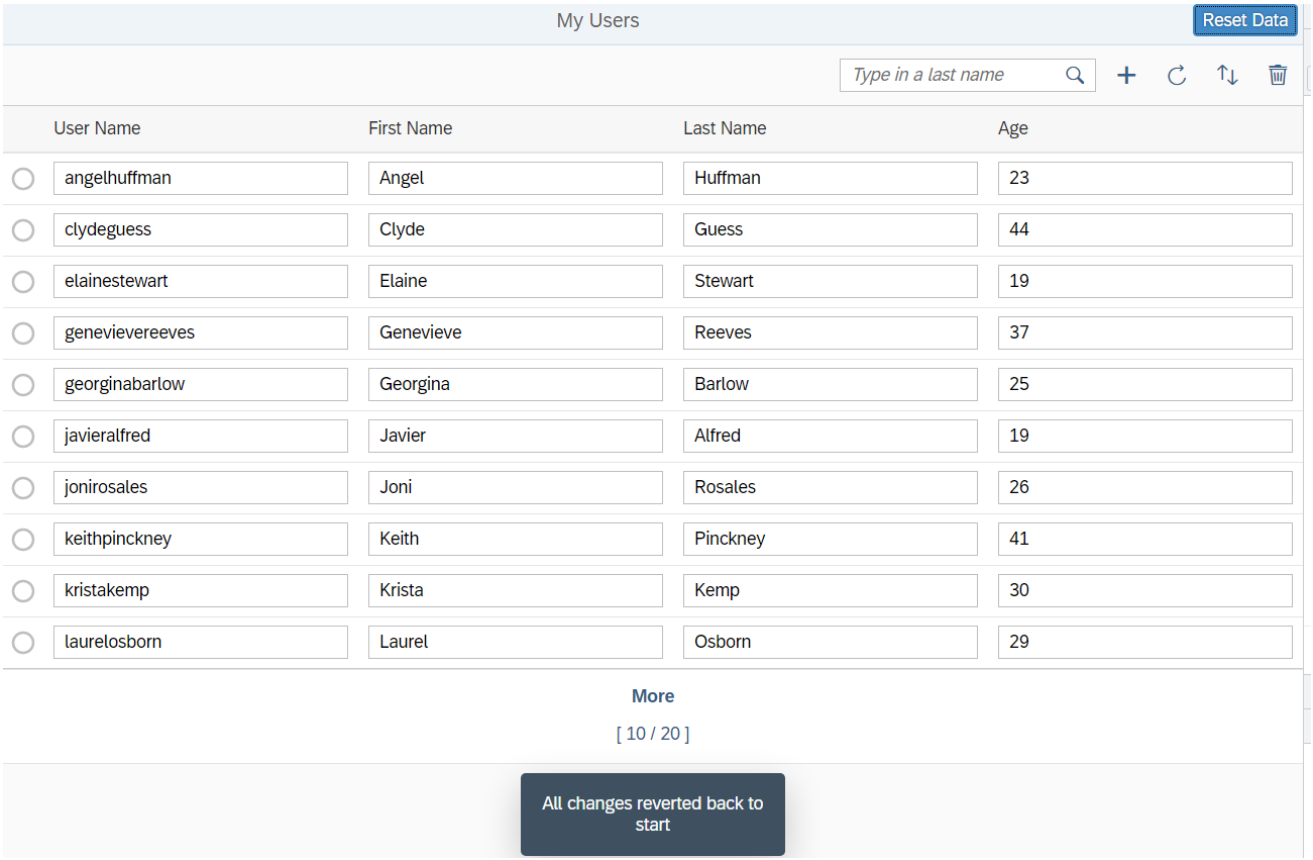
Before



After change



Reset



Clickable Detail Pane

```
App.controller.js > controller > App.controller.js > sap.ui.define() callback > function
230         onSelectionChange : function (oEvent) {
231             this._setDetailArea(oEvent.getParameter("listItem").getBindingContext());
232         },
233
234     ...
235         /**
236         * Toggles the visibility of the detail area
237         *
238         * @param {object} [oUserContext] - the current user context
239         */
240         _setDetailArea : function (oUserContext) {
241             var oDetailArea = this.byId("detailArea"),
242                 oLayout = this.byId("defaultLayout"),
243                 oSearchField = this.byId("searchField");
244
245             oDetailArea.setBindingContext(oUserContext || null);
246             // resize view
247             oDetailArea.setVisible(!!oUserContext);
248             oLayout.setSize(oUserContext ? "60%" : "100%");
249             oLayout.setResizable(!!oUserContext);
250             oSearchField.setWidth(oUserContext ? "40%" : "20%");
251         }
```

The `onSelectionChange` event handler retrieves the context of the selected list item and passes it to a new `_setDetailArea` function. Within `_setDetailArea`, the given context is passed as binding context for the semantic page detail area.

Afterwards the detail area is made visible and is resized.

The application also needs to close the detail area if its binding context is deleted. If the deleted context is restored after a failed DELETE request, or undeleted via `Context#resetChanges`, it could be shown in the detail area again, unless the user had selected another row in the meantime. Hence, we call `_setDetailArea` without a context once the context gets deleted, and with the restored context in the error handler of the `Context#delete` API.
In `_setDetailArea` we resize the view based on the given context in an appropriate way.

The idea is to split the pane into two and activate it based on selectionChange event handler

```
App.controller.js > view > App.view.xml > mvc:View > Shell > App > pages > Page > content > l:ResponsiveSplitter
16                  <content>
17                      <l:ResponsiveSplitter defaultPane="defaultPane">
18                          <l:PaneContainer orientation="Horizontal">
19                              <l:SplitPane id="defaultPane">
20                                  <l:layoutData>
21                                      <l:SplitterLayoutData id="defaultLayout" size="100%" resizable="false"/>
22                                  </l:layoutData>
23                                  <Table id="peopleList" growing="true" growingThreshold="10" items="{
24                              path: '/People',
25                              parameters: {
26                                  $count: true,
27                                  $$updateGroupId : 'peopleGroup'
28                              }
29                          }" mode="SingleSelectLeft" selectionChange=".onSelectionChange">
```

```
App.view.xml > mvc:View > Shell > App > pages > Page > content > l:ResponsiveSplitter > l:PaneContainer > l:SplitPane >
<f:formContainers>
    <f:FormContainer>
        <f:formElements>
            <f:FormElement label="{i18n>addressLabelText}">
                <f:fields>
                    <Text text="{HomeAddress/Address}" />
                </f:fields>
            </f:FormElement>
            <f:FormElement label="{i18n>cityLabelText}">
                <f:fields>
                    <Text text="{HomeAddress/City/Name}" />
                </f:fields>
            </f:FormElement>
            <f:FormElement label="{i18n>regionLabelText}">
                <f:fields>
                    <Text text="{HomeAddress/City/Region}" />
                </f:fields>
            </f:FormElement>
            <f:FormElement label="{i18n>countryLabelText}">
                <f:fields>
                    <Text text="{HomeAddress/City/CountryRegion}" />
                </f:fields>
            </f:FormElement>
        </f:formElements>
```

Several new namespaces are added to the `appView`. After the `<content>` and before the `<Table>` tag the first part of the `SplitPane` is added.

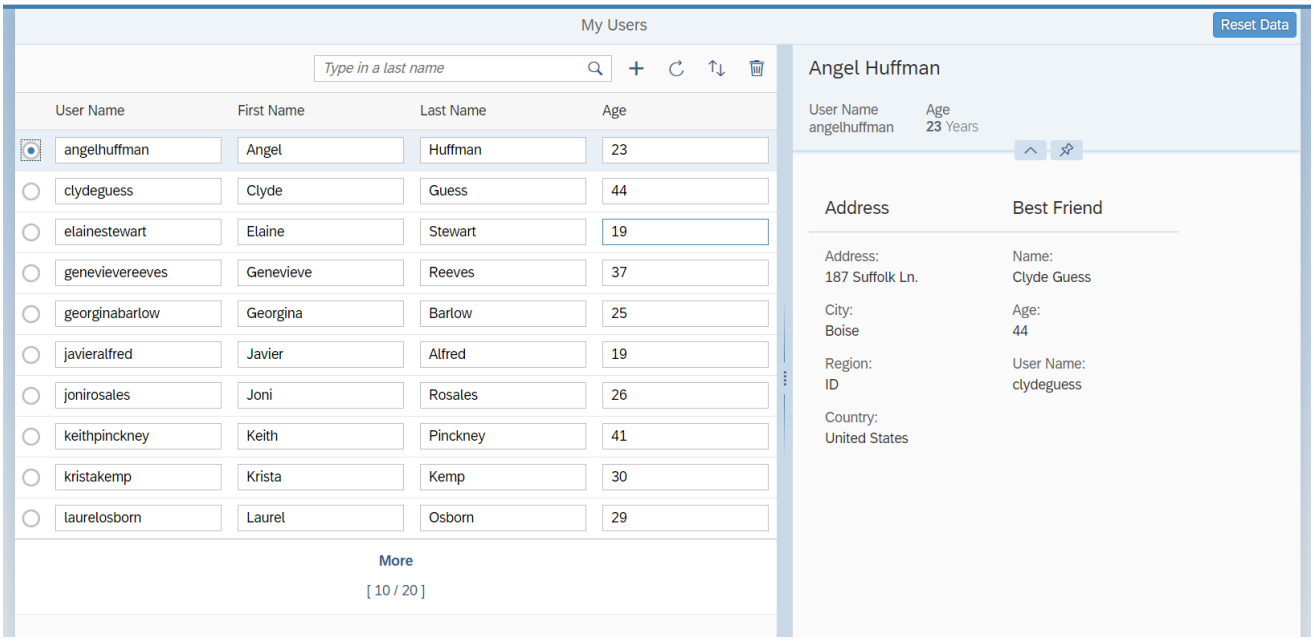
```
xmlns:l="sap.ui.layout"
xmlns:semantic="sap.f.semantic"
xmlns:f="sap.ui.layout.form"
xmlns:core="sap.ui.core">
```

We add a detail area after the user table. In the `appView` we add a splitter layout around the existing table. The first `SplitPane` contains the table with all users, and the second one contains the new detail area. It consists of two forms, one for the address information, and the other one for the best friend of the currently selected user. We also add the `onSelectionChange` event handler to the user table.

It is important that all bindings we introduced in the detail area are relative (property) bindings, so that we can reuse data of the list. This allows our application to share data like the user name or the age between the user selected in the table and the detail area. This helps to avoid redundant requests and to keep the data between the two areas in sync. Editing a property in the user table will thus automatically be reflected in the detail area as well.

One of the most vital parts of the data reuse functionality is the usage of the `autoExpandSelect` binding parameter. It permits us to put a tailored `$select` clause in the `GET` request, so that only missing properties are

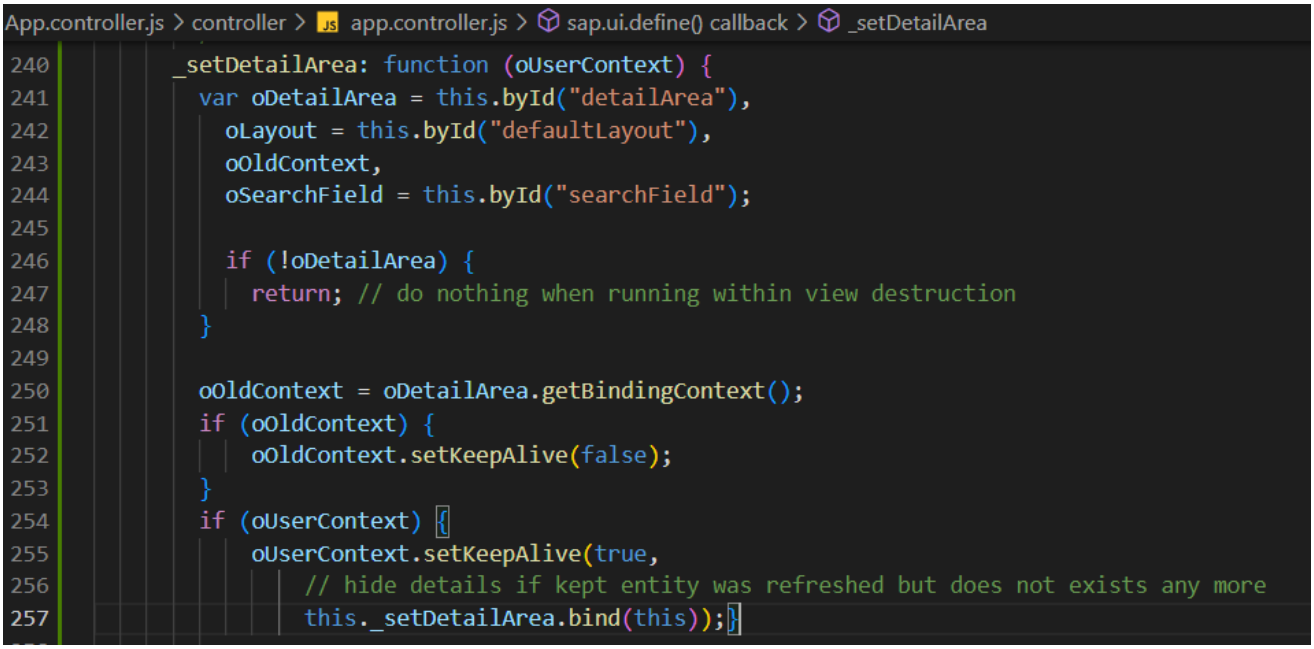
requested for display in the detail area.



Prevent Data Destruction

avoid unnecessary back-end requests by preventing the destruction of data shown in the detail area when sorting or filtering the list
We extend the logic of the `_setDetailArea` function. First, we check if there's an "old" binding context in the detail area. If so, the `keepAlive` for the old context is set to `false`.
For the new context we set `keepAlive` to `true` and add `_setDetailArea` as an `onBeforeDestroy` function to it, which hides the detail area when the user linked to it is deleted in the back end and the list is refreshed.
You can use the `Context#setKeepAlive` method to prevent the destruction of information shown in the detail area when the selected user is no longer part of the list from which the information was selected. This could otherwise happen if

you filter or sort the list.



Add Table with Additional Details in Detail Area

Nothing much, just add a table to the detail pane!!
. To this table we add a data binding for friends. It is important that we set the `$$ownRequest` binding parameter to `true`, so that the table containing all friends of the selected user makes its own OData requests separate from the request for best friend and best friend's address.

mvc:View > Shell > App > pages > Page > content > l:ResponsiveSplitter > l:PaneContainer > l:SplitPane > semantic:SemanticPage > seman

flex Aa ab .* 4 of 4

```
166                </f:Form>
167            </FlexBox>
168            <Table id="friendsTable" width="auto" items="{path: 'Friends',
169                    parameters: {
170                        $$ownRequest: true
171                    }}" noDataText="No Data" class="sapUiSmallMarginBottom">
172                <headerToolbar>
173                    <Toolbar>
174                        <Title text="Friends" titleStyle="H3" level="H3"/>
175                    </Toolbar>
176                </headerToolbar>
177                <columns>
178                    <Column>
179                        <Text text="User Name"/>
180                    </Column>
181                    <Column>
182                        <Text text="First Name"/>
183                    </Column>
184                    <Column>
185                        <Text text="Last Name"/>
186                    </Column>
187                    <Column>
188                        <Text text="Age"/>
```