Using Native SAP HANA Artifacts

Create or use an existing database object (table, view, table function, calculation view) and make use of it in your CDS model, for instance for exposing it in an OData service.

Introduction

To leverage CAP in combination with native SAP HANA artifacts, it is important to understand HDI containers. An HDI container is a database schema controlled by HDI. This has implications on the handling and you need to know the way HDI works , to fully understand the approach described in this advanced guide.

Adding Native SAP HANA Objects

You create a new database object or there's an existing object (table, view, table function, calculation view) and you want to use it in your CDS model, for instance for exposing it in an OData service.

Add Existing SAP HANA Objects from Other HDI Containers

To access database artifacts residing in other HDI containers, you need the permissions granted for that container and you need to introduce them into your own container using

synonyms. This synonym establishes a link between both needed HDI containers. The *.hdbsynonym* file you create for this, is a native SAP HANA object in your project.

> **TIP**
>
> Synonyms can be used to rename database objects.

Create Native SAP HANA Objects

To create SAP HANA native tables or use SAP HANA native features, use the folder *db/src* at design time and build, for example, your *.hdbtable* or *.hdbsynonym* files. This folder stays untouched during the `cds` build and the content is copied over to the *gen/db/src* folder during the build. Use this process for all tables and features that can't be modeled using *CDS*.

> **TIP**
>
> You can use "mapping views" (*.hdbview*) to rename database objects and column names, when you make the object known to CDS.

Make the Object Known to CDS

- Define an entity that matches the signature of the newly designed or already existing database object.
- Add the annotation `@cds.persistence.exists` to tell CDS that this object already exists on the database and must not be generated.

This entity then serves as a facade for the database object and can be used in the model like a regular entity. In the following, we refer to this entity as **facade entity**.

Steps to match the signature of a database object in a facade entity:

- Choose a name for the facade entity, which is identical to the resulting database name of the existing database object.
- Choose the names of the facade entity's elements, which are identical to the resulting database names of the existing database object's column names.

- After applying the CDS-to-DB type mapping, check that the types of the facade entity's elements match the types of the database object's columns.
- For a view, table function, or calculation view with parameters, check that the parameter names and types match, too. Functions with table-like input parameters are not supported.

> Note: If a field of that entity is defined as `not null` and you want to disable its runtime check, you can add `@assert.notNull: false`. This is important if you want to use, for example SAP HANA history tables .

As a result, the database name is defined by the name of the entity or its elements, after applying the SQL name mapping. We can distinguish two types of names - **plain** and **quoted**.

Naming Mode	Description
Plain (default)	When using **plain names**, the database name is converted to uppercase and all dots are replaced by underscores. This conversion is the default behavior. If a database name is all in uppercase and is a regular SQL identifier, then it's possible to construct a corresponding name in the CDS model that matches this name.
Quoted	If the existing database name also contains lower-case characters or characters that can't occur in regular SQL identifiers, it's not possible to choose a name in the CDS model that matches this name. Let's call such a database name "quoted", as the only possibility to create such a name is to quote it in the CREATE statement. In this case, it's necessary to introduce an additional database object (a synonym or a view) on top of the existing database object and construct the facade entity for this newly introduced mapping object.

↳ *Find here troubleshooting related to SAP HANA.*

Tables and Views Without Parameters

As the approach described here only depends on the signature of the existing database object, it applies to:

- Database tables
- SQL views without parameters
- Table functions without parameters
- Calculation views without parameters

For simplicity, we only talk about tables in this section, but everything applies to the other mentioned objects in the same way.

Plain Names

Assume that all names in the existing database table are plain names. Define the facade entity in such a way that the resulting database names match those of the table.

existing-table-without-params-plain.hdbtable

```sql
COLUMN TABLE DATA_MODEL_BOOKSHOP_BOOKS (
  ID integer,
  THE_TITLE nvarchar(100),
  primary key ( ID )
)
```

facade-entity-without-params.cds

```cds
namespace data.model;

context Bookshop {
  @cds.persistence.exists
  entity Books {
    key id        : Integer;
        the_title : String(100);
  }
}
```

Quoted Table Name, Plain Column Names

Assume that the name of the existing table contains lower case characters, ".", and "::". It isn't possible to define a CDS name that is mapped to this name. So, we introduce a synonym that maps *`data.model::Bookshop.Books`* to *`DATA_MODEL_BOOKSHOP_BOOKS`* .

existing-table-quoted-table-name.hdbtable

```sql
COLUMN TABLE "data.model::Bookshop.Books" (
  ID integer,
  THE_TITLE nvarchar(100),
  primary key ( ID )
)
```

existing-table-quoted-table-name.hdbsynonym

```json
{
  "DATA_MODEL_BOOKSHOP_BOOKS" : {
    "target": {
      "object" : "data.model::Bookshop.Books"
    }
  }
}
```

Now, define a facade entity in CDS. The table columns have plain names and thus need no mapping.

facade-entity-without-params.cds

```cds
namespace data.model;

context Bookshop {
  @cds.persistence.exists
  entity Books {
    key id        : Integer;
        the_title : String(100);
  }
}
```

Quoted Table Name, Quoted Column Names

Assume that the table name and column names are quoted names. There, a synonym isn't sufficient, because it can't map the column names. In this case, put a "mapping" view on top of the existing table that maps all the names to plain ones:

existing-table-quoted-names.hdbtable

```sql
COLUMN TABLE "data.model::Bookshop.Books" (
  "id" integer,
  "the.title" nvarchar(100),
  primary key ( "id" )
)
```

mapping-view-quoted-names-2.hdbview

```sql
VIEW DATA_MODEL_BOOKSHOP_BOOKS AS SELECT
  "id"        AS ID,
```

```
  "the.title" AS THE_TITLE
FROM "data.model::Bookshop.Books"
```

Now, define a facade entity in CDS.

facade-entity-without-params.cds

```cds
namespace data.model;

context Bookshop {
  @cds.persistence.exists
  entity Books {
    key id        : Integer;
        the_title : String(100);
  }
}
```

Views with Parameters

Assume that the existing database object is an SQL view with parameters, a table function with parameters, or a calculation view with parameters.

Parameters can be added to the CDS model starting with *`@sap/cds 3.4.1`* / *`@sap/cds-compiler 1.7.1`* .

Plain Names

Assume that all names are plain. You can directly define the facade entity with parameters.

existing-view-with-params-plain.hdbview

```sql
VIEW DATA_MODEL_BOOKSHOP_BOOKINFO (in AUTHOR nvarchar(100)) AS SELECT
  ID,
  'The book: ' || THE_TITLE || ' and the author ' || :AUTHOR AS BOOK_AUTHOR_I
FROM DATA_MODEL_BOOKSHOP_BOOKS;
```

facade-entity-with-params.cds

```cds
namespace data.model;
context Bookshop {
  @cds.persistence.exists
  entity Bookinfo (AUTHOR : String(100)) {
    key id : Integer;
    book_author_info : String(100);
  }
}
```

Quoted Names

Assume the SQL view with parameters has quoted names. Put a mapping view on top of the existing one that maps all the names, except the parameter names, to plain ones.

> Note: Names of parameters in SQL views and in table functions can't be quoted.

existing-view-quoted-names.hdbview

```sql
VIEW "data.model.Bookshop.Bookinfo" (in AUTHOR nvarchar(10)) AS SELECT
  ID AS "id",
  'The book: ' || THE_TITLE || ' and the author ' || :AUTHOR AS "book.author.
FROM DATA_MODEL_BOOKSHOP_BOOKS;
```

mapping-view-quoted-names.hdbview

```sql
VIEW DATA_MODEL_BOOKSHOP_BOOKINFO (in AUTHOR nvarchar(10)) AS SELECT
  "id"          AS ID,
  "book.author.info" AS BOOK_AUTHOR_INFO
FROM "data.model.Bookshop.Bookinfo"(AUTHOR => :AUTHOR)
```

Now, define a facade entity with parameters.

facade-entity-with-params.cds

```cds
namespace data.model;
context Bookshop {
  @cds.persistence.exists
  entity Bookinfo (AUTHOR : String(100)) {
    key id : Integer;
    book_author_info : String(100);
```

```
  }
}
```

In contrast to SQL views or table functions, the names of calculation view parameters can be quoted, too. The following is the definition of a calculation view `data.model.bookshop.CalcBooks` with elements `id`, `the.title`, and `calculated`, and in addition there's a parameter `Param`.

existing-calc-view-quoted.hdbcalculationview

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Calculation:scenario xmlns:Calculation="https://www.sap.com/ndb/BiModelCalcu
<descriptions defaultDescription="Calculation View w/ parameters and quoted na
<localVariables>
  <variable id="Param" parameter="true">
    <variableProperties datatype="INTEGER" mandatory="true">
      <valueDomain type="empty"/>
      <selection multiLine="false" type="Single"/>
      <defaultRange/>
    </variableProperties>
  </variable>
</localVariables>
<variableMappings/>
<dataSources>
  <DataSource id="DATA_MODEL_BOOKSHOP_BOOKS">
    <resourceUri>DATA_MODEL_BOOKSHOP_BOOKS</resourceUri>
  </DataSource>
</dataSources>
<calculationViews/>
<logicalModel id="DATA_MODEL_BOOKSHOP_BOOKS">
  <attributes>
    <attribute id="the.title" order="2" attributeHierarchyActive="false" disp
      <keyMapping columnObjectName="DATA_MODEL_BOOKSHOP_BOOKS" columnName="THI
    </attribute>
  </attributes>
  <calculatedAttributes>
    <calculatedAttribute id="calculated" order="3" semanticType="empty" displa
      <keyCalculation datatype="INTEGER" expressionLanguage="COLUMN_ENGINE">
        <formula>&quot;id&quot;+$$Param$$</formula>
      </keyCalculation>
    </calculatedAttribute>
  </calculatedAttributes>
  <baseMeasures>
```

```
      <measure id="id" order="1" aggregationType="sum" measureType="simple">
        <measureMapping columnObjectName="DATA_MODEL_BOOKSHOP_BOOKS" columnName
      </measure>
    </baseMeasures>
  </logicalModel>
  </Calculation:scenario>
```

For this calculation view, put a mapping view on top that maps all the names, except the parameter names, to plain ones. Note the weird syntax for passing parameters to a calculation view.

mapping-calc-view-quoted.hdbview

```sql
VIEW DATA_MODEL_BOOKSHOP_CALCBOOKS (in PARAM nvarchar(10)) AS SELECT
  "id"                AS ID,
  "the.title"         AS THE_TITLE,
  "calculated"        AS CALCULATED
FROM "data.model.bookshop.CalcBooks"(placeholder."$$Param$$" => :PARAM)
```

Define the facade entity in CDS.

facade-entity-calc-view.cds

```cds
namespace data.model;
context Bookshop {
  @cds.persistence.exists
  entity CalcBooks (PARAM : String(10)) {
    key id : Integer;
    the_title : String(100);
    calculated : Integer;
  }
}
```

Default Values of View Parameters

To set the default value of a parameter in a view, use the `default` keyword. This value will be evaluated at runtime and used as a fallback value in case if no other value was provided by the client.

facade-entity-with-def-val-params.cds

```cds
namespace data.model;
context Bookshop {
  @cds.persistence.exists
  entity Bookinfo (AUTHOR : String(100) default 'Unknown') {
    key id : Integer;
    book_author_info : String(100);
  }
}
```

Calculated Views and User-Defined Functions

Calculated view parameters need to be rendered as `PLACEHOLDER."$$$$"` . User-defined function parameters are rendered as ordinary parameters.

If a user-defined function has an empty or no parameter list, it still must be called with an empty parameter list '()'. This is the trigger for SAP HANA to execute the function instead of searching for a regular view, which eventually doesn't exist. Calculated views without parameters are called with no parameter list.

To produce the correct SQL view statement, use `@cds.persistence.exists` and one of the following annotations at the facade entity as a hint for the code generation:

- `@cds.persistence.udf` to specify that the facade entity represents a user-defined function
- `@cds.persistence.calcview` to specify that the facade entity represents a calculation view

Have a look at the following CDS sample and the generated view:

facade-entity-existing-calc-view.cds

```cds
@cds.persistence.exists
@cds.persistence.calcview
entity AddressCalcView (USERID: Integer) {
    key id: Integer;
};

view WeUseAddressCalcView  as select from AddressCalcView(USERID: 4711);

@cds.persistence.exists
@cds.persistence.udf
```

```
entity AddressUDF {
    key id: Integer;
};

view WeUseAddressUDF as select from AddressUDF;
```

mapping-calc-view.hdbview

```sql
VIEW WeUseAddressCalcView AS SELECT
  AddressCalcView_0.id
FROM AddressCalcView(PLACEHOLDER."$$USERID$$" => 4711) AS AddressCalcView_0;

VIEW WeUseAddressUDF AS SELECT
  AddressUDF_0.id
FROM AddressUDF() AS AddressUDF_0;
```

Associations and Compositions

This section describes how associations and compositions to artifacts with `@cds.persistence.skip/exists` are treated during the generation of the database model with `forHana`.

`@cds.persistence.skip`

Denotes that the artifact isn't available in the database but eventually implemented by custom code.

No association can point to a nonexisting database object and no query can be executed against such a nonexisting source. As `@cds.persistence.skip` is *propagated*, projections also don't become part of the database schema.

All association definitions to nonexisting database objects are removed from the defining database objects and all usages of such associations produce an error.

The following sample would throw an error:

```cds
entity Orders
{
  key id: Integer;
  orderName: String;
  items: Composition of Items on $self = items.parent;
};
@cds.persistence.skip
entity Items
{
  key id: Integer;
  name: String;
  parent: Association to Orders;
};
view OrdersView as select from Orders
{
  id,
  orderName,
  items.name
};
view ItemSelection as select from Items;
```

The view `Orders` will be rejected with an error message as `items.name` isn't resolvable to a valid JOIN expression and view `ItemSelection` is effectively annotated with `@cds.persistence.skip`.

`@cds.persistence.exists`

Denotes that there already exists a native database object, which should be used during runtime.

The CDS artifact merely acts as a *proxy* artifact, representing the signature of the native database artifact in the CDS model. Since the database object really exists, it's *indirectly* possible to associate these native database objects.

Associations to artifacts annotated with `@cds.persistence.exists` are removed from the defining database objects and all usages of such associations produce an error, as the following example shows:

```cds
entity Orders
{
  key id: Integer;
  orderName: String;
  items: Composition of Item on $self = items.parent;
```

```
};

@cds.persistence.exists
entity Items
{
  key id: Integer;
  name: String;
  parent: Association to Orders;
};

view OrdersView as select from Orders
{
  id,
  orderName,
  items.name
};

entity ItemSelection as projection on Items;
```

However, as the annotation *`@cds.persistence.exists`* **isn't** propagated, this allows using such proxy artifacts as query sources and to be valid association targets.

The example can now be rewritten to:

cds

```
entity Orders
{
  key id: Integer;
  orderName: String;
  items: Composition of ItemSelection on $self = items.parent; // <--- compos
};

@cds.persistence.exists
entity Items
{
  key id: Integer;
  name: String;
  parent: Association to Orders;
};

view OrdersView as select from Orders
{
  id,
  orderName,
```

```
  items.name // <--- is transformable into a valid JOIN expression
};

entity ItemSelection as projection on Items;
```

By composing `ItemSelection` instead of `Items`, it's possible to use this composition in `Orders`.

SAP HANA-Specific Data Types

The following SAP HANA-specific data types are primarily intended for porting existing SAP HANA CDS models into the CAP domain if the old SAP HANA types must be preserved in the existing database tables. If you're starting from scratch, these types shouldn't be used but only the predefined CDS types.

CDS Type	Arguments / Remarks	SQL	OData (V4)
`hana.SMALLINT`		*SMALLINT*	*Edm.Int16*
`hana.TINYINT`		*TINYINT*	*Edm.Byte*
`hana.SMALLDECIMAL`		*SMALLDECIMAL*	*Edm.Decimal*
`hana.REAL`		*REAL*	*Edm.Single*
`hana.CHAR`	(`length`)	*CHAR*	*Edm.String*
`hana.NCHAR`	(`length`)	*NCHAR*	*Edm.String*
`hana.VARCHAR`	(`length`)	*VARCHAR*	*Edm.String*
`hana.CLOB`		*CLOB*	*Edm.String*
`hana.BINARY`	(`length`)	*BINARY*	*Edm.Binary*
`hana.ST_POINT`	(`srid`) [1]	`ST_POINT`	*Edm.GeometryPoint*
`hana.ST_GEOMETRY`	(`srid`) [1]	`ST_GEOMETRY`	*Edm.Geometry*

[1] Optional, default: 0

Mapping UUIDs to SQL

By default, `cds` maps UUIDs to `nvarchar(36)` in SQL databases. The length is to accommodate representations with hyphens as well as any other representations. The choice of a string type over a raw/binary type is in line with this recommendation from SAP HANA:

> If the client side needs to work with the UUID, VARBINARY would lead to CAST operations or binary array handling at the client side. Here **NVARCHAR would be the data type of choice** to avoid handling binary arrays on the client side.

Example Index

> Download from here a fully fledged model with even more examples.

What	Database Object	Mapping Object
table	DATA_MODEL_BOOKSHOP_BOOKS	n/a
table	data.model::Bookshop.Books	DATA_MODEL_BOOKSHOP_BOOKS
view with param	DATA_MODEL_BOOKSHOP_BOOKINFO	n/a
view with param	data.model.Bookshop.Bookinfo	DATA_MODEL_BOOKSHOP_BOOKINFO
cv with param	data.model.bookshop.CalcBooks	DATA_MODEL_BOOKSHOP_CALCBOOKS