

HD2dC02wl – Creating a Persistence Model - Lite

Product and Focus

HANA Platform/CDS

Target Audience

Undergraduate/Graduate

Beginner to Intermediate

Author

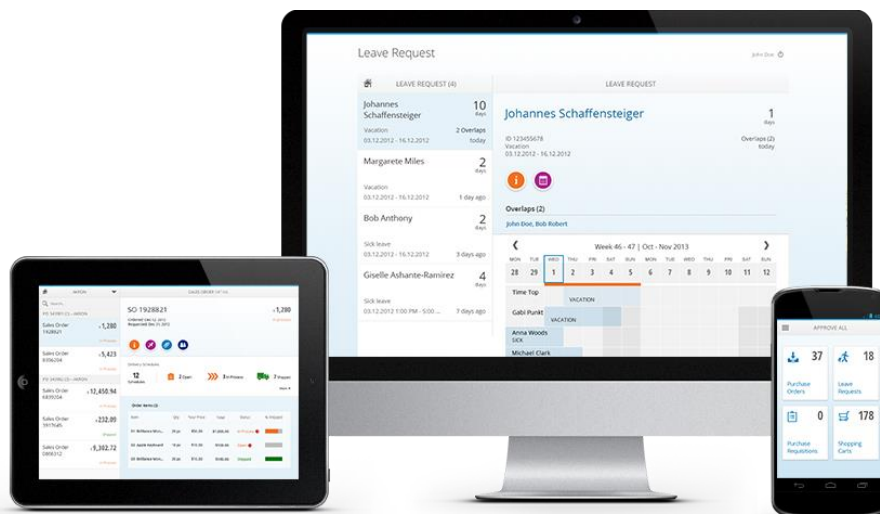
Ross Hightower

MOTIVATION

This case introduces HANA's [Core Data Services](#).

PREREQUISITES

None



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. The way that CDS works is 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 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 two views to demonstrate how they are created. The next case will show how oData associations and aggregations can provide the same results.

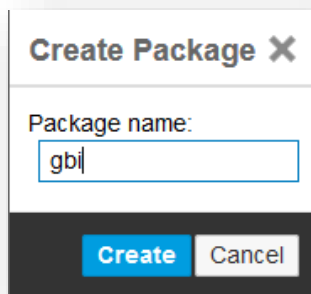


Create the Package

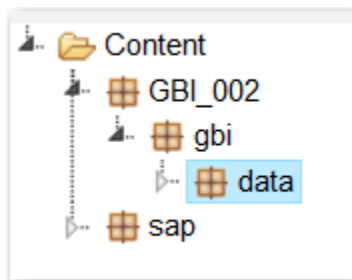
Login to the WDW and open the Editor.



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

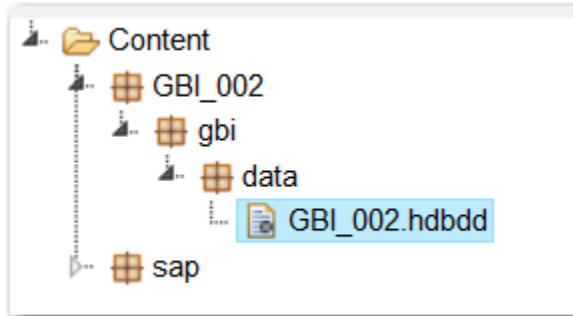
The first thing we'll do is create the overall structure of the persistence model. CDS database artifacts are described in [files](#) with an extension of `.hdbdd`. File extensions are important when working with HANA as they indicate what processes required when the files are activated. Right-click the **data** package and select **New→File**. Name the file **GBI_###.hdbdd**. Copy the code shown below into the file:

```
namespace GBI_002.gbi.data;  
  
@Schema: 'GBI_002'  
context GBI_002 {  
  
    context MASTERDATA {  
  
    };  
  
    context SALES {  
  
    };  
  
};
```

Listing 1

Substitute your values in the highlighted portions.

The first line defines the [namespace](#) 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.

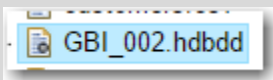


The @ on the next line indicates an [annotation](#) which provides the compiler information about how to process the file. The @SCHEMA annotation is required and indicates in which schema to create the artifacts. Your schema was created for you when your user id was created and has the same name as your user id. Substitute your schema name for the one in the sample code.

[Contexts](#) 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 the database artifacts. 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 pane below the editor. The first artifacts we will create are some user defined types.

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

[User defined types](#) 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 as our address type 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 2

```
@Schema: 'GBI_200'
context gbi_200 {
    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);
    };
};
```

Create the Tables

This section describes the creation of three tables: SALES_ORG, SALES_ORDERS, and CUSTOMERS. At the end of the section is an exercise in which you will create an additional table.

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 3

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](#) directive. Notice the use of the user defined type for the address. This will cause the creation of multiple fields with the names like `Address.Postal_code`.

There are number of [qualifiers](#) you can use to define the table fields. For example, the key qualifier in the SALESORGS table creates the primary key. Also the “not null” qualifier indicates the field cannot contain null values.

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

SAP HANA Web-based Development Workbench



Editor

Create, edit, execute, debug and manage HANA Repository artifacts



Catalog

Create, edit, execute and manage HANA DB SQL catalog artifacts



Security

Create users, create roles, assign objects and manage security



Traces

View, download traces for HANA applications, set trace levels

Drill into the Tables folder to find the table.

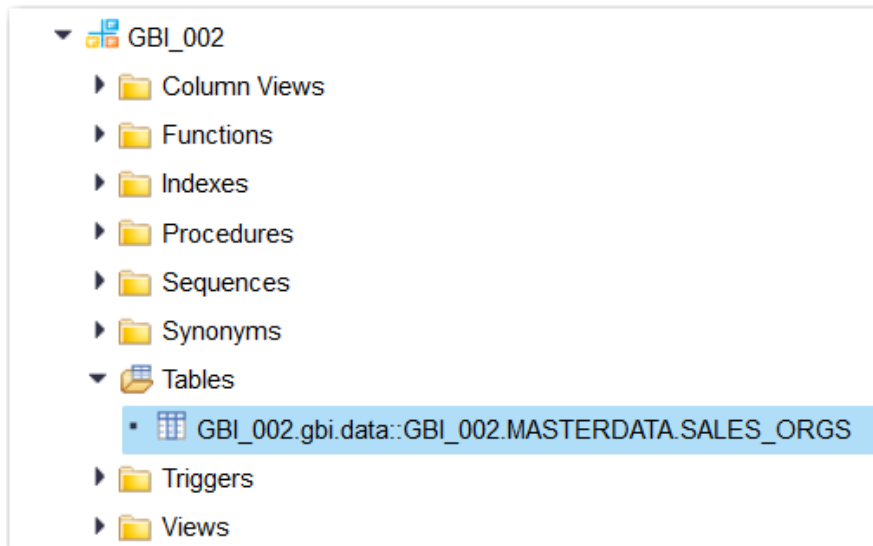


Table Name		Schema					
GBI_002.gbi.data::GBI_002.MASTERDATA.SALES_ORGS		GBI_002					
Columns		Indexes					
	Name	SQL Data Type	Dim	Column Store ...	Key	Not Null	
1	ID	NVARCHAR	4	STRING	(X1)	X	
2	Description	NVARCHAR	16	STRING		X	
3	Address.Address	NVARCHAR	35	STRING			
4	Address.City	NVARCHAR	20	STRING			
5	Address.Region	NVARCHAR	2	STRING			
6	Address.Country	NVARCHAR	2	STRING			
7	Address.Postal_code	NVARCHAR	5	STRING			
8	Phone	NVARCHAR	14	STRING			
9	Fax	NVARCHAR	14	STRING			

The name of the table is the namespace plus the context followed by the table name (there may be some inconsistencies between the image and the code provided in the case). 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 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 4

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,
  index : [ { name: 'CustomerIDIdx', order:#DESC, unique: false,
    elementNames:['CustomerID'] } ]
}
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 5

There are a number of new elements in this definition. The first is the creation of an [index](#). The index definition is included as part of the metadata defined with the @Catalog annotation. The properties of the object that defined the index is 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.

The other new element is the [association](#) created with the SALES_ORDER_DETAILS table. The qualifier creates a one-to-many [1..*] association with SALES_ORDER_DETAILS on that table's OrderNumber field.

When you save the file, the tables are created.

Table Name

Schema

GBI_002.gbi.data::GBI_002.SALES.SALES_ORDERS

GBI_002

Columns

Indexes

		Name	SQL Data Type	Dim	Column Store Data T...
	1	CreatedAt	DATE		DAYDATE
	2	CreatedBy	NVARCHAR	20	STRING
	3	CustomerID	NVARCHAR	10	STRING
	4	ID.OrderID	NVARCHAR	10	STRING
	5	GrossAmount.Amount	DECIMAL	17,3	FIXED
	6	GrossAmount.Currency	NVARCHAR	3	STRING
	7	Discount	DECIMAL	17,3	FIXED
	8	Status	NVARCHAR	15	STRING
	9	requiredDate	DATE		DAYDATE
	10	shipDate	DATE		DAYDATE

Notice the name of the field involved with the association. It consists of the name in the SALES definition and the name of the field in the other end of the association.

GBI_200.gbi.data::gbi_200.SALES.SALES_ORDER_DETAILS				GBI_200			
Columns		Indexes					
		Name	SQL Data Type	Dim	Column Store Data ...	Key	Not Null
	1	OrderID	NVARCHAR	10	STRING	(X1)	X
	2	OrderItem	NVARCHAR	3	STRING	(X2)	X
	3	ProductID	NVARCHAR	10	STRING		
	4	Quantity	INTEGER		INT		
	5	UnitOfMeasure	NVARCHAR	3	STRING		
	6	Revenue	DECIMAL	17,3	FIXED		
	7	Discount	DECIMAL	17,3	FIXED		

Table Creation Exercise

Now create a CUSTOMERS table. The structure and metadata are describe below.

Use the name of the table in the images to determine in which context they are created.

CUSTOMERS

GBI_002.gbi.data::GBI_002.MASTERDATA.CUSTOMERS					GBI_002		
Columns		Indexes					
		Name	SQL Data Type	Dim	Column Store Data T...	Key	Not N
	1	ID.CustomerID	NVARCHAR	10	STRING	(X1)	X
	2	CompanyName	NVARCHAR	35	STRING		X
	3	Address.Address	NVARCHAR	35	STRING		
	4	Address.City	NVARCHAR	20	STRING		
	5	Address.Region	NVARCHAR	2	STRING		
	6	Address.Country	NVARCHAR	2	STRING		
	7	Address.Postal_code	NVARCHAR	5	STRING		
	8	SalesOrgID	NVARCHAR	4	STRING		

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.

The CUSTOMERS table also has an ascending index based on the CompanyName field.

Import Data into the Tables

CDS includes the capability to [import](#) data into tables from csv file. 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 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 to 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 until the upload multifile 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,94105,1-415-555-7900,1-415-555-7901
```

Listing 6

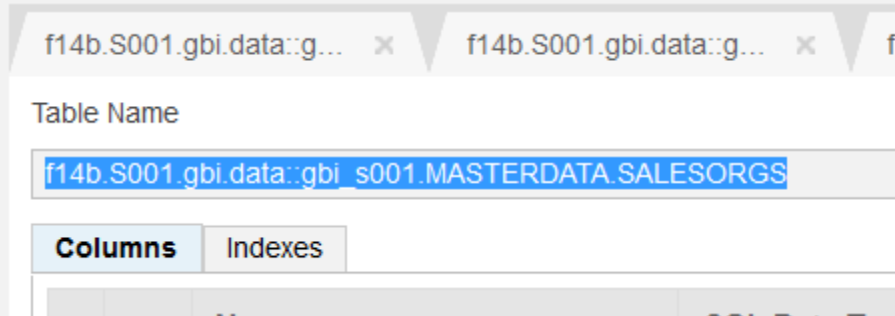
The values must align with the fields defined in the .hdbdd file.

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_002.gbi.data::GBI_002.MASTERDATA.SALES_ORGS";
    schema = "GBI_002";
    file = "GBI_002.gbi.data:salesorg.csv";
    header = false;
  }
];
```

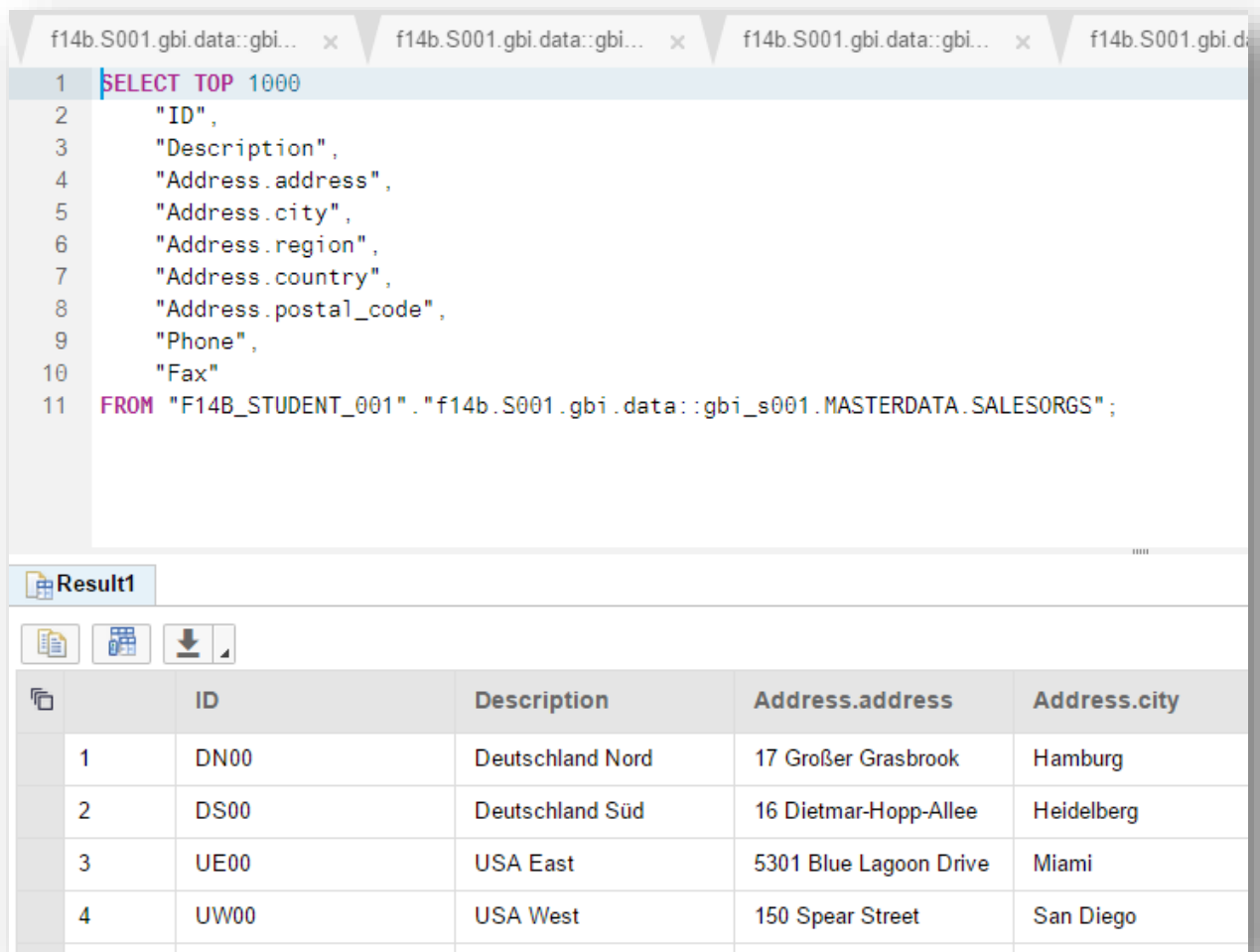
Listing 7

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



The 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 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** or click the file to open its structure and then click the Open Content button.



The screenshot shows the SAP HANA Studio interface. At the top, there are four tabs, all labeled 'f14b.S001.gbi.data::gbi...'. The active tab displays a SQL query in a text editor. The query is a 'SELECT TOP 1000' statement that lists columns: 'ID', 'Description', 'Address.address', 'Address.city', 'Address.region', 'Address.country', 'Address.postal_code', 'Phone', and 'Fax'. The data is sourced from the table 'MASTERDATA.SALESORGS' within the schema 'F14B_STUDENT_001'. Below the editor, a section titled 'Result1' shows the query's output as a table. The table has five columns: 'ID', 'Description', 'Address.address', and 'Address.city'. It displays the first four rows of data, which correspond to the 'top 1000' specified in the query.

```

1 SELECT TOP 1000
2     "ID",
3     "Description",
4     "Address.address",
5     "Address.city",
6     "Address.region",
7     "Address.country",
8     "Address.postal_code",
9     "Phone",
10    "Fax"
11 FROM "F14B_STUDENT_001"."f14b.S001.gbi.data::gbi_s001.MASTERDATA.SALESORGS";

```

	ID	Description	Address.address	Address.city
1	DN00	Deutschland Nord	17 Großer Grasbrook	Hamburg
2	DS00	Deutschland Süd	16 Dietmar-Hopp-Allee	Heidelberg
3	UE00	USA East	5301 Blue Lagoon Drive	Miami
4	UW00	USA West	150 Spear Street	San Diego

Import Data Exercise

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

```
import = [  
  {  
    table = "GBI_002.gbi.data::GBI_002.MASTERDATA.SALES_ORGS";  
    schema = "GBI_002";  
    file = "GBI_002.gbi.data:salesorg.csv";  
    header = false;  
  },  
  {  
    table = "GBI_002.gbi.data::GBI_002.SALES.SALES";  
    schema = "GBI_002";  
    file = "GBI_002.gbi.data:salesorders.csv";  
    header = false;  
  }  
];
```

Create Some Views

A [view](#) 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 8

This view makes use of the association between CUSTOMERS and SALES_ORDERS. Note the fields from SALES_ORDERS are prefixed with ID. The view shows the GrossAmount of all orders with a Status of New. It would be easier to do this with a WHERE clause but CDS does not support one-to-many associations in the WHERE clause of views yet. Note that the GROUP BY clause must include the list of fields that appear before the sum aggregation statement.

Create View Exercise

Create one more view:

SalesByCustomer

This view shows the sum GrossAmount, Discount and GrossAmount – Discount grouped by CustomerID and ordered by sum of GrossAmount in [descending order](#).

Appendix

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.

Fortunately, it is relatively easy to delete and create artifacts. In many cases the best course of action 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](#) on this topic.