HD3C05 – Creating a Persistence Model

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Product and Focus**  HANA Platform/CDS | **MOTIVATION**  This case introduces HANA’s [Core Data Services](http://help.sap.com/saphelp_hanaplatform/helpdata/en/b5/23afd66f5a40469573d9c47d7af831/content.htm).  **PREREQUISITES**  None | | **Target Audience**  Undergrduate/Graduate Beginner to Intermediate | | **Author**  Ross Hightower | | https://bgoerke.files.wordpress.com/2013/05/section1.png | | |  |

# Core Data Services

This case uses the core data services (CDS) infrastructure to create a persistence model that will be used in later cases to create services and applications. CDS is a semantically rich layer above SQL. CDS artifacts are design time objects that HANA uses to create the persistence objects in the HANA repository. CDS works by interpreting the descriptions of database objects that you create and then creating the SQL DDL statements to create the objects in the HANA database.

There are advantages to using CDS rather than using SQL directly, the two most important are that it’s easy to transport the CDS artifacts to other systems and allowing HANA to create the SQL allows it to optimize the SQL statements.

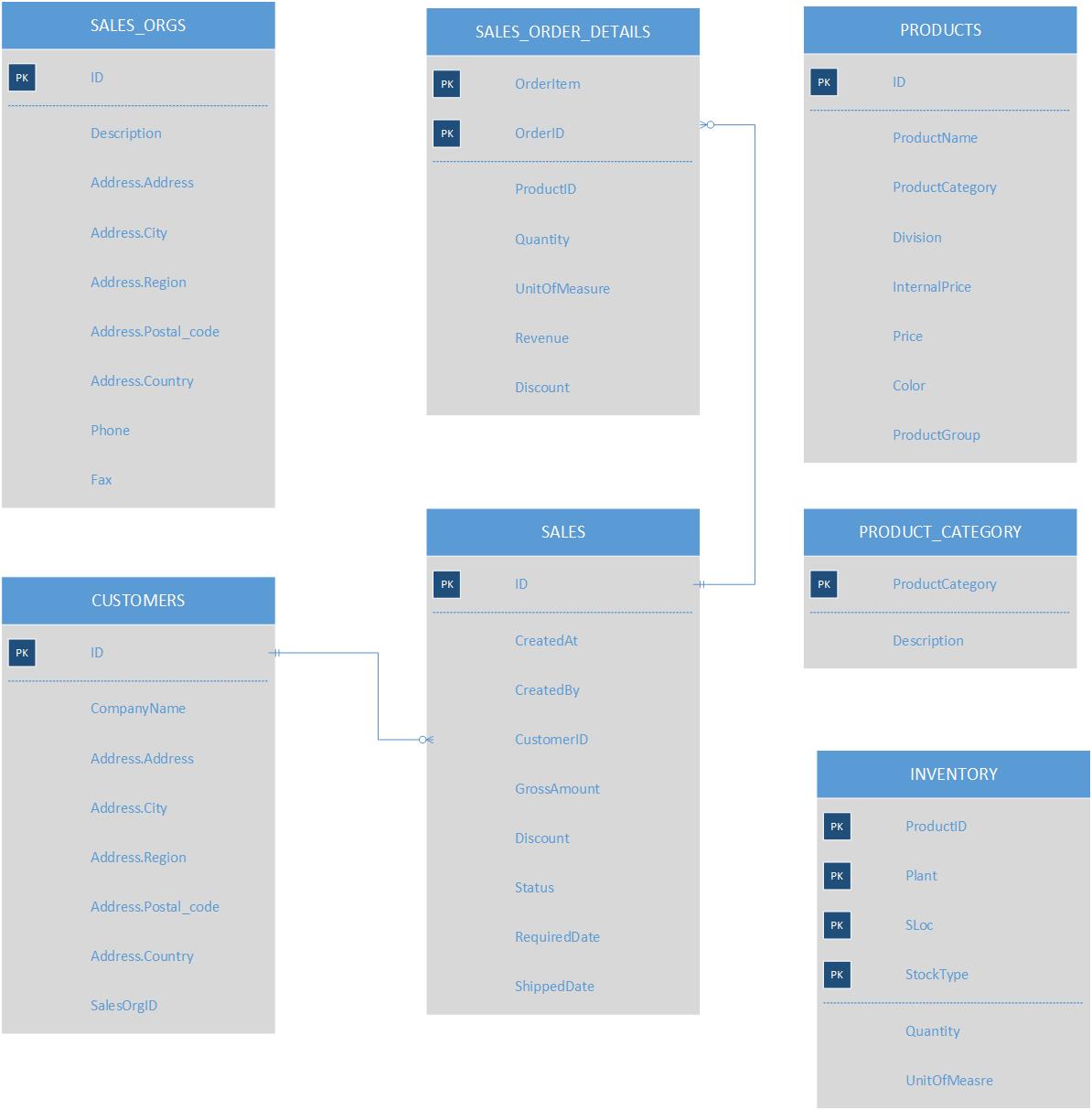
The persistence model created in this exercise will consist of a number of user defined types, database tables and views. This case will describe the creation process of some of those entities but will leave others for you to complete.

Any time there is an abstraction layer between the developer and the objects there are tradeoffs. In this case, the tradeoff is that you must follow the rules when altering existing objects. **The appendix provides a few tips for** [**working**](http://help.sap.com/saphelp_hanaplatform/helpdata/en/b5/7824feaa2d492382c6bfcf6bd7aa9a/content.htm?frameset=/en/10/fadeb42a7a4717982de96eee0e26be/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=124) **with CDS and troubleshooting problems. I recommend that you read the appendix before starting. However, if you insist on jumping right in, one word of advice: never edit the structure of a database object directly or delete an object directly. Always use the CDS files.**

# Create the Persistence Model

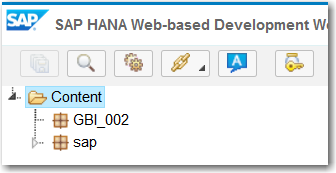
The full database model is shown in the diagram below. Note that there are not as many relationships that we would normally create in a relational database. The reason is that we will use associations created in the oData services rather than in the data model for many most associations we require. Relationships created in the data model cannot be exploited by oData services so there is no motivation to increase the complexity of the data model by creating relationships that are not necessary.

The two relationships we do create in the data model are used to create Views which can be accessed using oData services. Most of the functionality provided by Views can be provided by an oData service. The choice of using database Views or oData services often comes down to how the data will be accessed. If the data is accessed strictly via services then Views may not be necessary. However, if the data is accessed in other ways (i.e. from a SQL Console in the Catalog editor) then Views are necessary. In this case, we will create three views to demonstrate how they are created. The next case will show how oData associations and aggregations can provide the same results.



### Create the Package

Logon to the WDW and open the Editor.



Right-click your package and select **New → Package.** Name the package **gbi.**

Now create a new package inside the gbi package called **data.**



## Create the Overall Structure

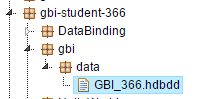
The first thing we’ll do is create the overall structure of the persistence model. CDS database artifacts are described in [files](http://help.sap.com/saphelp_hanaplatform/helpdata/en/c1/ecebbca9c14116992e87e9ae89090f/content.htm?frameset=/en/60/b2d45a8a964efea2e88e29140dbda6/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=96&show_children=false) with an extension of .hdbdd. File extensions are important when working with HANA as they indicate what processes required when the files are activated. Right-click the **data** package and select **New→File.** Name the file **GBI\_###.hdbdd.** Copy the code shown below into the file:

|  |
| --- |
| namespace "gbi-student-366".gbi.data;  @Schema: **'GBI\_366'**  context **GBI\_366** {    context MASTERDATA {    };    context SALES {    };    context LOGISTICS {    };  }; |

Listing

Substitute your values in the highlighted portions.

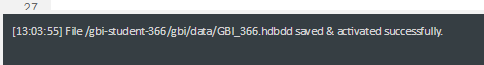
The first line defines the [namespace](http://help.sap.com/saphelp_hanaplatform/helpdata/en/70/31e3f7fa9c4830960907dc795d6360/content.htm?frameset=/en/9f/456278e9754ecfb26df3be10bdc2b7/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=100) of the model and is equal to the path of the packages to the .hdbdd file. Substitute your path for the one in the sample code. All names in CDS are case sensitive.



The @ on the next line indicates an [annotation](http://help.sap.com/saphelp_hanaplatform/helpdata/en/82/17aac86d9748d8b034797ecc8065b6/content.htm) which provides the compiler information about how to process the file. The @SCHEMA annotation is required and indicates in which schema to create the artifacts. Your schema was created for you when your user id was created and has the same name as your user id. Substitute your schema name for the one in the sample code.

[Contexts](http://help.sap.com/saphelp_hanaplatform/helpdata/en/95/2a2d4bba95447bbc5b7ce1be6f4f4d/content.htm?frameset=/en/70/31e3f7fa9c4830960907dc795d6360/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=101) are ways to organize the database artifacts. Contexts, as in our case, can be nested which allows the inner contexts to access artifacts created in their containing contexts. Even though our database is small, we’ll use contexts to illustrate the concept. The name of the outer context must be the same as the name of the .hdbdd file.

When you save the file, the file is saved and then the system will attempt to activate it. The activation process creates the objects defined in the file in the database. You can save the file at this point although there are no objects defined. You can see the result of saving the file in the console pane below the editor.



|  |
| --- |
| The small dot to the left of the file name as shown below then the file indicates the file has not been activated. This can happen if there was an error in the file when you saved it.      Sometimes the error that prevented activation can be corrected without editing the file so the Save option is not available. If you need to activate a file that has already been saved, right-click the file name and select Activate from the context menu. |

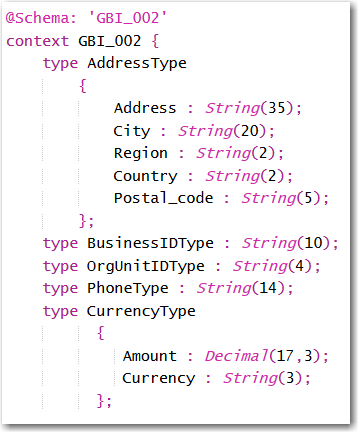
## Create User Defined Types

The first artifacts we will create are some user defined types. [User defined types](http://help.sap.com/saphelp_hanaplatform/helpdata/en/a4/7e85efd6214d2f957193e79883ca84/content.htm?frameset=/en/78/746b112d794f459978eb55b20a8a6f/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=113&show_children=false) can be used to enforce consistency across a database. For example, we will define a type for IDs. This will ensure that the IDs in all tables have the same data type. Not really useful in such a small database but with a large database, especially, one with multiple developers, this can prevent a lot of problems. We can also create more complex types such our AddressType which will ensure that a consistent set of data elements are used wherever the type is used.

Enter the code into the .hdbdd file immediately below the line context “GBI\_001 {“. By placing these in the outer context, they are available to the three inner contexts.

|  |
| --- |
| type AddressType  {  Address : String(35);  City : String(20);  Region : String(2);  Country : String(2);  Postal\_code : String(5);  };  type BusinessIDType : String(10);  type OrgUnitIDType : String(4);  type PhoneType : String(14);  type ValueType : Decimal(17,3);  type CurrencyType  {  Amount : Decimal(17,3);  Currency : String(3);  }; |

Listing



## Create the Tables

This section describes the creation of three tables: SALES\_ORG, SALES\_ORDERS\_DETAILS and SALES\_ORDERS. At the end of the section is an exercise in which you will create additional tables.

### SALES\_ORG

Insert the code shown below into the file **inside the MASTERDATA context**.

|  |
| --- |
| @Catalog.tableType: #COLUMN  entity SALES\_ORGS {  key ID : OrgUnitIDType;  Description : String(16) not null;  Address : AddressType;  Phone : String(14);  Fax: String(14);  }; |

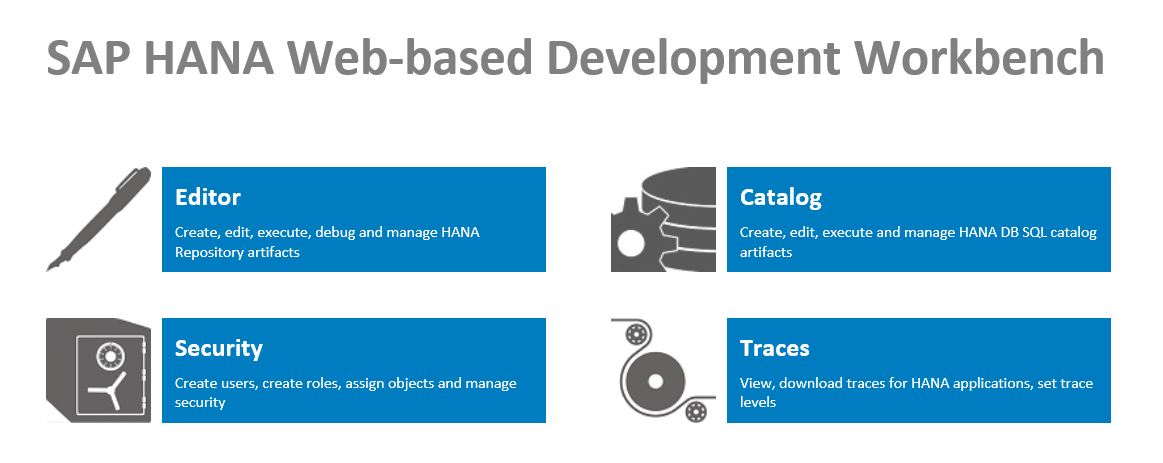
Listing

The @Catalog annotation is used to provide information about the database object you are creating. In this case, the @Catalog.tableType indicates the type of table. The options are #COLUMN, #ROW and #GLOBAL\_TEMPORARY. The last is a table created temporarily that is not saved in the database.

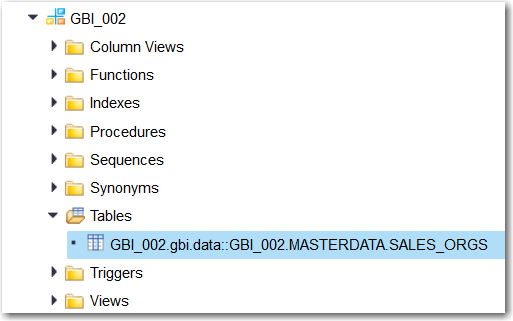
Tables are created using the [entity](http://help.sap.com/saphelp_hanaplatform/helpdata/en/e8/c150fde4614804831c63a67224ffa8/content.htm?frameset=/en/c1/ecebbca9c14116992e87e9ae89090f/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=106&show_children=false) directive. Notice the use of the user defined type for the address. This will cause the creation of multiple fields with names like Address.postal\_code (because postal\_code is defined in the user defined type).

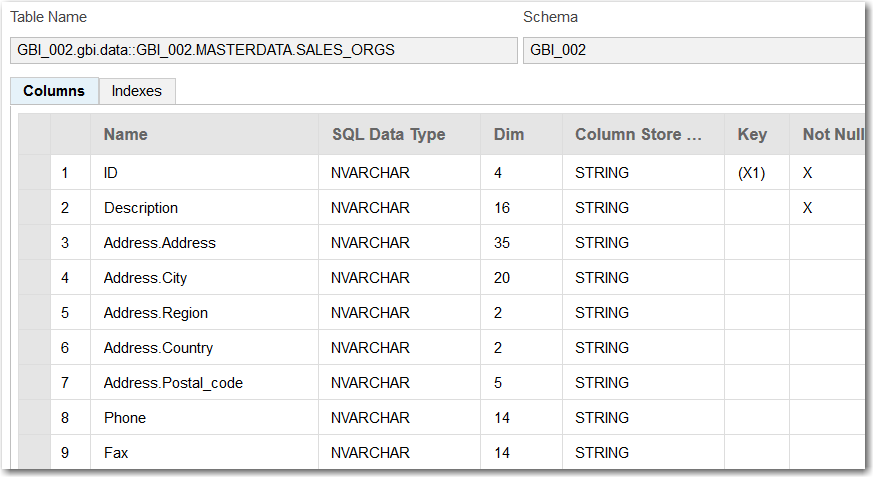
There are a number of [qualifiers](http://help.sap.com/saphelp_hanaplatform/helpdata/en/78/746b112d794f459978eb55b20a8a6f/content.htm?frameset=/en/e8/c150fde4614804831c63a67224ffa8/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=108) you can use to define the table fields. For example, the key qualifier in the SALES\_ORGS table creates the primary key. Also the “not null” qualifier indicates a field cannot contain null values.

When you save the file, the table will be created. To see it, open the Catalog editor.



Drill into the Tables folder to find the table.





The name of the table is the namespace plus the context followed by the table name. Also, notice the names of the fields created for the Address. Finally, notice that the String data type in the CDS file has been created as NVARCHARs.

### SALES\_ORDERS\_DETAILS

Next, we’ll create the SALES\_ORDERS\_DETAILS table **in the SALES context**. Copy the code below into the **SALES** context.

|  |
| --- |
| @Catalog.tableType: #COLUMN  entity SALES\_ORDER\_DETAILS {  key OrderID : BusinessIDType;  key OrderItem : String(3);  ProductID : BusinessIDType;  Quantity : Integer;  UnitOfMeasure : String(3);  Revenue : ValueType;  Discount : ValueType;  }; |

Listing

The only new aspects of this table definition is the use of the integer type. Note also the use of the user defined types and that the primary key consists of two fields.

### SALES\_ORDERS

Finally, we’ll create the SALES\_ORDERS table **in the SALES context**. Copy the code shown below into the SALES context.

|  |
| --- |
| @Catalog.tableType: #COLUMN  entity SALES\_ORDERS {  CreatedAt : LocalDate;  CreatedBy : String(20);  CustomerID : String(10) not null;  key ID : Association[1..\*] TO SALES\_ORDER\_DETAILS { OrderID };  GrossAmount : CurrencyType;  Discount : ValueType;  Status : String(15) default 'New';  requiredDate : LocalDate;  shipDate : LocalDate;  }; |

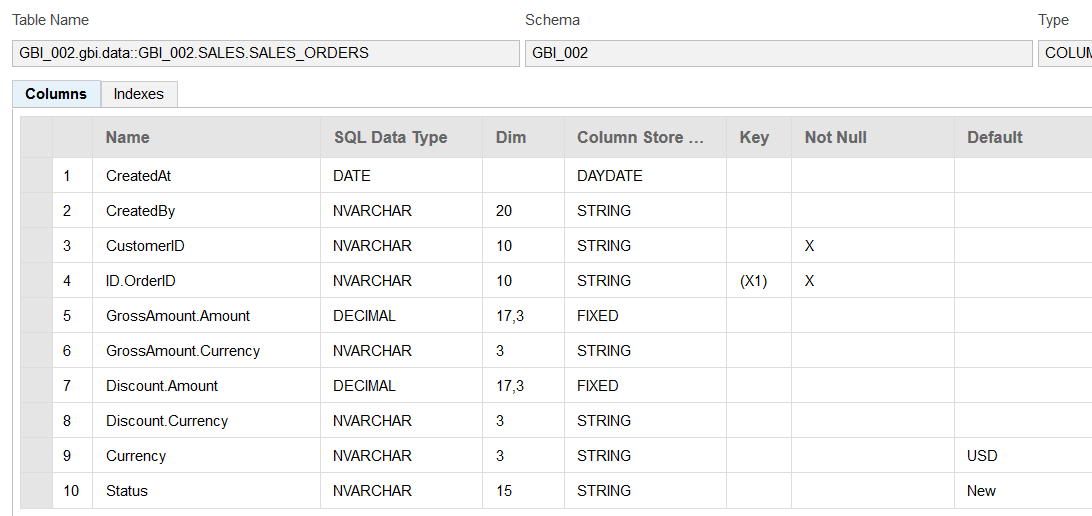
Listing

There are a number of new elements in this definition. The first is the creation of an [index](http://help.sap.com/saphelp_hanaplatform/helpdata/en/ad/036c56b5e545ae8b31ece0ab95379f/content.htm?frameset=/en/78/746b112d794f459978eb55b20a8a6f/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=109). The index definition is included as part of the metadata defined with the @Catalog annotation. The properties of the object that defined the index are the name (name of the index), the order (#DESC or #ASC), unique (whether the values in the indexed field are unique) and elementNames (one or more fields to include in the index. You can create multiple indexes by separating the definitions with a comma.

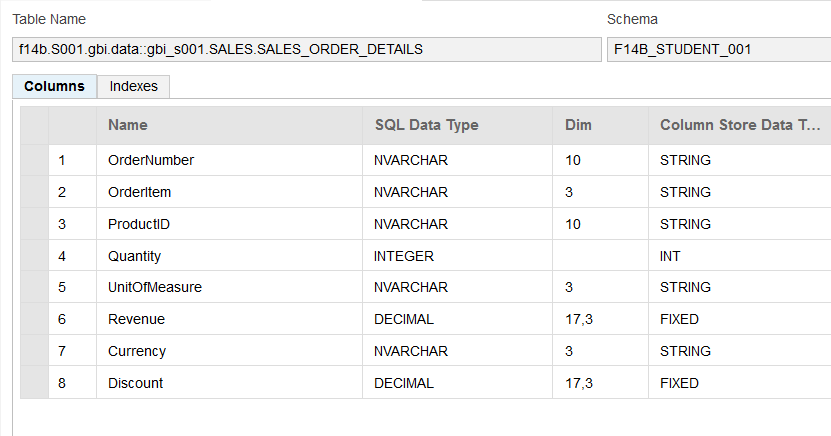
Another new element is the use of a default value for the Status field.

The last new element is the [association](http://help.sap.com/saphelp_hanaplatform/helpdata/en/10/fadeb42a7a4717982de96eee0e26be/content.htm?frameset=/en/95/2a2d4bba95447bbc5b7ce1be6f4f4d/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=118&show_children=false) created with the SALES\_ORDER\_DETAILS table. The qualifier creates a one-to-many [1..\*] association with SALES\_ORDER\_DETAILS on that table’s OrderID field.

When you save the file, the tables are created.



Notice the name of the field involved with the association. It consists of the name of the field in the SALES definition and the name of the field on the other end of the association in the SALES\_ORDERS\_DETAILS table.

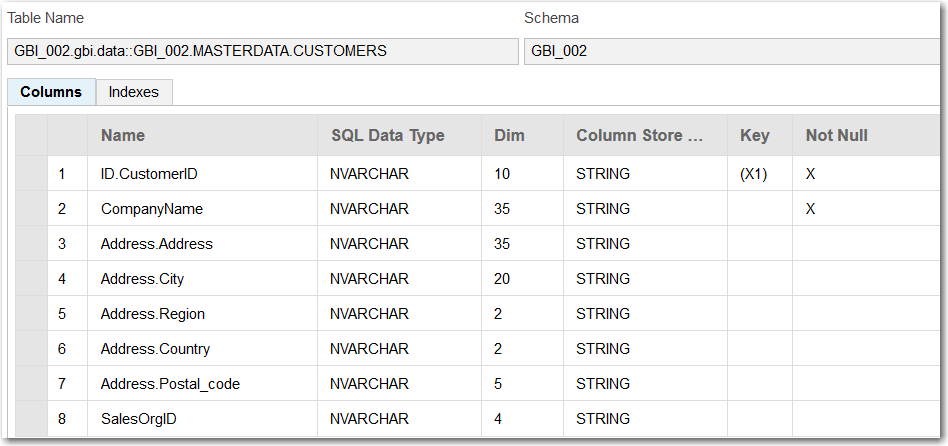


### Table Creation Exercise

Now create three more tables: **CUSTOMERS, PRODUCTS, PRODUCT\_CATEGORIES** and **INVENTORY**. The structures and metadata are describe below.

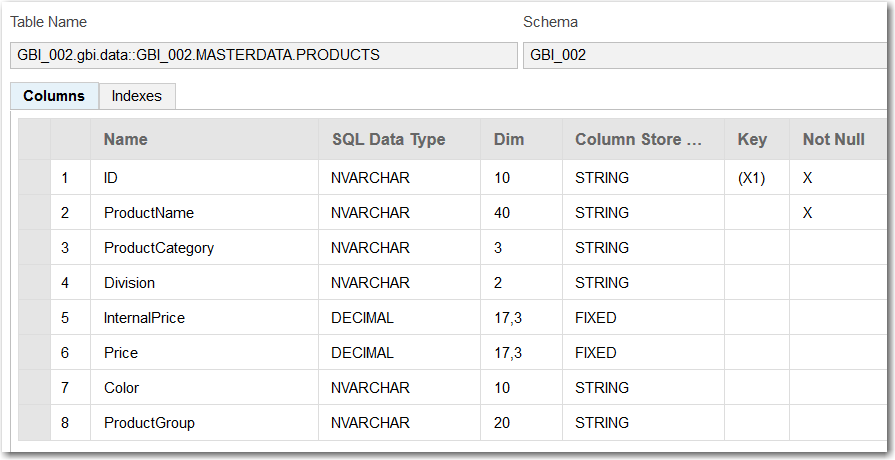
|  |
| --- |
| Note the names of the tables in the images to determine in which context they are created. |

#### CUSTOMERS

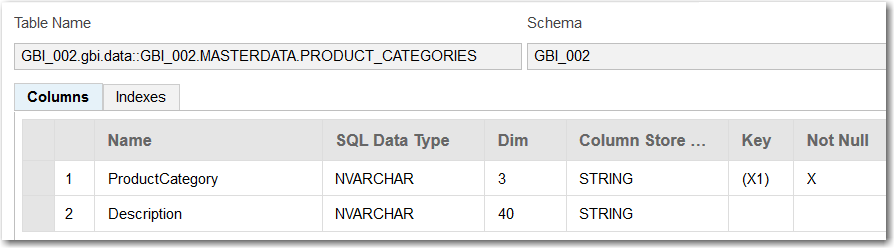


The CUSTOMERS table has a one-to-many association between the ID field and the CustomerID field in SALES\_ORDERS. Note that because the two entities are different contexts, you must refer to the SALES\_ORDERS table using the context name: SALES.SALES\_ORDERS.

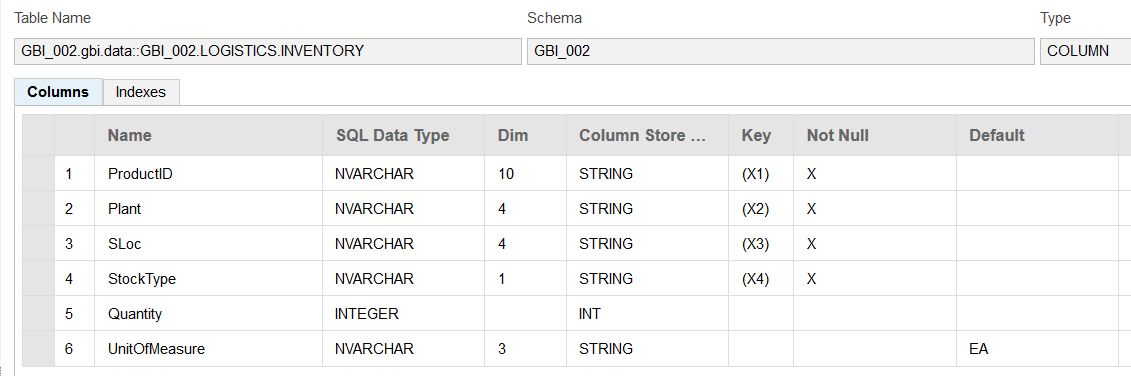
#### PRODUCTS



#### PRODUCT\_CATEGORY



#### INVENTORY



## Import Data into the Tables

CDS includes the capability to [import](http://help.sap.com/saphelp_hanaplatform/helpdata/en/42/a68d88ca894e539b267d1015d8f7d0/content.htm?frameset=/en/ad/036c56b5e545ae8b31ece0ab95379f/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=126&show_children=false) data into tables from csv files. The configuration of table imports is done in a file with an .hdbti extension. The data is included in this document but can also be found in files that are included with the curriculum.

|  |
| --- |
| Once you import data into a table, making changes to the table in the .hdbdd file becomes a bit more complex. Before making a change to the table, it’s best to delete any references to the table in the .hdbdd file (such as associations or views) and the .hdbti file first. |

As in the previous section, this document will describe importing data into one table and then leave the remaining tables for you to do.

### SALES\_ORG

Create a file called **salesorg.csv** in the **data** package. Copy the data shown below into the file and save it.

|  |
| --- |
| If you have local copies of the files you can copy them from your local computer into the data package. To do this, select the data package so that the upload multi-file drop zone for the package is displayed then drag the file into the drop zone. |

|  |
| --- |
| DN00,Deutschland Nord,17 Grosser Grasbrook,Hamburg,02,DE,20457,040-555-3200,040-555-3201  DS00,Deutschland Süd,16 Dietmar-Hopp-Allee,Heidelberg,08,DE,69118,06221-555-2200,06221-555-2201  UE00,USA East,5301 Blue Lagoon Drive,Miami,FL,US,33126,1-305-555-5000,1-305-555-5001  UW00,USA West,150 Spear Street,San Diego,CA,UW,94105,1-415-555-790,1-415-555-7901 |

Listing

The values must align with the fields defined in the .hdbdd file. If you’ve followed the directions carefully, they will align.

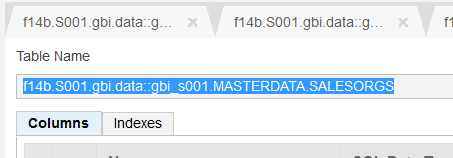
Create a file called **gbi\_import.hdbti** in the **data** package. Insert the code shown below and update it to reflect the name of your table, schema and the path to your salesorg.csv file.

|  |
| --- |
| import = [  {  table = "**gbi-student-366.gbi.data::GBI\_002**.MASTERDATA.SALES\_ORGS";  schema = "**GBI\_002**";  file = "**gbi-student-366.gbi.data**:salesorg.csv";  header = false;  }  ]; |

Listing

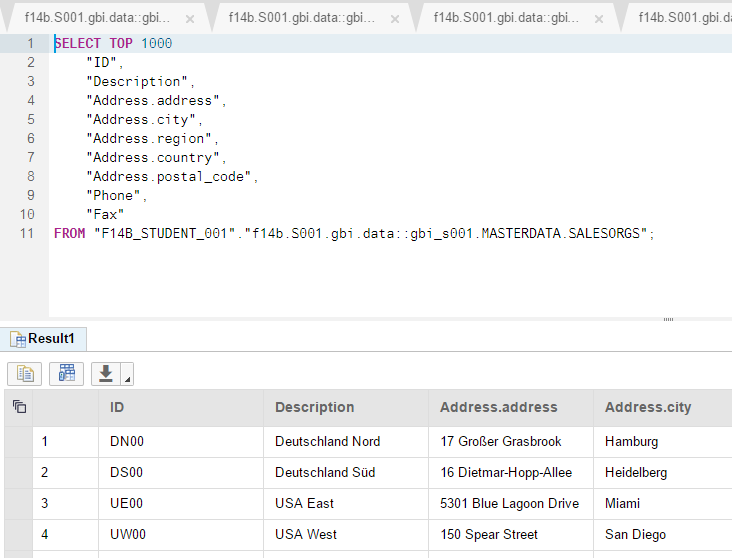
Substitute your values in the highlighted portions.

You can find the table name by opening the table in the Catalog editor.



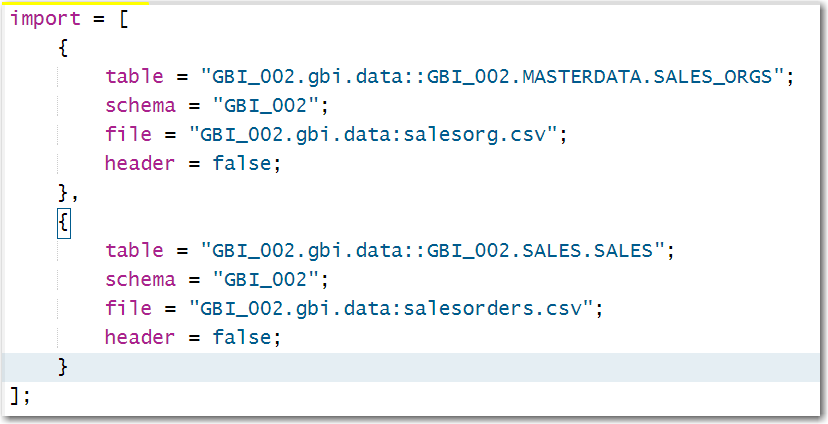
The csv file name includes the path to the file (equivalent to the namespace in the .hdbdd file) a colon and then the name of the csv file. Now when you save this file, it will be activated and the import will take place. Check the console pane below the editor to check for errors.

You can go to the Catalog editor now to see the data. Right-click the table and select **Open Content** orclick the fileto open its structureand then click the Open Content button.



### Import Data Exercise

Now import the data into the remaining tables. The data is included in separate files. You can add multiple import definitions in the file by separating them with commas as shown below.



## Create Some Views

A [view](http://help.sap.com/saphelp_hanaplatform/helpdata/en/22/47aca08f054d30b0f749f757ec1ccd/content.htm?frameset=/en/42/a68d88ca894e539b267d1015d8f7d0/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=121&show_children=false) is a virtual table that is created dynamically when opened.

### CustomerSales

Add the following view to the MASTERDATA context.

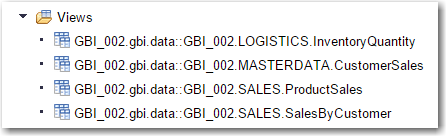
|  |
| --- |
| VIEW CustomerSales AS SELECT FROM CUSTOMERS  {  ID.Status,  CompanyName,  sum(ID.GrossAmount.Amount) AS GrossAmount  } GROUP BY ID.Status, CompanyName HAVING ID.Status = 'New'; |

Listing

This view makes use of the association between CUSTOMERS and SALES\_ORDERS. Note the fields from SALES\_ORDERS are prefixed with ID (the field on the CUSTOMERS side involved in the association). The view shows the GrossAmount of all orders with a Status of New.

Because you are including ID.Status and CompanyName in the GROUP By clause, these two fields must be the first two fields included in the view.

You can find the view in the Views folder of your schema in the Catalog editor.



### InventoryQty.

Add the following code to the LOGISTICS context.

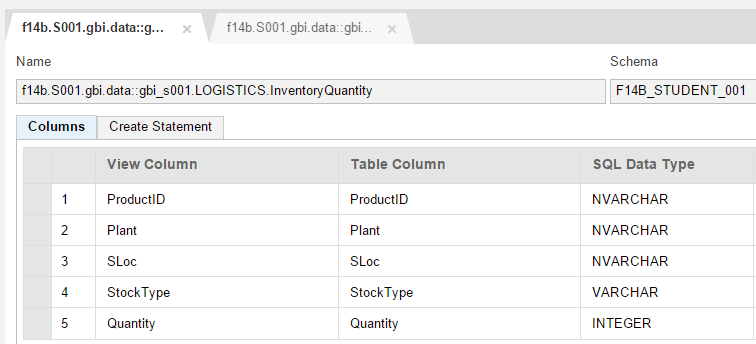
|  |
| --- |
| View InventoryQuantity AS SELECT FROM INVENTORY  {  ProductID,  CASE StockType  WHEN 'F' THEN 'Unrestricted Use'  WHEN 'Q' THEN 'In Quality Inspection'  WHEN 'X' THEN 'Blocked Stock'  END AS StockType,  sum(Quantity) AS Quantity  } GROUP BY ProductID, StockType  ORDER BY ProductID, StockType;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

Listing

The SELECT statement indicates the data for this view will come from the INVENTORY table. Inside the brackets, the fields included in the view are listed. The CASE statement will replace the letter stored in the StockType field with the corresponding strings.

The sum function will sum the Quantity field and name the result Quantity in the view. Other aggregation functions are available including count, min, max, avg, stddev and var. The aggregation is computed for the groups created by the fields in the GROUP BY clause. In this case, the view will show the sum of quantity for each unique combination of ProductID and StockType.

The ORDER BY statement indicates the sort order of the view.



### View CreationExercise

Create one more views

#### ProductSales

This view shows the sums of Quantity, Revenue and Discount grouped by ProductID and ordered by sum of Revenue.

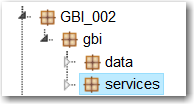
# Create OData Services

OData has been called ODBC of the Web because it is becoming a standard, cross-platform way to access and update data. It relies on web technologies such as RESTful services, Uniform Resource Identifiers (URI) for resource location, HTTP verbs (GET, PUT, POST and DELETE) for operations and JavaScript Object Notation (JSON) for data representation. OData is quickly becoming a widely used protocol because it is fast, efficient and flexible. Not only is the data easy to consume using oData, creating oData services on HANA is also very simple. A single line of code provides the ability to read, create, update and delete table. You can learn more about oData [here](http://odata.org) and [here](http://help.sap.com/saphelp_hanaplatform/helpdata/en/b8/0f8b626b3d44f882e8f2c3ff45952d/content.htm?frameset=/en/8d/78eb6f6e8f4307be939a6bc684b9ca/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=244&show_children=false).

## Create the Services

### Create the Services Package

Right-click the gbi package and create a new package called **services**.

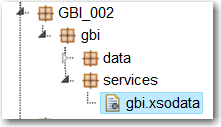


### Services for Tables

You can create services for tables as well as views. We will create the services for tables first.

#### Create oData Services

Right-click the services package and create a new file called **gbi.xsodata**.

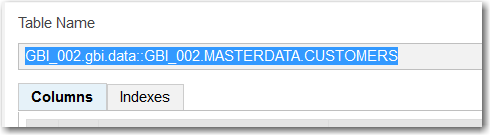


The first service we’ll create will be for the CUSTOMERS table. Enter the following code.

|  |
| --- |
| service {  "**gbi-student-366**.gbi.data::**GBI\_366**.MASTERDATA.CUSTOMERS" as "Customers";  } |

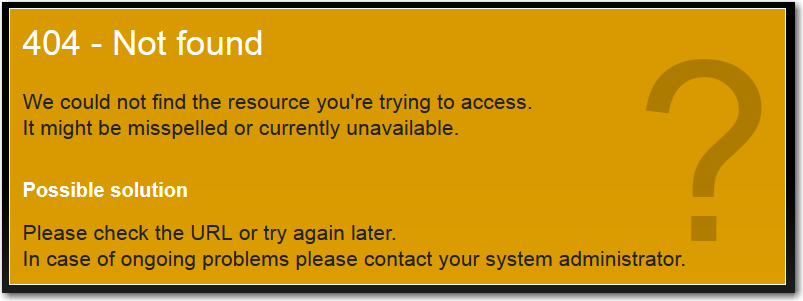
Listing

The part before the ‘as’ is the name of the Customers table you created above. You can find the correct name of the table by opening the table in the Catalog editor. If you open the table, you can copy the name:



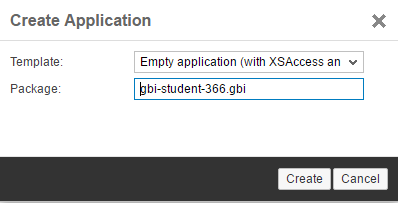
Change the path name to fit your situation. The Customers at the end will be name of the *endpoint* used to invoke the service.

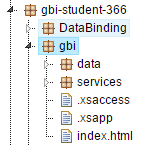
Click the run icon () to execute the service. You should receive a 404 error. The reason is that we have not made the package externally accessible.



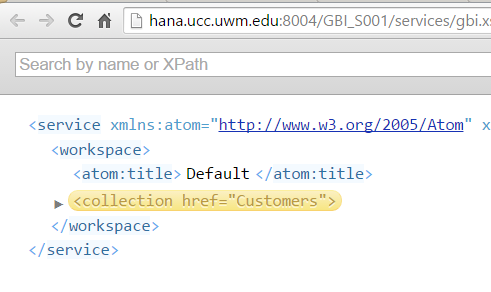
Right-click the gbi package and select **Create Application**. Choose the Empty application.

This creates three files.

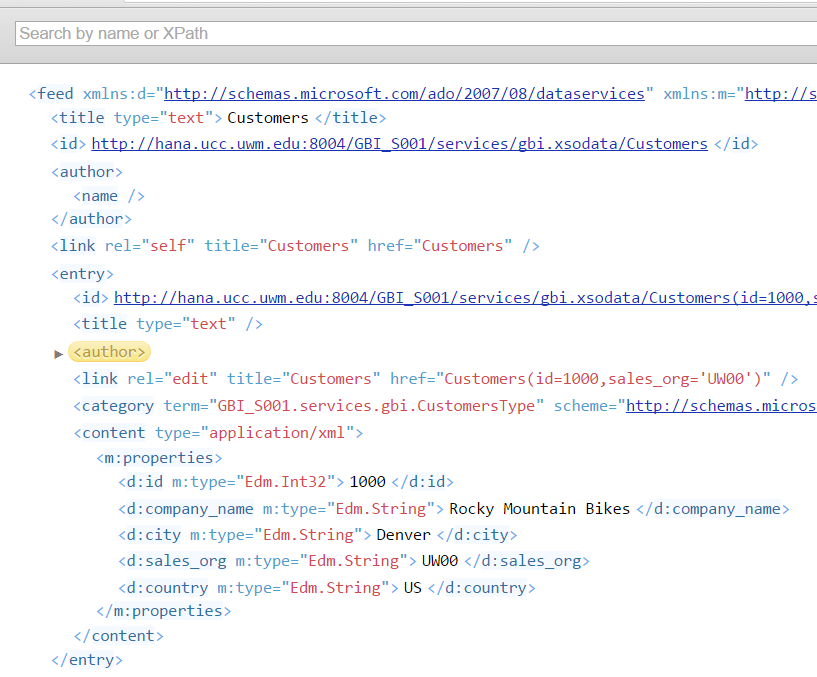




Now run the service again (Chrome was used for all the screenshots in this document. Other browsers will not show the data without installing extensions).



The service document shows one service, the Customers service we created. Add **/Customers** to the end of the URL and hit enter and the customer data will be retrieved.



One of the advantages of using RESTful services is that data access is achieved entirely through the URL. There are a number of parameters available to achieve your results. Try the examples in the following table.

|  |  |
| --- | --- |
| URL | Result |
| /Customers('1000') | Retrieves the customer with the matching primary key |
| /Customers?$format=json | Customers data in JSON format |
| /Customers?$orderby=CompanyName | Customers data sorted by the property CARRNAME |
| /Customers?$filter=CompanyName eq 'Rocky Mountain Bikes' | Customers with CompanyName property equal to ‘Rocky Mountain Bikes’ |
| /Customers?$select=CompanyName,SalesOrgID | Just the CompanyName and SalesOrgID properties of Customers |

You can also combine parameters. For example:

/Customers?$select=CompanyName,SalesOrgID&$filter=CompanyName eq 'Rocky Mountain Bikes'

To see the metadata of the service add the **/$metadata** option to the end of the service document URL.



The metadata document describes the various elements and services available in the oData model.

To see all the options refer to the documentation at <http://www.odata.org/>.

#### Add the Sales Org Service

Return to the gbi.xsodata file and modify the code in gbi.xsodata to look like the following:

|  |
| --- |
| service {  "**gbi-student-366**.gbi.data::**GBI\_366**.MASTERDATA.CUSTOMERS" as "Customers";  "**gbi-student-366**.gbi.data::**GBI\_366**.MASTERDATA.SALES\_ORGS" as "SalesOrg";  } |

Listing

Remember to update the highlighted code to reflect your table names. This adds a service for the SALES\_ORGS table accessible as /SalesOrg.

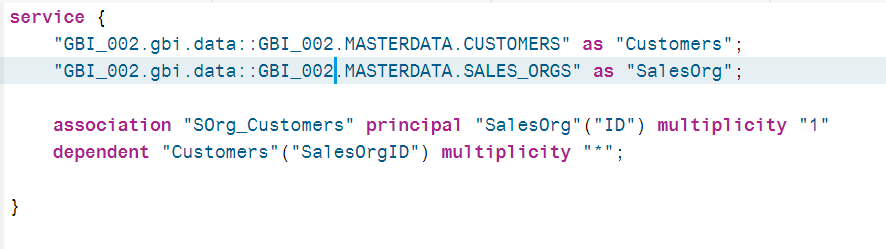


One thing we need to be able to do is to associate data from different services. For example, there is a one-to-many relationship between SalesOrg and Customers (a sales organization has multiple customers). So we want to be able to retrieve a sales organization and then retrieve all the customers that are assigned to that sales organization. We could create a view using CDS like we did above or we can create the association in the OData services.

Add the code below to gbi.xsodata.

|  |
| --- |
| association "SOrg\_Customers\_Association" principal "SalesOrg"("ID") multiplicity "1"  dependent "Customers"("SalesOrgID") multiplicity "\*"; |

Listing



This code creates an association between the SalesOrg and Customers services. Let’s break it down.

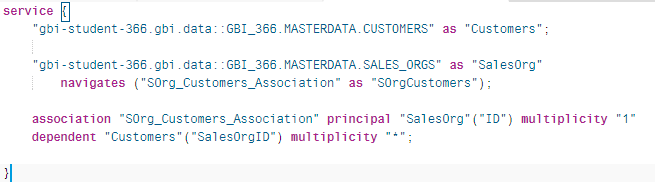
* The SOrg\_Customers\_Association is the name of the association and we could use anything for this.
* The term principle indicates the parent side of the association (for example the one side in a one-to-many relationship). Since, in our case SalesOrg (SalesOrg comes from the endpoint name assigned to the sales organization service) is the parent, the SalesOrg service comes next. The field that associates SalesOrg to Customers is ID so it is included in the parentheses.
* The term multiplicity “1” refers to fact that this is a one-to-many relationship and SalesOrg is the one side.
* The term dependent is the child side of the relationship and in our case it is the Customers service. The field in the Customers table that associated the customer with a sales organization is SalesOrgID so it is included in the parentheses after the service endpoint name.
* Finally, the Customer side of the association is a many and this is indicated by an \* after the multiplicity term.

There are a number of pieces to this but it’s logical.

Next we have to indicate that the SalesOrg service can navigate this association. Change the SalesOrg service as shown below. Insert the code that begins ‘navigates…’ before the semicolon at the end of the service line.

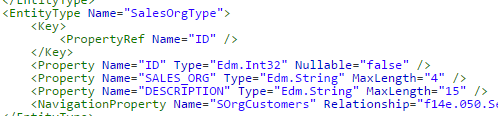
|  |
| --- |
| "**gbi-student-366**.gbi.data::**GBI\_366**.MASTERDATA.SALES\_ORGS" as "SalesOrg"  navigates ("SOrg\_Customers\_Association" as "SOrgCustomers"); |

Listing

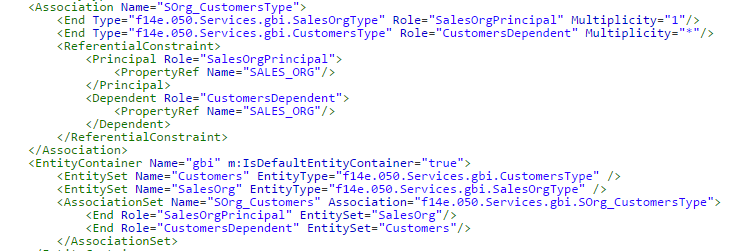


The term “SOrg\_Customers\_Association” in the navigates statement links the service to the association we created above.

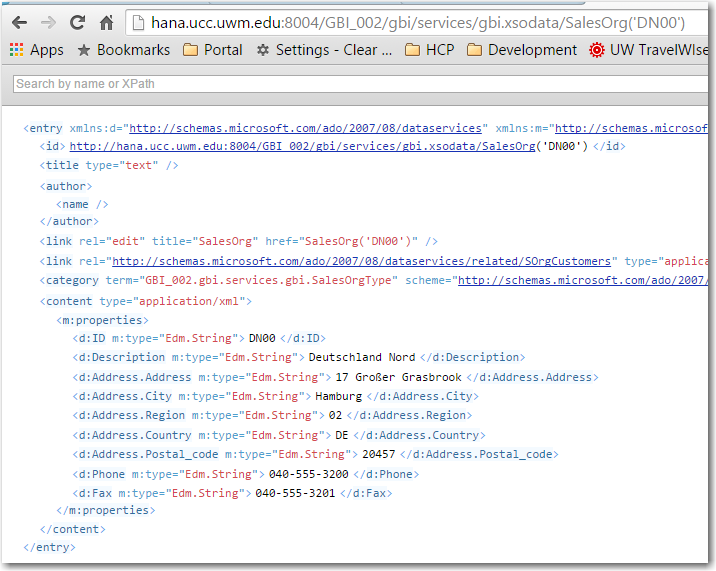
If you view the service metadata (add /$metadata to the end of the service URL) you can see the association metadata. The NavigationProperty is the parameter used to navigate the association. Note this value is the term after the “as” in navigates (“SOrg\_Customers\_Association” as “SOrgCustomers”).



A description of the association is also provided.



To use the association, first retrieve the sales organizations by adding **/SalesOrg** to the end of the service document URL. Then retrieve the DN00 sales organization by using **/SalesOrg('DN00')**. This construction works with the primary key of the table.



To retrieve the customers associated with sales organization DN00 add **/SalesOrg('DN00')/SOrgCustomers** to the service document URL.



## OData Exercise

1. Add services for the PRODUCTS, PRODUCT\_CATEGORIES, SALES\_ORDERS, SALES\_ORDER\_DETAILS and INVENTORY tables. Include the following associations:
   1. PRODUCTS and INVENTORY with NavigationProperty Inventory
   2. PRODUCT\_CATEGORIES and PRODUCTS with NavigationProperty Products
   3. SALES ORDERS and SALES\_ORDER\_DETAILS with NavigationProperty Details
   4. CUSTOMERS and SALES\_ORDERS with NavigationProperty Orders

## Services for Views

Database views do not have keys and oData services must have keys so you either must designate one of the columns as a key or you must have HANA generate a temporary key for the service.

### Add a Service for the CustomerSales View

The definition of the ProductSales view is shown below:

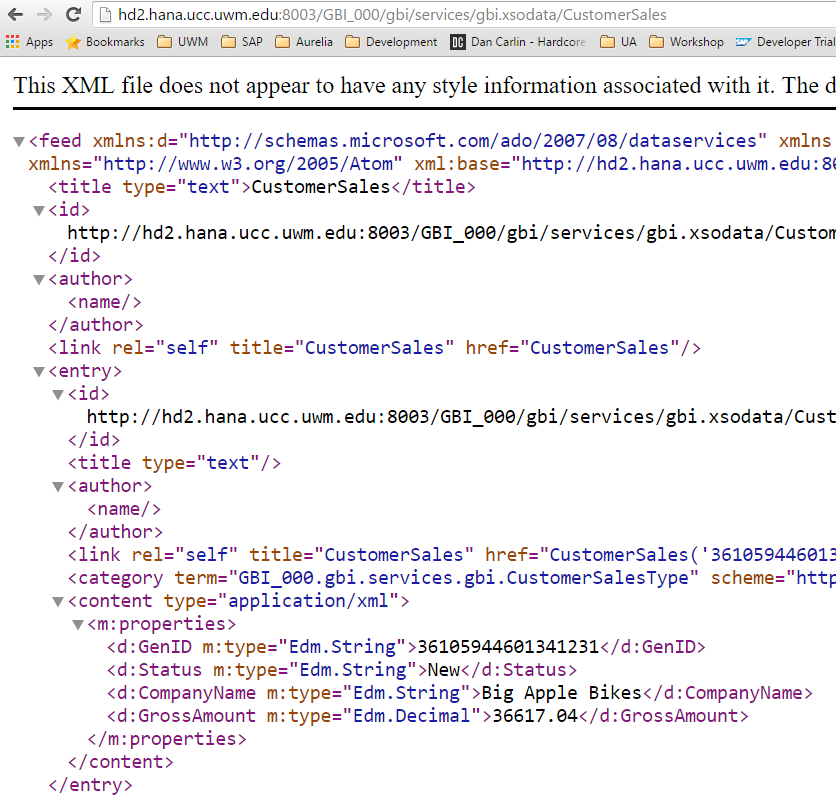
|  |
| --- |
| VIEW CustomerSales AS SELECT FROM CUSTOMERS  {  ID.Status,  CompanyName,  sum(ID.GrossAmount.Amount) AS GrossAmount  } GROUP BY ID.Status, CompanyName HAVING ID.Status = 'New'; |

Listing

There isn’t a field that can service as primary key for this view and all OData services must have a primary key so the service will have to generate one. Add the code below to gbi.xsodata.

|  |
| --- |
| "GBI\_002.gbi.data::GBI\_002.MASTERDATA.CustomerSales" as "CustomerSales"  key generate local "GenID"; |

Listing



## View Service Creation Exercise

1. Create services for the ProductSales and InventoryQuantity views.

## Create an Aggregation Service

You can also aggregate on fields with oData services even if you don’t have a view. The service below aggregates on the GrossAmount.Amount field of the SALES\_ORDERS table.

|  |
| --- |
| "**GBI\_002.gbi.data::GBI\_002**.SALES.SALES\_ORDERS" as "SalesRevenue"  key generate local "GenID"  aggregates always (SUM of "GrossAmount.Amount"); |

Listing

To extract the aggregated revenue for CustomerID add $select=CustomerID,GrossAmount.Amount to the URL.



## Aggregation Service Exercise

1. Create an aggregation service for the INVENTORY table that aggregate quantity.

# Appendix 1

## CUD Operations

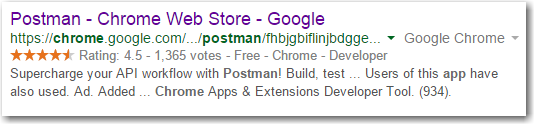
You can perform create, update and delete operations on the tables unless you specifically restrict the operations performed on the tables by adding code like the following:

|  |
| --- |
| "GBI\_002.gbi.data::GBI\_002.MASTERDATA.CUSTOMERS" as "Customers"  create forbidden  update forbidden  delete forbidden; |

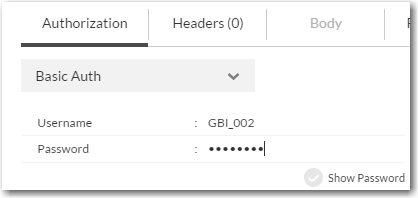
Don’t add this code because we want to be able to modify the data in the tables.

Testing in the remainder of this document is done using the [Postman](https://chrome.google.com/webstore/search/postman?hl=en) Google App. You can use any RESTful client to test the services but a RESTful client is required because we are going to perform POSTs, PUTs and DELETEs which cannot be submitted from a browser’s address field.

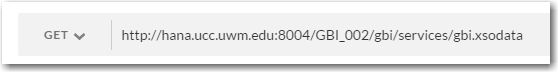
Google “postman chrome app” and open a link that looks like this:



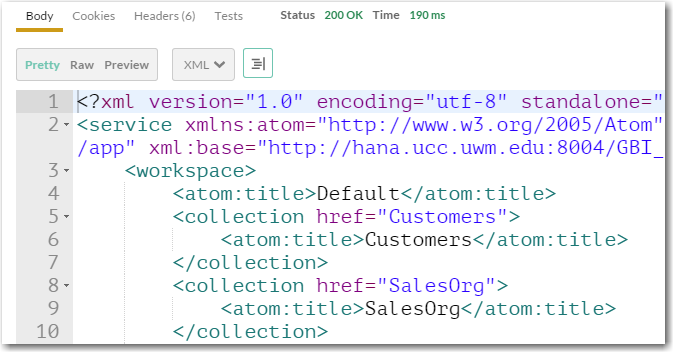
Install the app and follow the directions to open it. Click Authorization and change the authorization type to Basic Auth. Enter your HANA username and password.



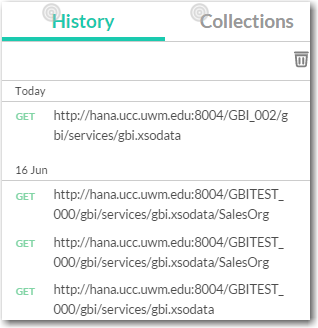
Enter you service document URL and make sure the method is set to GET.



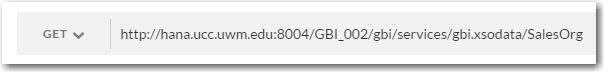
Click Send. The service document is retrieved and the status of 200 OK is returned



Postman keeps a history of the requests you perform on the left side of the screen so you can recall them.

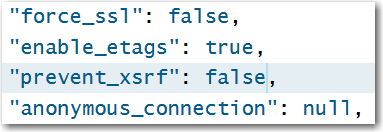


Now add /SalesOrg to the URL and click send to retrieve the contents of the SALES\_ORGS table.



### Update the .xsaccess File

Before we can perform creates, updates and deletes we have to update the .xsaccess file. Open the file and change the value of the **prevent\_xsrf** property to **false**. The xsrf token is a security mechanism to prevent cross-site forgery attacks.

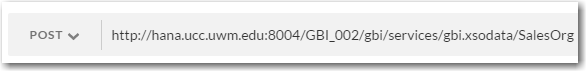


Of course, this leaves your application open to cross-site forgery attacks so, **once you have finished this exercise**, you should set this back to true.

### Create

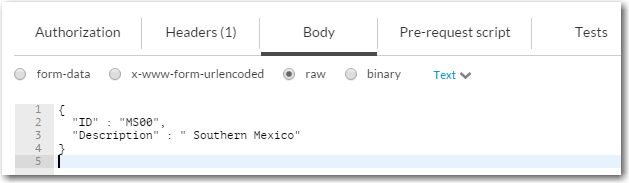
To create a record you use the POST HTTP verb and provide the data in the header of the request in JSON format.

In the Postman client, change the method to **POST** and add **/SalesOrg** to the service document URL.

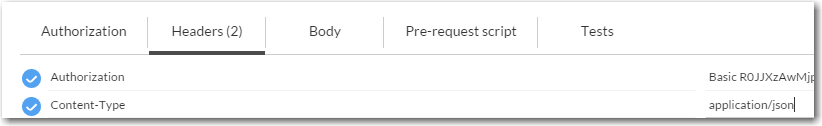


Click on the Body table and then the **Raw** option and enter the following data to the HTTP request header:

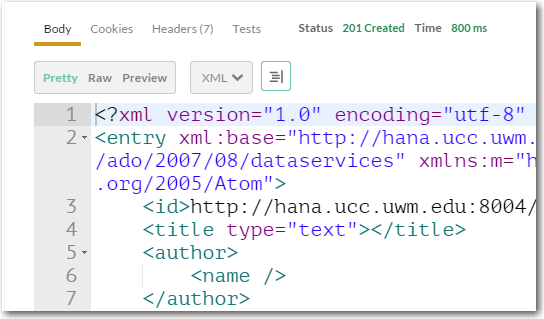
|  |
| --- |
| {  "ID" : "MS00",  "Description" : " Southern Mexico"  } |



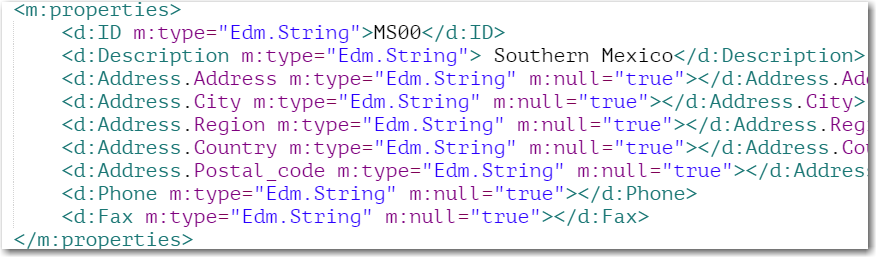
Click on the Headers tab and then enter the values shown below (Content-Type and applicaton/json). This tells HANA that the input data is in JSON format.



When you hit send the request, HANA responds with a status of 201 Created and the created record is returned.



Change the HTTP method to GET (or select the GET request in the history) and click Send again to see the contents of the table.

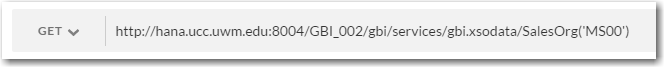


You can also open the table in the Catalog editor to see the table contents.

### Update

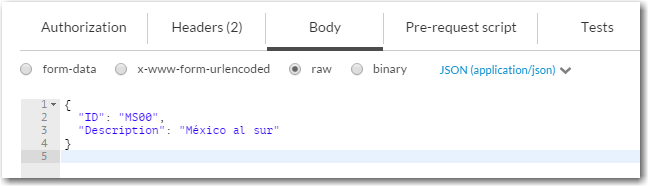
To perform an update you must use a URL that specifies the record to be updated and provide the updated data in HTTP request header in JSON formation.

In the Postman client, change the URL to add the id of the newly created record (/SalesOrg(‘MS00’)) and send a GET request.

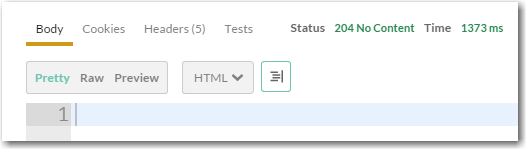


The response will be the newly created record. Change the method to PUT and enter the following data to the HTTP request header:

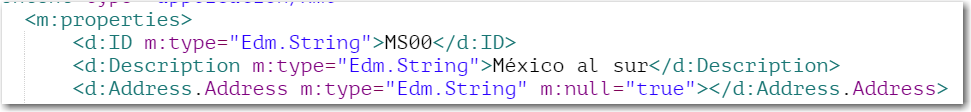
|  |
| --- |
| {  "ID": "MS00",  "Description": "México al sur"  } |



When you click Send the response will have a status of 204 No Content if the update was successful and no data will be returned.

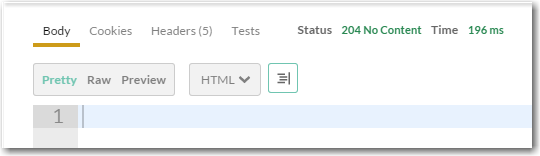


You can check that the update was successful by changing the method to GET and clicking Send again.

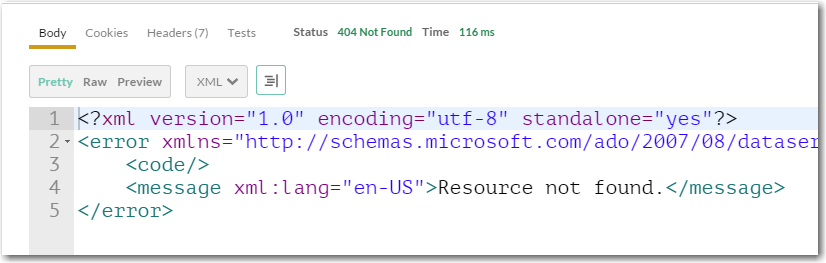


### Delete

To perform a DELETE you must change the method to DELETE and provide a URL that identifies the record to delete. Make sure you have included /SalesOrg(‘MS00’) on the URL to select the new record then change the method to DELETE and click Send. The response is:



If you change the method to GET and click Send you will receive a 404 Not Found error because the record no longer exists:



# Appendix 2

## Modifying database artifacts with CDS

While creating database artifacts with CDS is relatively simple, making modifications to those objects can be more problematic. Some changes cannot be accomplished in a straight forward way. For example, you cannot change the type of a column that already has data in a way that the data cannot be converted. Another frequent issue is when you attempt to make changes that affect related objects. For example, you cannot change the structure of a table if you have imported data using an .hdbti file if the change is incompatible with the data being imported. As a result, making changes to database artifacts can be complicated. **You should never edit the structure or delete database artifacts directly.**

Fortunately, it is relatively easy to delete and create artifacts. In many cases the best course of action is to delete the object then create it again with the changes. This may require you to delete any object that references the object you are attempting to change first. For example, assume you have table A and table B which has an association with table A. Also, assume you have imported data into both tables using an .hdbti file. If you now want to make a change to table A that is incompatible with the data you have imported you would have to follow this procedure:

1. Delete the references to both tables in the .hdbti file to eliminate the references to both tables.
2. Delete table B to eliminate the reference to table A.
3. Delete table A.

Then reverse the procedure to complete the change.

Another example is if you want to add fields to a table in which you’ve already imported data. The best course of action in this case is to:

1. Delete the reference to the table in the .hdbti file.
2. Open a SQL Console in the Catalog editor and use TRUNCATE TABLE <table name>; to delete the existing data.
3. Change the table structure in the .hdbdd file.
4. Update the data in the csv file.
5. Add the reference to the table back to the .hdbti file.

I encourage you to read the [documentation](http://help.sap.com/saphelp_hanaplatform/helpdata/en/b5/7824feaa2d492382c6bfcf6bd7aa9a/content.htm?frameset=/en/10/fadeb42a7a4717982de96eee0e26be/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=124) on this topic.