WEEK 2 UNIT 5
BINDING CONTEXTS WITH ELEMENT BINDING

Please perform the exercises below in your app project as shown in the video.

Table of Contents

Preview

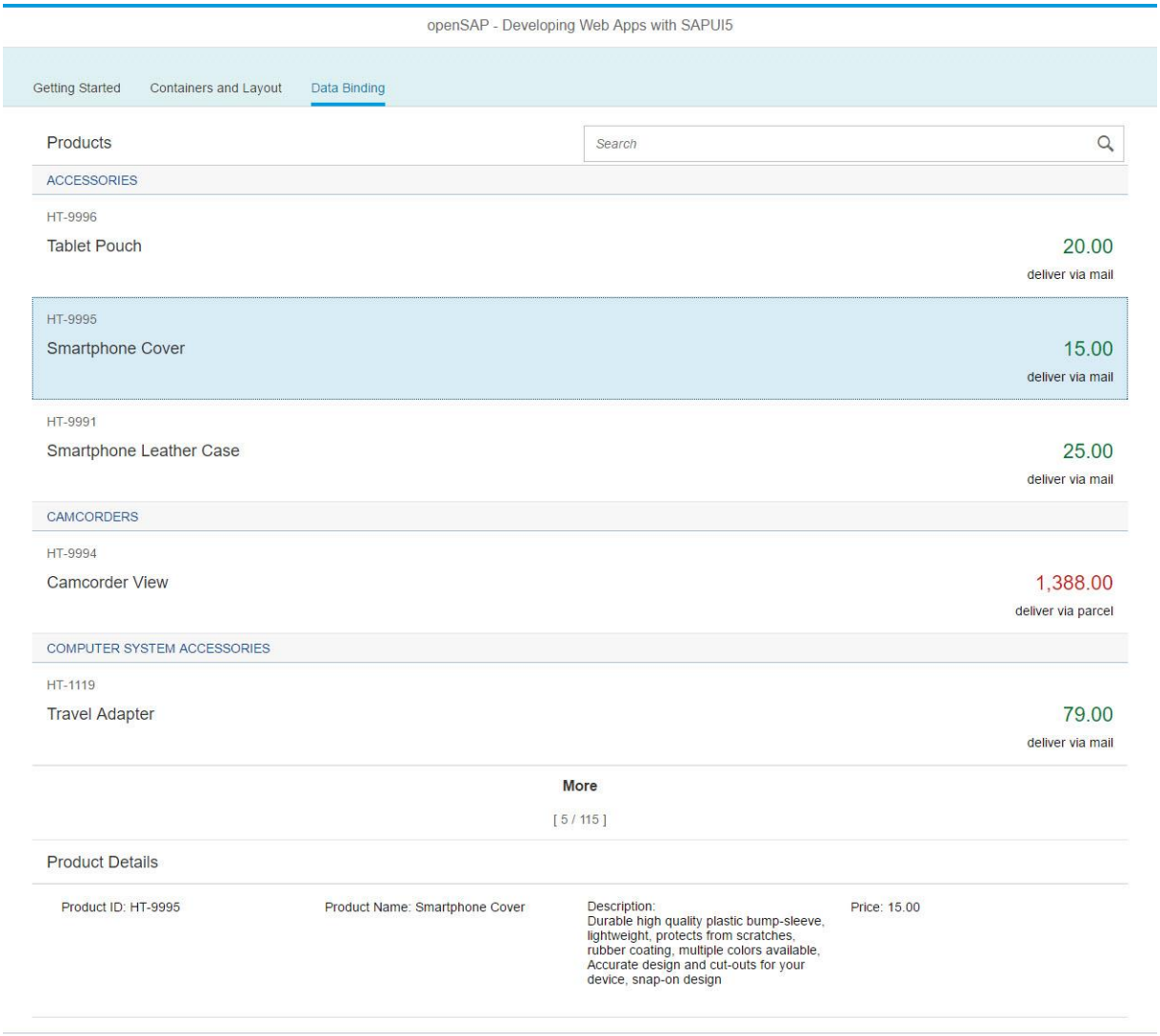

Figure 1 - Preview of the app after doing this unit's exercises

1 ADD PAGING TO THE LIST

In this step, we add paging functionality to the list in our data binding tab, in order to make the list not become too long. We want to add a panel below the list in the next step, and we do not want to scroll down so far every time we want to look at the panel.

Preview

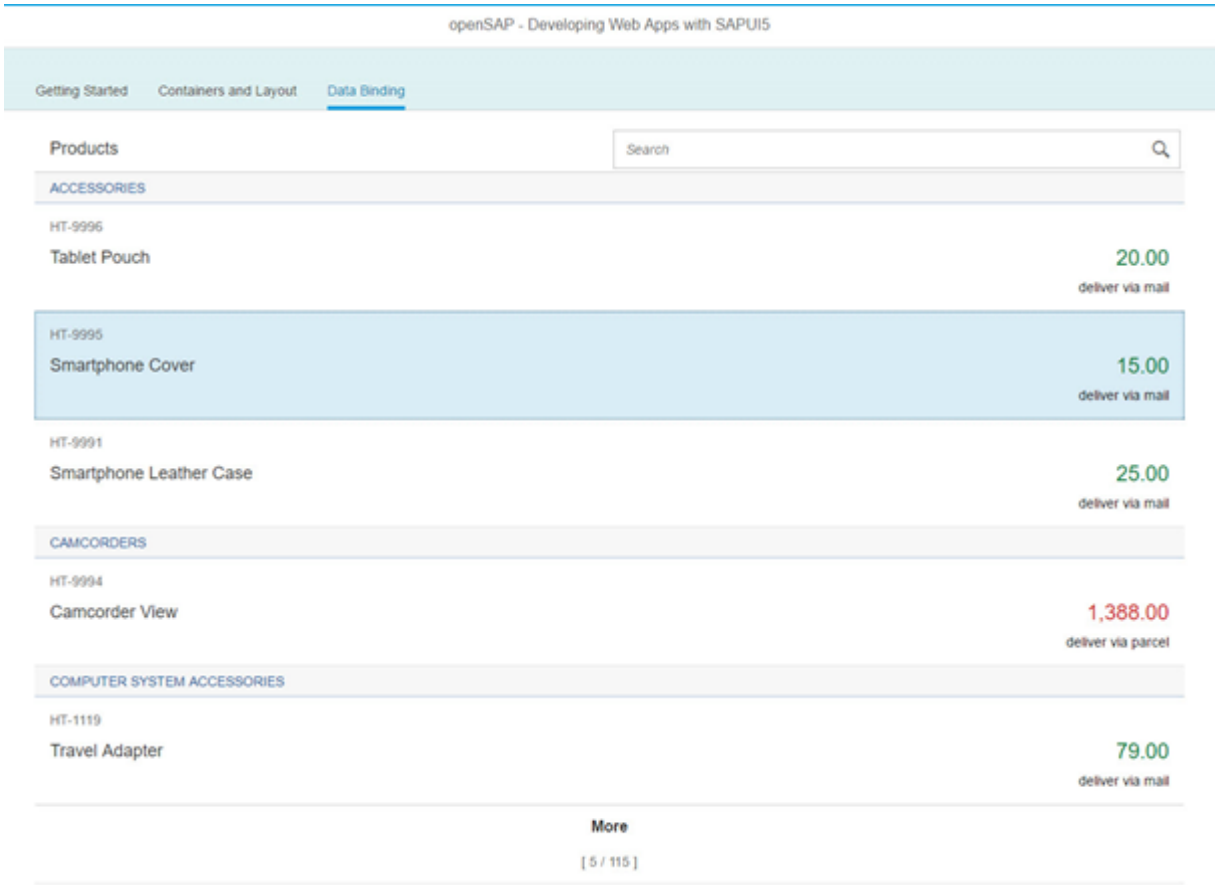

Figure 2: Paging is added to the List

webapp/view/App.view.xml

```
<List
  id="productsList"
  items="{
     path : '/ProductSet',
     sorter : {
        path : 'Category',
        group : true
     }
  }"
  growing="true"
  growingThreshold="5"
  growingScrollToLoad="false">
…
</List>
```

We add three new properties to the List in our App view. The `growing` property enables the paging mechanism. By setting the property `growingThreshold` to 5 we tell the list to initially display 5 items and add 5 more when a user clicks on the “More” button at the end of the list. Finally, the property `growingScrollToLoad` tells the list to only react on a click on the “More” button at the bottom instead of automatically loading more items when a user scrolls down in the list.

webapp/view/App.view.xml

```
<List …>
    …
  <items>
    <ObjectListItem
    …
          intro="{ProductID}"
          press="onItemSelected"
          type="Active"/>
```

t

Next, we will specify an event handler name for the `ObjectListItem` template within our list. We will not implement the handler just yet, but we can already add it in the view. The `Active` type will make the item selectable.

Related Information

List samples in Explored App

API Reference for sap.m.ListBase (The Growing feature is inherited from here)

2 ADDING THE DETAILS PANEL

In this step, we will add the area for the product details below our list. We will use a `sap.m.Panel` as container, and inside, we will put some fields of type `sap.m.Text`. To make these look prettier, we will organize them in a layout, the `sap.ui.layout.Grid`.

Preview

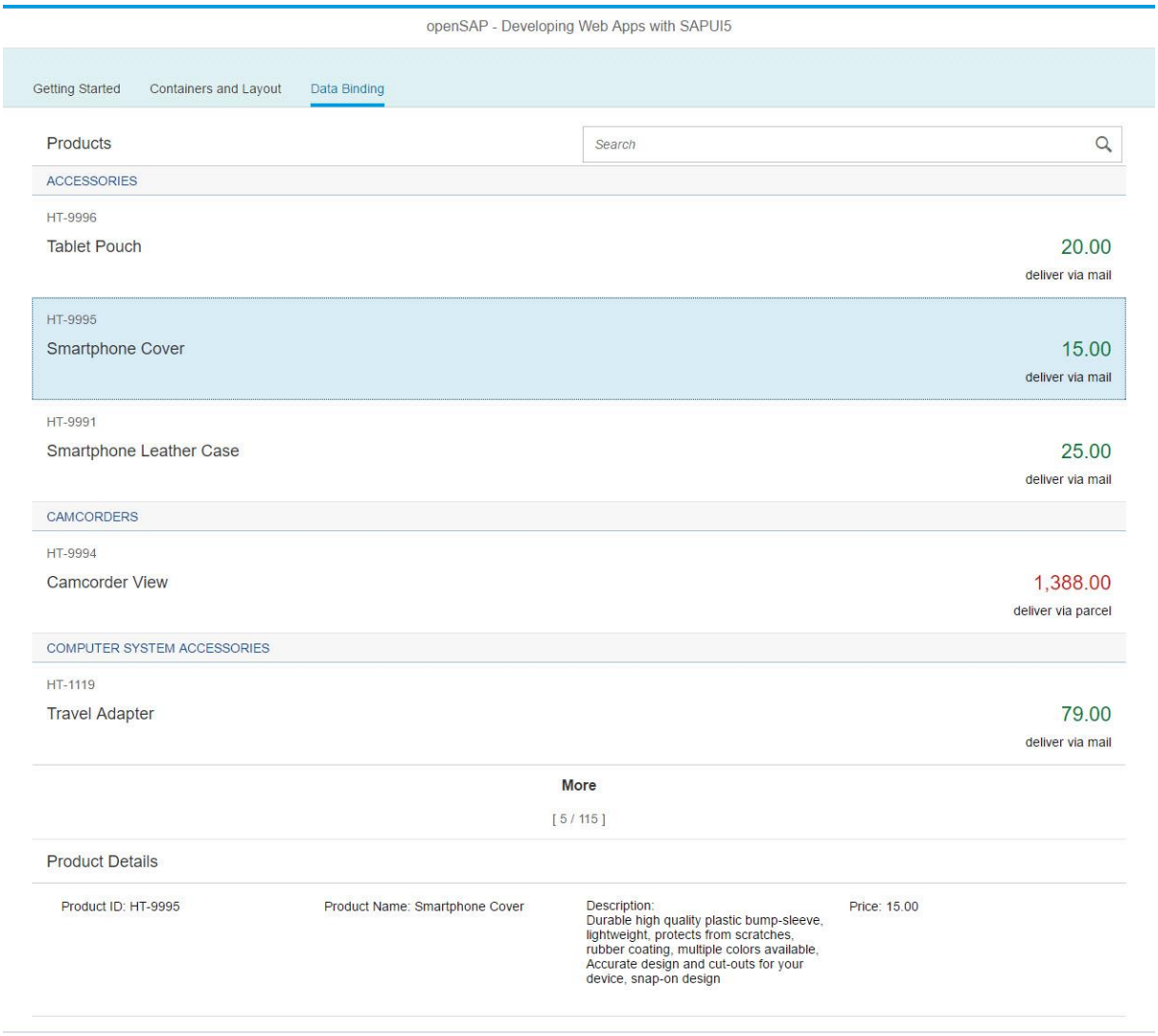

Figure 3: A panel below the list displaying details to a selected product

webapp/view/App.view.xml

```
…
  </List>
  <Panel id="productDetailsPanel"
    headerText="{i18n>productDetailPanelHeader}"
    visible="false">
  </Panel>
</content>
</IconTabFilter>
…
```

We add the panel, giving it an id (important) and a `headerText` property. The latter should be translatable, so we should take it from the resource model. We will define it in our properties file in a minute. We set its `visible` property to false, so the panel is initially not displayed. As the panel is not bound in the beginning, it would otherwise show up empty.

webapp/view/App.view.xml

```
<Panel id="productDetailsPanel"
  headerText="{i18n>productDetailPanelHeader}"
  visible="false">
  <l:Grid>
    <Text text="{i18n>productDetailPanelID}: {ProductID}"/>
    <Text text="{i18n>productDetailPanelName}: {Name}"/>
    <Text text="{i18n>productDetailPanelDescription}:\n {Description}"/>
    <Text text="{i18n>productDetailPanelPrice}: {Price}"/>
  </l:Grid>
</Panel>
```

Now, we can add the panel content. We start with the grid layout. As we have defined the namespace for the layout library as "`l`", we need to put this prefix in front of the control name, too:

Inside of the Grid, we now add our Text controls. We bind the text property to a combination of values, one providing the label text defined in the resource model, and the other one retrieved from the default model containing the product data. We separate the label text and value with a colon.

You can see that we are using relative paths for the property binding. In the following section, we will create the element binding for the parent control – the panel. When we do so, the child controls in the panel will resolve their bindings relative to their parents binding.

webapp/i18n/i18n.properties

```
...
# Data Binding Details Panel
productDetailPanelHeader=Product Details
productDetailPanelID=Product ID
productDetailPanelName=Product Name
productDetailPanelPrice=Price
productDetailPanelDescription=Description
```

We have added several controls using localized texts from the resource bundle, but the actual texts are still missing. Now we add these texts to the resource bundle.

3 BINDING THE DETAILS PANEL

In this step, we will enhance the controller of our App view with the `onItemSelected` method we have already specified in our view. When a user selects an item in our product list, we want to get the binding path from this item, and pass it to our panel, using the `bindElement` method.

webapp/controller/App.controller.js

```
…
return Controller.extend("opensap.myapp.controller.App", {

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

     // show message
     MessageToast.show(sMsg);
  },

  onItemSelected: function(oEvent) {
     var oSelectedItem = oEvent.getSource();
     var oContext = oSelectedItem.getBindingContext();
     var sPath = oContext.getPath();
     var oProductDetailPanel = this.byId("productDetailsPanel");

     oProductDetailPanel.bindElement({ path: sPath });
     this.byId("productDetailsPanel").setVisible(true);
  },

     onFilterProducts : function (oEvent) {…
```

As this is an event handler, it receives the Event object from the framework. From this object, we can get the control which has triggered the event by using `oEvent.getSource()`.

From the control instance, we can now determine its own binding context. As this is a child control of a List, and the list creates a binding context for each item, our currently selected `ObjectListItem` provides a binding context object.

Next, you need to retrieve the binding path from this object, respectively.

When we have the path, all we need to do is:

- Get the panel instance from the view by its id:
  `var oProductDetailPanel = this.byId("productDetailsPanel");`
- Bind the panel to the path we have retrieved from the list element:
  `oProductDetailPanel.bindElement({ path: sPath });`
- And finally make sure the panel is now visible:
  `this.byId("productDetailsPanel").setVisible(true);`

The `bindElement` function creates a binding context at the panel, and all paths at child controls of the panel will successfully resolve relatively to this context.

Related Information
Developer Guide – Essentials – Element Binding