

# HD2dC06w – Customer CRUD Application

## Product and Focus

HANA Platform/oData

## Target Audience

Undergraduate/Graduate  
Beginner to Intermediate

## Author

Ross Hightower

## MOTIVATION

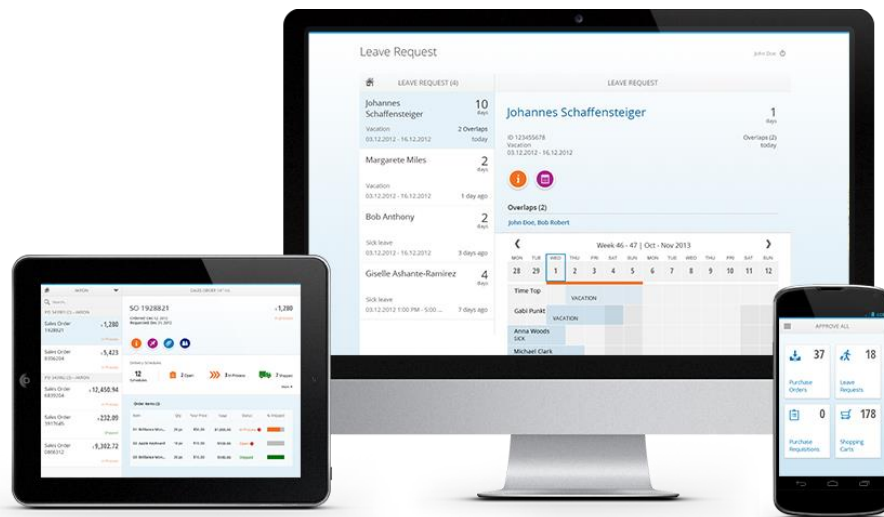
This case describes the user of oData services to implement CRUD operations.

## PREREQUISITES

HD1C01 – Hello World MVC App

HD1dC02 – Create the Persistence Model

HD1dC03 – oData Services

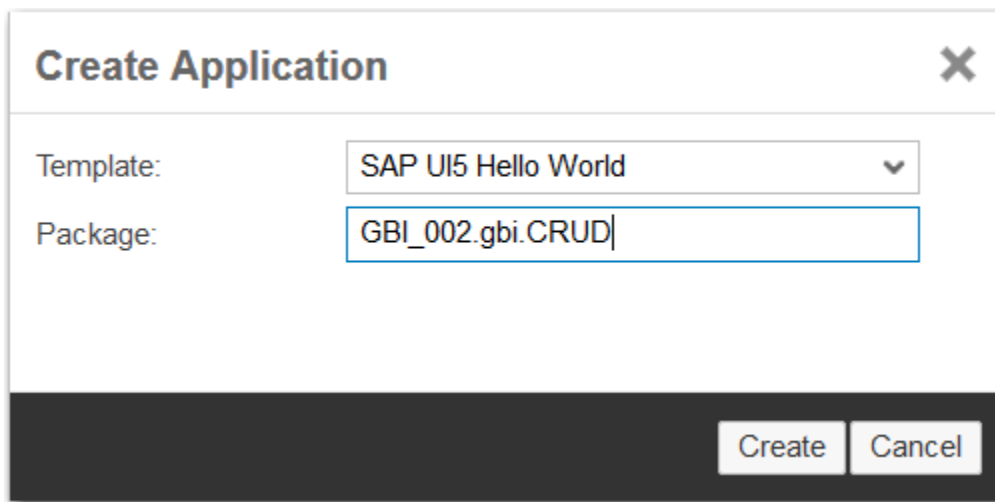


## Customer CRUD Application

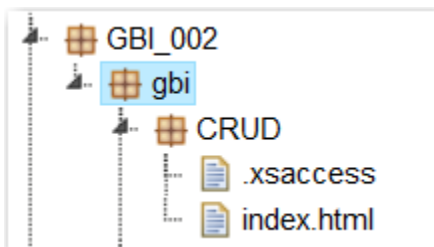
This cases builds on the previous two cases in this series to develop an application which allows the user to create, update and delete customers.

### Create the Application Packages


Login 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.



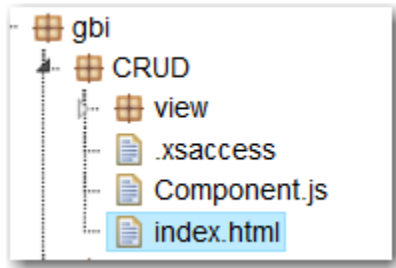
The package is created and basic application is created.



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** package 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 this application we will also load css file to illustrate how to use CSS styles.

#### 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 1

For an explanation of this code see 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", {

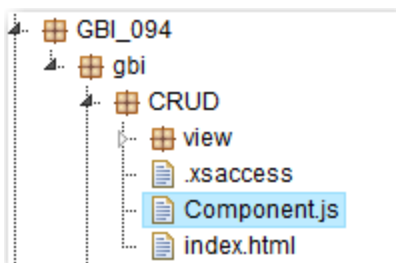
    createContent : function() {

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

        return oView;
    }
});
```

Listing 2

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



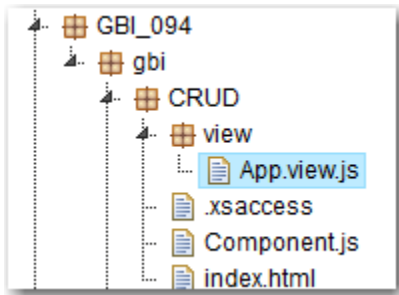
### App.view.js

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

```
sap.ui.jsview("gbi.view.App", {  
  
    createContent: function (oController) {  
  
        // to avoid scroll bars on desktop the root view must be set to block display  
        this.setDisplayBlock(true);  
  
        // create app  
        this.app = new sap.m.App();  
  
        // load the master page using an XML view  
        var customer = sap.ui.xmlview("Customers", "gbi.view.Customers");  
        this.app.addPage(customer, true);  
  
        // done  
        return this.app;  
    }  
});
```

Listing 3

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. Also unlike the previous cases, there is no need for navigation in this case so there is no router in the Component.js file and the main view of the application (Customers) is loaded in the App.view.js file.



#### 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:layout="sap.ui.layout"  
    xmlns:form="sap.ui.layout.form"  
    xmlns:core="sap.ui.core"  
    xmlns:c="sap.ui.commons">  
    <!:Grid  
        defaultSpan="L12 M12 S12"  
        width="auto">
```

```
</!: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>
</!:content>
```

```
</l:Grid>
</core:FragmentDefinition>
```

*Listing 4*

This code implements a SimpleForm control that allows the user to edit customer data. Fragments are blocks of code that can be loaded dynamically by SAPUI5. This form will be used to edit a customer and will be loaded when the user clicks the Edit button.

#### Display.fragment.xml

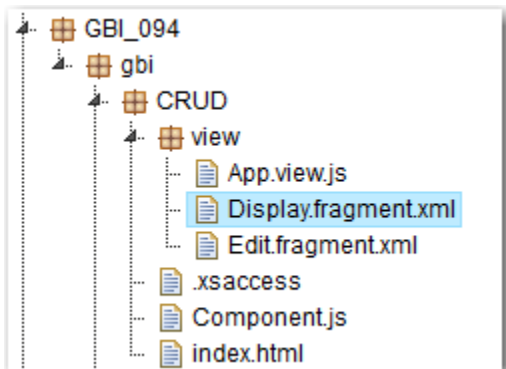
Create a file called Display.fragment.xml in the view folder and add the following code.

```
<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}" />
        </f:content>
      </f:SimpleForm>
    </l:content>
  </l:Grid>
</core:FragmentDefinition>
```

```
<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 5

This code implements a SimpleForm control that displays customer data. Unlike the form in Edit.fragment.xml, 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.view.xml

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

```
<mvc:View width="100%" controllerName="gbi.view.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%" fitContainer="true" 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 = "handleTableRowPress">
<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 6

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. The code that inserts the fragments is contained in the Customers.controller.js file.

```
<VBox width="80%" class="center" >
  <l:Grid
    id="idGrid"
    defaultSpan="L12 M12 S12"
    width="auto">
    <l:content>
    </l:content>
  </l:Grid>
```

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...

```
<Table id="idCustomersTable"
  inset="false"
  items="{gbi}/Customers?$orderby=ID.CustomerID}"
  itemPress = "handleTableRowPress">
  <columns>
    <Column>
      <header>
        <Text text="Customer No." />
      </header>
    </Column>
```

...and then the row structure is defined as items.

```
<items>
  <ColumnListItem type="Navigation">
    <Text text="{gbi}ID.CustomerID" />
    <Text text="{gbi}CompanyName" />
    <Text text="{gbi}SalesOrgID" />
  </ColumnListItem>
</items>
```

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

```
<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>
```

### Customers.controller.js

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

```
//Load some libraries
jQuery.sap.require("sap.m.MessageToast");
jQuery.sap.require("sap.m.MessageBox");

sap.ui.controller("gbi.view.Customers", {

  onInit : function(){
    //Create the model and assign it to the view
    var oModel = new
    sap.ui.model.odata.ODataModel("http://hana.ucc.uwm.edu:8004/GBI_002/gbi/services/gbi.xsodata"
    );

    this.getView().setModel(oModel,'gbi');

    //This model will be used to bind to the edit or create form
    var jModel = new sap.ui.model.json.JSONModel("");
    this.getView().setModel(jModel,'edit');

    //Load the display fragment
    this._showFormFragment("Display");
  },

  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');
  },

  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");
    },

    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");
    },

    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){
                sap.m.MessageToast.show("Update successful");
            },function(data){
                sap.m.MessageToast.show("Update failed");
            });
        }
    }
};
```

```

    } else {
        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");
        });
    }

    this._toggleButtonsAndView("Display");

},

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

handleCreatePress : function(){
    //Create pressed so set editFlag to false
    this.editFlag = false;
    this._toggleButtonsAndView("Edit");
},

_formFragments: {},

_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");
    } if(form === "Create"){
        oView.byId("edit").setVisible(false);
        oView.byId("create").setVisible(false);
        oView.byId("save").setVisible(true);
        oView.byId("cancel").setVisible(true);
        oView.byId("delete").setVisible(false);
        this._showFormFragment("Edit");
    } 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");
    }

    },

    _getFormFragment: function (sFragmentName) {
        //Retrieve the form fragment 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 current content
```

```
oGrid.removeAllContent();
//Insert the new fragment
oGrid.insertContent(this._getFormFragment(sFragmentName));
//If the form is the edit form, set the binding context to the selected item in the table
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 7

This is a complex controller. Review the comments in the code for an explanation. Much of the code is involved in switching between the two fragments and managing the display of the buttons.

### Test the Application

At this point you can test the Edit and Delete functions of the application. Before testing the Create operation, we will have to make some changes to the data model. Run the application and select an item on the table.

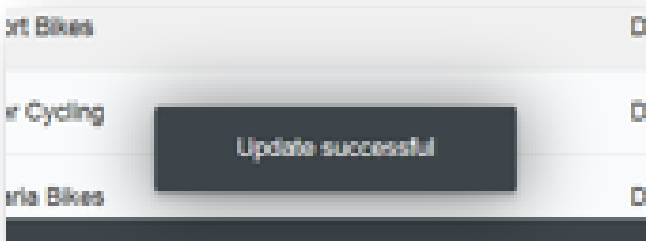
Customer No.	Company	Sales Org.
1000	Rocky Mountain Bikes	UW00
10000	Silicon Valley Bikes	UW00
11000	DC Bikes	UE00
12000	Northwest Bikes	UW00
13000	Airport Bikes	DS00
14000	Alster Cycling	DN00
15000	Bavaria Bikes	DS00

Click the Edit button and the edit form opens.

Customer Information

Company	Address
Customer No.: 21000	Address: Am Stadtwald
Company Name: Ostseerad	City: Anklam
Sales Org: DN00	Region: 13
	Postal Code: 17389
	Country: US

Edit the data and then click the Save button. A toast message will indicate the status of the update.



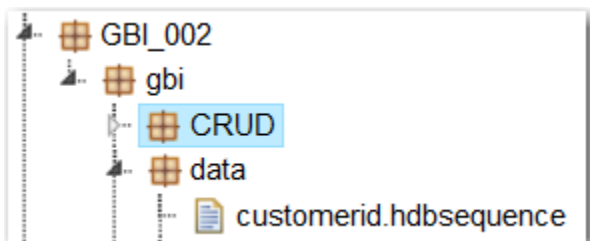
If you wish you can test the Delete operation as well.

### 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 procedure when creating a customer.

### Define the Sequence

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





Enter the code shown below.

```
schema = "GBI_002";  
start_with = 25000;  
increment_by = 1000;  
nomaxvalue = true;  
nominvalue = true;  
cycles = false;
```

*Listing 8*

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 9*

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.

```
View InventoryQuantity AS SELECT FROM INVENTORY
{
    ProductID,
    Plant,
    SLoc,
    CASE StockType
        WHEN 'F' THEN 'Unrestricted Use'
        WHEN 'Q' THEN 'In Quality Inspection'
        WHEN 'X' THEN 'Blocked Stock'
    END AS StockType,
    sum(Quantity) AS Quantity
} GROUP BY ProductID, Plant, SLoc, StockType
ORDER BY ProductID, Plant, SLoc, StockType;

};

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);
    };
};
```

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 10

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.

```

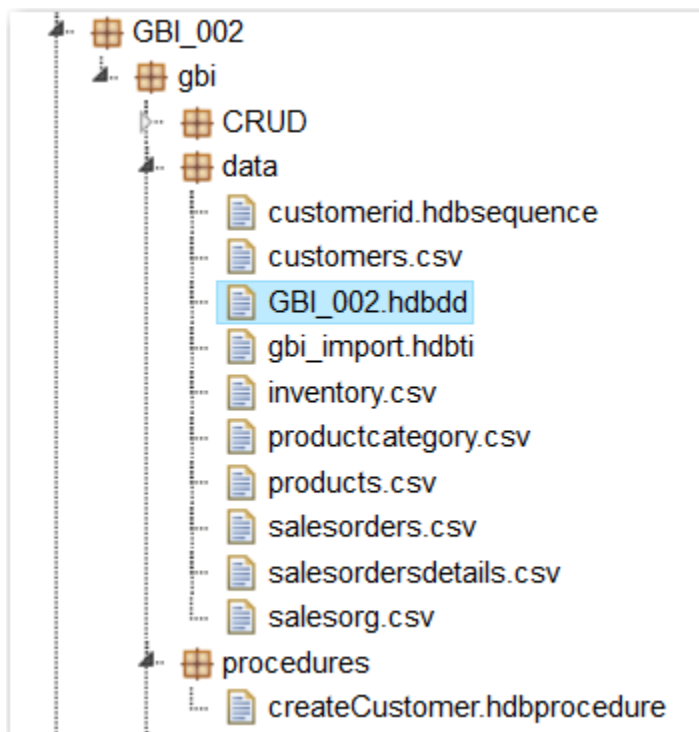
type CurrencyType
{
    Amount : Decimal(17,3);
    Currency : String(3);
};

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

context MASTERDATA {
    @Catalog.tableType: #COLUMN
    entity SALES_ORGS {
        key ID : OrgUnitIDType;
        Description : String(16) not null;
        Address : AddressType;
        Phone : String(14);
        Fax: String(14);
    };
};

```

Now, create a package called **procedures** in the gbi package and create a file called **createCustomer.hdbprocedure** in the procedures package.



Copy the code below into the file.

## PROCEDURE

```
"GBI_002"."GBI_002.gbi.procedures::createCustomer" (  
  IN intab "GBI_002.gbi.data::GBI_002.procedures.custStructure",  
  OUT outtab "GBI_002.gbi.data::GBI_002.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_002.gbi.data::GBI_002.MASTERDATA.CUSTOMERS"  
  values ("GBI_002.gbi.data::customerid".NEXTVAL, lv_companyname, lv_address, lv_city,  
lv_country, lv_region, lv_postalcode, lv_salesorg);  
end if;  
end;
```

Listing 11

Update the highlighted portions to reflect your id.

The first part of the code defines the name of the procedure. The first GBI\_002 is the schema name. The portion GBI\_002.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\_002.gbi.data is the path to the GBI\_###.hdbdd file and GBI\_###.procedures are the nested contexts in the GBI\_###.hdbdd file.

```
PROCEDURE
    "GBI_002"."GBI_002.gbi.procedures::createCustomer" (
        IN intab "GBI_002.gbi.data::GBI_002.procedures.custStructure",
        OUT outtab "GBI_002.gbi.data::GBI_002.procedures.errorsStructure"
    )
LANGUAGE SQLSCRIPT
SQL SECURITY INVOKER AS
--DEFAULT SCHEMA <schema>
--READS SQL DATA AS
```

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.

```
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"
into lv_id, lv_companyname, lv_address, lv_city, lv_country, lv_region,
```

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.

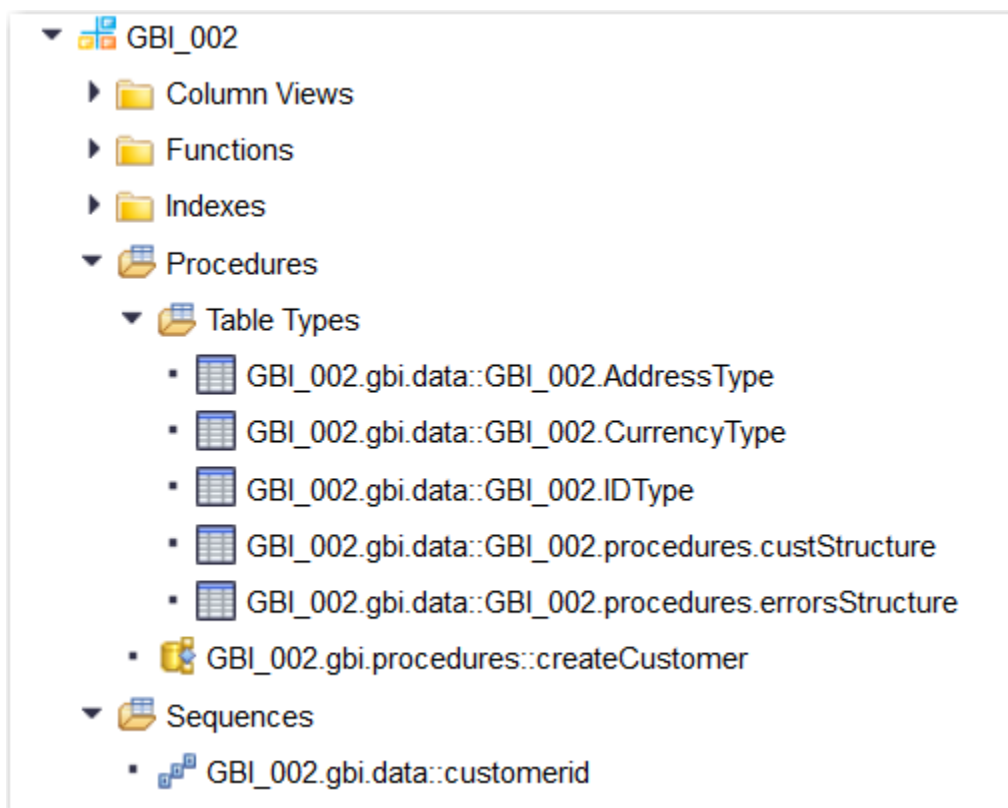
```
if :lv_companyname = '' then
    outtab = select 500 as http_status_code,
        'Company Name cannot be null ' as error_message,
        'No Way! CompanyName field must not be empty' as detail from dummy;
else
```

Finally, the record is inserted into the CUSTOMERS table.

```
insert into "GBI_002.gbi.data::GBI_002.MASTERDATA.CUSTOMERS"  
    values ("GBI_002.gbi.data::customerid".NEXTVAL, 1v_  
end if;
```

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_002.gbi.data::GBI_002.MASTERDATA.CUSTOMERS" as "Customers"  
  navigates ("CustomerOrders" as "Orders")  
  create using "GBI_002.gbi.procedures::createCustomer";
```

Listing 12

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