HD3C16 – Products Application

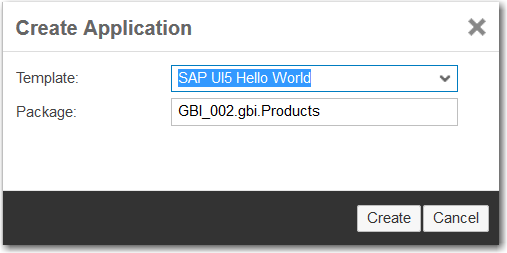
|  |  |
| --- | --- |
| **Product and Focus**  HANA Platform/oData | **MOTIVATION**  This case demonstrates the creation of a master/master/detail application using SAPUI5.  **PREREQUISITES**  HD3C05 – Create the Persistence Model |
| **Target Audience**  Undergrduate/Graduate Beginner to Intermediate |
| **Author**  Ross Hightower |
| https://bgoerke.files.wordpress.com/2013/05/section1.png | |

# Master/Master/Detail Application

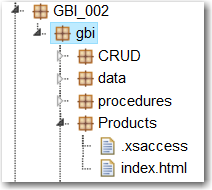
This case builds on the previous two cases in this series to develop a master/master/detail application which allows the user view information about products.

## Create the Application Packages

Logon to the WDW and locate the gbi package you created in case HD1dC02w. Right-click the gbi package and choose **Create Application.** Choose the SAP UI5 Hello World Template and then add **Products** to the Package.



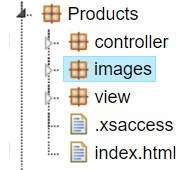
The package is created and basic application is created.



Note there is no .xsapp file. That file is unnecessary because in previous cases we created that file in the gbi package. It applies to the entire directory structure below it. However, the .xsaccess file was created because it is possible to create different levels of access for different packages.

You can run the application if you want by selecting the index.html file and clicking . This is the standard SAPUI5 Hello World app.

Now create the **controller**, **view** and **images** packages shown in the image below.



The basic structure of the application is complete. Now let’s add some content.

## Create the Application

The application follows a standard structure for an SAPUI5 application. The index.html file bootstraps the SAPUI5 libraries and creates a Component which encapsulates the application. The definition of the component is included in a file called Component.js. The name of this file is standard and cannot be altered. The various view and controller files are located in a package called view. For this application we will also include some images of products in the images package.

### index.html

The index.html file will be identical to those you created in other cases.

### Component.js

The Component.js file is very similar to those you created in previous cases. The code below shows the routes you will need. They are included here because you haven’t created a master/master/detail application before. Remember to create a model named gbi that uses your OData services.

|  |
| --- |
| routing: {  config: {  viewType: "XML",  viewPath: "gbi.view",  targetControl: "splitApp",  clearTarget: true,  transition: "slide"  },  routes: [  {  pattern: "",  name: "ProductCategories",  view: "Master",  targetAggregation: "masterPages"  },  {  pattern: "ProductCategory/{entity}",  name: "Products",  view: "Products",  targetAggregation: "masterPages"  },  {  pattern: "Products/{entity}",  name: "Detail",  view: "Detail",  targetAggregation: "detailPages"  }  ] |

Listing

### App.view.xml

The App view is identical to App view you created in the Routing case. Note that this is an XML view so the code that loads the App view in the Component.js file should look the same code in the Routing case.

### Master.view.xml

Create a file called Mater.view.xml in the view package and add the following code.

|  |
| --- |
| <mvc:View controllerName="gbi.controller.Master" xmlns:mvc="sap.ui.core.mvc"  xmlns="sap.m">  <Page title="Orders">  <List  id="ShortProductList"  headerText="Product Categories"  items="{gbi>/ProductCategories}" >  <StandardListItem  type="Active"  press="handleListItemPress"  title="{gbi>ProductCategory}"  description="{gbi>Description}" />  </List>  </Page>  </mvc:View> |

Listing

This code implements the initial view visible in the left side (master) of the screen. It consists of a List control that is bound to the ProductCategories collection. The list items show the ProductCategory and Description fields. Setting the type attribute to Active makes the list items clickable and the press attribute assigns a function called handleListItemPress to handle the click event. This function is defined in the Master.controller.js file.

### Master.controller.js

Create a file called Master.controller.js in the view package and insert the following code.

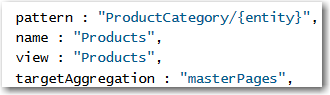
|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"  ], function (Controller) {  "use strict";  return Controller.extend("gbi.controller.Master", {    onInit: function() {  this.router = sap.ui.core.UIComponent.getRouterFor(this);  },    handleListItemPress: function(oItem){    var entity = oItem.getSource().getBindingContext("gbi").getPath().split("'");    this.router.navTo("Products", {  from: "Master",  entity: entity[1]  });  }  });  }); |

Listing

In the onInit function, a reference to the application’s router object is retrieved.

The handleListItemPress event is invoked when a list item is clicked. The oItem argument is a reference to the list item that was clicked. It is used to retrieve the list item’s binding context. One of the properties of binding context is the path within the oData collection to the object. This path will be of the form "/ProductCategories('ORB')" where ORB is a product category ID. The getPath function retrieves this path and the split function breaks the path into parts delimited by the ‘ character. The result is that entity is an array with three elements and the second element will be ORB.

The navTo method of the router object is used to navigate to the route that has the name Products and ORB is passed as the parameter entity. This matches the route:



The route loads the Products view into the masterPages collection. If you look at the URL when this route is invoked you will see the pattern indicated by the route.



### Products.view.xml

Create a file called Products.view.xml in the view package and insert the following code.

|  |
| --- |
| <mvc:View  controllerName="gbi.controller.Products"  xmlns:l="sap.ui.layout"  xmlns:core="sap.ui.core"  xmlns:mvc="sap.ui.core.mvc"  xmlns:f="sap.ui.layout.form"  xmlns="sap.m">  <Page title="Products"  navButtonPress="handleNavButtonPress"  showNavButton="true">  <List  id="idProductList"  headerText="Products"  items="{gbi>Products}" >  <StandardListItem  type="Active"  press="handleListItemPress"  title="{gbi>ID}"  description="{gbi>ProductName}" />  </List>  </Page>  </mvc:View> |

Listing

The Products view implements a list bound to the Products entity collection. See below for an explanation of the binding.

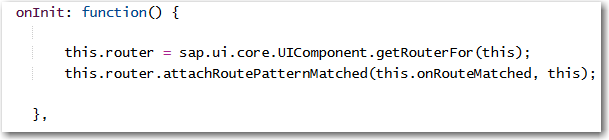
### Products.controller.js

Create a file called Products.controller.js in the view package and insert the following code.

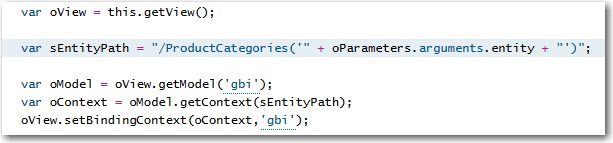
|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"  ], function(Controller) {  "use strict";  return Controller.extend("gbi.controller.Products", {  onInit: function() {  this.router = sap.ui.core.UIComponent.getRouterFor(this);  this.router.attachRoutePatternMatched(this.onRouteMatched, this);  },  handleNavButtonPress: function() {  this.router.navTo("");  },  onRouteMatched: function(oEvent) {  var oParameters = oEvent.getParameters();  if (oParameters.name !== "Products") {  return;  }  var oView = this.getView();  var sEntityPath = "/ProductCategories('" + oParameters.arguments.entity + "')";  var oModel = oView.getModel('gbi');  var oContext = oModel.getContext(sEntityPath);  oView.setBindingContext(oContext, 'gbi');  },  handleListItemPress: function(evt) {  this.showDetail(evt.getParameter("listItem") || evt.getSource());  },  showDetail: function(oItem) {  var entity = oItem.getBindingContext("gbi").getPath().split("'");  this.router.navTo("Detail", {  from: "Products",  entity: entity[1]  });  }  });  }); |

Listing

The onInit function retrieves a reference to the application’s router object then uses the router’s attachRoutePatternMatched method to assign the onRouteMatched function in the Products controller to run when a route to this view is navigated.



In the onRouteMatched function the parameters of the route matched event are retrieved. The parameters include the entity value (which contains the product category) that was passed to the route by the Master.controller.js code.



This value is used to create the variable sEntityPath which will contain a value like /ProductCategories(‘ORB’). You should remember from the oData case that this is what is added to the end of the service document URL to retrieve the data from product categories with ID equal to ORB. Next, a reference to the gbi model is retrieved and the model and sEntityPath are used to create a binding context that references the product category. The binding context is bound to the Products view.

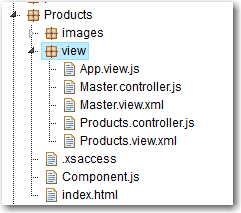
Getting the products in the list makes use of the association between the ProductCategories service and the Products service. The image below shows the definition of the Customers service from the gbi.xsodata file.



Notice that, in order to retrieve a products, you would use something like this:

<service document URL>/ProductCategores(‘ORB’)/Products

We just saw how the code in onRouteMatched bound the view to the product categories (e.g. /ProductCategories(‘ORB’)). If you look at the binding for items in the List control in the Products.view.xml file you will see that it is bound to Products. Since the List control is embedded in the view, the combined bindings mean the table items are bound to something like /ProductCategores(‘ORB’)/Products.



### Detail.view.xml

Create a file called Detail.view.xml in the view package and insert the following code.

|  |
| --- |
| <mvc:View  controllerName="gbi.controller.Detail"  xmlns:l="sap.ui.layout"  xmlns:core="sap.ui.core"  xmlns:mvc="sap.ui.core.mvc"  xmlns:f="sap.ui.layout.form"  xmlns="sap.m">  <Page title="Product Details" >  <l:Grid  defaultSpan="L12 M12 S12"  width="auto">  <l:content>  <f:SimpleForm id="idProductForm"  minWidth="1024"  maxContainerCols="2"  editable="false"  layout="ResponsiveGridLayout"  title="Product Details"  labelSpanL="4"  labelSpanM="4"  emptySpanL="0"  emptySpanM="0"  columnsL="2"  columnsM="2">  <f:content>  <core:Title text="" />  <core:Title text="" />  <Label text="Number" />  <Text text="{gbi>ID}" />  <Label text="Name" />  <Text text="{gbi>ProductName}" />  <Label text="Color" />  <Text text="{gbi>Color}" />  <core:Title text="" />  <core:Title text="" />  <Label text="Price" />  <ObjectNumber number = "{gbi>Price}" />  <Label text="Internal Price" />  <ObjectNumber number = "{gbi>InternalPrice}" />  </f:content>  </f:SimpleForm>  </l:content>  </l:Grid>    </Page>  </mvc:View> |

Listing

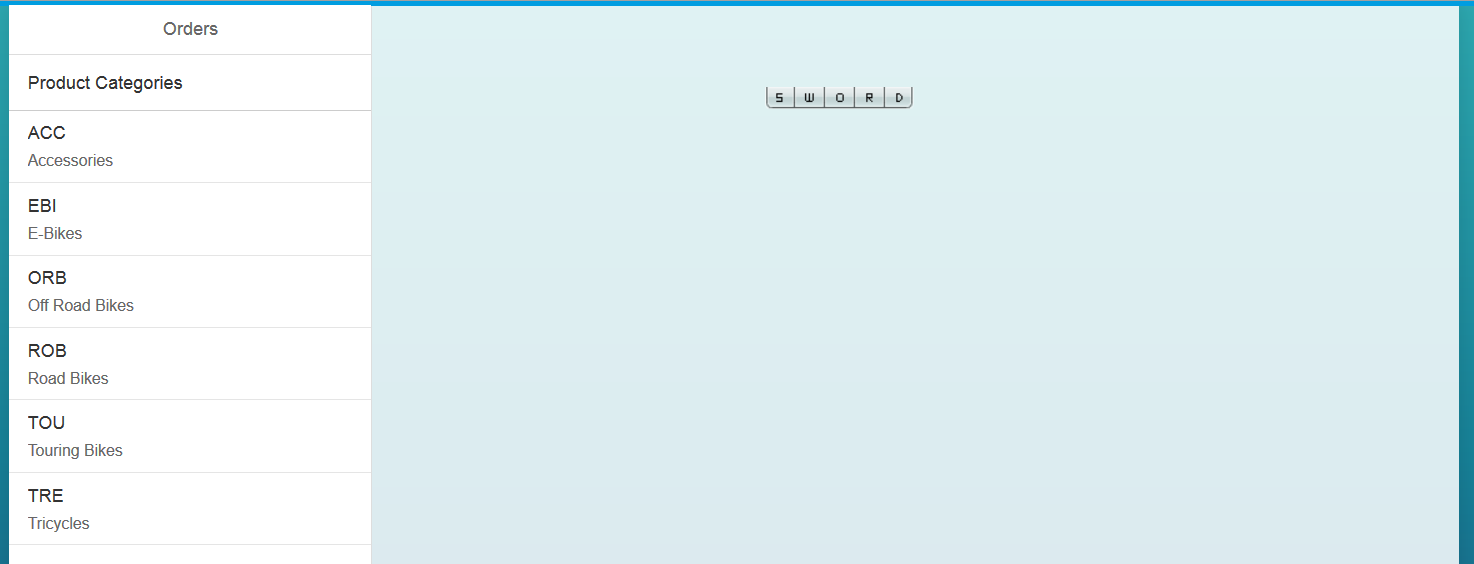
### Detail.controller.js

Create a file called Detail.controller.js in the view package and insert the following code.

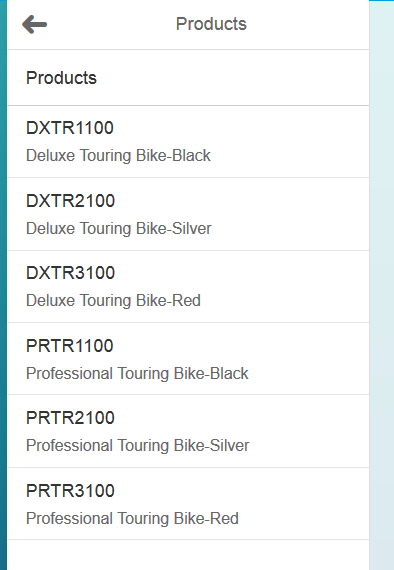
|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"  ], function (Controller) {  "use strict";  return Controller.extend("gbi.controller.Detail", {    onInit: function() {    this.router = sap.ui.core.UIComponent.getRouterFor(this);  this.router.attachRoutePatternMatched(this.onRouteMatched, this);    },  onRouteMatched : function(oEvent) {  var oParameters = oEvent.getParameters();  var oView = this.getView();  // When navigating in the Detail page, update the binding context  if (oParameters.name !== "Detail") {  return;  }  var sEntityPath = "/Products('" + oParameters.arguments.entity + "')";  var oModel = oView.getModel('gbi');  var context = new sap.ui.model.Context(oModel , sEntityPath);    oView.setBindingContext(context,'gbi');  }  });  }); |

Listing

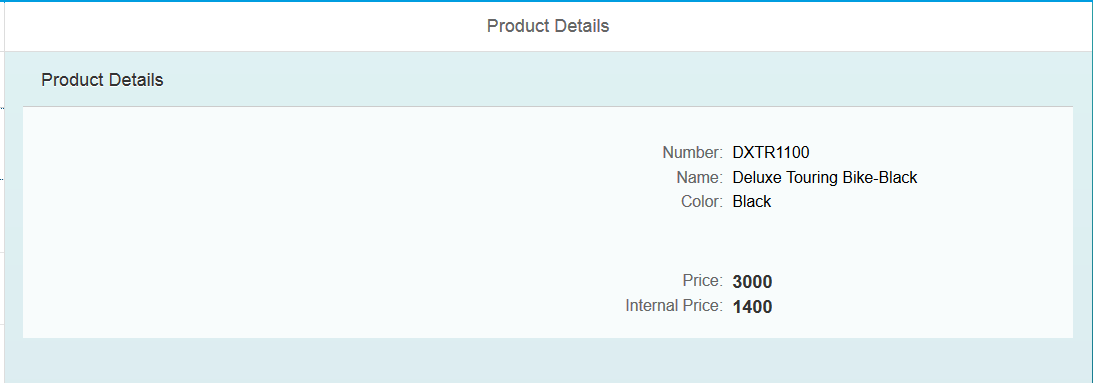
Run the application by clicking the index.html file and clicking the run icon. When the application loads initially you will see a list of product categories in the master page section.



If you click a product category, a list of products in that category are loaded.



If you click a product, the detail view is loaded with product information.



Pretty nice but let’s dress it up a bit.

## Format the Numbers

To format the Price and Internal Price we need to add a formatter function. Locate {gbi>Price}, the binding, in the Detail.view.xml files. Change the code to look like the following:



Do the same for the Internal Price field.

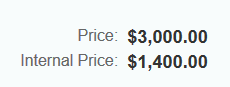
The formatter property refers to the function called formatCurrency in the Details.controller.js file. Add the following function to that file.

|  |
| --- |
| formatCurrency : function(value){  var d = ".";  var t = ",";  var c = 2;  var p = "$";  c = isNaN(c = Math.abs(c)) ? 2 : c;  var s = value < 0 ? "-" : "";  var i = parseInt(value = Math.abs(+value || 0).toFixed(2)) + "";  var j = (j = i.length) > 3 ? j % 3 : 0;  return p + s + (j ? i.substr(0, j) + t : "") + i.substr(j).replace(/(\d{3})(?=\d)/g, "$1" + t) + (c ? d + Math.abs(value - i).toFixed(2).slice(2) : "");  } |

Listing

Remember to separate functions with a comma. The value of the field is passed to this function in the value parameter. The function formats the value as USD. You can change the values of d, t and p to change the currency format.

Now the numbers are formatted:



## Add Product Images

Add the following code immediately following the **first** <core:Title text="" /> in the Details.view.xml file on or about line 26.

|  |
| --- |
| <Image src="{ path : 'gbi>ID', formatter: '.imageURL' }" height="150px" width="150px"/> |

Listing

This adds an Image control. The src attribute of the control will be computed by the function imageURL using the ID of the product. Add the following function to the Details.controller.js file.

|  |
| --- |
| imageURL : function(file){  return "**http://hana.ucc.uwm.edu:8004/GBI\_002/gbi/Products/images/**" + file + ".jpg";  } |

Listing

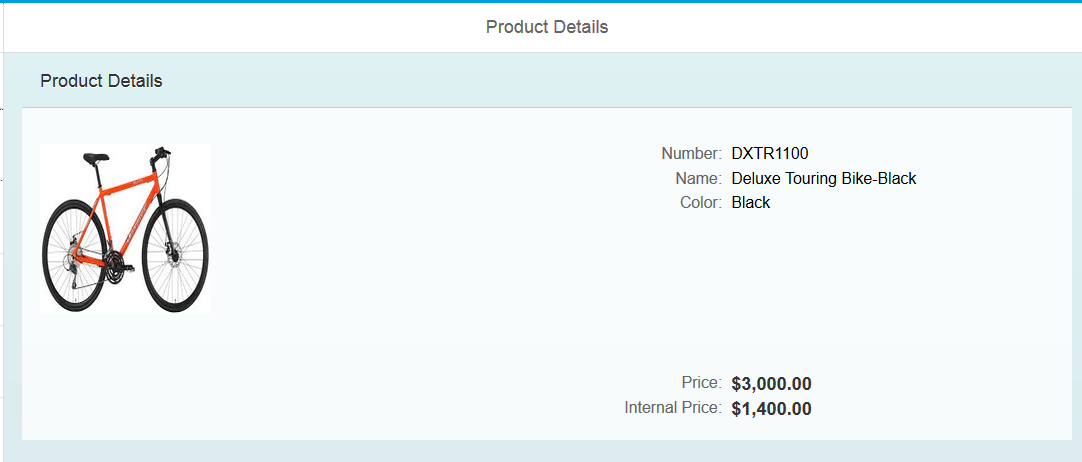
**Update the URL for your situation**. The path, GBI\_002/gbi/Products/images is the path of the packages from the Content folder to your images folder.

Finally, copy the images provided with the curriculum into the images package. The easiest way to do this is the select the images package so that you see the Multi-File Drop Zone.



Drag and drop the files onto the drop zone. If you want to retrieve your own image files they be jpg files and be named the name of the product’s ID.

The result is shown below. Note that not all products have images.



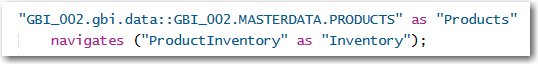
## Add an Inventory Table

Add the following code below the Grid control in the Details.view.xml file.

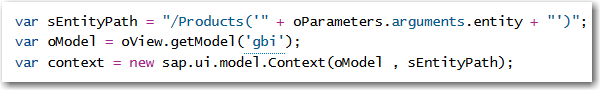
|  |
| --- |
| <Table id="idProductsTable"  inset="false"  itemPress = "handleTableRowPress"  items='{gbi>Inventory}'>  <columns>  <Column  minScreenWidth="Tablet"  demandPopin='true'  mergeDuplicates="true">  <header>  <Text text="Product" />  </header>  </Column>  <Column>  <header>  <Text text="Plant" />  </header>  </Column>  <Column>  <header>  <Text text="SLoc" />  </header>  </Column>  <Column>  <header>  <Text text="Quantity" />  </header>  </Column>  <Column>  <header>  <Text text="Stock Type" />  </header>  </Column>    </columns>  <items>  <ColumnListItem>  <Text  text="{gbi>ProductID}" />  <Text  text="{gbi>Plant}" />  <Text  text="{gbi>SLoc}" />  <ObjectNumber  number = "{gbi>Quantity}"  numberUnit = "{gbi>UnitOfMeasure}" />  <Text  text = "{gbi>StockType}" />  </ColumnListItem>  </items>  </Table> |

Listing

This code creates a Table control that is bound to Inventory. It makes use of the association between the Products table and the Inventory table. The Inventory in gbi>Inventory which is bound to the Table controls items aggregation, refers to the NavigationProperty between Products and Inventory in the oData service.

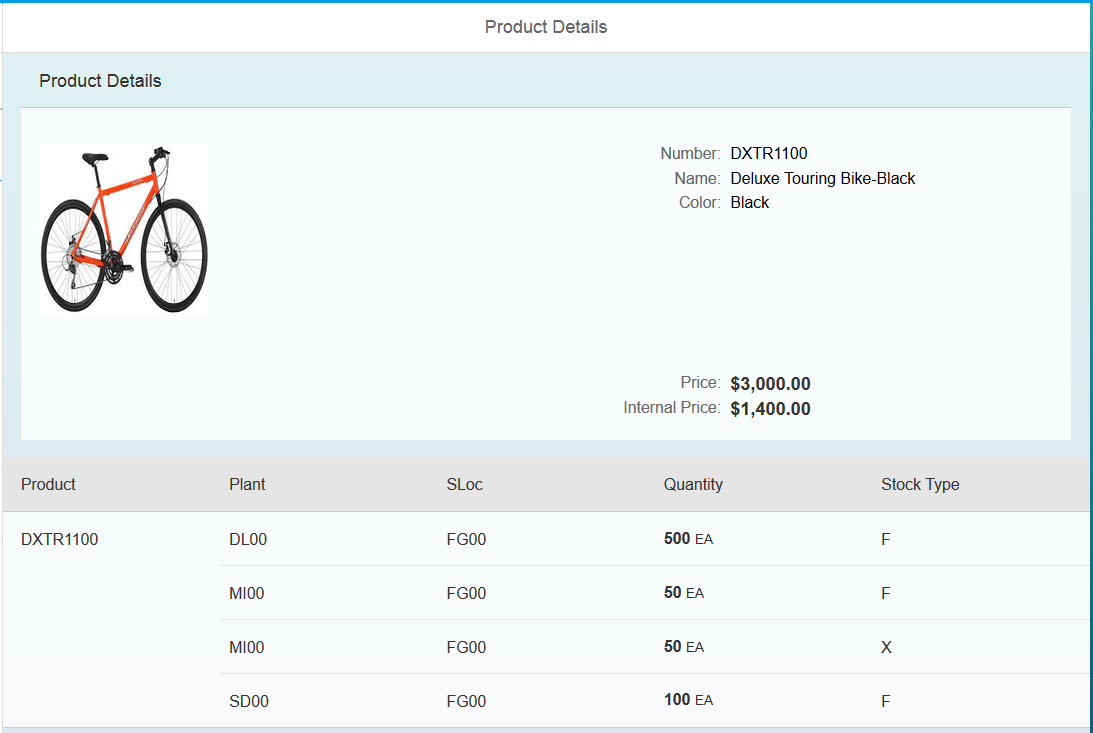


The binding context of the view is set to /Products(<product id>) by this code:



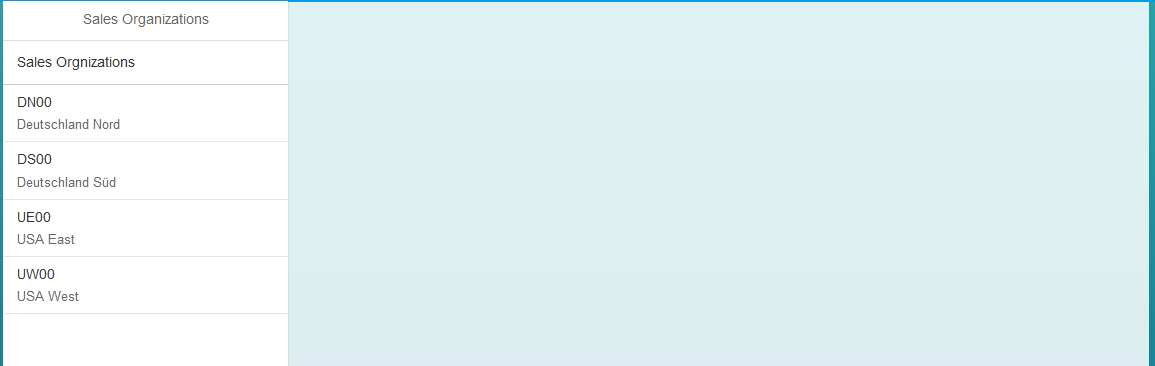
Binding the Table to Inventory binds the table to /Products(<product id>)/Inventory, navigating the association.

The completed Details view.

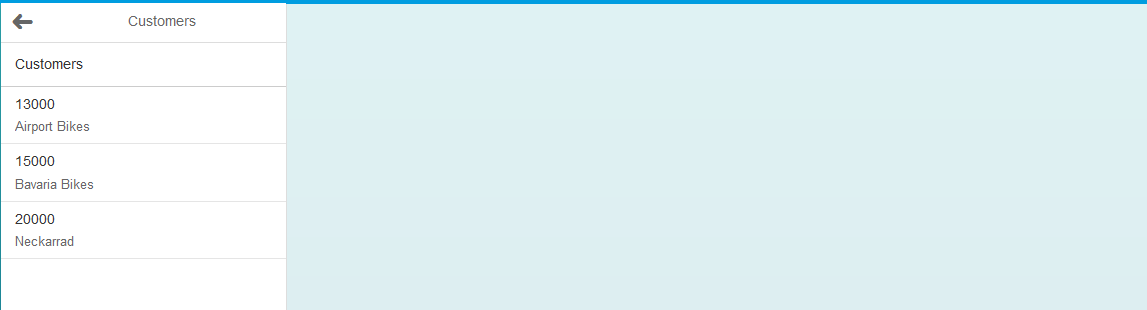


# Exercise

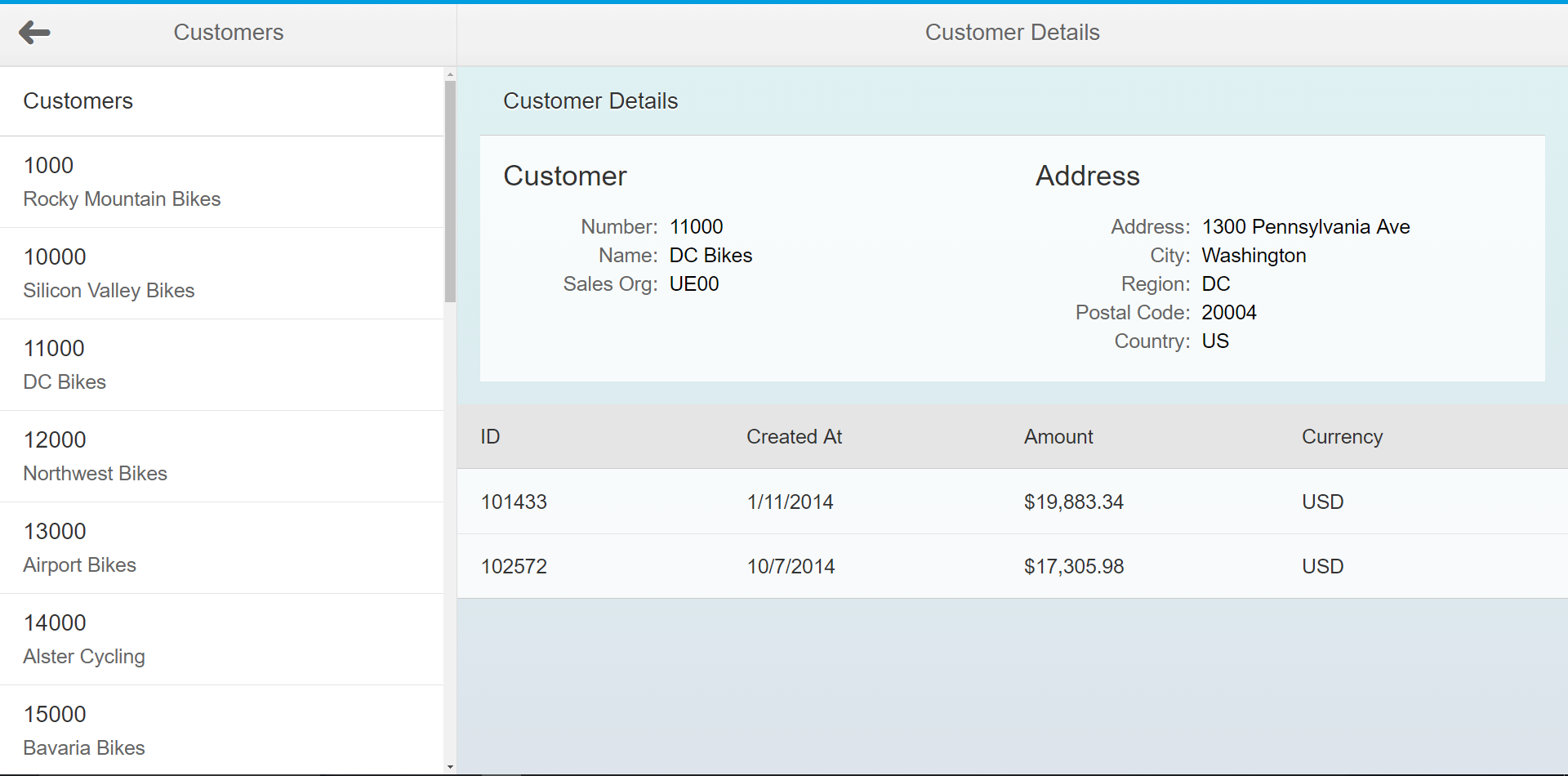
Create a master/master/detail application using Sales Organizations, Customers and Sales Orders. The initial view shows the sales organizations in the master view.



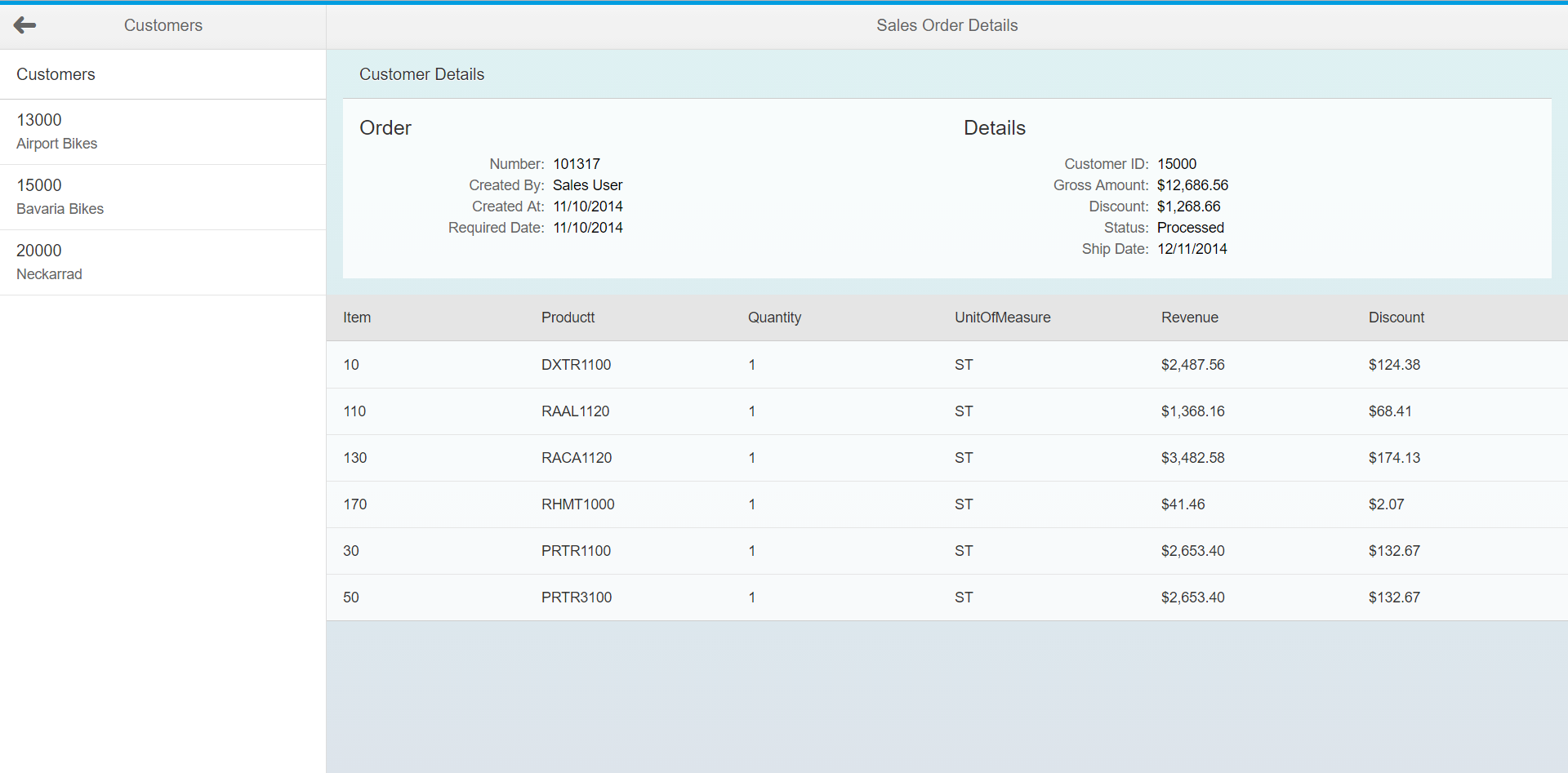
When you click a Sales Organization, a list of the customers associated with that sales organization is shown in the master view:



Note the navigation button that returns you to the Sales Organization list. If you click on a customer you should see the customer details and the customer’s sales orders. This is similar to the Customers application.



Now the tricky part. When you click a Sales Order in the Customer details view table, you get a new view in the details section that shows the sales order details.



You need a couple of hints to make this work:

First, in order to make the table row clickable you have to alter the ColumnListItem control for the table as shown:

<ColumnListItem type="Navigation">

Second, when you click the table with the sales order, you need to retrieve the sales order ID in the first column to pass to the entity in the route. Use the following code to retrieve that value:

var id = evt.getParameters().listItem.getCells()[0].getText();