

HD2dC05w – Products Application

Product and Focus

HANA Platform/oData

Target Audience

Undergraduate/Graduate
Beginner to Intermediate

Author

Ross Hightower

MOTIVATION

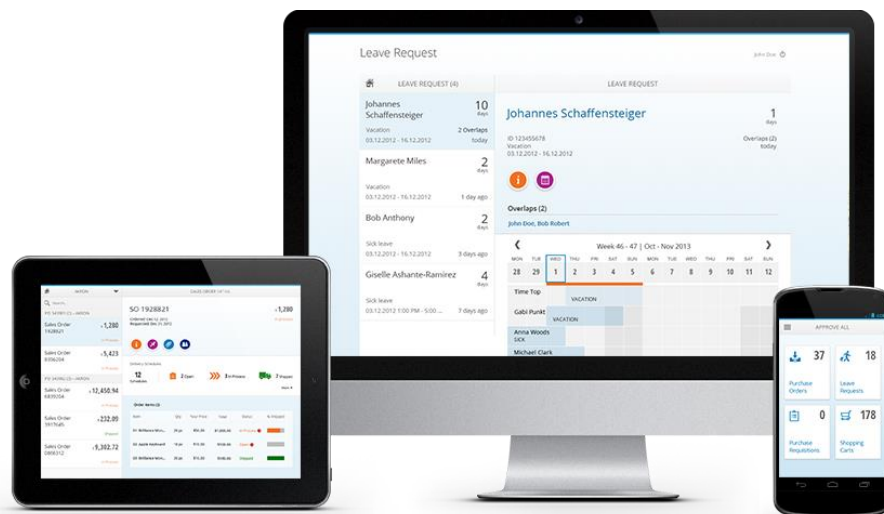
This case demonstrates the creation of a master/master/detail application using SAPUI5.

PREREQUISITES

HD1C01 – Hello World MVC

HD1dC02 – Create the Persistence Model

HD1dC03 – oData Services¹



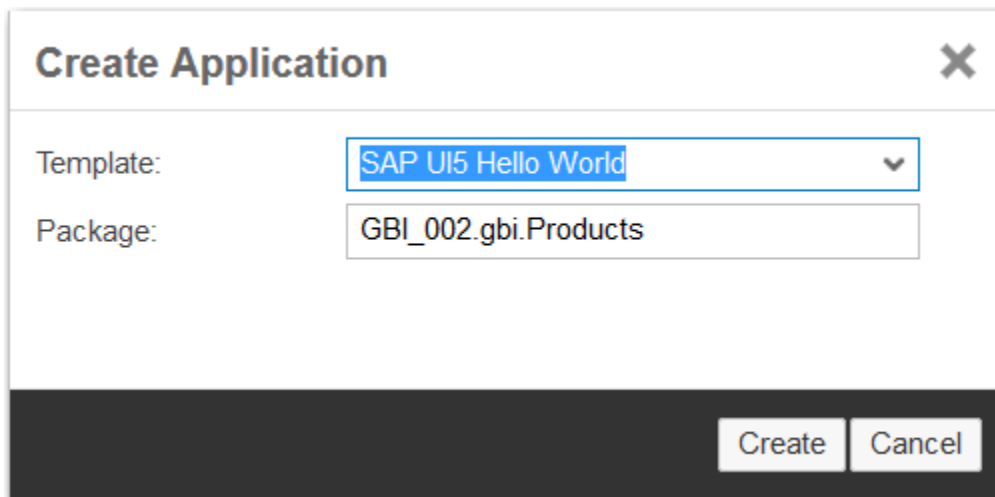
¹ If you haven't completed the HD1dC02 and HD1dC03 cases you can use the HANA GBI oData service provided by the UCC.

Master/Master/Detail Application

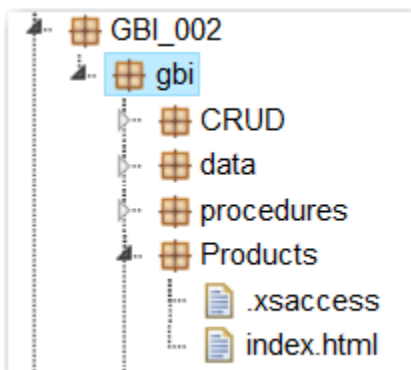
This case builds on the previous two cases in this series to develop a master/master/detail application which allows the user view information about products.

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.

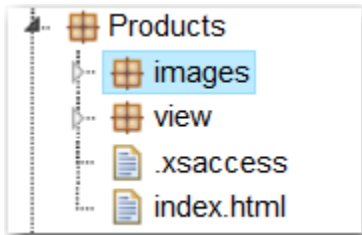


Note there is no .xsapp file. That file is unnecessary because in previous cases we created that file in the gbi package. It applies to the entire directory structure below it. However, the .xsaccess file was created because it is possible to create different levels of access for different packages.



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 **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

Replace the code in the index.html file with this code.

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <title>GBI Products</title>

  <script id="sap-ui-bootstrap"
    type="text/javascript"
    src="/sap/ui5/1/resources/sap-ui-core.js"
    data-sap-ui-theme="sap_bluecrystal"
    data-sap-ui-libs="sap.m"
    data-sap-ui-xx-bindingSyntax="complex"
    data-sap-ui-resourceroots = '{
      "gbi" : "./"
    }'></script>

  <script>

    new sap.m.Shell("Shell",{
      app: new sap.ui.core.ComponentContainer({
        name: 'gbi'
      })
    })
```

```
        }).placeAt('uiArea');

    </script>

</head>
<body class="sapUiBody">
    <div id="uiArea"></div>
</body>
</html>
```

Listing 1

For an explanation of this code see the case HD1C01 – Hello World MVC.

Component.js

Create the Component.js file in the CRUD package and add the following code.

```
jQuery.sap.declare("gbi.Component");

sap.ui.core.UIComponent.extend("gbi.Component",{

    metadata: {

        routing: {

            config: {
                viewType: "XML",
                viewPath: "gbi.view",
                targetControl: "splitApp",
                clearTarget: false,
                transition: "slide"
            },

            routes: [
                {
                    pattern : "",
                    name : "ProductCategories",
                    view : "Master",
                    targetAggregation : "masterPages"
                },
                {
                    pattern : "ProductCategory/{entity}",
                    name : "Products",
                    view : "Products",
```

```
        targetAggregation : "masterPages",
        subroutes : [
            {
                pattern : "Products/{entity}",
                name : "Detail",
                view : "Detail",
                targetAggregation : "detailPages"
            }
        ]
    }
}

},

init: function() {

    jQuery.sap.require("sap.m.routing.RouteMatchedHandler");
    jQuery.sap.require("sap.ui.core.routing.HashChanger");

    //call createContent
    sap.ui.core.UIComponent.prototype.init.apply(this, arguments);

    this._router = this.getRouter();

    //initialize the router
    this._routeHandler = new sap.m.routing.RouteMatchedHandler(this._router);
    this._router.initialize();

},

createContent: function() {

    var oView = sap.ui.view({
        id: "app",
        viewName: "gbi.view.App",
        type: "JS",
        viewData: {component: this}
    });

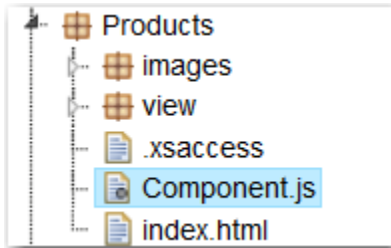
    var oModel = new
sap.ui.model.odata.ODataModel("http://hana.ucc.uwm.edu:8004/GBI_002/gbi/services/gbi.xsodata"
);

    oView.setModel(oModel,'gbi');

    return oView;
}
```

```
}  
  
});
```

Listing 2



Update the highlighted portion of the code to reflect your oData service URL.

This code declares a component called gbi.Component and then extends it. Note the gbi. at the beginning of the component name corresponds to ./ as declared in the index.html file. This means that SAPUI5 will look for the file in the same package as the index.html file.

```
jQuery.sap.declare("gbi.Component");  
  
sap.ui.core.UIComponent.extend("gbi.Component", {
```

Next, the code defines metadata for the component. In this case the only metadata is configuration data for a router. The router object will be used to navigate between views. The config section defines default values for the router. We will use XML views and locate them in the view package. The views will be displayed in a splitApp control which is used to organize a master/detail application. The clearTarget attribute indicates that the application shouldn't delete the contents of the target control before navigation and the transition defines the type of transition to use when navigation occurs. You can find the configuration options [here](#).

```
routing: {  
  
  config: {  
    viewType: "XML",  
    viewPath: "view",  
    targetControl: "splitApp",  
    clearTarget: false,  
    transition: "slide"  
  },  
  
  routes: [  
    {  
      pattern: "",  
      name: "ProductCategories",  
      view: "Master",  
      targetAggregation: "masterPages"  
    },  
    {  
      pattern: "ProductCategory/{entity}",  
      name: "Products",  
      view: "Products",  
      targetAggregation: "masterPages",  
      subroutes: [  
        {  
          pattern: "Products/{entity}",  
          name: "Detail",  
          view: "Detail",  
          targetAggregation: "detailPages"  
        }  
      ]  
    }  
  ]  
}
```

A SplitApp control has two aggregations: masterPages and detailPages. An aggregation is a property to which multiple items can be assigned. In this case, the views that make up the application's interface area assigned to the aggregations. The masterPages aggregation is shown on the left side of the application and the detailPages aggregation is shown on the right side of the screen.

The first route in the image above has an empty pattern which causes this route to load automatically when the application loads. The name of the route is ProductCategories and the view that will be loaded is Master.view.xml which will be loaded into the masterPages (left) aggregation.

The second route is triggered with the pattern ProductCategory/{entity}. The value {entity} will contain the id of the selected ProductCategory when the route is invoked. The route loads the Products.view.xml view into the masterPages aggregation.

The Products route has a subroute called Detail. The pattern indicates that a parameter, also called {entity} will be passed to the route. In this case, entity will contain the id of the selected product. The Detail.view.xml view will be loaded into the detailPages (right side of the screen) aggregation.

The init function is called when the Component is first created. The first two lines in this function tell SAPUI5 to load the two libraries that implement the router. Next, the constructor for the prototype of the Component class is called. The result is to initialize the component.

Finally, the router is created then initialized. The operation of the router will become clearer a little later.

```
init: function() {  
  
    jQuery.sap.require("sap.m.routing.RouteMatchedHandler");  
    jQuery.sap.require("sap.ui.core.routing.HashChanger");  
  
    //call createContent  
    sap.ui.core.UIComponent.prototype.init.apply(this, arguments);  
  
    this._router = this.getRouter();  
  
    //initialize the router  
    this._routeHandler = new sap.m.routing.RouteMatchedHandler(this._router);  
    this._router.initialize();  
  
},
```

The createContent function creates the content of the component which consists of a single view called App. The App view defines the overall structure of the application and can be considered the root view of the application. The code then creates the application model using the URL of the oData service document.

```
createContent: function() {  
  
    var oView = sap.ui.view({  
        id: "app",  
        viewName: "view.App",  
        type: "JS",  
        viewData: {component: this}  
    });  
  
    var oModel = new sap.ui.model.odata.ODataModel("http://hana.ucc.uwm.edu:8004/GBI_002/gbi/services/gbi.xsodata");  
    oView.setModel(oModel, 'gbi');  
  
    return oView;  
  
}
```

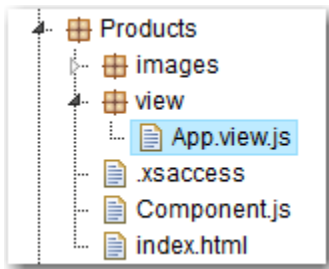

App.view.js

Create the App.view.js file in the view package and add the following code.

```
sap.ui.jsview("gbi.view.App", {  
  
    createContent : function() {  
  
        this.setDisplayBlock(true);  
  
        return new sap.m.SplitApp("splitApp",{  
        });  
    }  
});
```

Listing 3

This code creates the application object based on the SplitApp master/detail structure.



Master.view.xml

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

```
<mvc:View controllerName="gbi.view.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 4

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.controller("gbi.view.Master", {

    onInit: function() {

        this.router = sap.ui.core.UIComponent.getRouterFor(this);
    },

    handleListItemPress: function(oItem){

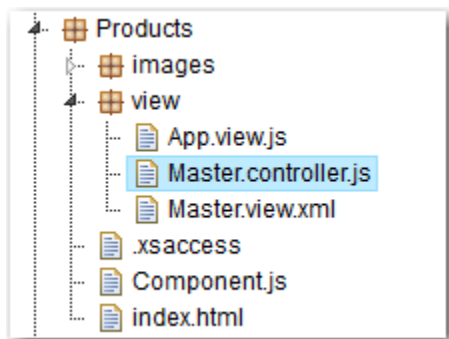
        var entity = oItem.getSource().getBindingContext("gbi").getPath().split("");

        this.router.navTo("Products", {
            from: "Master",
            entity: entity[1]
        });

    }

});
```

Listing 5



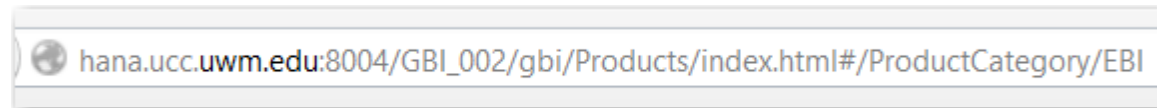
In the onInit function, a reference to the application's router object is retrieved.

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:

```
pattern : "ProductCategory/{entity}",
name : "Products",
view : "Products",
targetAggregation : "masterPages",
```

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.view.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 6

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.controller("gbi.view.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 7

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.

```
onInit: function() {  
  
    this.router = sap.ui.core.UIComponent.getRouterFor(this);  
    this.router.attachRoutePatternMatched(this.onRouteMatched, this);  
  
},
```

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.

```
var oView = this.getView();  
  
var sEntityPath = "/ProductCategories('" + oParameters.arguments.entity + "')";  
  
var oModel = oView.getModel('gbi');  
var oContext = oModel.getContext(sEntityPath);  
oView.setBindingContext(oContext, 'gbi');
```

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.

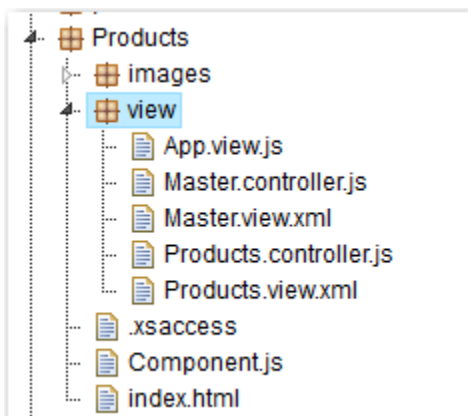
Getting the products in the list makes use of the association between the ProductCategories service and the Products service. The image below shows the definition of the Customers service from the gbi.xsodata file.

```
"GBI_002.gbi.data::GBI_002.MASTERDATA.PRODUCT_CATEGORIES" as "ProductCategories"
  navigates ("CategoriesProducts" as "Products");
```

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

```
<service document URL>/ProductCategories('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 /ProductCategories('ORB')/Products.



Detail.view.xml

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

```
<mvc:View
  controllerName="view.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 8

Detail.controller.js

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

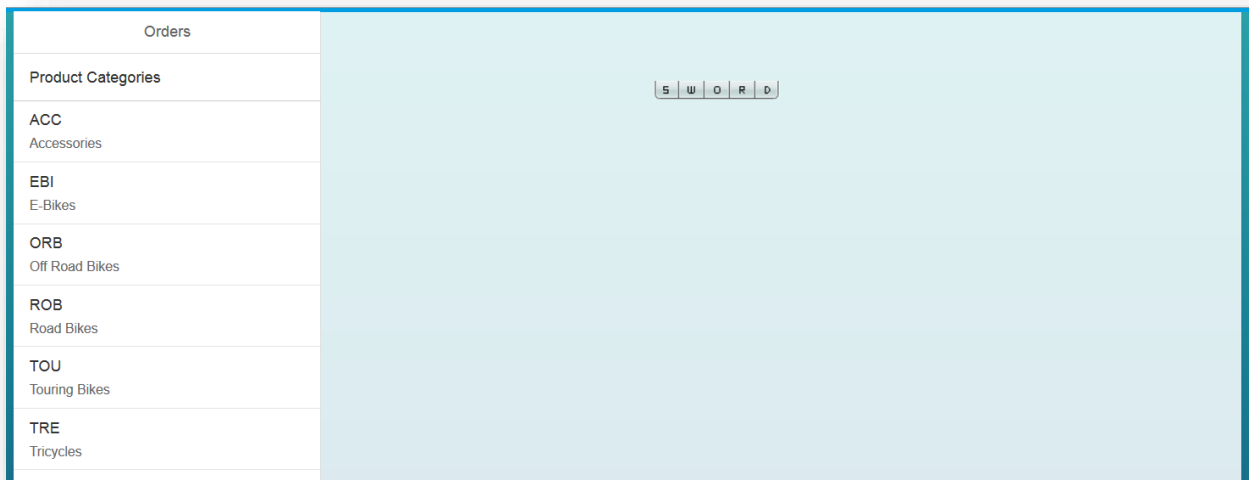
```
sap.ui.controller("view.Detail", {
    onInit: function() {

        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 9

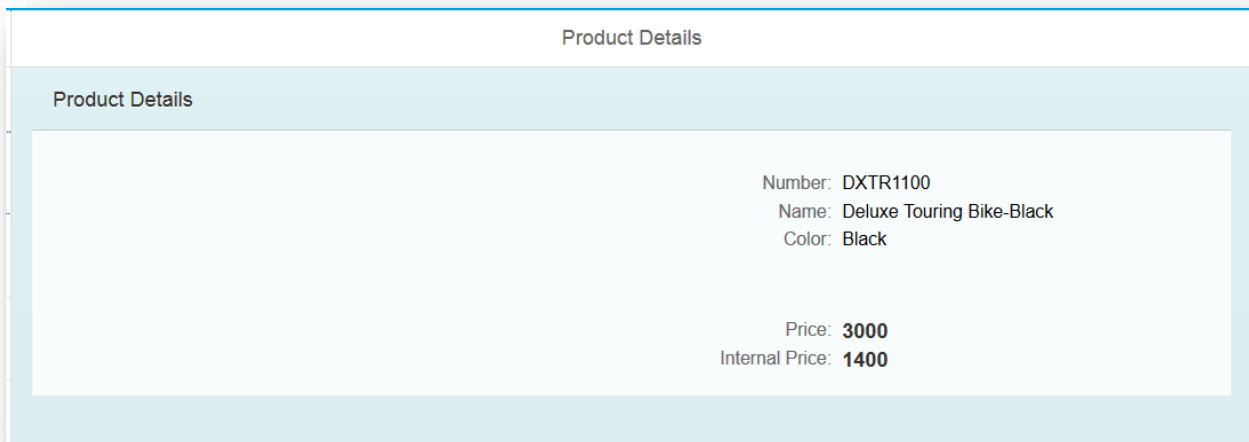
Run the application by clicking the index.html file and clicking the run icon. When the application loads initially you will see a list of product categories in the master page section.



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

←	Products
Products	
DXTR1100	Deluxe Touring Bike-Black
DXTR2100	Deluxe Touring Bike-Silver
DXTR3100	Deluxe Touring Bike-Red
PRTR1100	Professional Touring Bike-Black
PRTR2100	Professional Touring Bike-Silver
PRTR3100	Professional Touring Bike-Red

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:

```
<Label text="Price" />  
<ObjectNumber number = "{ path : 'gbi>Price', formatter : '.formatCurrency' }" />
```

Do the same for the Internal Price field.

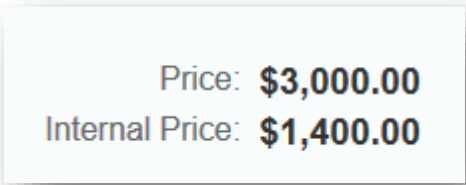
The formatter property refers to the function called `formatCurrency` in the `Details.controller.js` file. Add the following function to that file.

```
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 10

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:



Price: \$3,000.00
Internal Price: \$1,400.00

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 11

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 12

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.

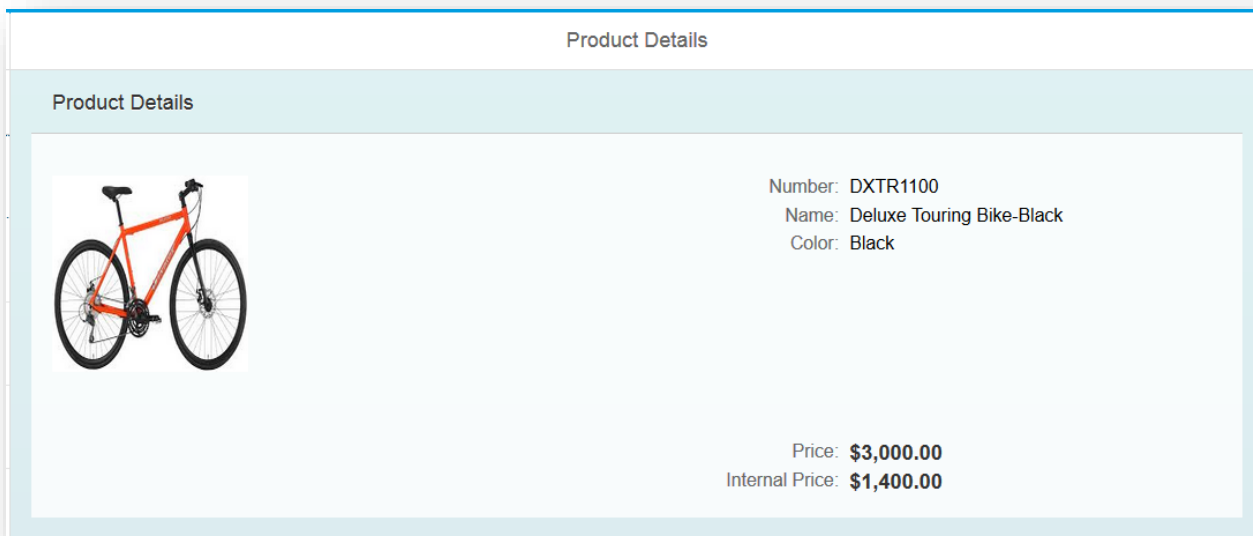
Finally, copy the images provided with the curriculum into the images package. The easiest way to do this is to select the images package so that you see the Multi-File Drop Zone.

Package: GBI_002/gbi/Products/images

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 13

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.


```
"GBI_002.gbi.data::GBI_002.MASTERDATA.PRODUCTS" as "Products"  
    navigates ("ProductInventory" as "Inventory");
```

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

```
var sEntityPath = "/Products('" + oParameters.arguments.entity + "')";  
var oModel = oView.getModel('gbi');  
var context = new sap.ui.model.Context(oModel , sEntityPath);
```

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

The completed Details view.

Product Details				
<div> <div>Product Details</div> <div>  <div> Number: DXTR1100 Name: Deluxe Touring Bike-Black Color: Black </div> <div> Price: \$3,000.00 Internal Price: \$1,400.00 </div> </div> </div>				
Product	Plant	SLoc	Quantity	Stock Type
DXTR1100	DL00	FG00	500 EA	F
	MI00	FG00	50 EA	F
	MI00	FG00	50 EA	X
	SD00	FG00	100 EA	F

Exercise

Create a master/detail application using Customer and Sales Orders. The table shows the Orders for the selected Customer. The Customer and Address headings in the form are created using controls like this:

```
<core:Title text="Customer" />
```

Note that the Amount and CreatedAt fields are formatted. The function to format the date is:

```
formatDate : function(oDate){
    return new Date(oDate).toDateString();
}
```

Listing 14

Customers	Customer Details			
Customers	Customer Details			
1000 Rocky Mountain Bikes	<div> <div>Customer</div> <div> Number: 12000 Name: Northwest Bikes Sales Org: UW00 </div> <div>Address</div> <div> Address: 601 108th Ave City: Seattle Region: WA Postal Code: 98004 Country: US </div> </div>			
10000 Silicon Valley Bikes				
11000 DC Bikes				
12000 Northwest Bikes				
13000 Airport Bikes				
14000 Alster Cycling				
15000 Bavaria Bikes				
16000 Capital Bikes				
17000 Cruiser Bikes				
18000 Drahtesel				
	ID	Created At	Amount	Currency
	101347	Mon Dec 01 2014	\$4,060.00	USD
	101606	Sun Apr 20 2014	\$102,352.79	USD