HD3C17 – Customer CRUD Application

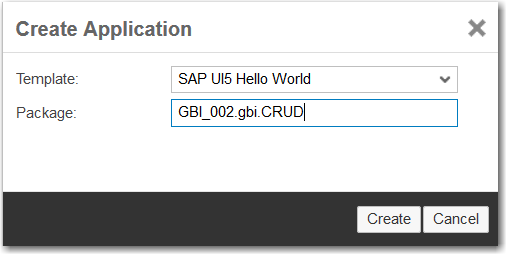
|  |  |
| --- | --- |
| **Product and Focus**  HANA Platform/oData | **MOTIVATION**  This case describes the user of oData services to implement CRUD operations.  **PREREQUISITES**  HD3C05 – Create the Persistence Model |
| **Target Audience**  Undergraduate/Graduate Beginner to Intermediate |
| **Author**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

# Customer CRUD Application

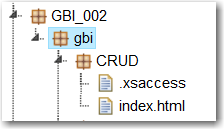
This cases builds on the previous two cases in this series to develop an application which allows the user to create, update and delete customers.

## 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 **CRUD** to the Package.



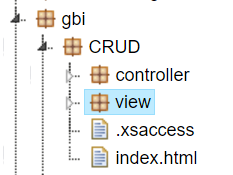
The package is created and basic application is created.



Note there is no .xsapp file. That file is unnecessary because in the previous 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.

### 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" />  <meta charset="UTF-8">  <title>Customer CRUD App</title>    <!-- Bootstrap the SAPUI5 libraries and create the namespaces for the application -->  <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-resourceroots = '{  "gbi" : "./"  }'>  </script>  <!-- Load the component defined in the Component.js file -->  <script>  new sap.ui.core.ComponentContainer({  name: "gbi"  }).placeAt("content")    </script>  </head>  <body class="sapUiBody" id="content">  </body>  </html> |

Listing

### Component.js

Create the Component.js file in the CRUD 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: "idAppControl",  clearTarget: false,  transition: "slide"  },  routes: [  {  pattern: "",  name: "Customers",  view: "Customers",  targetAggregation: "pages"  }  ]  }  },  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

Replace the highlighted code with your own service URL.

### 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" >  <App id="idAppControl" />  </mvc:View> |

Listing

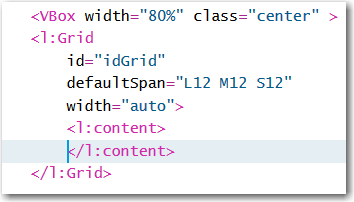
This code creates the App control which defines the overall structure of the interface which, unlike the previous cases that use a SplitApp control, is a full screen application.

### Customers.view.xml

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

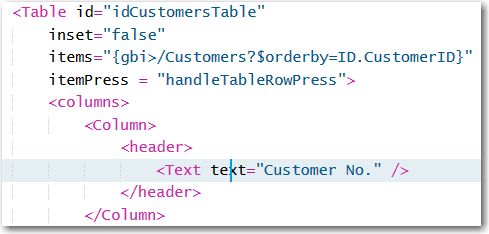
|  |
| --- |
| <mvc:View width="100%" controllerName="gbi.controller.Customers" xmlns="sap.m"  xmlns:l="sap.ui.layout" xmlns:core="sap.ui.core" xmlns:f="sap.ui.layout.form" xmlns:mvc="sap.ui.core.mvc" >  <Page title="GBI Customers" id="idPage">    <VBox width="80%" justifyContent="Center">    <l:Grid  id="idGrid"  defaultSpan="L10 M12 S12"  width="auto">  <l:content>  </l:content>  </l:Grid>    <Table id="idCustomersTable"    inset="true"  items="{gbi>/Customers?$orderby='ID.CustomerID'}"  itemPress = "">  <columns>  <Column>  <header> <Text text="Customer No." />  </header>  </Column>  <Column>  <header>  <Text text="Company" />  </header>  </Column>    <Column>  <header>  <Text text="Sales Org." />  </header>  </Column>  </columns>  <items>  <ColumnListItem type="Navigation">  <Text text="{gbi>ID.CustomerID}" />  <Text text="{gbi>CompanyName}" />  <Text text="{gbi>SalesOrgID}" />  </ColumnListItem>  </items>  </Table>  </VBox>  <footer>  <Bar>  <contentRight>  <Button id="create" text="Create" press="handleCreatePress" />  <Button id="edit" text="Edit" press="handleEditPress" />  <Button id="delete" text="Delete" type="Reject" visible="false" press="handleDeletePress" />  <Button id="save" text="Save" type="Emphasized" visible="false" press="handleSavePress" />  <Button id="cancel" text="Cancel" visible="false" press="handleCancelPress" />  </contentRight>  </Bar>  </footer>  </Page>  </mvc:View> |

Listing

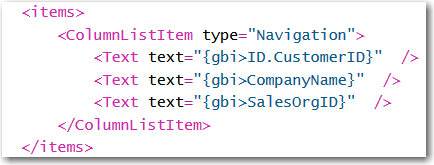
This code implements the main interface of the application. There are two main parts of the interface. These are organized using a VBox control which is used to arrange elements vertically. The first part inside the VBox control is a Grid control. This control is a placeholder into which the Edit and Display fragments are inserted (we’ll create these below). 

The second part is a Table control that is bound to the Customers entity. This is an aggregation binding, because it will return multiple objects. The Table’s aggregation binding is items so an item (defined below) will be created for each object returned by the service.

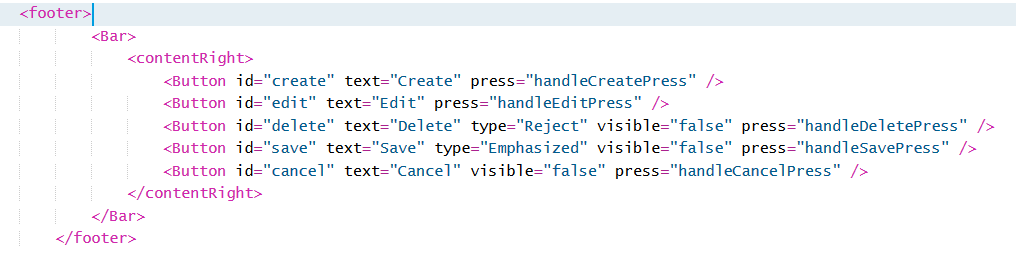
The itemPress event is fired when the user clicks an item in the table. This is assigned to a function called handleTableRowPress which will be defined in the Customers.controller.js file. Within the Table control the columns are defined…



…and then the row structure is defined as items.



Finally, the footer of the page is defined. Note that three of the Buttons will not be visible when the application loads.



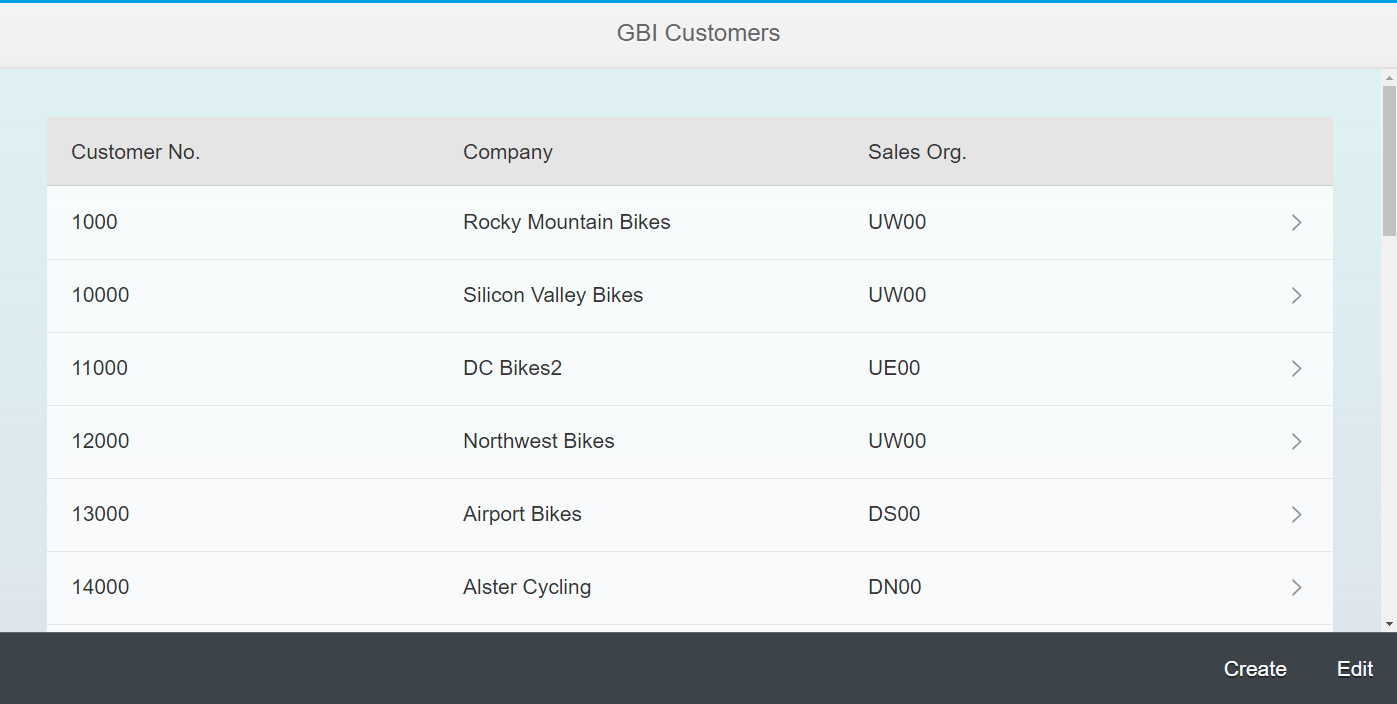
### Customers.controller.js

Create the Customers.controller.js file in the view package and add the following code to it.

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"  ], function(Controller) {  "use strict";  return Controller.extend("gbi.controller.Customers", {    });  }); |

Listing

If you run the application now, it should like this:



Not much so far, so let’s add to it.

## Create Fragments

Next we need to create two fragments: one to display customer details and one to edit customer details.

### Display.fragment.xml

The first thing to do is create a fragment that will be loaded above the table to display the details of the selected customer when a user clicks an item on the list. Create a file called Display.fragment.xml in the view folder and add the following code.

|  |
| --- |
| <core:FragmentDefinition  xmlns="sap.m"  xmlns:l="sap.ui.layout"  xmlns:f="sap.ui.layout.form"  xmlns:core="sap.ui.core">  <l:Grid  defaultSpan="L12 M12 S12"  width="auto">  <l:content>  <f:SimpleForm id="idCustomerForm"  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="Company" />  <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>  </core:FragmentDefinition> |

Listing

This code implements a SimpleForm control that displays customer data. Unlike the form in Edit.fragment.xml (described later), this form uses Text controls rather than Input controls. This fragment is loaded dynamically when the application first loads and when the user clicks cancel or save while editing.

#### Customers.controller.js

Next, we have to add code to the Customers.controller.js file that loads the display fragment. Add the following code to the end of the controller file. Make sure you add a comma to the end of the onInit function.

|  |
| --- |
| \_formFragments: {},    \_getFormFragment: function (sFragmentName) {  //Retrieve the form fragement from the \_formFragments object  //If it has been created before it will exist in the object  var oFormFragment = this.\_formFragments[sFragmentName];    //If the form fragment has already been created, return it  if (oFormFragment) {  return oFormFragment;  }  //If it hasn't been created before load it from the file in the view package  oFormFragment = sap.ui.xmlfragment(this.getView().getId(), "gbi.view." + sFragmentName);  //Add it to the \_formFragments object and return it  return this.\_formFragments[sFragmentName] = oFormFragment;  },  \_showFormFragment : function (sFragmentName) {  //Get a reference to the grid control  var oGrid = this.getView().byId("idGrid");  //Delete the currrent content  oGrid.removeAllContent();  //Insert the new fragment  oGrid.insertContent(this.\_getFormFragment(sFragmentName));    } |

Listing

The first line of code creates an object to hold the fragments so that we don’t have to create them over and over. The \_getFormFragment function checks to see if the fragment already exists in \_fomrFragments. If it does, it is returned. If it doesn’t, it creates the fragment, stores it and returns it.

The \_showFormFragment function deletes anything that is already displayed, retrieves the fragment and displays it.

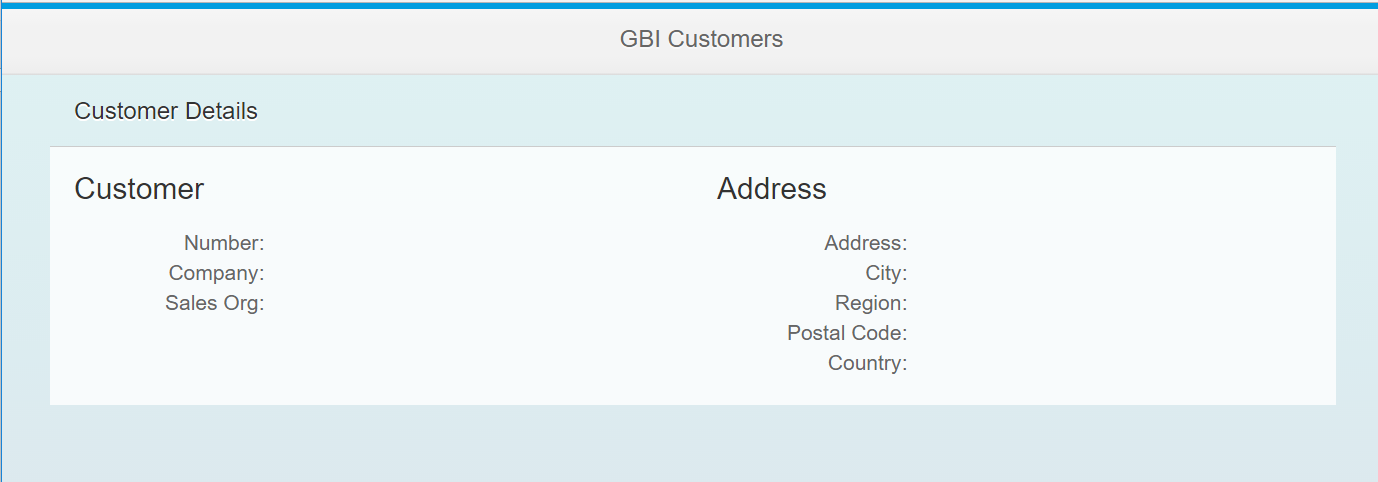
Finally, add onInit function.

|  |
| --- |
| onInit: function() {  this.\_showFormFragment("Display");  } |

Listing

This code will display the Display fragment when the view loads.

If you run the app now, you should see the fragment but it doesn’t show any data.



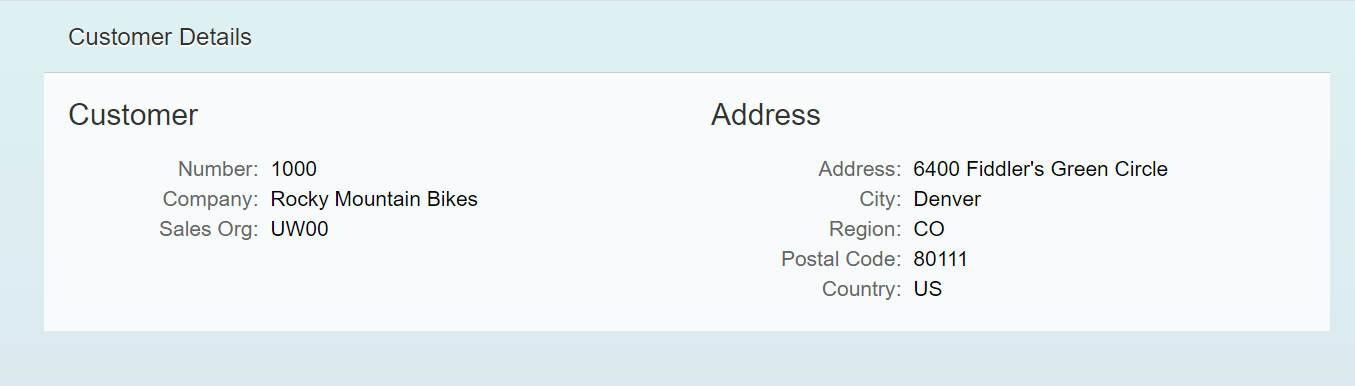
To show data we need to add a press event handler to the table row. Add the function shown below to the Customers.controller.js and then update the itemPress property in the Table control in the Customer view with the function name.

|  |
| --- |
| handleTableRowPress : function(oEvent){  //When a table row is clicked, get its binding context and set the display form's binding context  var context = oEvent.getParameter("listItem").getBindingContext('gbi');  this.getView().byId("idCustomerForm").setBindingContext(context,'gbi');  } |

Listing

The code gets the binding context from the list item clicked which will include the customer’s data and then binds it to the CustomerForm in the display fragment.

Now, if you click a list item, the data is displayed.



### Edit.fragment.xml

Create a file called Edit.fragment.xml in the view folder and add the following code.

|  |
| --- |
| <core:FragmentDefinition  xmlns="sap.m"  xmlns:l="sap.ui.layout"  xmlns:f="sap.ui.layout.form"  xmlns:core="sap.ui.core"  xmlns:c="sap.ui.commons">  <l:Grid  defaultSpan="L12 M12 S12"  width="auto">  <l:content>  <f:SimpleForm id="idEditForm"  minWidth="1024"  maxContainerCols="2"  editable="true"  layout="ResponsiveGridLayout"  title="Customer Information"  labelSpanL="2"  labelSpanM="2"  emptySpanL="4"  emptySpanM="4"  columnsL="2"  columnsM="1"  class="editableForm">  <f:content>  <core:Title text="Company" />  <Label text="Customer No." />  <Text id="idCustomerID"  text="{edit>ID.CustomerID}" />  <Label text="Company Name:" />  <Input id="idCompanyName"  maxLength="35"  value="{edit>CompanyName}" />  <Label text="Sales Org" />  <Input id="idSalesOrg"  maxLength="4"  value="{edit>SalesOrgID}" />  <core:Title text="Address" />  <Label text="Address" />  <Input id="idAddress"  maxLength="35"  value="{edit>Address.Address}" />  <Label text="City" />  <Input id="idCity"  maxLength="20"  value="{edit>Address.City}" />  <Label text="Region" />  <Input id="idRegion"  maxLength="2"  value="{edit>Address.Region}" />  <Label text="Postal Code" />  <Input id="idPostalCode"  maxLength="5"  value="{edit>Address.Postal\_code}" />  <Label text="Country" />  <Input id="idCountry"  maxLength="5"  value="{edit>Address.Country}" />  </f:content>  </f:SimpleForm>  </l:content>  </l:Grid>  </core:FragmentDefinition> |

Listing

This code implements a SimpleForm control that allows the user to edit customer data.

#### Customers.controller.js

The code to load the fragment already exists. However, we need to add code to the Edit buttons press event handler that will initiate the load. Add this function to the Customers controller.

|  |
| --- |
| handleEditPress : function () {  //Create a flag so we know that a customer is being edited  this.editFlag = true;  //Clone the data so we can cancel the changes if necessary  this.\_oCustomer = jQuery.extend({}, this.getView().byId("idCustomerForm").getBindingContext('gbi').getObject());    //Add the object ot edit to the edit model  this.getView().getModel("edit").setData(this.\_oCustomer);    //Save the binding context of the display form so we can reset it if necessary  this.\_context = this.getView().byId("idCustomerForm").getBindingContext('gbi');  //Save the binding path so we can perform puts and posts  this.\_sPath = this.getView().byId("idCustomerForm").getBindingContext('gbi').sPath.slice(1);  this.\_toggleButtonsAndView("Edit");    },  \_toggleButtonsAndView : function (form) {  var oView = this.getView();  if(form === "Edit"){  oView.byId("edit").setVisible(false);  oView.byId("create").setVisible(false);  oView.byId("save").setVisible(true);  oView.byId("cancel").setVisible(true);  oView.byId("delete").setVisible(true);  this.\_showFormFragment("Edit");  }  } |

Listing

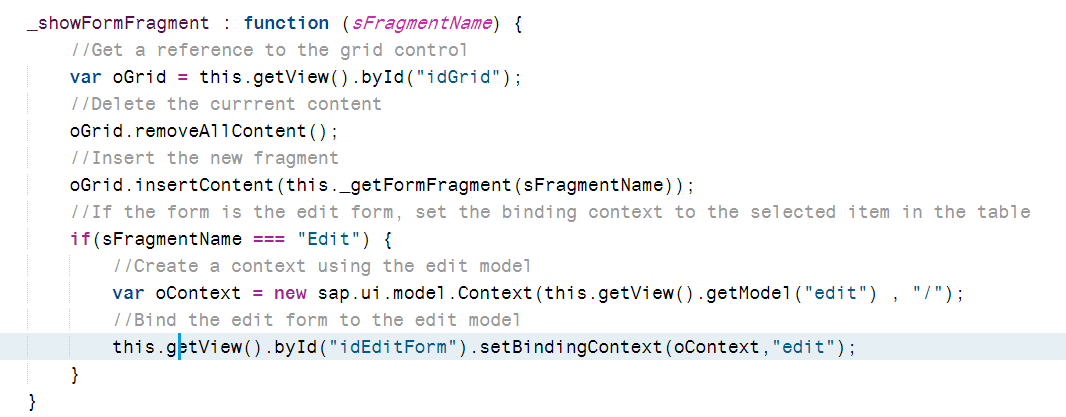
This code set’s a flag so that we know that we are editing an existing record. It then clones the customer object so we can cancel edits and restore the original data. This uses a model named edit which we will add in a minute. It retrieves the binding context and saves the path to the selected customer. It then calls \_toggleButtonsAndView to show the fragment and toggles button visibilities.

Update the \_showFormFragment function by adding this to the end of the function

|  |
| --- |
| if(sFragmentName === "Edit") {  //Create a context using the edit model  var oContext = new sap.ui.model.Context(this.getView().getModel("edit") , "/");  //Bind the edit form to the edit model  this.getView().byId("idEditForm").setBindingContext(oContext,"edit");  } |

Listing

This code retrieves a binding context from the edit model and binds it to the edit form.

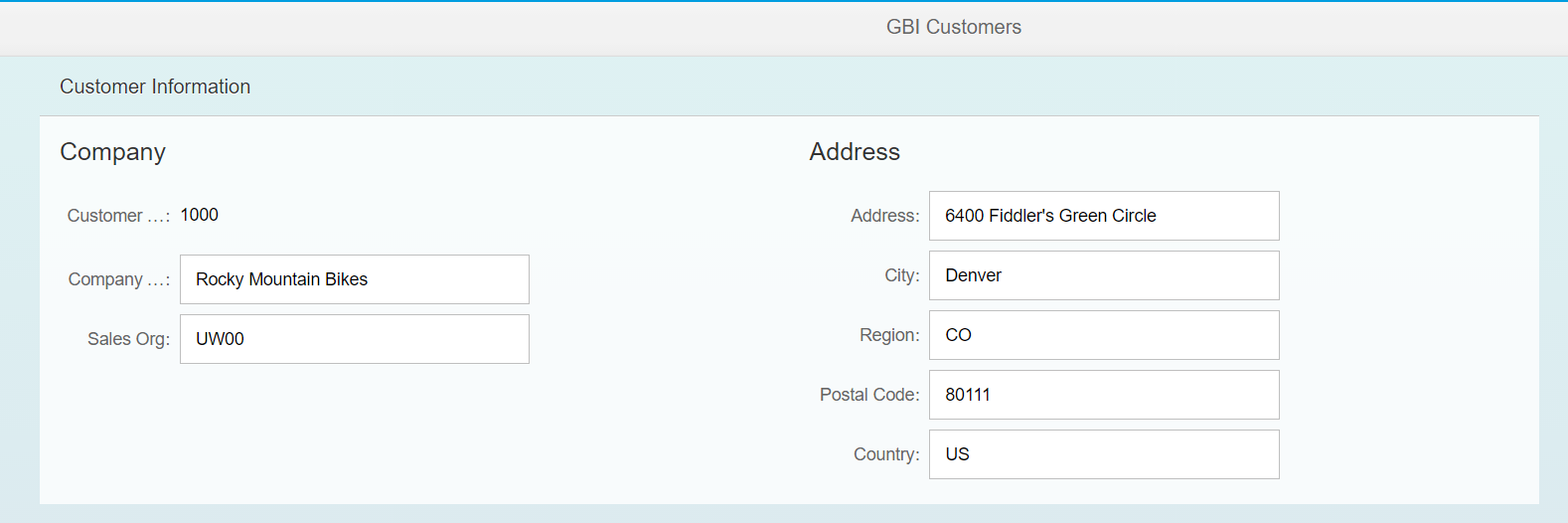


Next you need to create a model that will hold the record that is being edited. This is necessary so you can cancel the edit. Make the highlighted changes to the Component declaration on onInit function.

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"**,**  **"sap/ui/model/json/JSONModel"**  ], function(Controller, **JSONModel**) {  "use strict";  return Controller.extend("gbi.controller.Customers", {  onInit: function() {  **var jModel = new JSONModel("");**  **this.getView().setModel(jModel, 'edit');**  this.\_showFormFragment("Display");  } |

Listing

Run the application, select a customer and click the Edit button.



### Cancel Button

Add the following function to implement the Cancel button press event handler.

|  |
| --- |
| handleCancelPress : function () {  //Restore the data  var oModel = this.getView().getModel('gbi');  //Put back the original data  oModel.oData[this.\_sPath] = this.\_oCustomer;  this.\_toggleButtonsAndView("Display");  } |

Listing

This code restores the customer data, discarding edits and then calls \_toggleButtonsAndView to adjust the interface.

Then add this code to \_toggleButtonsAndView.

|  |
| --- |
| if(form === "Display"){  oView.byId("edit").setVisible(true);  oView.byId("create").setVisible(true);  oView.byId("save").setVisible(false);  oView.byId("cancel").setVisible(false);  oView.byId("delete").setVisible(false);  this.\_showFormFragment("Display");  } |

Listing

## CUD Operations

Now that we can display the data in a form, we can implement the CUD operations.

### Delete

The code below shows the Delete button event handler.

|  |
| --- |
| handleDeletePress: function() {  //Confirm the user wants to delete the customer  var path = this.\_sPath;  var oModel = this.getView().getModel("gbi");  MessageBox.confirm(  "Are you sure you want to delete the customer?", {  icon: MessageBox.Icon.INFORMATION,  title: "Confirm Delete",  initialFocus: MessageBox.Action.CANCEL,  onClose: function(oAction) {  if (oAction === "OK") {  oModel.remove('/' + path, null, function() {  MessageToast.show("Delete successful");  }, function() {  MessageToast.show("Delete failed");  });  }  }  }  );  this.\_toggleButtonsAndView("Display");  } |

Listing

This code retrieves the path that was saved when the table row was clicked. It then shows a MessageBox so the user can confirm the deletion. If the user clicked OK the model is retrieved and the and the record at the saved path is removed. The model will ensure the database is updated. A toast message shows whether the operation was successful.

Make the highlighted changes below to the top of the controller code in order to make the MessageBox and Toast available.

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller",  "sap/ui/model/json/JSONModel"**,**  **"sap/m/MessageBox",**  **"sap/m/MessageToast"**  ], function(Controller, JSONModel, MessageBox, MessageToast) { |

Listing

### Update

Add the following Save event handler to the controller file.

|  |
| --- |
| handleSavePress : function () {  //Retrieve the view's model  var oModel = this.getView().getModel('gbi');    //Create an object and add the customer properties to it from the form  var oEntry = {};    //Check whether the customer is being edited or created  oEntry["CompanyName"] = this.byId("idCompanyName").getValue();  oEntry["SalesOrgID"] = this.byId("idSalesOrg").getValue();  oEntry["Address.Address"] = this.byId("idAddress").getValue();  oEntry["Address.City"] = this.byId("idCity").getValue();  oEntry["Address.Region"] = this.byId("idRegion").getValue();  oEntry["Address.Country"] = this.byId("idCountry").getValue();  oEntry["Address.Postal\_code"] = this.byId("idPostalCode").getValue();    if(this.editFlag){  oEntry["ID.CustomerID"] = this.byId('idCustomerID').getText();  //Perform a PUT  oModel.update('/' + this.\_sPath, oEntry, null, function(data){  MessageToast.show("Update successful");  },function(data){  MessageToast.show("Update failed");  });  } else {  }  this.\_toggleButtonsAndView("Display");  } |

Listing

This code retrieves a reference to the gbi model then creates an object to hold the data we will send to the server. It then transfers the property values from the form controls to this object.

Next, we have an if statement that checks the editFlag variable. If editFlag is true, then this is an existing record so we issue a PUT to the server. We’ll add the else portion that issues a POST later. The results of the PUT are shown using a Toast.

Now, you can edit a customer record and save it to the database. You can open the developer tools and examine the results of the PUT service call.



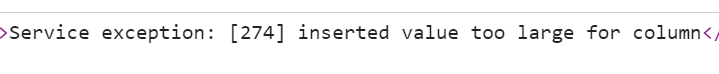


The 204 status code means that the update was successful but no content was returned from the server.

Keep in mind that you must adhere to the constraints of the data in the database. For example, we set the length of the Address.Country field to 2 so you will receive an error if you attempt to save a longer string.



The 500 status code means there was an error on the server. If you open the call and look at the response, you get a little more information although it could be more helpful.



### Create

The create Button event handler is shown below.

|  |
| --- |
| handleCreatePress: function() {  var oEntry = {};  //Check whether the customer is being edited or created  oEntry["CompanyName"] = "";  oEntry["SalesOrgID"] = "";  oEntry["Address.Address"] = "";  oEntry["Address.City"] = "";  oEntry["Address.Region"] = "";  oEntry["Address.Country"] = "";  oEntry["Address.Postal\_code"] = "";  this.getView().getModel("edit").setData(oEntry);  //Create pressed so set editFlag to false  this.editFlag = false;  this.\_toggleButtonsAndView("Edit");  } |

Listing

This code sets the editFlag variable to false so we know this is a new customer and then toggles the interface.

Add the code below to the else block of the if statement in the handleSavePress function.

|  |
| --- |
| oEntry["ID.CustomerID"] = 0;  //Perform a POST  oModel.create('/Customers', oEntry, null, function(data){  sap.m.MessageToast.show("Update successful");  },function(data){  sap.m.MessageToast.show("Update failed");  }); |

Listing

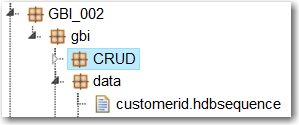
This code performs the POST call to the server. Note that it sets the ID.CustomerID field to 0. This field is the primary key so when we create a new record we have to be sure we use a unique value. To make that happen, we have to create a sequence on the server that will automatically create a unique value for the field and then we have to create a procedure to create the customer record so that we can utilize the sequence.

#### Implement the Create Procedure

As the application stands now you can create a customer but its ID will be 0 and you will not be able to create a second customer because the ID field is the primary key. In order to create multiple customers with valid ID fields we will use a sequence. A sequence will automatically choose the next value in a defined sequence each time it is invoked. To use a sequence we must create the sequence definition, bypass the oData customer create process with a SQLScript procedure and update the oData service to use the procudure when creating a customer.

##### Define the Sequence

Create a file in the GBI\_###\gbi\data package called **customerid.hdbsequence**.



Enter the code shown below.

|  |
| --- |
| schema = "GBI\_002";  start\_with = 25000;  increment\_by = 1000;  nomaxvalue = true;  nominvalue = true;  cycles = false; |

Listing

Update the highlighted code to use your schema name.

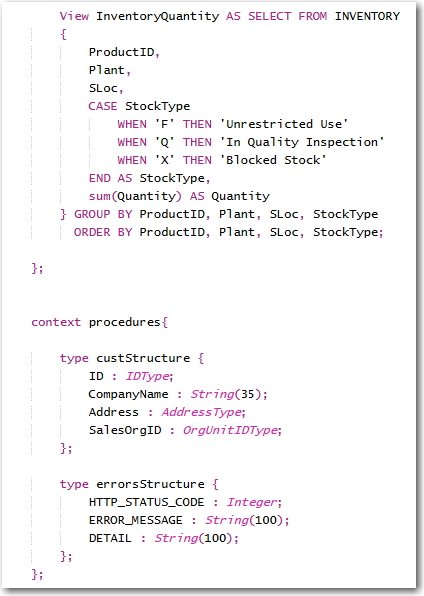
This will create a sequence that begins with the number 25000 and increments by 1000 each time. No min or max value are specified and the sequence will not cycle. It’s possible to set up a sequence that will cycle back to a min value when it reaches a max value.

##### Create the Procedure

To use a procedure, we must first create some table types that will contain the input and error values. Open the **GBI\_###.hdbdd** file from the data package. Add the following code just inside the closing } at the bottom of the file. This creates a new context so make sure it is not inside the MASTERDATA, SALES or LOGISTICS contexts.

|  |
| --- |
| context procedures{    type custStructure {  ID : IDType;  CompanyName : String(35);  Address : AddressType;  SalesOrgID : OrgUnitIDType;  };    type errorsStructure {  HTTP\_STATUS\_CODE : Integer;  ERROR\_MESSAGE : String(100);  DETAIL : String(100);  };  }; |

Listing

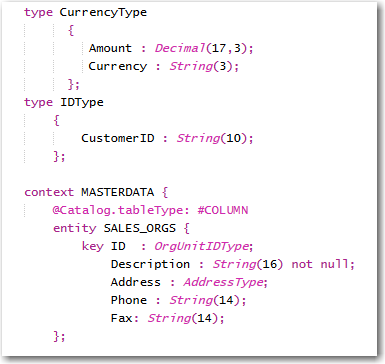


Notice the IDType custom data type for the ID field of the custStructure. We’ll define that type by adding the code shown below to the other custom types at the top of the file.

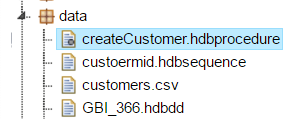
|  |
| --- |
| type IDType  {  CustomerID : String(10);  }; |

Listing

We must use this custom type because the ID field in the CUSTOMER field is a compound field (ID.CustomerID) because of the association with SALES\_ORDERS. The code that defines the custStructure type does not allow compound field names.



Now, create a file called **createCustomer.hdbprocedure** in the data package.



Copy the code below into the file.

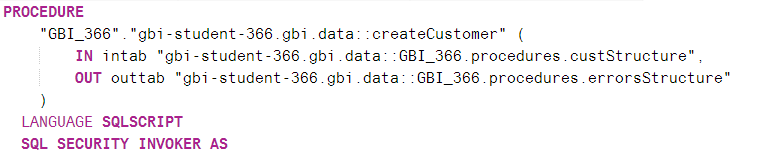
|  |
| --- |
| PROCEDURE  "**GBI\_366"."gbi-student-366**.gbi.data::createCustomer" (  IN intab "**gbi-student-366.gbi.data::GBI\_366**.procedures.custStructure",  OUT outtab "**gbi-student-366.gbi.data::GBI\_366**.procedures.errorsStructure"  )  LANGUAGE SQLSCRIPT  SQL SECURITY INVOKER AS  --DEFAULT SCHEMA <schema>  --READS SQL DATA AS  begin  declare lv\_id string;  declare lv\_companyname string;  declare lv\_address string;  declare lv\_region string;  declare lv\_city string;  declare lv\_country string;  declare lv\_postalcode string;  declare lv\_salesorg string;  select "ID.CustomerID", "CompanyName", "Address.Address", "Address.City", "Address.Country", "Address.Region", "Address.Postal\_code", "SalesOrgID"  into lv\_id, lv\_companyname, lv\_address, lv\_city, lv\_country, lv\_region, lv\_postalcode, lv\_salesorg from :intab;  if :lv\_companyname = '' then  outtab = select 500 as http\_status\_code,  'Company Name cannot be null ' as error\_message,  ' No Way! CompanyName field must not be empty' as detail from dummy;  else  insert into "**gbi-student-366.gbi.data::GBI\_366**.MASTERDATA.CUSTOMERS"  values ("**gbi-student-366.**gbi.data::customerid".NEXTVAL, lv\_companyname, lv\_address, lv\_city, lv\_country, lv\_region, lv\_postalcode, lv\_salesorg);  end if;  end; |

Listing

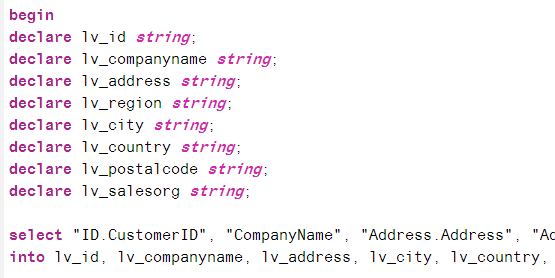
Update the highlighted portions to reflect your id.

The first part of the code defines the name of the procedure. The first GBI\_366 is the schema name. The portion gbi-student-366.gbi.procedures is the path to the createCustomer.ndbprocedure file.

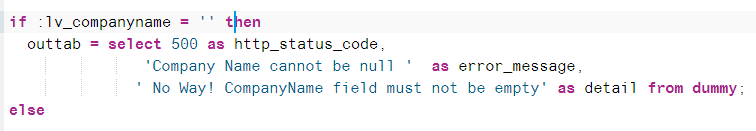
The IN parameter is used to pass the input values to the procedure and makes use of the custStructure type you created above. Again, gbi-student-366.gbi.data is the path to the GBI\_###.hdbdd file and GBI\_###.procedures are the nested contexts in the GBI\_###.hdbdd file.



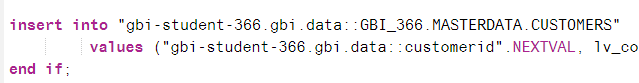
The next portion defines variables to hold the input values and then uses a SELECT statement to retrieve the input values from the intab input parameter.



The CompanyName field cannot be null so the next step in the procedure is to make sure a value was passed in for CompanyName. If it wasn’t, an error is passed back to the application.

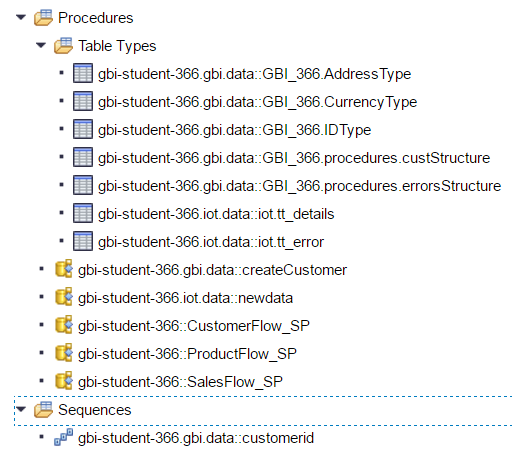


Finally, the record is inserted into the CUSTOMERS table.



Note the reference to the customerid sequence for the ID field value.

If you look at your schema in the Catalog editor you will see the custom types, the sequence and the procedure.

****

### Update the oData Service

Open the **gbi.xsodata** file and modify the Customers service as shown below.

|  |
| --- |
| "gbi-student-366.gbi.data::GBI\_366.MASTERDATA.CUSTOMERS" as "Customers"  navigates ("CustomerOrd" as "Orders")  create using "**gbi-student-366**.gbi.data::createCustomer"; |

Listing

Now, you can create customers with the application and the sequence will insert a new value for the ID of each new customer.