HD3C15 – Customers Application

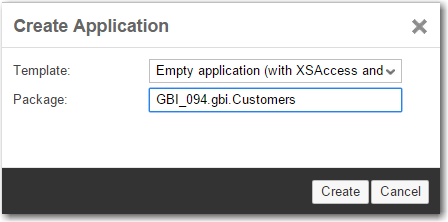
|  |  |
| --- | --- |
| **Product and Focus**  HANA Platform/oData | **MOTIVATION**  This case describes the user of oData services to create a simple SAPUI5 application.  **PREREQUISITES**  HD3C05 – Create the Persistence Model |
| **Target Audience**  Undergraduate/Graduate Beginner to Intermediate |
| **Author**  Ross Hightower |
| https://bgoerke.files.wordpress.com/2013/05/section1.png | |

# Master/Detail Application

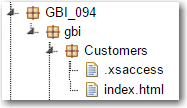
This cases builds on the previous two cases in this series to develop a master/detail application which allows the user view information about customers.

## Create the Application Packages

Logon to the WDW and locate the gbi package you created in case HD1dC2w. Right-click the gbi package and choose **Create Application.** Choose the SAP UI5 Hello World Template and then add **Customers** to the Package.



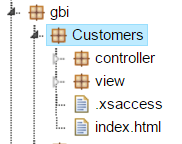
The package is created and basic application is created.



Note there is no .xsapp file. That file is unnecessary because in the oData services case 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 **controller** 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 and location of this file is standard and cannot be altered. The various view and controller files are located in a package called view.

|  |
| --- |
| For a detailed explanation of the code see the case HD1C01 – Hello World MVC |

### 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 Customers</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

### Component.js

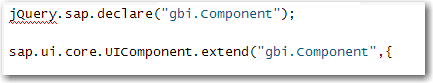
Create the Component.js file in the Customer package and add the following code.

|  |
| --- |
| sap.ui.define([  "sap/ui/core/UIComponent",  "sap/ui/model/odata/ODataModel"  ], function(UIComponent, ODataModel) {  "use strict";  return UIComponent.extend("gbi.Component", {  metadata: {  rootView: "gbi.view.App",    routing: {    config: {  viewType: "XML",  viewPath: "gbi.view",  targetControl: "splitApp",  clearTarget: false,  transition: "slide"  },    routes: [  {  pattern : "",  name : "Customers",  view : "Master",  targetAggregation : "masterPages"  },  {  pattern : "Orders/{entity}",  name : "Details",  view : "Details",  targetAggregation : "detailPages"  }  ]    }    },    init: function() {  UIComponent.prototype.init.apply(this, arguments);    this.getRouter().initialize();    var oModel = new ODataModel("**http://db1.hana2.ucc.uwm.edu:8002/gbi-student-366/gbi/services/gbi.xsodata**");  this.setModel(oModel, 'gbi');  }    });    }); |

Listing

**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.



Next, metadata for the component is defined. 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 views defined using XML 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](http://help.sap.com/saphelp_uiaddon10/helpdata/en/90/2313063d6f45aeaa3388cc4c13c34e/content.htm?frameset=/en/ec/165265ce394490b20dadb2e45a6185/frameset.htm&current_toc=/en/e4/843b8c3d05411c83f58033bac7f072/plain.htm&node_id=305).

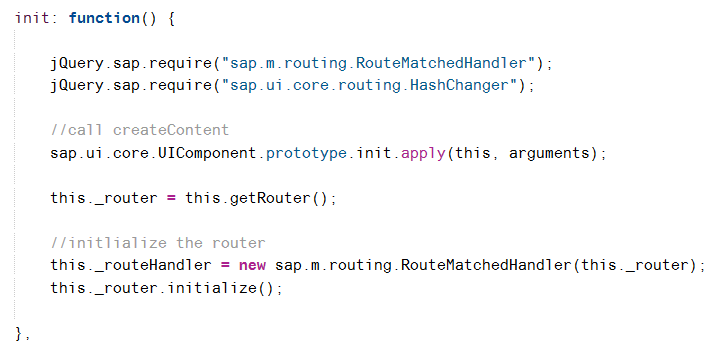


A SplitApp control has two aggregations: masterPages and detailPages. An aggregation is a collection to which multiple items can be bound. These collections can be bound to the views which make up the visual interface. The masterPages aggregation is shown on the left side of the application and the detailPages aggregation is shown on the right side of the screen. Each aggregation can have multiple views assigned to it and the routes define when each view is shown.

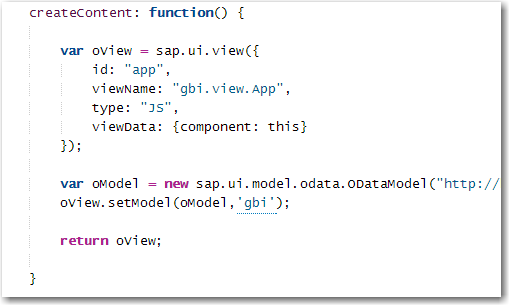
The first route in the image above has an empty pattern which causes this route to load automatically when the application loads. The name of the route is Customers and the view that will be loaded is Master.xml.view which will be loaded into the masterPages (left) aggregation.

The second route, a subroute of the Customers route is triggered with the pattern Orders/{entity}. The value {entity} will contain the id of the selected Customer when the route is invoked. The route loads the Details.view.xml view into the detailPages 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.

The createContent function creates the content of the component which consists of a single view called App. The App view encapsulates the other views in the application. It then creates the application model using the URL of the oData service document.



### App.view.xml

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

|  |
| --- |
| <mvc:View  xmlns:mvc="sap.ui.core.mvc"  displayBlock="true"  xmlns="sap.m" >  <SplitApp id="splitApp" />  </mvc:View> |

Listing

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.controller.Master" xmlns:mvc="sap.ui.core.mvc"  xmlns="sap.m">  <Page title="Customers">  <List  id="ShortCustomerList"  headerText="Customers"  items="{gbi>/Customers}" >  <StandardListItem  type="Active"  press="handleListItemPress"  title="{gbi>ID.CustomerID}"  description="{gbi>CompanyName}" />  </List>  </Page>  </mvc:View> |

Listing

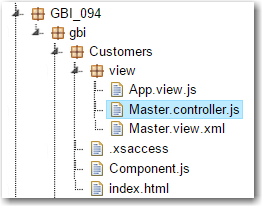
This code implements the initial view visible in the left side (master) of the screen. It consists of a List control that is bound to the Customers collection. The list items show the CustomerID and CompanyName fields. Setting the type attribute to Active makes the list items clickable and the press 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 controller 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("Details", {  from: "Master",  entity: entity[1]  });  }  });  }); |

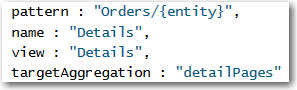
Listing



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 refers to the list item that was clicked. The argument is used to retrieve the binding context of the clicked list item. One of the properties of the binding context is the path within the oData collection to the object. This path will be of the form " /Customers('1000')". 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 1000.

The navTo method of the router object is used to navigate to the route that has the name Details and 1000 is passed as the parameter named entity. This matches the route (from the Component.js file):



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



The 12000 is the id of the customer who’s list item was clicked.

### Details.view.xml

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

|  |
| --- |
| <mvc:View  controllerName="gbi.controller.Details"  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="Customer Details" >  <l:Grid  defaultSpan="L12 M12 S12"  width="auto">  <l:content>  <f:SimpleForm id="idCusotmerForm"  minWidth="1024"  maxContainerCols="2"  editable="false"  layout="ResponsiveGridLayout"  title="Customer Details"  labelSpanL="4"  labelSpanM="4"  emptySpanL="0"  emptySpanM="0"  columnsL="2"  columnsM="2">  <f:content>    <core:Title text="Customer" />  <Label text="Number" />  <Text text="{gbi>ID.CustomerID}" />  <Label text="Name" />  <Text text="{gbi>CompanyName}" />  <Label text="Sales Org" />  <Text text="{gbi>SalesOrgID}" />  <core:Title text="Address" />  <Label text="Address" />  <Text text="{gbi>Address.Address}" />  <Label text="City" />  <Text text="{gbi>Address.City}" />  <Label text="Region" />  <Text text="{gbi>Address.Region}" />  <Label text="Postal Code" />  <Text text="{gbi>Address.Postal\_code}" />  <Label text="Country" />  <Text text="{gbi>Address.Country}" />  </f:content>  </f:SimpleForm>  </l:content>  </l:Grid>  <Table id="idOrdersTable"  inset="false"  itemPress = "handleTableRowPress"  items='{gbi>Orders}'>  <columns>  <Column>  <header>  <Text text="ID" />  </header>  </Column>  <Column>  <header>  <Text text="Created At" />  </header>  </Column>  <Column>  <header>  <Text text="Amount" />  </header>  </Column>  <Column>  <header>  <Text text="Currency" />  </header>  </Column>    </columns>  <items>  <ColumnListItem>    <Text text="{gbi>ID.OrderID}" />  <Text text="{gbi>CreatedAt}" />  <Text text="{gbi>GrossAmount.Amount}" />  <Text text="{gbi>GrossAmount.Currency}" />    </ColumnListItem>  </items>  </Table>    </Page>  </mvc:View> |

Listing

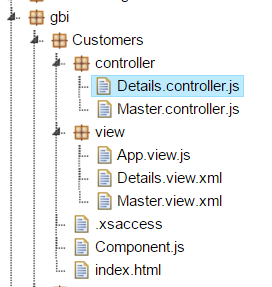
This view consists of a SimpleForm control at the top that shows the customer details. The form is enclosed within a Grid control that controls the form’s width. Below the form is a table that shows the customer’s orders.

### Details.controller.js

Create a file called Details.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.Details", {    onInit: function() {    this.router = sap.ui.core.UIComponent.getRouterFor(this);  this.router.attachRoutePatternMatched(this.onRouteMatched, this);    },    onRouteMatched : function(oEvent) {  var oParameters = oEvent.getParameters();  this.getView().bindElement("gbi>/Customers('" + oParameters.arguments.entity + "')");    }  });  }); |

Listing



In the onInit function, a reference to the application’s router is retrieved and then the onRouteMatched function is assigned to run when the router routes to this view.

In the onRouteMatched function the parameters of the route matched event are retrieved. The parameters include the entity value (which contains the customer id) 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 /Customers(‘12000’). 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 customer with ID equal to 12000. Next, a reference to the gbi model is retrieved and the model and sEntityPath are used to create a binding context that references the customer. The binding context is bound to the Details view. This makes the customer data available to the SimpleForm control.

Getting the customer’s orders in the table makes use of the association between the Customers service and the SalesOrders service. The image below shows the definition of the Customers service from the gbi.xsodata file.



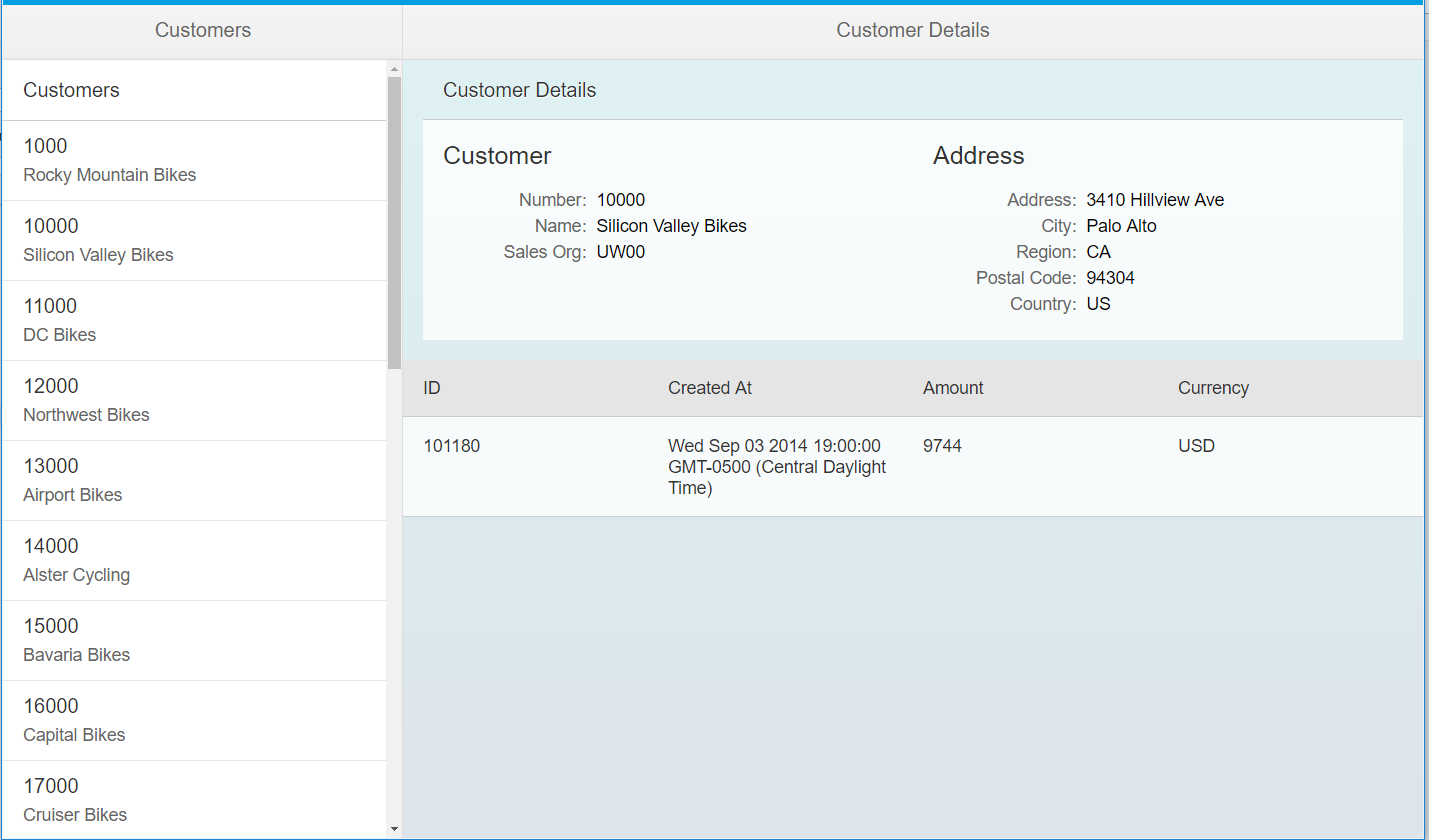
Notice that, in order to retrieve a customer’s orders, you would use something like this:

<service document URL>/Customers(‘12000’)/Orders

We just saw how the code in onRouteMatched bound the view to the customer (e.g. /Customers(‘12000’)). If you look at the binding for items in the Table control in the Details.view.xml file you will see that it is bound to Orders. Since the Table control is embedded in the view, the combined bindings mean the table items are bound to something like /Customers(‘12000’)/Orders.

## Run the Application

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 customers in the master page section. When you click a customer, the customer’s details and orders are shown.



## Add Some Formatting

We can improve the app by adding some formatting to dates and currency values. Formatting in XML views is handled by value converters which are functions in the view’s controller that receive the value before it is displayed, formats it and returns it to be displayed.

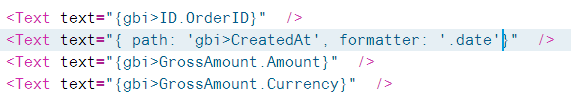
### Format the dates

Add the following function

|  |
| --- |
| date: function(date){  return new Date(date).toLocaleDateString();  } |

Listing 8

Now update the code in the view that displays the date:



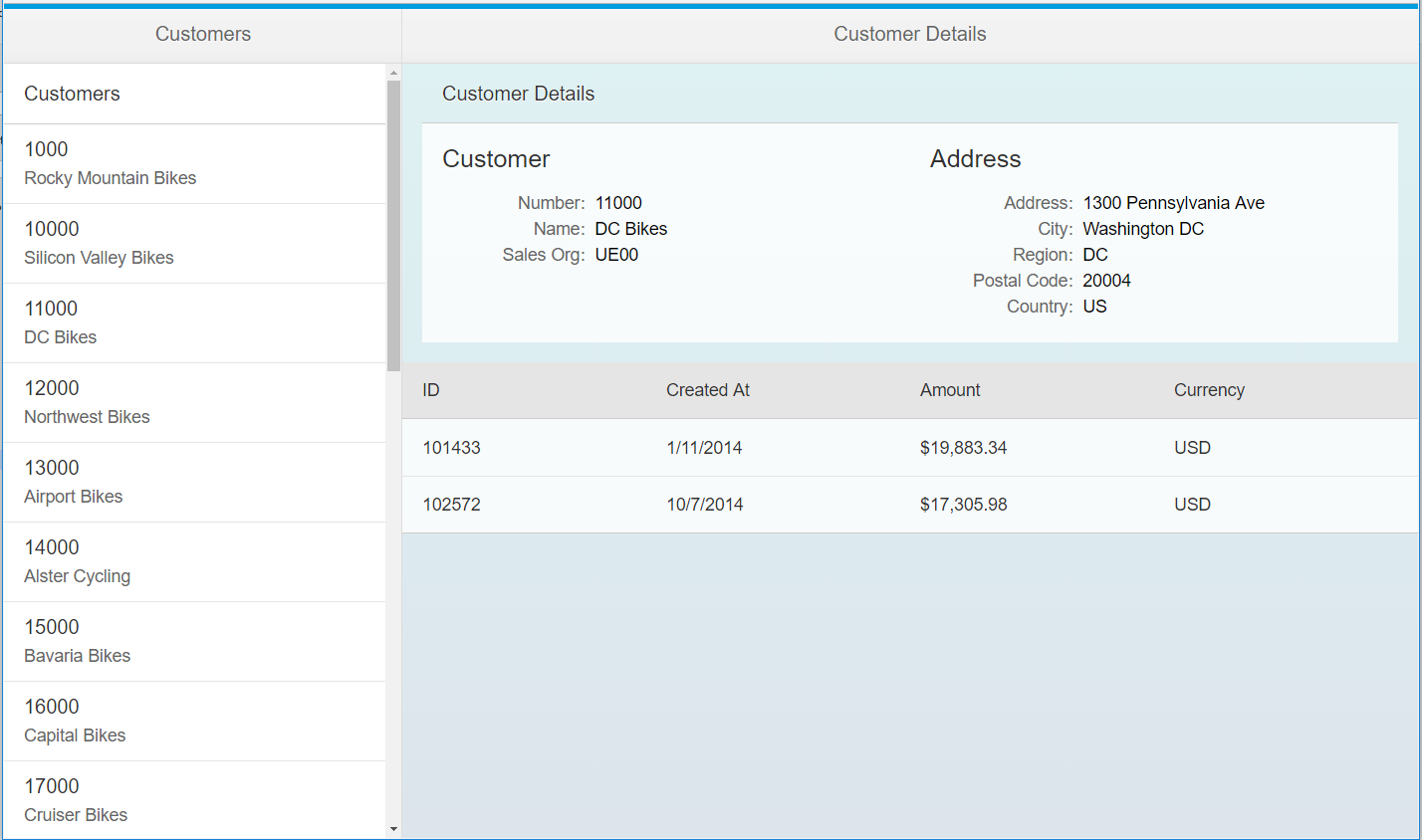
### Format Currencies

Add the following function to the Details controller.

|  |
| --- |
| 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 9

Update the view to use the formatter function for the Gross Amount in the table. The application should now look like this:



## Exercise

Create the master/detail application using Sales Orders and Sales Order Details depicted in the image below.

