PUBLIC
SAP HANA Platform 2.0 SPS 05
Document Version: 1.1 – 2021-07-09

SAP HANA Core Data Services (CDS) Reference

THE BEST RUN SAP

Content

1 SAP HANA Core Data Services (CDS) Reference

Use SAP HANA Core Data Services (CDS) to build design-time data-persistence models in SAP HANA Extended Application Services.

The *SAP HANA Core Data Services (CDS) Reference* explains how to use SAP HANA CDS to define and consume semantically rich, design-time data models in SAP HANA (on-premise). The model described in CDS enables you to use the Data Definition Language to define the artifacts that make up the data-persistence model. Data is exposed in response to client requests via HTTP, for example, from a Fiori or an SAPUI5-based application.

The information in the guide is organized as follows:

- Getting started
  An overview of the process of developing data models for application for SAP HANA XS; some information about the roles and permissions required for XS development; and an introduction to the process of setting up the data-persistence model in SAP HANA
- Defining data models in XS classic
  Detailed, step-by-step information about defining the data model, managing the data model in the SAP HANA repository, activating the data model and managing the resulting objects in the database catalog, and consuming the data model (for example, in a client UI)
- Defining Data models in XS advanced
  Detailed, step-by-step information about defining the data model, setting up the SAP HANA deployment infrastructure (HDI), deploying the data model, and consuming the data model (for example, in a client UI)

> **i Note**
>
> SAP HANA CDS (on-premise) is not compatible or interchangeable with SAP CAP CDS, the Core Data Services used by the SAP Cloud Application Programming Model (SAP CAP). Although SAP CAP CDS is syntactically similar to the SAP HANA CDS used in XS classic and XS advanced, the two CDS dialects are not compatible: they use different artifact types (`.hdbcds` and `.cds`, respectively) and are intended for use in different development scenarios. For more information about SAP CAP CDS, see *About SAP CAP* in *Related Information* below.

Related Information

About CAP (SAP Cloud Application Programming Model)
Getting Started with Core Data Services [page 6]

2 Getting Started with Core Data Services

Core Data Services (CDS) is an infrastructure that can be used by database developers to create the underlying (persistent) data model which the application services expose to UI clients.

The database developer defines the data-persistence and analytic models that are used to expose data in response to client requests via HTTP. With CDS, you can define a persistence model that includes objects such as tables, views, and structured types; the database objects specify what data to make accessible for consumption by applications and how. This guide takes you through the tasks required to create CDS documents that define the objects most often used in a data persistence model, for example:

- Create tables (entities)
- Create SQL views
- Create associations between entities or views
- Create user-defined structured types

The *SAP HANA Core Data Services (CDS) Reference* also provides code examples that illustrate how to specify the various object types. In addition, the CDS Reference also includes the complete specification of the CDS syntax required for each object type.

> → Tip
>
> For more information about the tutorials and reference guides available for SAP HANA developers, see *The SAP HANA Developer's Information Atlas* in *Related Information* below.

Building the data model is the first step in the overall process of developing applications that provide access to the SAP HANA database. When you have created the underlying data persistence model, application developers can build the application services that expose selected elements of the data model to client application by means of so-called “data end-points”. The client applications bind UI controls such as buttons or charts and graphs to the application services which in turn retrieve and display the requested data.

Prerequisites

Before you can start using CDS to define the objects that comprise your persistence model, you need to ensure that the following prerequisites are met:

- You must have access to an SAP HANA system.
- You must have already created a development workspace and a project.
- You must have shared a project for the CDS artifacts so that the newly created files can be committed to (and synchronized with) the repository (XS classic only).
- You must have created a schema for the CDS catalog objects created when the CDS document is activated in the repository, for example, `MYSCHEMA` (XS classic only).

  > i Note
  >
  > It is not possible to use the CDS syntax to define a design-time representation of a database schema.

- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects (XS classic only).

Related Information

Setting up the Data Persistence Model in SAP HANA [page 7]
Creating the Persistence Model in Core Data Services [page 10]
Creating Data Persistence Artifacts with CDS in XS Advanced [page 134]
The SAP HANA Developer's Information Atlas

2.1 Setting up the Data Persistence Model in SAP HANA

The persistence model defines the schema, tables, sequences, and views that specify what data to make accessible for consumption by XS applications and how.

In SAP HANA Extended Application Services (SAP HANA XS), the persistence model is mapped to the consumption model that is exposed to client applications and users so that data can be analyzed and displayed in the appropriate form in the client application interface. The way you design and develop the database objects required for your data model depends on whether you are developing applications that run in the SAP HANA XS classic or XS advanced run-time environment.

- SAP HANA XS Classic Model [page 7]
- SAP HANA XS Advanced Model [page 8]

SAP HANA XS Classic Model

SAP HANA XS classic model enables you to create database schema, tables, views, and sequences as design-time files in the SAP HANA repository. Repository files can be read by applications that you develop. When implementing the data persistence model in XS classic, you can use either the Core Data Services (CDS) syntax or HDBtable syntax (or both). “HDBtable syntax” is a collective term; it includes the different configuration schema for each of the various design-time data artifacts, for example: schema (`.hdbschema`), sequence (`.hdbsequence`), table (`.hdbtable`), and view (`.hdbview`).

All repository files including your view definition can be transported (along with tables, schema, and sequences) to other SAP HANA systems, for example, in a delivery unit. A delivery unit is the medium SAP HANA provides to enable you to assemble all your application-related repository artifacts together into an archive that can be easily exported to other systems.

> **i** Note
>
> You can also set up data-provisioning rules and save them as design-time objects so that they can be included in the delivery unit that you transport between systems.

The rules you define for a data-provisioning scenario enable you to import data from comma-separated values (CSV) files directly into SAP HANA tables using the SAP HANA XS table-import feature. The complete data-import configuration can be included in a delivery unit and transported between SAP HANA systems for reuse.

As part of the process of setting up the basic persistence model for SAP HANA XS, you create the following artifacts in the XS classic repository:

XS Classic Data Persistence Artifacts by Language Syntax and File Suffix

XS Classic Artifact Type	CDS	HDBTable
Schema	`.hdbschema` *	`.hdbschema`
Synonym	`.hdbsynonym`*	`.hdbsynonym`
Table	`.hdbdd`	`.hdbtable`
Table Type	`.hdbdd`	`.hdbstructure`
View	`.hdbdd`	`.hdbview`
Association	`.hdbdd`	-
Sequence	`.hdbsequence`*	`.hdbsequence`
Structured Types	`.hdbdd`	-
Data import	`.hdbti`	`.hdbti`

> **i Note**
>
> (*) To create a schema, a synonym, or a sequence, you must use the appropriate HDBTable syntax, for example, `.hdbschema`, `.hdbsynonym`, or `.hdbsequence`. In a CDS document, you can include references to both CDS and HDBTable artifacts.

On activation of a repository artifact, the file suffix (for example, `.hdbdd` or `.hdb[table|view]`) is used to determine which run-time plug-in to call during the activation process. When you activate a design-time artifact in the SAP HANA Repository, the plug-in corresponding to the artifact's file suffix reads the contents of repository artifact selected for activation (for example, a table, a view, or a complete CDS document that contains multiple artifact definitions), interprets the artifact definitions in the file, and creates the appropriate corresponding run-time objects in the catalog.

SAP HANA XS Advanced Model

For the XS advanced run time, you develop multi-target applications (MTA), which contain modules, for example: a database module, a module for your business logic (Node.js), and a UI module for your client interface (HTML5). The modules enable you to group together in logical subpackages the artifacts that you need for the various elements of your multi-target application. You can deploy the whole package or the individual subpackages.

As part of the process of defining the database persistence model for your XS advanced application, you use the database module to store database design-time artifacts such as tables and views, which you define using Core Data Services (CDS). However, you can also create procedures and functions, for example, using SQLScript, which can be used to insert data into (and remove data from) tables or views.

> **i Note**
>
> In general, CDS works in XS advanced (HDI) in the same way that it does in the SAP HANA XS classic Repository. For XS advanced, however, there are some incompatible changes and additions, for example, in the definition and use of name spaces, the use of annotations, the definition of entities (tables) and structure types. For more information, see *CDS Documents in XS Advanced* in the list of *Related Links* below.

In XS advanced, application development takes place in the context of a project. The project brings together individual applications in a so-called Multi-Target Application (MTA), which includes a module in which you define and store the database objects required by your data model.

1. Define the data model.
   Set up the folder structure for the design-time representations of your database objects; this could include CDS documents that define tables, data types, views, and so on. But it could also include other database artifacts, too, for example: your stored procedures, synonyms, sequences, scalar (or table) functions, and any other artifacts your application requires.

   > **→ Tip**
   >
   > You can also define the analytic model, for example, the calculation views and analytic privileges that are to be used to analyze the underlying data model and specify who (or what) is allowed access.

2. Set up the SAP HANA HDI deployment infrastructure.
   This includes the following components:
   - The HDI configuration
     Map the design-time database artifact type (determined by the file extension, for example, `.hdbprocedure`, or `.hdbcds` in XS advanced) to the corresponding HDI build plug-in in the HDI configuration file (`.hdiconfig`).
   - Run-time name space configuration (**optional**)
     Define rules that determine how the run-time name space of the deployed database object is formed. For example, you can specify a base prefix for the run-time name space and, if desired, specify if the name of the folder containing the design-time artifact is reflected in the run-time name space that the deployed object uses.
     Alternatively, you can specify the use of freestyle names, for example, names that do not adhere to any name-space rules.
3. Deploy the data model.
   Use the design-time representations of your database artifacts to generate the corresponding active objects in the database catalog.
4. Consume the data model.
   Reference the deployed database objects from your application, for example, using OData services bound to UI elements.

Related Information

Creating the Persistence Model in Core Data Services [page 10]
Creating Data Persistence Artifacts with CDS in XS Advanced [page 134]
CDS Documents in XS Advanced [page 148]

3 Creating the Persistence Model in Core Data Services

Core data services (CDS) is an infrastructure that can be used to define and consume semantically rich data models in SAP HANA.

The model described in CDS enables you to use the Data Definition Language to define the artifacts that make up the data-persistence model. You can save the data-persistence object definition as a CDS artifact, that is; a design-time object that you manage in the SAP HANA repository and activate when necessary. Using a data definition language (DDL), a query language (QL), and an expression language (EL), CDS enables write operations, transaction semantics, and more.

You can use the CDS specification to create a CDS document which defines the following artifacts and elements:

- Entities (tables)
- Views
- User-defined data types (including structured types)
- Contexts
- Associations
- Annotations

> **i Note**
>
> To create a schema, a synonym, or a sequence, you must use the appropriate `.hdbtable` artifact, for example, `.hdbschema`, `.hdbsynonym`, or `.hdbsequence`. You can reference these artifacts in a CDS document.

CDS artifacts are design-time definitions that are used to generate the corresponding run-time objects, when the CDS document that contains the artifact definitions is activated in the SAP HANA repository. In CDS, the objects can be referenced using the name of the design-time artifact in the repository; in SQL, only the name of the catalog object can be used. The CDS document containing the design-time definitions that you create using the CDS-compliant syntax must have the file extension `.hdbdd`, for example, `MyCDSTable.hdbdd`.

Related Information

Create a CDS Document [page 14]
Create an Entity in CDS [page 39]
Create a User-defined Structured Type in CDS [page 62]
Create an Association in CDS [page 76]
Create a View in CDS [page 91]
CDS Annotations [page 28]

3.1 CDS Editors

The SAP Web IDE for SAP HANA provides editing tools specially designed to help you create and modify CDS documents.

SAP Web IDE for SAP HANA includes dedicated editors that you can use to define data-persistence objects in CDS documents using the DDL-compliant Core Data Services syntax. SAP HANA XS advanced model recognizes the `.hdbcds` file extension required for CDS object definitions and, at deployment time, calls the appropriate plug-in to parse the content defined in the CDS document and create the corresponding run-time object in the catalog. If you right-click a file with the `.hdbcds` extension in the *Project Explorer* view of your application project, SAP Web IDE for SAP HANA provides the following choice of editors in the context-sensitive menu.

- CDS Text Editor [page 11]
  View and edit DDL source code in a CDS document as text with the syntax elements highlighted for easier visual scanning.
  Right-click a CDS document: *Open With* > *Text Editor*
- CDS Graphical Editor [page 12]
  View a graphical representation of the contents of a CDS source file, with the option to edit the source code as text with the syntax elements highlighted for easier visual scanning.
  Right-click a CDS document: *Open With* > *Graphical Editor*

CDS Text Editor

SAP Web IDE for SAP HANA includes a dedicated editor that you can use to define data-persistence objects using the CDS syntax. SAP HANA recognizes the `.hdbcds` file extension required for CDS object definitions and calls the appropriate repository plug-in. If you double-click a file with the `.hdbcds` extension in the *Project Explorer* view, SAP Web IDE for SAP HANA automatically displays the selected file in the CDS text editor.

The CDS editor provides the following features:

- Syntax highlights
  The CDS DDL editor supports syntax highlighting, for example, for keywords and any assigned values. To customize the colors and fonts used in the CDS text editor, choose *Tools* > *Preferences* > *Code Editor* > *Editor Appearance* and select a theme and font size.

  > **i Note**
  >
  > The CDS DDL editor automatically inserts the keyword *namespace* into any new DDL source file that you create using the *New* > *CDS Artifact* dialog.
  >
  > The following values are assumed:
  >
  > - *namespace* = `<ProjectName>.<ApplDBModuleName>`
  > - *context* = `<NewCDSFileName>`

- Keyword completion
  The editor displays a list of DDL suggestions that could be used to complete the keyword you start to enter. To change the settings, choose *Tools* > *Code Completion* in the toolbar menu.

- Code validity
  The CDS text editor provides syntax validation, which checks for parser errors as you type. Semantic errors are only shown when you build the XS advanced application module to which the CDS artifacts belong; the errors are shown in the console tab.
- Comments
  Text that appears after a double forward slash (`//`) or between a forward slash and an asterisk (`/*...*/`) is interpreted as a comment and highlighted in the CDS editor (for example, `//this is a comment`).

CDS Graphical Editor

The CDS graphical editor provides graphical modeling tools that help you to design and create database models using standard CDS artifacts with minimal or no coding at all. You can use the CDS graphical editor to create CDS artifacts such as entities, contexts, associations, structured types, and so on.

The built-in tools provided with the CDS Graphical Editor enable you to perform the following operations:

- Create CDS files (with the extension `.hdbcds`) using a file-creation wizard.
- Create standard CDS artifacts, for example: entities, contexts, associations (to internal and external entities), structured types, scalar types, ...
- Define technical configuration properties for entities, for example: indexes, partitions, and table groupings.
- Generate the relevant CDS source code in the text editor for the corresponding database model.
- Open in the CDS graphical editor data models that were created using the CDS text editor.

> → Tip
>
> The built-in tools included with the CDS Graphical Editor are context-sensitive; right-click an element displayed in the CDS Graphical editor to display the tool options that are available.

Related Information

Getting Started with the CDS Graphical Editor [page 282]

3.1.1 CDS Text Editor

The CDS text editor displays the source code of your CDS documents in a dedicated text-based editor.

SAP HANA studio includes a dedicated editor that you can use to define data-persistence objects using the CDS syntax. SAP HANA studio recognizes the `.hdbdd` file extension required for CDS object definitions and calls the appropriate repository plugin. If you double-click a file with the `.hdbdd` extension in the *Project Explorer* view, SAP HANA studio automatically displays the selected file in the CDS editor.

The CDS editor provides the following features:

- Syntax highlights

The CDS DDL editor supports syntax highlighting, for example, for keywords and any assigned values (`@Schema: 'MySchema'`). You can customize the colors and fonts used in the Eclipse Preferences (*Window* > *Preferences* > *General* > *Appearance* > *Colors and Fonts* > *CDS DDL*).

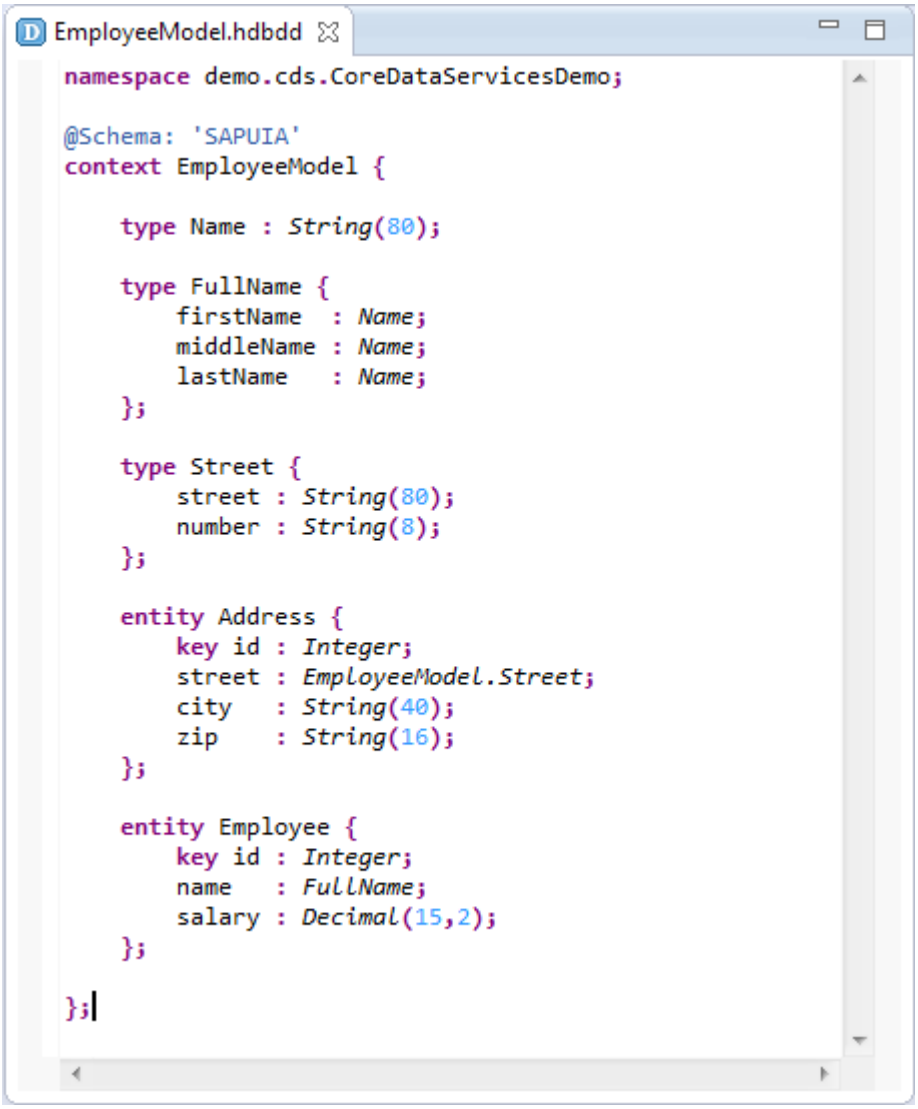
> **i Note**
>
> The CDS DDL editor automatically inserts the mandatory keyword *namespace* into any new DDL source file that you create using the *New DDL Source File* dialog. The following values are assumed:
>
> - *namespace* = `<repository package name>`

- Keyword completion
  The editor displays a list of DDL suggestions that could be used to complete the keyword you start to enter. You can insert any of the suggestions using the `SPACE` + `TAB` keys.
- Code validity
  You can check the validity of the syntax in your DDL source file before activating the changes in the SAP HANA repository. Right-click the file containing the syntax to check and use the *Team* > *Check* option in the context menu.

  > **i Note**
  >
  > Activating a file automatically commits the file first.

- Comments
  Text that appears after a double forward slash (`//`) or between a forward slash and an asterisk (`/*...*/`) is interpreted as a comment and highlighted in the CDS editor (for example, `//this is a comment`).

> **→ Tip**
>
> The *Project Explorer* view associates the `.hdbdd` file extension with the DDL icon. You can use this icon to determine which files contain CDS-compliant DDL code.



3.2 Create a CDS Document

A CDS document is a design-time source file that contains definitions of the objects you want to create in the SAP HANA catalog.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have already created a development workspace and a project.

- You must have shared a project for the CDS artifacts so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema for the CDS catalog objects created when the CDS document is activated in the repository, for example, `MYSCHEMA`
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.

Context

CDS documents are design-time source files that contain DDL code that describes a persistence model according to rules defined in Core Data Services. CDS documents have the file suffix `.hdbdd`. Activating the CDS document creates the corresponding catalog objects in the specified schema. To create a CDS document in the repository, perform the following steps:

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS document.

   Browse to the folder in your project workspace where you want to create the new CDS document and perform the following steps:

   a. Right-click the folder where you want to save the CDS document and choose *New* > *Other...* > *Database Development* > *DDL Source File* in the context-sensitive popup menu.

   b. Enter the name of the CDS document in the *File Name* box, for example, `MyModel`.

   > → Tip
   >
   > File extensions are important. If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically (for example, `MyModel.hdbdd`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

   c. Choose *Finish* to save the changes and commit the new CDS document to the repository.
   The file-creation wizard creates a basic CDS document with the following elements:
   - Namespace
     The name of the repository package in which you created the new CDS document, for example, `acme.com.hana.cds.data`
   - Top-level element
     The name of the top-level element in a CDS document must match the name of the CDS document itself; this is the name you enter when using the file-creation wizard to create the new CDS document, for example, `MyModel`, `MyContext`, or `MyEntity`. In this example, the top-level element is a context.

   ```
   namespace acme.com.hana.cds.data;
   ```

```
context MyModel {

};
```

5. Define the details of the CDS artifacts.

   Open the CDS document you created in the previous step, for example, `MyModel.hdbdd`, and add the CDS-definition code to the file. The CDS code describes the CDS artifacts you want to add, for example: entity definitions, type definitions, view definitions and so on:

   > **i Note**
   >
   > The following code examples are provided for illustration purposes only.

   a. Add a schema name.

      The `@Schema` annotation defines the name of the schema to use to store the artifacts that are generated when the CDS document is activated. The schema name must be inserted before the top-level element in the CDS document; in this example, the context `MyModel`.

      > **i Note**
      >
      > If the schema you specify does not exist, you cannot activate the new CDS document.

      ```
      namespace acme.com.hana.cds.data;
      @Schema: 'SAP_HANA_CDS'
      context MyModel {

      };
      ```

   b. Add structured types, if required.

      Use the `type` keyword to define a type artifact in a CDS document. In this example, you add the user-defined types and structured types to the top-level entry in the CDS document, the context `MyModel`.

      ```
      namespace acme.com.hana.cds.data;
      @Schema: 'SAP_HANA_CDS'
      context MyModel {
       type BusinessKey : String(10);
       type SString  : String(40);
       type <[...]>
      <[...]>
      };
      ```

   c. Add a new context, if required.

      Contexts enable you to group together related artifacts. A CDS document can only contain one top-level context, for example, `MyModel {};`. Any new context must be **nested** within the top-level entry in the CDS document, as illustrated in the following example.

      ```
      namespace acme.com.hana.cds.data;
      @Schema: 'SAP_HANA_CDS'
      context MyModel {
       type BusinessKey : String(10);
       type SString  : String(40);
       type <[...]>
       context MasterData {
       <[...]>
       };
       context Sales {
       <[...]>
       };
       context Purchases {
      ```

```
  <[...]>
  };
};
```

d. Add new entities.

You can add the entities either to the top-level entry in the CDS document; in this example, the context `MyModel` or to any other context, for example, `MasterData`, `Sales`, or `Purchases`. In this example, the new entities are column-based tables in the `MasterData` context.

```
namespace acme.com.hana.cds.data;
@Schema: 'SAP_HANA_CDS'
context MyModel {
 type BusinessKey : String(10);
 type SString  : String(40);
 type <[...]>
 context MasterData {
   @Catalog.tableType : #COLUMN
    Entity Addresses {
        key  AddressId: BusinessKey;
        City: SString;
        PostalCode: BusinessKey;
        <[...]>
    };
  @Catalog.tableType : #COLUMN
    Entity BusinessPartner {
        key  PartnerId: BusinessKey;
        PartnerRole: String(3);
        <[...]>
    };
 };
 context Sales {
 <[...]>
 };
 context Purchases {
 <[...]>
 };
};
```

6. Save the CDS document.

> **i Note**
>
> Saving a file in a shared project automatically commits the saved version of the file to the repository. You do not need to explicitly commit it again.

7. Activate the changes in the repository.
   a. Locate and right-click the new CDS document in the *Project Explorer* view.
   b. In the context-sensitive pop-up menu, choose *Team* > *Activate*.

> **i Note**
>
> If you cannot activate the new CDS document, check that the specified schema already exists and that there are no illegal characters in the name space, for example, the hyphen (-).

8. Ensure access to the schema where the new CDS catalog objects are created.

   After activation in the repository, a schema object is only visible in the catalog to the _SYS_REPO user. To enable other users, for example the schema owner, to view the newly created schema and the objects it contains, you must grant the user the required SELECT privilege for the schema object.

> **Note**
>
> If you already have the appropriate `SELECT` privilege for the schema, you do not need to perform this step.

a. In the SAP HANA studio *Systems* view, right-click the SAP HANA system hosting the repository where the schema was activated and choose *SQL Console* in the context-sensitive popup menu.
b. In the *SQL console*, execute the statement illustrated in the following example, where `<SCHEMANAME>` is the name of the newly activated schema, and `<username>` is the database user ID of the schema owner:

```
call
_SYS_REPO.GRANT_SCHEMA_PRIVILEGE_ON_ACTIVATED_CONTENT('select','<SCHEMANAME
>','<username>');
```

9. Check that a catalog objects has been successfully created for each of the artifacts defined in the CDS document.

   When a CDS document is activated, the activation process generates a corresponding catalog object where appropriate for the artifacts defined in the document; the location in the catalog is determined by the type of object generated.

> **Note**
>
> Non-generated catalog objects include: scalar types, structured types, and annotations.

a. In the *SAP HANA Development* perspective, open the *Systems* view.
b. Navigate to the catalog location where new object has been created, for example:

Catalog Object	Location
Entities	`<SID>` ▶ *Catalog* ▶ `<MYSCHEMA>` ▶ *Tables*
Types	`<SID>` ▶ *Catalog* ▶ `<MYSCHEMA>` ▶ *Procedures* ▶ *Table Types*

c. Open a data preview for the new object.

   Right-click the new object and choose *Open Data Preview* in the pop-up menu.

Related Information

CDS Namespaces [page 24]
CDS Naming Conventions [page 23]
CDS Contexts [page 25]
CDS Annotations [page 28]
CDS Comment Types [page 37]

3.2.1 CDS Documents

CDS documents are design-time source files that contain DDL code that describes a persistence model according to rules defined in Core Data Services.

CDS documents have the file suffix `.hdbdd`. Each CDS document must contain the following basic elements:

- A name space declaration
  The name space you define must be the first declaration in the CDS document and match the absolute package path to the location of the CDS document in the repository. It is possible to enclose parts of the name space in quotes (“”), for example, to solve the problem of illegal characters in name spaces.

  > **i Note**
  >
  > If you use the file-creation wizard to create a new CDS document, the name space is inserted automatically; the inserted name space reflects the repository location you select to create the new CDS document.

- A schema definition
  The schema you specify is used to store the catalog objects that are defined in the CDS document, for example: entities, structured types, and views. The objects are generated in the catalog when the CDS document is activated in the SAP HANA repository.
- CDS artifact definitions
  The objects that make up your persistence model, for example: contexts, entities, structured types, and views

Each CDS document must contain one top-level artifact, for example: a context, a type, an entity, or a view. The name of the top-level artifact in the CDS document must match the file name of the CDS document, without the suffix. For example, if the top-level artifact is a context named `MyModel`, the name of the CDS document must be `MyModel.hdbdd`.

> **i Note**
>
> On activation of a repository file in, the file suffix, for example, `.hdbdd`, is used to determine which runtime plug-in to call during the activation process. The plug-in reads the repository file selected for activation, in this case a CDS-compliant document, parses the object descriptions in the file, and creates the appropriate runtime objects in the catalog.

If you want to define multiple CDS artifacts within a single CDS document (for example, multiple types, structured types, and entities), the top-level artifact must be a context. A CDS document can contain multiple contexts and any number and type of artifacts. A context can also contain nested sub-contexts, each of which can also contain any number and type of artifacts.

When a CDS document is activated, the activation process generates a corresponding catalog object for each of the artifacts defined in the document; the location in the catalog is determined by the type of object generated. The following table shows the catalog location for objects generated by the activation of common CDS artifacts.

Catalog Location for CDS-generated Artifacts

CDS Artifact	Catalog Location
Entity	

CDS Artifact	Catalog Location
View	<SID> > *Catalog* > <MYSCHEMA> > *Views*
Structured type	<SID> > *Catalog* > <MYSCHEMA> > *Procedures* > *Table Types*

The following example shows the basic structure of a single CDS document that resides in the package `acme.com.hana.cds.data` in the SAP HANA repository. the CDS document defines the following CDS artifacts:

- Types:
  - `BusinessKey` and `SString`
- Entities:
  - `Addresses`, `BusinessPartners`, `Header`, and `Item`
- Contexts:
  - `MyModel`, which contains the nested contexts: `MasterData`, `Sales`, and `Purchases`
- External references
  The `using` keyword enables you to refer to artifacts defined in separate CDS documents, for example, `MyModelB.hdbdd`. You can also assign an alias to the reference, for example, `AS <alias>`.
- Annotations
  Built-in annotations, for example, *@Catalog*, *@Schema*, and *@nokey*, are important elements of the CDS syntax used to define CDS-compliant catalog objects. You can define your own custom annotations, too.

Note

The following code snippet is incomplete `[...]`; it is intended for illustration purposes only.

Sample Code

```
namespace acme.com.hana.cds.data;
using acme.com.hana.cds.data::MyModelB.MyContextB1 as ic;
@Schema: 'SAP_HANA_CDS'
context MyModel {
 type BusinessKey : String(10);
 type SString  : String(40);
 type <[...]>
 context MasterData {
  @Catalog.tableType : #COLUMN
    Entity Addresses {
        key  AddressId: BusinessKey;
        City: SString;
        PostalCode: BusinessKey;
        <[...]>
    };
  @Catalog.tableType : #COLUMN
    Entity BusinessPartner {
        key  PartnerId: BusinessKey;
        PartnerRole: String(3);
        <[...]>
    };
 };
 context Sales {
  @Catalog.tableType : #COLUMN
    Entity Header {
        key  SalesOrderId: BusinessKey;
        <[...]>
    };
 @Catalog.tableType : #COLUMN
```

```
  @MyAnnotation : 'foo'
     Entity Item {
         key  SalesOrderId: BusinessKey;
         key  SalesOrderItem: BusinessKey;
         <[...]>
     };
  };
  context Purchases {
  <[...]>
  };
};
```

Related Information

Create a CDS Document [page 14]
CDS Namespaces [page 24]
CDS Annotations [page 28]
External Artifacts in CDS [page 21]

3.2.2 External Artifacts in CDS

You can define an artifact in one CDS document by referring to an artifact that is defined in another CDS document.

The CDS syntax enables you to define a CDS artifact in one document by basing it on an “external” artifact - an artifact that is defined in a separate CDS document. Each external artifact must be explicitly declared in the source CDS document with the `using` keyword, which specifies the location of the external artifact, its name, and where appropriate its CDS context.

> → Tip
>
> The `using` declarations must be located in the header of the CDS document between the `namespace` declaration and the beginning of the top-level artifact, for example, the `context`.

The external artifact can be either a single object (for example, a type, an entity, or a view) or a context. You can also include an optional alias in the `using` declaration, for example, `ContextA.ContextA1 as ic`. The alias (`ic`) can then be used in subsequent type definitions in the source CDS document.

```
//Filename = Pack1/Distributed/ContextB.hdbdd
namespace Pack1.Distributed;
using Pack1.Distributed::ContextA.T1;
using Pack1.Distributed::ContextA.ContextAI as ic;
using Pack1.Distributed::ContextA.ContextAI.T3 as ict3;
using Pack1.Distributed::ContextA.ContextAI.T3.a as a;  // error, is not an
artifact
context ContextB {
  type T10 {
    a : T1;               // Integer
    b : ic.T2;            // String(20)
    c : ic.T3;            // structured
    d : type of ic.T3.b;  // String(88)
```

```
    e : ict3;              // structured
    x : Pack1.Distributed::ContextA.T1;  // error, direct reference not allowed
  };
  context ContextBI {
    type T1 : String(7);  // hides the T1 coming from the first using declaration
    type T2 : T1;         // String(7)
  };
};
```

The CDS document `ContextB.hdbdd` shown above uses external artifacts (data types `T1` and `T3`) that are defined in the "target" CDS document `ContextA.hdbdd` shown below. Two `using` declarations are present in the CDS document `ContextB.hdbdd`; one with no alias and one with an explicitly specified alias (`ic`). The first `using` declaration introduces the scalar type `Pack1.Distributed::ContextA.T1`. The second `using` declaration introduces the context `Pack1.Distributed::ContextA.ContextAI` and makes it accessible by means of the explicitly specified alias `ic`.

> **i Note**
>
> If no explicit alias is specified, the last part of the fully qualified name is assumed as the alias, for example `T1`.

The `using` keyword is the only way to refer to an externally defined artifact in CDS. In the example above, the type `x` would cause an activation error; you cannot refer to an externally defined CDS artifact directly by using its fully qualified name in an artifact definition.

```
//Filename = Pack1/Distributed/ContextA.hdbdd
namespace Pack1.Distributed;
context ContextA {
  type T1 : Integer;
  context ContextAI {
    type T2 : String(20);
    type T3 {
      a : Integer;
      b : String(88);
    };
  };
};
```

> **i Note**
>
> Whether you use a single or multiple CDS documents to define your data-persistence model, each CDS document must contain only **one** top-level artifact, and the name of the top-level artifact must correspond to the name of the CDS document. For example, if the top-level artifact in a CDS document is `ContextA`, then the CDS document itself must be named `ContextA.hdbdd`.

3.2.3 CDS Naming Conventions

Rules and restrictions apply to the names of CDS documents and the package in which the CDS document resides.

The rules that apply for naming CDS documents are the same as the rules for naming the packages in which the CDS document is located. When specifying the name of a package or a CDS document (or referencing the name of an existing CDS object, for example, within a CDS document), bear in mind the following rules:

- CDS source-file name
  From SAP HANA 2.0 SPS 01, it is possible to define **multiple** top-level artifacts (for example, contexts, entities, etc.) in a single CDS document. For this reason, you can choose any name for the CDS source file; there is no longer any requirement that the name of the CDS source file must be the same as the name of a top-level artifact.
- File suffix
  The file suffix differs according to SAP HANA XS version:
  - XS classic
    `.hdbdd`, for example, `MyModel.hdbdd`.
  - XS advanced
    `.hdbcds`, for example, `MyModel.hdbcds`.
- Permitted characters
  CDS object and package names can include the following characters:
  - Lower or upper case letters (aA-zZ) and the underscore character (_)
  - Digits (0-9)
- Forbidden characters
  The following restrictions apply to the characters you can use (and their position) in the name of a CDS document or a package:
  - You cannot use either the hyphen (-) or the dot (.) in the name of a CDS document.
  - You cannot use a digit (0-9) as the first character of the name of either a CDS document or a package, for example, `2CDSobjectname.hdbdd` (XS classic) or `acme.com.1package.hdbcds` (XS advanced).
  - The CDS parser does not recognize either CDS document names or package names that consist **exclusively** of digits, for example, `1234.hdbdd` (XS classic) or `acme.com.999.hdbcds` (XS advanced).

> ⚠ Caution
>
> Although it is possible to use quotation marks (“”) to wrap a name that includes forbidden characters, as a general rule, it is recommended to follow the naming conventions for CDS documents specified here in order to avoid problems during activation in the repository.

Related Information

Create a CDS Document [page 14]
CDS Documents [page 19]
CDS Namespaces [page 24]

3.2.4 CDS Namespaces

The namespace is the path to the package in the SAP HANA Repository that contains CDS artifacts such as entities, contexts, and views.

In a CDS document, the first statement must declare the namespace that contains the CDS elements which the document defines, for example: a context, a type, an entity, or a view. The namespace must match the package name where the CDS elements specified in the CDS document are located. If the package path specified in a namespace declaration does not already exist in the SAP HANA Repository, the activation process for the elements specified in the CDS document fails.

It is possible to enclose in quotation marks (“”) individual parts of the namespace identifier, for example, `"Pack1".pack2`. Quotes enable the use of characters that are not allowed in regular CDS identifiers; in CDS, a quoted identifier can include all characters except the dot (.) and the double colon (::). If you need to use a reserved keyword as an identifier, you **must** enclose it in quotes, for example, “`Entity`”. However, it is recommended to avoid the use of reserved keywords as identifiers.

> **i** Note
>
> You can also use quotation marks (“”) to wrap the names of CDS artifacts (entities, views) and elements (columns...).

The following code snippet applies to artifacts created in the Repository package `/Pack1/pack2/` and shows some examples of **valid** namespace declarations, including namespaces that use quotation marks (“”).

> **i** Note
>
> A CDS document cannot contain more than one namespace declaration.

```
namespace Pack1.pack2;
namespace "Pack1".pack2;
namespace Pack1."pack2";
namespace "Pack1"."pack2";
```

The following code snippet applies to artifacts created in the Repository package `/Pack1/pack2/` and shows some examples of **invalid** namespace declarations.

```
namespace pack1.pack2;              // wrong spelling
namespace "Pack1.pack2";            // incorrect use of quotes
namespace Pack1.pack2.MyDataModel;  // CDS file name not allowed in namespace
namespace Jack.Jill;                // package does not exist
```

The examples of namespace declarations in the code snippet above are invalid for the following reasons:

- `pack1.pack2;`
  `pack1` is spelled incorrectly; the namespace element requires a capital P to match the corresponding location in the Repository, for example, `Pack1`.
- `"Pack1.pack2";`
  You cannot quote the entire namespace path; only individual elements of the namespace path can be quoted, for example, `"Pack1".pack2;` or `Pack1."pack2";`.
- `Pack1.pack2.MyDataModel;`
  The namespace declaration must not include the names of elements specified in the CDS document itself, for example, `MyDataModel`.

- `Jack.Jill;`
  The package path `Jack.Jill;` does not exist in the Repository.

Related Information

Create a CDS Document [page 14]
CDS Documents [page 19]

3.2.5 CDS Contexts

You can define multiple CDS-compliant *entities* (tables) in a single file by assigning them to a *context*.

The following example illustrates how to assign two simple entities to a context using the CDS-compliant `.hdbdd` syntax; you store the context-definition file with a specific name and the file extension `.hdbdd`, for example, `MyContext.hdbdd`.

> **i Note**
>
> If you are using a CDS document to define a CDS context, the name of the CDS document must match the name of the context defined in the CDS document, for example, with the "`context`" keyword.

In the example below, you must save the context definition "Books" in the CDS document `Books.hdbdd`. In addition, the name space declared in a CDS document must match the repository package in which the object the document defines is located.

The following code example illustrates how to use the CDS syntax to define multiple design-time entities in a context named `Books`.

```
namespace com.acme.myapp1;
@Schema : 'MYSCHEMA'
context Books {
  @Catalog.tableType: #COLUMN
  @Catalog.index : [ { name : 'MYINDEX1', unique : true, order : #DESC,
elementNames : ['ISBN'] }  ]
  entity Book {
    key AuthorID  : String(10);
    key BookTitle : String(100);
        ISBN      : Integer  not null;
        Publisher : String(100);
    };
  @Catalog.tableType: #COLUMN
  @Catalog.index : [ { name: 'MYINDEX2', unique: true, order: #DESC,
elementNames: ['AuthorNationality'] }  ]
  entity Author {
    key AuthorName        : String(100);
        AuthorNationality : String(20);
        AuthorBirthday    : String(100);
        AuthorAddress     : String(100);
    };
};
```

Activation of the file `Books.hdbdd` containing the context and entity definitions creates the catalog objects "Book" and "Author".

> **i Note**
>
> The namespace specified at the start of the file, for example, `com.acme.myapp1` corresponds to the location of the entity definition file (`Books.hdbdd`) in the application-package hierarchy .

Nested Contexts

The following code example shows you how to define a nested context called `InnerCtx` in the parent context `MyContext`. The example also shows the syntax required when making a reference to a user-defined data type in the nested context, for example, (`field6 : type of InnerCtx.CtxType.b;`).

The `type of` keyword is only required if referencing an element in an entity or in a structured type; types in another context can be referenced directly, without the `type of` keyword. The nesting depth for CDS contexts is restricted by the limits imposed on the length of the database identifier for the name of the corresponding SAP HANA database artifact (for example, table, view, or type); this is currently limited to 126 characters (including delimiters).

> **i Note**
>
> The context itself does not have a corresponding artifact in the SAP HANA catalog; the context only influences the names of SAP HANA catalog artifacts that are generated from the artifacts defined in a given CDS context, for example, a table or a structured type.

```
namespace com.acme.myapp1;
@Schema: 'MySchema'
context MyContext {
// Nested contexts
   context InnerCtx {

      Entity MyEntity {
         …
      };
      Type CtxType {
        a : Integer;
        b : String(59);
      };
   };
    type MyType1 {
      field1 : Integer;
      field2 : String(40);
      field3 : Decimal(22,11);
      field4 : Binary(11);
    };

    type MyType2 {
      field1 : String(50);
      field2 : MyType1;
    };

    type MyType3 {
      field1 : UTCTimestamp;
      field2 : MyType2;
    };


    @Catalog.index : [{ name : 'IndexA', order : #ASC, unique: true,
                        elementNames : ['field1'] }]
```

```
    entity MyEntity1 {
      key id  : Integer;
      field1  : MyType3 not null;
      field2  : String(24);
      field3  : LocalDate;
      field4  : type of field3;
      field5  : type of MyType1.field2;
      field6  : type of InnerCtx.CtxType.b;  // refers to nested context
      field7  : InnerCtx.CtxType;           //  more context references
    };
};
```

Name Resolution Rules

The sequence of definitions inside a block of CDS code (for example, `entity` or `context`) does not matter for the scope rules; a binding of an artifact type and name is valid within the confines of the smallest block of code containing the definition, except in inner code blocks where a binding for the same identifier remains valid. This rules means that the definition of `nameX` in an inner block of code hides any definitions of `nameX` in outer code blocks.

> **i Note**
>
> An identifier may be used before its definition without the need for forward declarations.

```
context OuterCtx
{
  type MyType1 : Integer;
  type MyType2 : LocalDate;
  context InnerCtx
  {
    type Use1 : MyType1;           // is a String(20)
    type Use2 : MyType2;           // is a LocalDate
    type MyType1 : String(20);
  };
   type invalidUse : Use1;       // invalid: Use1 is not
                                 // visible outside of InnerCtx
   type validUse   : InnerCtx.Use1;   // ok
};
```

No two artifacts (including namespaces) can be defined whose absolute names are the same or are different only in case (for example, `MyArtifact` and `myartifact`), even if their artifact type is different (entity and view). When searching for artifacts, CDS makes no assumptions which artifact kinds can be expected at certain source positions; it simply searches for the artifact with the given name and performs a final check of the artifact type.

The following example demonstrates how name resolution works with multiple nested contexts, Inside context `NameB`, the local definition of `NameA` shadows the definition of the context `NameA` in the surrounding scope. This means that the definition of the identifier `NameA` is resolved to `Integer`, which does not have a sub-component `T1`. The result is an error, and the compiler does not continue the search for a “better” definition of `NameA` in the scope of an outer (parent) context.

```
context OuterCtx
{
  context NameA
  {
    type T1 : Integer;
```

```
    type T2 : String(20);
  };
  context NameB
  {
    type NameA : Integer;
    type Use   : NameA.T1;  // invalid: NameA is an Integer
    type Use2  : OuterCtx.NameA.T2;  // ok
  };
};
```

Related Information

CDS User-Defined Data Types [page 64]
Create a CDS Document [page 14]

3.2.6 CDS Annotations

CDS supports built-in annotations, for example, *@Catalog*, *@Schema*, and *@nokey*, which are important elements of the CDS documents used to define CDS-compliant catalog objects. However, you can define your own custom annotations, too.

Example

```
namespace mycompany.myapp1;
@Schema : 'MYSCHEMA'
context Books {
  @Catalog.tableType: #COLUMN
  @Catalog.index: [ { name : 'MYINDEX1', unique : true, order : #DESC,
elementNames : ['ISBN'] } ]
  entity BOOK {
    key Author : String(100);
    key BookTitle : String(100);
        ISBN : Integer not null;
        Publisher : String(100);
  };
  @Catalog.tableType : #COLUMN
  @nokey
  entity MyKeylessEntity
  {
    element1 : Integer;
    element2 : UTCTimestamp;
    @SearchIndex.text: { enabled: true }
    element3 : String(7);
  };
  @GenerateTableType : false
  Type MyType1 {
    field1 : Integer;
    field2 : Integer;
    field3 : Integer;
  };
};
```

Overview

The following list indicates the annotations you can use in a CDS document:

- `@Catalog`
- `@nokey`
- `@Schema`
- `@GenerateTableType`
- `@SearchIndex`
- `@WithStructuredPrivilegeCheck`

@Catalog

The *@Catalog* annotation supports the following parameters, each of which is described in detail in a dedicated section below:

- *@Catalog.index*
  Specify the type and scope of index to be created for the CDS entity, for example: name, order, unique/ non-unique
- *@Catalog.tableType*
  Specify the table type for the CDS entity, for example, column, row, global temporary.

You use the *@Catalog.index* annotation to define an index for a CDS entity. The *@Catalog.index* annotation used in the following code example ensures that an index called `Index1` is created for the entity `MyEntity1` along with the index fields `fint` and `futcshrt`. The order for the index is ascending (`#ASC`) and the index is unique.

```
namespace com.acme.myapp1;
@Catalog.tableType : #COLUMN
@Schema: 'MYSCHEMA'
@Catalog.index:[ { name:'Index1', unique:true, order:#ASC, elementNames:['fint',
'futcshrt' ] } ]
entity MyEntity1 {
    key fint:Integer;
    fstr    :String(5000);
    fstr15  :String(51);
    fbin    :Binary(4000);
    fbin15  :Binary(51);
    fint32  :Integer64;
    fdec53  :Decimal(5,3);
    fdecf   :DecimalFloat;
    fbinf   :BinaryFloat;
    futcshrt:UTCDateTime not null;
    flstr   :LargeString;
    flbin   :LargeBinary;
};
```

You can define the following values for the *@Catalog.index* annotation:

- `elementNames : ['<name1>', '<name2>' ]`
  The names of the fields to use in the index; the elements are specified for the entity definition, for example, `elementNames:['fint', 'futcshrt' ]`
- `name : '<IndexName>'`
  The names of the index to be generated for the specified entity, for example, `name:'myIndex'`

- `order`
  Create a table index sorted in ascending or descending order. The order keywords `#ASC` and `#DESC` can be only used in the **BTREE** index (for the maintenance of sorted data) and can be specified only once for each index.
  - `order : #ASC`
    Creates an index for the CDS entity and sorts the index fields in **ascending** logical order, for example: 1, 2, 3...
  - `order : #DESC`
    Creates a index for the CDS entity and sorts the index fields in **descending** logical order, for example: 3, 2, 1...
- `unique`
  Creates a unique index for the CDS entity. In a unique index, two rows of data in a table cannot have identical key values.
  - `unique : true`
    Creates a unique index for the CDS entity. The uniqueness is checked and, if necessary, enforced each time a key is added to (or changed in) the index.
  - `unique : false`
    Creates a non-unique index for the CDS entity. A non-unique index is intended primarily to improve query performance, for example, by maintaining a sorted order of values for data that is queried frequently.

You use the *@Catalog.tableType* annotation to define the type of CDS entity you want to create. The *@Catalog.tableType* annotation determines the storage engine in which the underlying table is created.

```
namespace com.acme.myapp1;
@Schema: 'MYSCHEMA'
context MyContext1 {
    @Catalog.tableType : #COLUMN
    entity MyEntity1 {
        key ID : Integer;
        name : String(30);
    };
    @Catalog.tableType : #ROW
    entity MyEntity2 {
        key ID : Integer;
        name : String(30);
    };
    @Catalog.tableType : #GLOBAL_TEMPORARY
    entity MyEntity3 {
        ID : Integer;
        name : String(30);
    };
};
```

You can define the following values for the *@Catalog.tableType* annotation:

- `#COLUMN`
  Create a column-based table. If the majority of table access is through a large number of tuples, with only a few selected attributes, use COLUMN-based storage for your table type.
- `#ROW`
  Create a row-based table. If the majority of table access involves selecting a few records, with all attributes selected, use ROW-based storage for your table type.
- `#GLOBAL_TEMPORARY`
  Set the scope of the created table. Data in a **global** temporary table is session-specific; only the owner session of the global temporary table is allowed to insert/read/truncate the data. A global temporary table

exists for the duration of the session, and data from the global temporary table is automatically dropped when the session is terminated. A global temporary table can be dropped only when the table does not have any records in it.

> **i** Note
>
> The SAP HANA database uses a combination of table types to enable storage and interpretation in both ROW and COLUMN forms. If no table type is specified in the CDS entity definition, the default value *#COLUMN* is applied to the table created on activation of the design-time entity definition.

@nokey

An entity usually has one or more key elements, which are flagged in the CDS entity definition with the *key* keyword. The key elements become the primary key of the generated SAP HANA table and are automatically flagged as "not null". Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

> **i** Note

However, you can also define an entity that has no key elements. If you want to define an entity without a key, use the *@nokey* annotation. In the following code example, the *@nokey* annotation ensures that the entity `MyKeylessEntity` defined in the CDS document creates a column-based table where no key element is defined.

```
namespace com.acme.myapp1;
@Schema: 'MYSCHEMA'
@Catalog.tableType : #COLUMN
@nokey
entity MyKeylessEntity
{
  element1 : Integer;
  element2 : UTCTimestamp;
  element3 : String(7);
};
```

@Schema

The `@Schema` annotation is only allowed as a top-level definition in a CDS document. In the following code example `@Schema` ensures that the schema `MYSCHEMA` is used to contain the entity `MyEntity1`, a column-based table.

```
namespace com.acme.myapp1;
@Schema: 'MYSCHEMA'
@Catalog.tableType : #COLUMN
entity MyEntity1 {
    key ID : Integer;
    name : String(30);
};
```

> **i** Note
>
> If the schema specified with the @Schema annotation does not already exist, an activation error is displayed and the entity-creation process fails.

The schema name must adhere to the SAP HANA rules for database identifiers. In addition, a schema name must not start with the letters `SAP*`; the SAP* namespace is reserved for schemas used by SAP products and applications.

@GenerateTableType

For each structured type defined in a CDS document, an SAP HANA table type is generated, whose name is built by concatenating the elements of the CDS document containing the structured-type definition and separating the elements by a dot delimiter (.). The new SAP HANA table types are generated in the schema that is specified in the schema annotation of the respective top-level artifact in the CDS document containing the structured types.

> **i** Note
>
> Table types are only generated for **direct** structure definitions; no table types are generated for **derived** types that are based on structured types.

If you want to use the structured types inside a CDS document **without** generating table types in the catalog, use the annotation `@GenerateTableType : false`.

@SearchIndex

The annotation `@SearchIndex` enables you to define which of the columns should be indexed for search capabilities, for example, `{enabled : true}`. To extend the index search definition, you can use the properties `text` or `fuzzy` to specify if the index should support text-based or fuzzy search, as illustrated in the following example:

```
entity MyEntity100
  {
    element1 : Integer;
    @SearchIndex.text: { enabled: true }
    element2 : LargeString;
    @SearchIndex.fuzzy: { enabled: true }
    element3 : String(7);
  };
```

> **→** Tip
>
> For more information about setting up search features and using the search capability, see the *SAP HANA Search Developer Guide* .

@WithStructuredPrivilegeCheck

The annotation `@WithStructuredPrivilegeCheck` enables you to control access to data (for example, in a view) by means of privileges defined with the Data Control Language (DCL), as illustrated in the following example:

```
@WithStructuredPrivilegeCheck
view MyView as select from Foo {
  <select_list>
} <where_groupBy_Having_OrderBy>;
```

Related Information

Create a CDS Document [page 14]
User-Defined CDS Annotations [page 33]
CDS Structured Type Definition [page 67]

3.2.6.1 User-Defined CDS Annotations

In CDS, you can define your own custom annotations.

The built-in **core** annotations that SAP HANA provides, for example, `@Schema`, `@Catalog`, or `@nokey`, are located in the namespace `sap.cds`; the same namespace is used to store all the primitive types, for example, `sap.cds::integer` and `sap.cds::SMALLINT`.

However, the CDS syntax also enables you to define your own annotations, which you can use in addition to the existing "core" annotations. The rules for defining a custom annotation in CDS are very similar way the rules that govern the definition of a user-defined type. In CDS, an annotation can be defined either inside a CDS context or as the single, top-level artifact in a CDS document. The custom annotation you define can then be assigned to other artifacts in a CDS document, in the same way as the core annotations, as illustrated in the following example:

```
@Catalog.tableType : #ROW
@MyAnnotation : 'foo'
entity MyEntity {
       key Author : String(100);
       key BookTitle : String(100);
       ISBN : Integer not null;
       Publisher : String(100);
}
```

CDS supports the following types of user-defined annotations:

- Scalar annotations
- Structured annotations
- Annotation arrays

Scalar Annotations

The following example shows how to define a scalar annotation.

```
annotation MyAnnotation_1 : Integer;
annotation MyAnnotation_2 : String(20);
```

In annotation definitions, you can use both the **enumeration** type and the **Boolean** type, as illustrated in the following example.

```
type Color : String(10) enum { red = 'rot'; green = 'grün'; blue = 'blau'; };
annotation MyAnnotation_3 : Color;
annotation MyAnnotation_4 : Boolean;
```

Structured Annotations

The following example shows how to define a structured annotation.

```
annotation MyAnnotation_5 {
  a : Integer;
  b : String(20);
  c : Color;
  d : Boolean;
};
```

The following example shows how to nest annotations in an anonymous annotation structure.

```
annotation MyAnnotation_7 {
   a : Integer;
   b : String(20);
   c : Color;
   d : Boolean;
   s {
       a1 : Integer;
       b1 : String(20);
       c1 : Color;
       d1 : Boolean;
   };
};
```

Array Annotations

The following example shows how to define an array-like annotation.

```
annotation MyAnnotation_8 : array of Integer;
annotation MyAnnotation_9 : array of String(12);
annotation MyAnnotation_10 : array of { a: Integer; b: String(10); };
```

3.2.6.2 CDS Annotation Usage Examples

Reference examples of the use of user-defined CDS annotations.

When you have defined an annotation, the user-defined annotation can be used to annotate other definitions. It is possible to use the following types of user-defined annotations in a CDS document:

User-defined CDS Annotations

CDS Annotation Type	Description
Scalar annotations [page 35]	For use with simple integer or string annotations and enumeration or Boolean types
Structured annotations [page 36]	For use where you need to create a simple annotation structure or nest an annotation in an anonymous annotation structure
Annotation arrays [page 36]	For use where you need to assign the same annotation several times to the same object.

Scalar Annotations

The following examples show how to use a scalar annotation:

```
@MyAnnotation_1 : 18
type MyType1 : Integer;
@MyAnnotation_2 : 'sun'
@MyAnnotation_1 : 77
type MyType2 : Integer;
@MyAnnotation_2 : 'sun'
@MyAnnotation_2 : 'moon'   // error: assigning the same annotation twice is not
allowed.
type MyType3 : Integer;
```

> **i Note**
>
> It is not allowed to assign an annotation to the same object more than once. If several values of the same type are to be annotated to a single object, use an array-like annotation.

For annotations that have enumeration type, the `enum` values can be addressed either by means of their fully qualified name, or by means of the shortcut notation (using the hash (#) sign. It is not allowed to use a literal value, even if it matches a literal of the `enum` definition.

```
@MyAnnotation_3 : #red
type MyType4 : Integer;
@MyAnnotation_3 : Color.red
type MyType5 : Integer;
@MyAnnotation_3 : 'rot' // error: no literals allowed, use enum symbols
type MyType6 : Integer;
```

For Boolean annotations, only the values "`true`" or "`false`" are allowed, and a shortcut notation is available for the value "`true`", as illustrated in the following examples:

```
@MyAnnotation_4 : true
type MyType7 : Integer;
@MyAnnotation_4 // same as explicitly assigning the value "true"
```

```
type MyType8 : Integer;
@MyAnnotation_4 : false
type MyType9 : Integer;
```

Structured Annotations

Structured annotations can be assigned either as a complete unit or, alternatively, one element at a time. The following example show how to assign a **whole** structured annotation:

```
@MyAnnotation_5 : { a : 12, b : 'Jupiter', c : #blue, d : false }
type MyType10 : Integer;
@MyAnnotation_5 : { c : #green }  // not all elements need to be filled
type MyType11 : Integer;
```

The following example shows how to assign the same structured annotation element by element.

> **i** Note
>
> It is not necessary to assign a value for each element.

```
@MyAnnotation_5.a : 12
@MyAnnotation_5.b : 'Jupiter'
@MyAnnotation_5.c : #blue
@MyAnnotation_5.d : false
type MyType12 : Integer;
@MyAnnotation_5.c : #green
type MyType13 : Integer;
@MyAnnotation_5.c : #blue
@MyAnnotation_5.d        // shortcut notation for Boolean (true)
type MyType14 : Integer;
```

It is not permitted to assign the same annotation element more than once; assigning the same annotation element more than once in a structured annotation causes an activation error.

```
@MyAnnotation_5 : { c : #green, c : #green }  // error, assign an element once
only
type MyType15 : Integer;
@MyAnnotation_5.c : #green
@MyAnnotation_5.c : #blue  // error, assign an element once only
type MyType16 : Integer;
```

Array-like Annotations

Although it is not allowed to assign the same annotation several times to the same object, you can achieve the same effect with an array-like annotation, as illustrated in the following example:

```
@MyAnnotation_8 : [1,3,5,7]
type MyType30 : Integer;
@MyAnnotation_9 : ['Earth', 'Moon']
type MyType31 : Integer;
@MyAnnotation_10 : [{ a: 52, b: 'Mercury'}, { a: 53, b: 'Venus'}]
type MyType32 : Integer;
```

Related Information

CDS Annotations [page 28]
CDS Documents [page 19]
Create a CDS Document [page 14]

3.2.7 CDS Comment Types

The Core Data Services (CDS) syntax enables you to insert comments into object definitions.

Example: Comment Formats in CDS Object Definitions

```
namespace com.acme.myapp1;

  /**
   *  multi-line comment,
   *  for doxygen-style,
   *  comments and annotations
   */
  type Type1 {
       element Fstr:       String( 5000 ); // end-of-line comment
               Flstr:      LargeString;
       /*inline comment*/  Fbin:       Binary( 4000 );
       element Flbin:      LargeBinary;
               Fint:       Integer;
       element Fint64:     Integer64;
               Ffixdec:    Decimal( 34, 34 /* another inline comment */);
       element Fdec:       DecimalFloat;
               Fflt:       BinaryFloat;
       //complete line comment element Flocdat:    LocalDate;    LocalDate
temporarily switched off
       //complete line comment        Floctim:    LocalTime;
       element Futcdatim:  UTCDateTime;
               Futctstmp:  UTCTimestamp;
  };
```

Overview

You can use the forward slash (`/`) and the asterisk (`*`) characters to add comments and general information to CDS object-definition files. The following types of comment are allowed:

- In-line comment
- End-of-line comment
- Complete-line comment
- Multi-line comment

In-line Comments

The in-line comment enables you to insert a comment into the middle of a line of code in a CDS document. To indicate the start of the in-line comment, insert a forward-slash (`/`) followed by an asterisk (`*`) before the comment text. To signal the end of the in-line comment, insert an asterisk followed by a forward-slash character (`*/`) after the comment text, as illustrated by the following example:.

```
element Flocdat:  /*comment text*/  LocalDate;
```

End-of-Line Comment

The end-of-line comment enables you to insert a comment at the end of a line of code in a CDS document. To indicate the start of the end-of-line comment, insert two forward slashes (`//`) before the comment text, as illustrated by the following example:.

```
element Flocdat:    LocalDate; // Comment text
```

Complete-Line Comment

The complete-line comment enables you to tell the parser to ignore the contents of an entire line of CDS code. The comment out a complete line, insert two forward slashes (`//`) at the start of the line, as illustrated in the following example:

```
//  element Flocdat:    LocalDate;      Additional comment text
```

Multi-Line Comments

The multi-line comment enables you to insert comment text that extends over multiple lines of a CDS document. To indicate the start of the multi-line comment, insert a forward-slash (`/`) followed by an asterisk (`*`) at the start of the group of lines you want to use for an extended comment (for example, `/*`). To signal the end of the multi-line comment, insert an asterisk followed by a forward-slash character (`*/`). Each line between the start and end of the multi-line comment must start with an asterisk (`*`), as illustrated in the following example:

```
/*
*  multiline,
*  doxygen-style
*  comments and annotations
*/
```

Related Information

Create a CDS Document [page 14]

3.3 Create an Entity in CDS

The **entity** is the core artifact for persistence-model definition using the CDS syntax. You create a database entity as a design-time file in the SAP HANA repository.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have already created a development workspace and a project.
- You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema for the CDS catalog objects, for example, `MYSCHEMA`
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.

Context

In the SAP HANA database, as in other relational databases, a CDS entity is a table with a set of data elements that are organized using columns and rows. SAP HANA Extended Application Services (SAP HANA XS) enables you to use the CDS syntax to create a database entity as a design-time file in the repository. Activating the CDS entity creates the corresponding table in the specified schema. To create a CDS entity-definition file in the repository, perform the following steps:

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS entity-definition file.

   Browse to the folder in your project workspace where you want to create the new CDS entity-definition file and perform the following steps:

a. Right-click the folder where you want to save the entity-definition file and choose *New* > *Other...* > *Database Development* > *DDL Source File* in the context-sensitive popup menu.
b. Enter the name of the entity-definition file in the *File Name* box, for example, `MyEntity`.

> → Tip
>
> File extensions are important. If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically (for example, `MyEntity.hdbdd`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

c. Choose *Finish* to save the changes and commit the new entity-definition file in the repository.

5. Define the structure of the CDS entity.

   If the new entity-definition file is not automatically displayed by the file-creation wizard, in the *Project Explorer* view double-click the entity-definition file you created in the previous step, for example, `MyEntity.hdbdd`, and add the catalog- and entity-definition code to the file:

> i Note
>
> The following code example is provided for illustration purposes only. If the schema you specify does not exist, you cannot activate the new CDS entity.

```
namespace acme.com.apps.myapp1;
@Schema : 'MYSCHEMA'
@Catalog.tableType : #COLUMN
@Catalog.index : [ { name : 'MYINDEX1', unique : true, order :#DESC,
elementNames : ['ISBN'] } ]
entity MyEntity {
  key Author    : String(100);
  key BookTitle : String(100);
      ISBN      : Integer not null;
      Publisher : String(100);
};
```

6. Save the CDS entity-definition file.

> i Note
>
> Saving a file in a shared project automatically commits the saved version of the file to the repository. You do not need to explicitly commit it again.

7. Activate the changes in the repository.
   a. Locate and right-click the new CDS entity-definition file in the *Project Explorer* view.
   b. In the context-sensitive pop-up menu, choose *Team* > *Activate*.

> i Note
>
> If you cannot activate the new CDS artifact, check that the specified schema already exists and that there are no illegal characters in the name space, for example, the hyphen (-).

8. Ensure access to the schema where the new CDS catalog objects are created.

   After activation in the repository, a schema object is only visible in the catalog to the _SYS_REPO user. To enable other users, for example the schema owner, to view the newly created schema and the objects it contains, you must grant the user the required SELECT privilege for the appropriate schema object.

> **i Note**
>
> If you already have the appropriate SELECT privilege, you do not need to perform this step.

a. In the SAP HANA studio *Systems* view, right-click the SAP HANA system hosting the repository where the schema was activated and choose *SQL Console* in the context-sensitive popup menu.
b. In the *SQL console*, execute the statement illustrated in the following example, where `<SCHEMANAME>` is the name of the newly activated schema, and `<username>` is the database user ID of the schema owner:

```
call
_SYS_REPO.GRANT_SCHEMA_PRIVILEGE_ON_ACTIVATED_CONTENT('select','<SCHEMANAME
>','<username>');
```

9. Check that the new entity has been successfully created.

   CDS entities are created in the `Tables` folder in the catalog.

   a. In the *SAP HANA Development* perspective, open the *Systems* view.
   b. Navigate to the catalog location where you created the new entity.

      `<SID>` > *Catalog* > `<MYSCHEMA>` > *Tables*
   c. Open a data preview for the new entity `MyEntity`.

      Right-click the new entity `<package.path>::MyEntity` and choose *Open Data Preview* in the pop-up menu.

      > **→ Tip**
      >
      > Alternatively, to open the table-definition view of the SAP HANA catalog tools, press `F3` when the CDS entity is in focus in the CDS editor.

Related Information

CDS Entities [page 41]
Entity Element Modifiers [page 43]
CDS Entity Syntax Options [page 49]

3.3.1 CDS Entities

In the SAP HANA database, as in other relational databases, a CDS entity is a table with a set of data elements that are organized using columns and rows.

A CDS entity has a specified number of columns, defined at the time of entity creation, but can have any number of rows. Database entities also typically have meta-data associated with them; the meta-data might include constraints on the entity or on the values within particular columns. SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database entity as a design-time file in the repository. All repository files including your entity definition can be transported to other SAP HANA systems, for example, in a delivery unit. You can define the entity using CDS-compliant DDL.

> **i Note**
>
> A delivery unit is the medium SAP HANA provides to enable you to assemble all your application-related repository artifacts together into an archive that can be easily exported to other systems.

The following code illustrates an example of a single design-time entity definition using CDS-compliant DDL. In the example below, you must save the entity definition "MyTable" in the CDS document `MyTable.hdbdd`. In addition, the name space declared in a CDS document must match the repository package in which the object the document defines is located.

```
namespace com.acme.myapp1;
@Schema : 'MYSCHEMA'
@Catalog.tableType : #COLUMN
@Catalog.index : [ { name : 'MYINDEX1', unique : true, order :#DESC,
elementNames : ['ISBN'] }  ]
entity MyTable {
       key Author    : String(100);
       key BookTitle : String(100);
           ISBN      : Integer not null;
           Publisher : String(100);
};
```

If you want to create a CDS-compliant database entity definition as a repository file, you must create the entity as a flat file and save the file containing the DDL entity dimensions with the suffix `.hdbdd`, for example, `MyTable.hdbdd`. The new file is located in the package hierarchy you establish in the SAP HANA repository. The file location corresponds to the namespace specified at the start of the file, for example, `com.acme.myapp1` or `sap.hana.xs.app2`. You can activate the repository files at any point in time to create the corresponding runtime object for the defined table.

> **i Note**
>
> On activation of a repository file, the file suffix, for example, `.hdbdd`, is used to determine which runtime plug-in to call during the activation process. The plug-in reads the repository file selected for activation, in this case a CDS-compliant entity, parses the object descriptions in the file, and creates the appropriate runtime objects.

When a CDS document is activated, the activation process generates a corresponding catalog object for each of the artifacts defined in the document; the location in the catalog is determined by the type of object generated. For example, the corresponding database table for a CDS entity definition is generated in the following catalog location:

`<SID>` > *Catalog* > `<MYSCHEMA>` > *Tables*

Entity Element Definition

You can expand the definition of an entity element beyond the element's name and type by using element **modifiers**. For example, you can specify if an entity element is the primary key or **part** of the primary key. The following entity element modifiers are available:

- `key`
  Defines if the specified element is the **primary** key or **part** of the primary key for the specified entity.

> **Note**
>
> Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

- `null`
  Defines if an entity element can (`null`) or cannot (`not null`) have the value NULL. If neither `null` nor `not null` is specified for the element, the default value `null` applies (except for the `key` element).
- `default <literal_value>`
  Defines the default value for an entity element in the event that no value is provided during an INSERT operation. The syntax for the literals is defined in the primitive data-type specification.

```
entity MyEntity {
    key   MyKey  : Integer;
    key   MyKey2 : Integer null;                        // illegal combination
    key   MyKey3 : Integer default 2;
          elem2  : String(20) default 'John Doe';
          elem3  : String(20) default 'John Doe' null;
          elem4  : String default 'Jane Doe' not null;
};
```

Spatial Data

CDS entities support the use of spatial data types such as `hana.ST_POINT` or `hana.ST_GEOMETRY` to store geo-spatial coordinates. Spatial data is data that describes the position, shape, and orientation of objects in a defined space; the data is represented as two-dimensional geometries in the form of points, line strings, and polygons.

Related Information

CDS Primitive Data Types [page 72]
Entity Element Modifiers [page 43]
CDS Entity Syntax Options [page 49]

3.3.2 Entity Element Modifiers

Element **modifiers** enable you to expand the definition of an entity element beyond the element's name and type. For example, you can specify if an entity element is the primary key or **part** of the primary key.

Example

```
entity MyEntity {
```

```
    key   MyKey  : Integer;
          elem2  : String(20) default 'John Doe';
          elem3  : String(20) default 'John Doe' null;
          elem4  : String default 'Jane Doe' not null;
};
entity MyEntity1 {
    key id : Integer;
    a  : integer;
    b  : integer;
    c  : integer generated always as a+b;
};
entity MyEntity2 {
    autoId : Integer generated [always|by default] as identity ( start with 10
increment by 2 );
    name : String(100);
};
```

key

```
key   MyKey  : Integer;
key   MyKey2 : Integer null;                // illegal combination
key   MyKey3 : Integer default 2;
```

You can expand the definition of an entity element beyond the element's name and type by using element **modifiers**. For example, you can specify if an entity element is the primary key or **part** of the primary key. The following entity element modifiers are available:

- `key`
  Defines if the element is the **primary** key or **part** of the primary key for the specified entity. You **cannot** use the `key` modifier in the following cases:
  - In combination with a `null` modifier. The `key` element is `non null` by default because NULL cannot be used in the `key` element.

> **i** Note
>
> Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

null

```
elem3  : String(20) default 'John Doe' null;
elem4  : String default 'Jane Doe' not null;
```

`null` defines if the entity element can (`null`) or cannot (`not null`) have the value NULL. If neither `null` nor `not null` is specified for the element, the default value `null` applies (except for the `key` element), which means the element **can** have the value NULL. If you use the `null` modifier, note the following points:

> ⚠ Caution
>
> The keywords `nullable` and `not nullable` are no longer valid; they have been replaced for SPS07 with the keywords `null` and `not null`, respectively. The keywords `null` and `not null` must appear at the end of the entity element definition, for example, `field2 : Integer null;`.

- The `not null` modifier can only be added if the following is true:
  - A `default` it also defined
  - no `null` data is already in the table
- Unless the table is empty, bear in mind that when adding a new `not null` element to an existing entity, you must declare a default value because there might already be existing rows that do not accept NULL as a value for the new element.
- `null` elements with default values are permitted
- You cannot combine the element `key` with the element modifier `null`.
- The elements used for a unique index must have the `not null` property.

```
entity WithNullAndNotNull
{
  key id :  Integer;
  field1 :  Integer;
  field2 :  Integer null;  // same as field1, null is default
  field3 :  Integer not null;
};
```

default

```
default <literal_value>
```

For each scalar element of an entity, a default value can be specified. The `default` element identifier defines the default value for the element in the event that no value is provided during an INSERT operation.

> ℹ Note
>
> The syntax for the literals is defined in the primitive data-type specification.

```
entity WithDefaults
{
  key id :  Integer;
  field1 :  Integer       default -42;
  field2 :  Integer64     default 9223372036854775807;
  field3 :  Decimal(5, 3) default 12.345;
  field4 :  BinaryFloat   default 123.456e-1;
  field5 :  LocalDate     default date'2013-04-29';
  field6 :  LocalTime     default time'17:04:03';
  field7 :  UTCDateTime   default timestamp'2013-05-01 01:02:03';
  field8 :  UTCTimestamp  default timestamp'2013-05-01 01:02:03';
  field9 :  Binary(32)    default x'0102030405060708090a0b0c0d0e0[...]';
  field10 : String(10)    default 'foo';
```

```
};
```

generated always as <expression>

```
entity MyEntity1 {
    key id : Integer;
    a  : integer;
    b  : integer;
    c  : integer generated always as a+b;
};
```

The SAP HANA SQL clause `generated always as <expression>` is available for use in CDS entity definitions; it specifies the expression to use to generate the column value at run time. An element that is defined with `generated always as <expression>` corresponds to a field in the database table that is present in the persistence and has a value that is computed as specified in the expression, for example, "`a+b`".

> **!** Restriction
>
> For use in XS advanced only; it is not possible to use generated calculated elements in XS classic. Please also note that the `generated always as <expression>` clause is only for use with column-based tables.

"Generated" fields and "calculated" field differ in the following way. **Generated** fields are physically present in the database table; values are computed on `INSERT` and need not be computed on `SELECT`. **Calculated** fields are not actually stored in the database table; they are computed when the element is "selected". Since the value of the **generated** field is computed on `INSERT`, the expression used to generate the value must not contain any non-deterministic functions, for example: `current_timestamp`, `current_user`, `current_schema`, and so on.

generated [always | by default] as identity

```
entity MyEntity2 {
    autoId : Integer generated always as identity ( start with 10 increment by
2 );
    name : String(100);
};
```

The SAP HANA SQL clause `generated as identity` is available for use in CDS entity definitions; it enables you to specify an identity column. An element that is defined with `generated as identity` corresponds to a field in the database table that is present in the persistence and has a value that is computed as specified in the sequence options defined in the `identity` expression, for example, `( start with 10 increment by 2 )`.

In the example illustrated here, the name of the generated column is `autoID`, the first value in the column is "10"; the `identity` expression `( start with 10 increment by 2 )` ensures that subsequent values in the column are incremented by 2, for example: 12, 14, and so on.

> ! Restriction
>
> For use in XS advanced only; it is not possible to define an element with `IDENTITY` in XS classic. Please also note that the `generated always as identity` clause is only for use with column-based tables.

You can use either `always` or `by default` in the clause `generated as identity`, as illustrated in the examples in this section. If `always` is specified, then values are **always** generated; if `by default` is specified, then values are generated **by default**.

```
entity MyEntity2 {
    autoId : Integer generated by default as identity ( start with 10 increment
by 2 );
    name : String(100);
};
```

> ! Restriction
>
> CDS does not support the use of reset queries, for example, `RESET BY <subquery>`.

Column Migration Behavior

The following table shows the migration strategy that is used for modifications to any given column; the information shows which actions are performed and what strategy is used to preserve content. During the migration, a comparison is performed on the column **type**, the generation **kind**, and the expression, if available. From an end-user perspective, the result of a column modification is either a preserved or new value. The aim of any modification to an entity (table) is to cause as little loss as possible.

- Change to the column **type**
  In case of a column type change, the content is converted into the new type. HANA conversion rules apply.
- Change to the expression clause
  The expression is re-evaluated in the following way:
  - “early”
    As part of the column change
  - “late”
    As part of a query
- Change to a calculated column
  A calculated column is transformed into a plain column; the new column is initialized with NULL.

Technically, columns are either dropped and added or a completely new “shadow” table is created into which the existing content is copied. The shadow table will then replace the original table.

Before column/ After row	Plain	As <expr>	generated always as <expr>	generated always as identity <expr>	generated by default as identity <expr>
Plain	Migrate Keep content	Drop/add Evaluate on select	Drop/add Evaluate on add	Migrate Keep content	Migrate Keep content
generated by default as identity <expr>	Migrate Keep content	Drop/add Evaluate on select	Drop/add Evaluate on add	Migrate Keep content	Migrate Keep content
generated always as identity <expr>	Migrate Keep content	Drop/add Evaluate on select	Drop/add Evaluate on add	Migrate Keep content	Migrate Keep content
generated always as <expr>	Drop/add NULL	Drop/add Evaluate on select	Drop/add Evaluate on add	Drop/add Keep content	Migrate Keep content
as <expr>	Drop/add NULL	Drop/add Evaluate on select	Drop/add Evaluate on add	Drop/add Keep content	Migrate Keep content

Related Information

Create an Entity in CDS [page 39]
Create an Entity in CDS
CDS Entity Syntax Options [page 49]
SAP HANA SQL and System Views Reference (CREATE TABLE)

3.3.3 CDS Entity Syntax Options

The **entity** is the core design-time artifact for persistence model definition using the CDS syntax.

Example

> **i** Note
>
> This example is not a working example; it is intended for illustration purposes only.

```
namespace Pack1."pack-age2";
@Schema: 'MySchema'
context MyContext {
  entity MyEntity1
  {
    key id : Integer;
    name   : String(80);
  };
  @Catalog:
   { tableType : #COLUMN,
     index : [
     { name:'Index1', order:#DESC, unique:true,  elementNames:['x', 'y'] },
     { name:'Index2', order:#DESC, unique:false, elementNames:['x', 'a'] }
             ]
   }
  entity MyEntity2 {
    key id : Integer;
    x      : Integer;
    y      : Integer;
    a      : Integer;
    field7 : Decimal(20,10) = power(ln(x)*sin(y), a);
  };
  entity MyEntity {
    key id : Integer;
    a : Integer;
    b : Integer;
    c : Integer;
    s {
      m : Integer;
      n : Integer;
    };
    } technical configuration {
      row store;
      index MyIndex1 on (a, b) asc;
      unique index MyIndex2 on (c, s) desc;
    };
  context MySpatialContext {
    entity Address {
      key id        : Integer;
      street_number : Integer;
      street_name   : String(100);
      zip           : String(10);
      city          : String(100);
      loc           : hana.ST_POINT(4326);
    };
  }
  context MySeriesContext {
    entity MySeriesEntity {
      key setId : Integer;
      t : UTCTimestamp;
```

```
      value : Decimal(10,4);
      series (
        series key (setId)
        period for series (t)
        equidistant increment by interval 0.1 second
        equidistant piecewise //increment or piecewise; not both
      )
    };
  }
}
```

> **i** Note
>
> For series data, you can use either `equidistant` or `equidistant piecewise`, but not both at the same time. The example above is for illustration purposes only.

Overview

Entity definitions resemble the definition of structured types, but with the following additional features:

- Key definition [page 50]
- Index definition [page 51]
- Table type specification [page 52]
- Calculated Fields [page 53]
- Technical Configuration [page 53]
- Spatial data * [page 55]
- Series Data * [page 56]

On activation in the SAP HANA repository, each entity definition in CDS generates a database table. The name of the generated table is built according to the same rules as for table types, for example, `Pack1.Pack2::MyModel.MyContext.MyTable`.

> **i** Note
>
> The CDS name is restricted by the limits imposed on the length of the database identifier for the name of the corresponding SAP HANA database artifact (for example, table, view, or type); this is currently limited to 126 characters (including delimiters).

Key Definition

```
type MyStruc2
{
  field1 : Integer;
  field2 : String(20);
};
entity MyEntity2
{
  key id  : Integer;
  name    : String(80);
  key str : MyStruc2;
```

```
};
```

Usually an entity must have a key; you use the keyword `key` to mark the respective elements. The key elements become the primary key of the generated SAP HANA table and are automatically flagged as `not null`. Key elements are also used for managed associations. Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured element are part of the primary key.

> **i** Note
>
> To define an entity without a key, use the `@nokey` annotation.

Index Definition

```
@Catalog:
   { tableType : #COLUMN,
     index : [
     { name:'Index1', order:#DESC, unique:true,  elementNames:['field1',
'field2'] },
     { name:'Index2', order:#ASC,  unique:false, elementNames:['field1',
'field7'] }
            ]
   }
```

You use the `@Catalog.index` or `@Catalog: { index: [...]}` annotation to define an index for a CDS entity. You can define the following values for the `@Catalog.index` annotation:

- `name : '<IndexName>'`
  The name of the index to be generated for the specified entity, for example, `name:'myIndex'`
- `order`
  Create a table index sorted in ascending or descending order. The order keywords `#ASC` and `#DESC` can be only used in the **BTREE** index (for the maintenance of sorted data) and can be specified only once for each index.
  - `order : #ASC`
    Creates an index for the CDS entity and sorts the index fields in **ascending** logical order, for example: 1, 2, 3...
  - `order : #DESC`
    Creates a index for the CDS entity and sorts the index fields in **descending** logical order, for example: 3, 2, 1...
- `unique`
  Creates a unique index for the CDS entity. In a unique index, two rows of data in a table cannot have identical key values.
  - `unique : true`
    Creates a unique index for the CDS entity. The uniqueness is checked and, if necessary, enforced each time a key is added to (or changed in) the index and, in addition, each time a row is added to the table.
  - `unique : false`
    Creates a non-unique index for the CDS entity. A non-unique index is intended primarily to improve query performance, for example, by maintaining a sorted order of values for data that is queried frequently.
- `elementNames : ['<name1>', '<name2>' ]`

The names of the fields to use in the index; the elements are specified for the entity definition, for example, `elementNames:['field1', 'field2' ]`

Table-Type Definition

```
namespace com.acme.myapp1;
@Schema: 'MYSCHEMA'
context MyContext1 {
    @Catalog.tableType : #COLUMN
    entity MyEntity1 {
        key ID : Integer;
        name : String(30);
    };
    @Catalog.tableType : #ROW
    entity MyEntity2 {
        key ID : Integer;
        name : String(30);
    };
    @Catalog.tableType : #GLOBAL_TEMPORARY
    entity MyEntity3 {
        ID : Integer;
        name : String(30);
    };
     @Catalog.tableType : #GLOBAL_TEMPORARY_COLUMN
     entity MyTempEntity {
          a : Integer;
          b : String(20);
     };
};
```

You use the `@Catalog.tableType` or `@Catalog: { tableType: #<TYPE> }` annotation to define the type of CDS entity you want to create, for example: column- or row-based or global temporary. The `@Catalog.tableType` annotation determines the storage engine in which the underlying table is created. The following table lists and explains the permitted values for the `@Catalog.tableType` annotation:

Table-Type Syntax Options

Table-Type Option	Description
`#COLUMN`	@Catalog:Create a column-based table. If the majority of table access is through a large number of tuples, with only a few selected attributes, use COLUMN-based storage for your table type.
`#ROW`	Create a row-based table. If the majority of table access involves selecting a few records, with all attributes selected, use ROW-based storage for your table type.
`#GLOBAL_TEMPORARY`	Set the scope of the created table. Data in a **global** temporary table is session-specific; only the owner session of the global temporary table is allowed to insert/read/truncate the data. A global temporary table exists for the duration of the session, and data from the global temporary table is automatically dropped when the session is terminated. Note that a temporary table cannot be changed when the table is in use by an open session, and a global temporary table can only be dropped if the table does not have any records.
`#GLOBAL_TEMPORARY_COLUMN`	Set the scope of the table column. Global temporary column tables **cannot** have either a key or an index.

> **Note**
>
> The SAP HANA database uses a combination of table types to enable storage and interpretation in both ROW and COLUMN forms. If no table type is specified in the CDS entity definition, the default value *#COLUMN* is applied to the table created on activation of the design-time entity definition.

Calculated Fields

The definition of an entity can contain calculated fields, as illustrated in type "`z`" the following example:

> **Sample Code**

```
entity MyCalcField {
  a : Integer;
  b : Integer;
  c : Integer = a + b;
  s : String(10);
  t : String(10) = upper(s);
  x : Decimal(20,10);
  y : Decimal(20,10);
  z : Decimal(20,10) = power(ln(x)*sin(y), a);
};
```

The calculation expression can contain arbitrary expressions and SQL functions. The following restrictions apply to the expression you include in a calculated field:

- The definition of a calculated field must not contain other calculated fields, associations, aggregations, or subqueries.
- A calculated field cannot be key.
- No index can be defined on a calculated field.
- A calculated field cannot be used as foreign key for a managed association.

In a query, calculated fields can be used like ordinary elements.

> **Note**
>
> In SAP HANA tables, you can define columns with the additional configuration "`GENERATED ALWAYS AS`". These columns are physically present in the table, and all the values are stored. Although these columns behave for the most part like ordinary columns, their value is computed upon insertion rather than specified in the `INSERT` statement. This is in contrast to calculated field, for which no values are actually stored; the values are computed upon `SELECT`.

technical configuration

The definition of an entity can contain a section called `technical configuration`, which you use to define the elements listed in the following table:

- Storage type

- Indexes
- Full text indexes

> **Note**
>
> The syntax in the technical configuration section is as close as possible to the corresponding clauses in the SAP HANA SQL `Create Table` statement. Each clause in the technical configuration must end with a semicolon.

Storage type

In the technical configuration for an entity, you can use the `store` keyword to specify the storage type ("`row`" or "`column`") for the generated table, as illustrated in the following example. If no store type is specified, a "column" store table is generated by default.

> **Sample Code**

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   row store;
};
```

> **Restriction**
>
> It is not possible to use both the `@Catalog.tableType` annotation and the technical configuration (for example, `row store`) at the same time to define the storage type for an entity.

Indexes

In the technical configuration for an entity, you can use the `index` and `unique index` keywords to specify the index type for the generated table. For example: "`asc`" (ascending) or "`desc`" (descending) describes the index order, and `unique` specifies that the index is unique, where no two rows of data in the indexed entity can have identical key values.

> **Sample Code**

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
  index MyIndex1 on (a, b) asc;
  unique index MyIndex2 on (c, s) desc;
};
```

! Restriction

It is not possible to use both the `@Catalog.index` annotation **and** the technical configuration (for example, `index`) at the same time to define the index type for an entity.

Full text indexes

In the technical configuration for an entity, you can use the `fulltext index` keyword to specify the full-text index type for the generated table, as illustrated in the following example.

‹≡› Sample Code

```
entity MyEntity {
    key id : Integer;
    a : Integer;
    b : Integer;
    t : String(100);
    s {
        u : String(100);
    };
} technical configuration {
    row store;
    index MyIndex1 on (a, b) asc;
    unique index MyIndex2 on (a, b) asc;
    fulltext index MYFTI1 on (t)
        LANGUAGE COLUMN t
        LANGUAGE DETECTION ('de', 'en')
        MIME TYPE COLUMN s.u
        FUZZY SEARCH INDEX off
        PHRASE INDEX RATIO 0.721
        SEARCH ONLY off
        FAST PREPROCESS off
        TEXT ANALYSIS off;
    fuzzy search index on (s.u);
};
```

The `<fulltext_parameter_list>` is identical to the standard SAP HANA SQL syntax for `CREATE FULLTEXT INDEX`. A `fuzzy search index` in the technical configuration section of an entity definition corresponds to the `@SearchIndex` annotation in XS classic and the statement "`FUZZY SEARCH INDEX ON`" for a table column in SAP HANA SQL. It is not possible to specify both a full-text index and a fuzzy search index for the same element.

! Restriction

It is not possible to use both the `@SearchIndex` annotation **and** the technical configuration (for example, `fulltext index`) at the same time.

Spatial Types *

The following example shows how to use the spatial type `ST_POINT` in a CDS entity definition. In the example entity `Person`, each person has a home address and a business address, each of which is accessible via the corresponding associations. In the `Address` entity, the geo-spatial coordinates for each person are stored in element `loc` using the spatial type `ST_POINT` (*).

Sample Code

```
context SpatialData {
    entity Person {
        key id : Integer;
        name : String(100);
        homeAddress : Association[1] to Address;
        officeAddress : Association[1] to Address;
    };
    entity Address {
        key id : Integer;
        street_number : Integer;
        street_name : String(100);
        zip : String(10);
        city : String(100);
        loc : hana.ST_POINT(4326);
    };
    view CommuteDistance as select from Person {
        name,
        homeAddress.loc.ST_Distance(officeAddress.loc) as distance
    };
};
```

Series Data *

CDS enables you to create a table to store series data by defining an entity that includes a `series ()` clause as an table option and then defining the appropriate parameters and options.

Note

The `period for series` must be unique and should not be affected by any shift in timestamps.

Sample Code

```
context SeriesData {
    entity MySeriesEntity1 {
        key setId : Integer;
        t : UTCTimestamp;
        value : Decimal(10,4);
        series (
            series key (setId)
            period for series (t)
            equidistant increment by interval 0.1 second
        );
    };
    entity MySeriesEntity2 {
        key setId : Integer;
        t : UTCTimestamp;
        value : Decimal(10,4);
        series (
            series key (setId)
            period for series (t)
            equidistant piecewise
        );
    };
};
```

CDS also supports the creation of a series table called `equidistant piecewise` using Formula-Encoded Timestamps (FET). This enables support for data that is not loaded in an order that ensures good compression. There is no a-priori restriction on the timestamps that are stored, but the data is expected to be well approximated as piecewise linear with some jitter. The timestamps do not have a single slope/offset throughout the table; rather, they can change within and among series in the table.

> **!** Restriction
>
> The `equidistant piecewise` specification can only be used in CDS; it cannot be used to create a table with the SQL command `CREATE TABLE`.

When a series table is defined as `equidistant piecewise`, the following restrictions apply:

1. The `period` includes one column (instant); there is no support for interval periods.
2. There is no support for `missing elements`. These could logically be defined if the period includes an interval start and end. Missing elements then occur when we have adjacent rows where the end of the `interval` **does not** equal the start of the `interval`.
3. The type of the period column must map to the one of the following types: `DATE`, `SECONDDATE`, or `TIMESTAMP`.

> ⚠ Caution
>
> (*) For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Related Information

Create an Entity in CDS [page 39]
Create an Entity in CDS
CDS Annotations [page 28]
CDS Primitive Data Types [page 72]

3.4 Migrate an Entity from hdbtable to CDS (hdbdd)

Migrate a design-time representation of a table from the `.hdbtable` syntax to the CDS-compliant `.hdbdd` syntax while retaining the underlying catalog table and its content.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.

- You must have already created a development workspace and a project.
- You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema for the CDS catalog objects, for example, `MYSCHEMA`
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.
- You must have a design-time definition of the `hdbtable` entity you want to migrate to CDS.

Context

In this procedure you replace a design-time representation of a database table that was defined using the `hdbtable` syntax with a CDS document that describes the same table (entity) with the CDS-compliant `hdbdd` syntax. To migrate an `hdbtable` artifact to CDS, you must delete the inactive version of the `hdbtable` object and create a new `hdbdd` artifact with the same name and structure.

You must define the target CDS entity manually. The name of the entity and the names of the elements can be reused from the `hdbtable` definition. The same applies for the element modifiers, for example, NULL/NOT NULL, and the default values.

> **i Note**
>
> In CDS, there is no way to reproduce the column-comments defined in an `hdbtable` artifact. You can use source code comments, for example, `'/* */'` or `'//'`, however, the comments do not appear in the catalog table after activation of the new CDS artifact.

Procedure

1. Use CDS syntax to create a duplicate of the table you originally defined using the `hdbtable` syntax.

   > **i Note**
   >
   > The new CDS document must have the same name as the original `hdbtable` artifact, for example, `Employee.hdbdd` and `Employee.hdbtable`.

   The following code shows a simple table `Employee.hdbtable` that is defined using the `hdbtable` syntax. This is the "source" table for the migration. When you have recreated this table in CDS using the `.hdbdd` syntax, you can delete the artifact `Employee.hdbtable`.

   ```
   table.schemaName = "MYSCHEMA";
   table.tableType = COLUMNSTORE;
   table.columns = [
        {name = "firstname"; sqlType = NVARCHAR; nullable = false; length = 20;},
        {name = "lastname"; sqlType = NVARCHAR; nullable = true; length = 20;
   defaultValue = "doe";},
        {name = "age"; sqlType = INTEGER; nullable = false;},
        {name = "salary"; sqlType = DECIMAL; nullable = false; precision = 7;
   scale = 2;}
                   ];
   ```

The following code shows the same simple table recreated with the CDS-compliant `hdbdd` syntax. The new design-time artifact is called `Employee.hdbdd` and is the “target” for the migration operation. Note that all column names remain the same.

```
namespace sample.cds.tutorial;
@Schema:'MYSCHEMA'
@Catalog.tableType:#COLUMN
@nokey
entity Employee {
     firstname : String(20) not null;
     lastname : String(20) default 'doe';
     age : Integer not null;
     salary : Decimal(7,2) not null;
};
```

2. Activate the source (`hdbtable`) and target (CDS) artifacts of the migration operation.

   To replace the old `hdbtable` artifact with the new `hdbdd` (CDS) artifact, you must activate both artifacts (the deleted `hdbtable` artifact and the new new CDS document) together in a single activation operation, for example, by performing the activation operation on the folder that contains the two objects. If you do not activate both artifacts together in one single activation operation, data stored in the table will be lost since the table is deleted and recreated during the migration process.

   > → Tip
   >
   > In SAP HANA studio, choose the *Team* > *Activate all...* option to list all inactive objects and select the objects you want to activate. In the SAP HANA Web-based Workbench, the default setting is *Activate on save*, however you can change this behavior to *Save only*.

3. Check that the table is in the catalog in the specified schema.

   To ensure that the new CDS-defined table is identical to the old (HDBtable-defined) table, check the contents of the table in the catalog.

Related Information

Migration Guidelines: hdbtable to CDS Entity [page 59]
SAP HANA to CDS Data-Type Mapping [page 60]

3.4.1 Migration Guidelines: hdbtable to CDS Entity

Replace an existing `hdbtable` definition with the equivalent CDS document.

It is possible to migrate your SAP HANA `hdbtable` definition to a Core Data Services (CDS) entity that has equally named but differently typed elements. When recreating the new CDS document, you cannot choose an arbitrary data type; you must follow the guidelines for valid data-type mappings in the SAP HANA SQL data-type conversion documentation. Since the SAP HANA SQL documentation does not cover CDS data types you must map the target type names to CDS types manually.

> **i Note**
>
> Remember that most of the data-type conversions depend on the data that is present in the catalog table on the target system.

If you are planning to migrate SAP HANA (hdbtable) tables to CDS entities, bear in mind the following important points:

- CDS document structure
  The new entity (that replaces the old `hdbtable` definition) must be defined at the top-level of the new CDS document; it cannot be defined deeper in the CDS document, for example, nested inside a CDS context. If the table (entity) is not defined as the top-level element in the CDS document, the resulting catalog name of the entity (on activation) will not match the name of the runtime table that should be taken over by the new CDS object. Instead, the name of the new table would also include the name of the CDS context in which it was defined, which could lead to unintended consequences after the migration.
  If the top-level element of the target CDS entity is not an entity (for example, a context or a type), the activation of the CDS document creates the specified artifact (a context or a type) and does not take over the catalog table defined by the source (`hdbtable`) definition.
- Structural compatibility
  The new CDS document (defined in the `hdbdd` artifact) must be structurally compatible with the table definition in the old `hdbtable` artifact (that is, with the runtime table).
  - Data types
    All elements of the new CDS entity that have equally named counterparts in the old `hdbtable` definition must be convertible with respect to their data type. The implicit conversion rules described in the SAP HANA SQL documentation apply.
  - Elements/Columns
    Elements/columns that exist in the runtime table but are **not** defined in the CDS entity will be dropped.
    Elements/columns that do **not** exist in the runtime table but are defined in the CDS entity are added to the runtime table.

Related Information

SAP HANA to CDS Data-Type Mapping [page 60]
SAP HANA SQL Data Type Conversion

3.4.2 SAP HANA to CDS Data-Type Mapping

Mapping table for SAP HANA (hdbtable) and Core Data Services (CDS) types.

Although CDS defines its own system of data types, the list of types is roughly equivalent to the data types available in SAP HANA (`hdbtable`); the difference between CDS data types and SAP HANA data types is mostly in the type names. The following table lists the SAP HANA (`hdbtable`) data types and indicates what the equivalent type is in CDS.

Mapping SAP HANA and CDS Types

SAP HANA Type (hdbtable)	CDS Type (hdbdd)
NVARCHAR	String
SHORTTEXT	String
NCLOB	LargeString
TEXT	LargeString
VARBINARY	Binary
BLOB	LargeBinary
INTEGER	Integer
INT	Integer
BIGINT	Integer64
DECIMAL(p,s)	Decimal(p,s)
DECIMAL	DecimalFloat
DOUBLE	BinaryFloat
DAYDATE	LocalDate
DATE	LocalDate
SECONDTIME	LocalTime
TIME	LocalTime
SECONDDATE	UTCDateTime
LONGDATE	UTCTimestamp
TIMESTAMP	UTCTimestamp
ALPHANUM	hana.ALPHANUM
SMALLINT	hana.SMALLINT
TINYINT	hana.TINYINT
SMALLDECIMAL	hana.SMALLDECIMAL
REAL	hana.REAL
VARCHAR	hana.VARCHAR
CLOB	hana.CLOB
BINARY	hana.BINARY
ST_POINT	hana.ST_POINT
ST_GEOMETRY	hana.ST_GEOMETRY

Related Information

Migrate an Entity from hdbtable to CDS (hdbdd) [page 57]
CDS Entity Syntax Options [page 49]
SAP HANA SQL Data Type Conversion

3.5 Create a User-Defined Structured Type in CDS

A structured type is a data type comprising a list of attributes, each of which has its own data type. You create a user-defined structured type as a design-time file in the SAP HANA repository.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have already created a development workspace and a project.
- You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema for the CDS catalog objects, for example, `MYSCHEMA`
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.

Context

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the CDS syntax to create a user-defined structured type as a design-time file in the repository. Repository files are transportable. Activating the CDS document creates the corresponding types in the specified schema. To create a CDS document that defines one or more structured types and save the document in the repository, perform the following steps:

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS definition file for the user-defined structured type.

   Browse to the folder in your project workspace where you want to create the CDS definition file for the new user-defined structured type and perform the following steps:

   a. Right-click the folder where you want to save the definition file for the user-defined structured type and choose *New* > *Other...* > *Database Development* > *DDL Source File* in the context-sensitive popup menu.

   b. Enter the name of the user-defined structured type in the *File Name* box, for example, `MyStructuredType`.

   > → Tip
   >
   > File extensions are important. If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically (for

example, `MyCDSFile.hdbdd`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

c. Choose *Finish* to save the changes and commit the new the user-defined structured type in the repository.

5. Define the user-defined structured type in CDS.

   If the new user-defined structured type is not automatically displayed by the file-creation wizard, in the *Project Explorer* view double-click the user-defined structured type you created in the previous step, for example, `MyStructuredType.hdbdd`, and add the definition code for the user-defined structured type to the file:

   > **i Note**
   >
   > The following code example is provided for illustration purposes only. If the schema you specify does not exist, you cannot activate the new CDS document and the structured types are not created.

   ```
   namespace Package1.Package2;
   @Schema: 'MYSCHEMA'
   type MyStructuredType
   {
     aNumber   : Integer;
     someText  : String(80);
     otherText : String(80);
   };
   ```

6. Save the definition file for the CDS user-defined structured type.

   > **i Note**
   >
   > Saving a file in a shared project automatically commits the saved version of the file to the repository. You do not need to explicitly commit the file again.

7. Activate the changes in the repository.
   a. Locate and right-click the new CDS definition file in the *Project Explorer* view.
   b. In the context-sensitive pop-up menu, choose *Team* > *Activate*.

      If you cannot activate the new CDS artifact, check that the specified schema already exists and that there are no illegal characters in the name space, for example, the hyphen (-).

   On activation, the data types appear in the *Systems* view of the *SAP HANA Development* perspective under <SID> > *Catalog* > *SchemaName* > *Procedures* > *Table Types*.

8. Ensure access to the schema where the new CDS catalog objects are created.

   After activation in the repository, a schema object is only visible in the catalog to the _SYS_REPO user. To enable other users, for example the schema owner, to view the newly created schema and the objects it contains, you must grant the user the required SELECT privilege for the schema object.

   > **i Note**
   >
   > If you already have the appropriate `SELECT` privilege, you do not need to perform this step.

   a. In the SAP HANA studio *Systems* view, right-click the SAP HANA system hosting the repository where the schema was activated and choose *SQL Console* in the context-sensitive popup menu.

b. In the *SQL console*, execute the statement illustrated in the following example, where `<SCHEMANAME>` is the name of the newly activated schema, and `<username>` is the database user ID of the schema owner:

```
call
_SYS_REPO.GRANT_SCHEMA_PRIVILEGE_ON_ACTIVATED_CONTENT('select','<SCHEMANAME
>','<username>');
```

Related Information

CDS User-Defined Data Types [page 64]
CDS Structured Type Definition [page 67]
CDS Structured Types [page 70]

3.5.1 CDS User-Defined Data Types

User-defined data types reference existing structured types (for example, user-defined) or the individual types (for example, field, type, or context) used in another data-type definition.

You can use the *type* keyword to define a new data type in CDS-compliant DDL syntax. You can define the data type in the following ways:

- Using allowed structured types (for example, user-defined)
- Referencing another data type

In the following example, the element definition `field2 : MyType1;` specifies a new element `field2` that is based on the specification in the user-defined data type `MyType1`.

> **i Note**
>
> If you are using a CDS document to define a single CDS-compliant user-defined data type, the name of the CDS document must match the name of the top-level data type defined in the CDS document, for example, with the *type* keyword.

In the following example, you must save the data-type definition "MyType1" in the CDS document `MyType1.hdbdd`. In addition, the name space declared in a CDS document must match the repository package in which the object the document defines is located.

```
namespace com.acme.myapp1;
@Schema: 'MYSCHEMA' // user-defined structured data types
type MyType1 {
      field1 : Integer;
      field2 : String(40);
      field3 : Decimal(22,11);
      field4 : Binary(11);
    };
```

In the following example, you must save the data-type definition "MyType2" in the CDS document `MyType2.hdbdd`; the document contains a using directive pointing to the data-type "MyType1" defined in CDS document `MyType1.hdbdd`.

```
namespace com.acme.myapp1;
using com.acme.myapp1::MyType1;
@Schema: 'MYSCHEMA' // user-defined structured data types
type MyType2 {
    field1 : String(50);
    field2 : MyType1;
};
```

In the following example, you must save the data-type definition "MyType3" in the CDS document `MyType3.hdbdd`; the document contains a using directive pointing to the data-type "MyType2" defined in CDS document `MyType2.hdbdd`.

```
namespace com.acme.myapp1;
using com.acme.myapp1::MyType2;
@Schema: 'MYSCHEMA' // user-defined structured data types
type MyType3 {
    field1 : UTCTimestamp;
    field2 : MyType2;
};
```

The following code example shows how to use the `type of` keyword to define an element using the definition specified in another user-defined data-type field. For example, `field4 : type of field3;` indicates that, like `field3`, `field4` is a `LocalDate` data type.

```
namespace com.acme.myapp1;
using com.acme.myapp1::MyType1;
@Schema: 'MYSCHEMA' // Simple user-defined data types
entity MyEntity1 {
  key id  : Integer;
  field1  : MyType3;
  field2  : String(24);
  field3  : LocalDate;
  field4  : type of field3;
  field5  : type of MyType1.field2;
  field6  : type of InnerCtx.CtxType.b;  // context reference
};
```

You can use the `type of` keyword in the following ways:

- Define a new element (`field4`) using the definition specified in another user-defined element `field3`:
  `field4 : type of field3;`
- Define a new element `field5` using the definition specified in a **field** (`field2`) that belongs to another user-defined data type (`MyType1`):
  `field5 : type of MyType1.field2;`
- Define a new element (`field6`) using an existing field (`b`) that belongs to a data type (`CtxType`) in another **context** (`InnerCtx`):
  `field6 : type of InnerCtx.CtxType.b;`

The following code example shows you how to define nested contexts (`MyContext.InnerCtx`) and refer to data types defined by a user in the specified context.

```
namespace com.acme.myapp1;
@Schema: 'MYSCHEMA'
context MyContext {
// Nested contexts
```

```
   context InnerCtx {

      Entity MyEntity {
         …
      };
      Type CtxType {
        a : Integer;
        b : String(59);
      };
   };
   type MyType1 {
      field1 : Integer;
      field2 : String(40);
      field3 : Decimal(22,11);
      field4 : Binary(11);
   };
   type MyType2 {
      field1 : String(50);
      field2 : MyType1;
   };
   type MyType3 {
      field1 : UTCTimestamp;
      field2 : MyType2;
   };

   @Catalog.index : [{ name : 'IndexA', order : #ASC, unique: true,
                       elementNames : ['field1'] }]
   entity MyEntity1 {
      key id  : Integer;
      field1  : MyType3 not null;
      field2  : String(24);
      field3  : LocalDate;
      field4  : type of field3;
      field5  : type of MyType1.field2;
      field6  : type of InnerCtx.CtxType.b;  // refers to nested context
      field7  : InnerCtx.CtxType;           //  more context references
   };
};
```

Restrictions

CDS name resolution does not distinguish between CDS `elements` and CDS `types`. If you define a CDS element based on a CDS data type that has the same name as the new CDS element, CDS displays an error message and the activation of the CDS document fails.

> ⚠ Caution
>
> In an CDS document, you cannot define a CDS element using a CDS type of the same name; you must specify the **context** where the target type is defined, for example, `MyContext.doobidoo`.

The following example defines an association between a CDS element and a CDS data `type` both of which are named `doobidoo`. The result is an error when resolving the names in the CDS document; CDS expects a type named `doobidoo` but finds an CDS entity element with the same name that is **not** a type.

```
context MyContext2 {
  type doobidoo : Integer;
  entity MyEntity {
    key id : Integer;
    doobidoo : doobidoo;  // error: type expected; doobidoo is not a type
```

```
  };
};
```

The following example works, since the explicit reference to the context where the type definition is located (`MyContext.doobidoo`) enables CDS to resolve the definition target.

```
context MyContext {
  type doobidoo : Integer;
  entity MyEntity {
    key id : Integer;
    doobidoo : MyContext.doobidoo;  // OK
  };
};
```

> **i** Note
>
> To prevent name clashes between artifacts that **are** types and those that **have** a type assigned to them, make sure you keep to strict naming conventions. For example, use an **uppercase** first letter for `MyEntity`, `MyView` and `MyType`; use a lowercase first letter for elements `myElement`.

Related Information

Create a User-Defined Structured Type in CDS
Create a User-Defined Structured Type in CDS [page 62]
CDS Structured Type Definition [page 67]
CDS Primitive Data Types [page 72]

3.5.2 CDS Structured Type Definition

A structured type is a data type comprising a list of attributes, each of which has its own data type. The attributes of the structured type can be defined manually in the structured type itself and reused either by another structured type or an entity.

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database structured type as a design-time file in the repository. All repository files including your structured-type definition can be transported to other SAP HANA systems, for example, in a delivery unit. You can define the structured type using CDS-compliant DDL.

> **i** Note
>
> A delivery unit is the medium SAP HANA provides to enable you to assemble all your application-related repository artifacts together into an archive that can be easily exported to other systems.

When a CDS document is activated, the activation process generates a corresponding catalog object for each of the artifacts defined in the document; the location in the catalog is determined by the type of object generated. For example, the corresponding table type for a CDS type definition is generated in the following catalog location:

`<SID>` ▸ *Catalog* ▸ `<MYSCHEMA>` ▸ *Procedures* ▸ *Table Types*

Structured User-Defined Types

In a structured user-defined type, you can define original types (`aNumber` in the following example) or reference existing types defined elsewhere in the same type definition or another, separate type definition (`MyString80`). If you define multiple types in a single CDS document, for example, in a parent context, each structure-type definition must be separated by a semi-colon (;).

The type `MyString80` is defined in the following CDS document:

```
namespace Package1.Package2;
@Schema: 'MySchema'
type MyString80: String(80);
```

A using directive is required to resolve the reference to the data type specified in `otherText : MyString80;`, as illustrated in the following example:

```
namespace Package1.Package2;
using Package1.Package2::MyString80;  //contains definition of MyString80
@Schema: 'MySchema'
type MyStruct
{
  aNumber   : Integer;
  someText  : String(80);
  otherText : MyString80;  // defined in a separate type
};
```

> **i** Note
>
> If you are using a CDS document to specify a single CDS-compliant data type, the name of the CDS document (`MyStruct.hdbdd`) must match the name of the top-level data type defined in the CDS document, for example, with the *type* keyword.

Nested Structured Types

Since user-defined types can make use of other user-defined types, you can build nested structured types, as illustrated in the following example:

```
namespace Package1.Package2;
using Package1.Package2::MyString80;
using Package1.Package2::MyStruct;
@Schema: 'MYSCHEMA'
context NestedStructs {
    type MyNestedStruct
    {
        name : MyString80;
        nested : MyStruct; // defined in a separate type
    };
    type MyDeepNestedStruct
    {
        text : LargeString;
        nested : MyNestedStruct;
    };
    type MyOtherInt    : type of MyStruct.aNumber; // => Integer
    type MyOtherStruct : type of MyDeepNestedStruct.nested.nested; // => MyStruct
};
```

You can also define a type based on an existing type that is already defined in another user-defined structured type, for example, by using the `type of` keyword, as illustrated in the following example:

```
type MyOtherInt    : type of MyStruct.aNumber;                  // => Integer
type MyOtherStruct : type of MyDeepNestedStruct.nested.nested;  // => MyStruct
```

Generated Table Types

For each structured type, a SAP HANA table type is generated, whose name is built by concatenating the following elements of the CDS document containing the structured-type definition and separating the elements by a dot delimiter (.):

- the name space (for example, `Pack1.Pack2`)
- the names of all artifacts that enclose the type (for example, `MyModel`)
- the name of the type itself (for example, `MyNestedStruct`)

```
create type "Pack1.Pack2::MyModel.MyNestedStruct" as table (
  name             nvarchar(80),
  nested.aNumber   integer,
  nested.someText  nvarchar(80),
  nested.otherText nvarchar(80)
);
```

The new SAP HANA table types are generated in the schema that is specified in the schema annotation of the respective top-level artifact in the CDS document containing the structured types.

> **i** Note
>
> To view the newly created objects, you must have the required SELECT privilege for the schema object in which the objects are generated.

The columns of the table type are built by flattening the elements of the type. Elements with structured types are mapped to one column per nested element, with the column names built by concatenating the element names and separating the names by dots ".".

> **→** Tip
>
> If you want to use the structured types inside a CDS document without generating table types in the catalog, use the annotation `@GenerateTableType : false`.

Table types are only generated for direct structure definitions; in the following example, this would include: `MyStruct`, `MyNestedStruct`, and `MyDeepNestedStruct`. No table types are generated for **derived** types that are based on structured types; in the following example, the derived types include: `MyS`, `MyOtherInt`, `MyOtherStruct`.

Example

```
namespace Pack1."pack-age2";
```

```
@Schema: 'MySchema'
context MyModel
{
  type MyInteger  : Integer;
  type MyString80 : String(80);
  type MyDecimal  : Decimal(10,2);
  type MyStruct
  {
    aNumber   : Integer;
    someText  : String(80);
    otherText : MyString80;  // defined in example above
  };
  type MyS           : MyStruct;
  type MyOtherInt    : type of MyStruct.aNumber;
  type MyOtherStruct : type of MyDeepNestedStruct.nested.nested;
  type MyNestedStruct
  {
    name   : MyString80;
    nested : MyS;
  };
  type MyDeepNestedStruct
  {
    text   : LargeString;
    nested : MyNestedStruct;
  };
};
```

Related Information

Create a User-Defined Structured Type in CDS [page 62]
Create a User-Defined Structured Type in CDS
CDS User-Defined Data Types [page 64]
CDS Structured Types [page 70]
CDS Primitive Data Types [page 72]

3.5.3 CDS Structured Types

A structured type is a data type comprising a list of attributes, each of which has its own data type. The attributes of the structured type can be defined manually in the structured type itself and reused either by another structured type or an entity.

Example

```
namespace examples;
@Schema: 'MYSCHEMA'
context StructuredTypes {
    type MyOtherInt : type of MyStruct.aNumber; // => Integer
    type MyOtherStruct : type of MyDeepNestedStruct.nested.nested; // => MyStruct
    @GenerateTableType: false
    type EmptyStruct { };
```

```
    type MyStruct
    {
        aNumber : Integer;
        aText : String(80);
        anotherText : MyString80; // defined in a separate type
    };
    entity E {
        a : Integer;
        s : EmptyStruct;
    };
    type MyString80 : String(80);
    type MyS : MyStruct;
    type MyNestedStruct
    {
        name : MyString80;
        nested : MyS;
    };
    type MyDeepNestedStruct
    {
        text : LargeString;
        nested : MyNestedStruct;
    };
};
```

type

In a structured user-defined type, you can define original types (`aNumber` in the following example) or reference existing types defined elsewhere in the same type definition or another, separate type definition, for example, `MyString80` in the following code snippet. If you define multiple types in a single CDS document, each structure definition must be separated by a semi-colon (;).

```
type MyStruct
{
  aNumber     : Integer;
  aText       : String(80);
  anotherText : MyString80;  // defined in a separate type
};
```

You can define structured types that do not contain any elements, for example, using the keywords `type EmptyStruct { };`. In the example, below the generated table for entity “E” contains only one column: “a”.

> → Tip
>
> It is not possible to generate an SAP HANA table type for an empty structured type. This means you must disable the generation of the table type in the Repository, for example, with the `@GenerateTableType` annotation.

```
@GenerateTableType : false
type EmptyStruct { };
entity E {
  a : Integer;
  s : EmptyStruct;
};
```

type of

You can define a type based on an existing type that is already defined in another user-defined structured type, for example, by using the `type of` keyword, as illustrated in the following example:

```
Context StructuredTypes
{
type MyOtherInt    : type of MyStruct.aNumber;                  // => Integer
type MyOtherStruct : type of MyDeepNestedStruct.nested.nested;  // => MyStruct
};
```

Related Information

Create a User-Defined Structured Type in CDS [page 62]
Create a User-Defined Structured Type in CDS
CDS Primitive Data Types [page 72]
CDS User-Defined Data Types [page 64]
CDS Structured Type Definition [page 67]

3.5.4 CDS Primitive Data Types

In the Data Definition Language (DDL), primitive (or core) data types are the basic building blocks that you use to define *entities* or *structure types* with DDL.

When you are specifying a design-time table (entity) or a view definition using the CDS syntax, you use data types such as *String*, *Binary*, or *Integer* to specify the type of content in the entity columns. CDS supports the use of the following primitive data types:

- DDL data types [page 72]
- Native SAP HANA data types [page 74]

The following table lists all currently supported simple DDL primitive data types. Additional information provided in this table includes the SQL syntax required as well as the equivalent SQL and EDM names for the listed types.

Supported SAP HANA DDL Primitive Types

Name	Description	SQL Literal Syntax	SQL Name	EDM Name
String (n)	Variable-length Unicode string with a specified maximum length of n=1-1333 characters (5000 for SAP HANA specific objects). Default = maximum length. String length (n) is mandatory.	'text with “quote”'	NVARCHAR	String
LargeString	Variable length string of up to 2 GB (no comparison)	'text with “quote”'	NCLOB	String

Name	Description	SQL Literal Syntax	SQL Name	EDM Name
Binary(n)	Variable length byte string with user-defined length limit of up to 4000 bytes. Binary length (n) is mandatory.	x'01Cafe', X'01Cafe'	VARBINARY	Binary
LargeBinary	Variable length byte string of up to 2 GB (no comparison)	x'01Cafe', X'01Cafe'	BLOB	Binary
Integer	Respective container's standard signed integer. Signed 32 bit integers in 2's complement, -2**31 .. 2**31-1. Default=NULL	13, -1234567	INTEGER	Int64
Integer64	Signed 64-bit integer with a value range of -2^63 to 2^63-1. Default=NULL.	13, -1234567	BIGINT	Int64
Decimal(p, s)	Decimal number with fixed precision (p) in range of 1 to 34 and fixed scale (s) in range of 0 to p. Values for precision and scale are mandatory.	12.345, -9.876	DECIMAL(p, s)	Decimal
DecimalFloat	Decimal floating-point number (IEEE 754-2008) with 34 mantissa digits; range is roughly ±1e-6143 through ±9.99e+6144	12.345, -9.876	DECIMAL	Decimal
BinaryFloat	Binary floating-point number (IEEE 754), 8 bytes (roughly 16 decimal digits precision); range is roughly ±2.2207e-308 through ±1.7977e+308	1.2, -3.4, 5.6e+7	DOUBLE	Double
LocalDate	Local date with values ranging from 0001-01-01 through 9999-12-31	date'1234-12-31'	DATE	DateTimeOffset Combines date and time; with time zone must be converted to offset
LocalTime	Time values (with seconds precision) and values ranging from 00:00:00 through 24:00:00	time'23:59:59', time'12:15'	TIME	Time For duration/ period of time (==xsd:duration). Use DateTimeOffset if there is a date, too.

Name	Description	SQL Literal Syntax	SQL Name	EDM Name
UTCDateTime	UTC date and time (with seconds precision) and values ranging from 0001-01-01 00:00:00 through 9999-12-31 23:59:59	timestamp'2011-12-31 23:59:59'	SECONDDATE	DateTimeOffset Values ending with “Z” for UTC. Values before 1753-01-01T00:00:00 are not supported; transmitted as NULL.
UTCTimestamp	UTC date and time (with a precision of 0.1 microseconds) and values ranging from 0001-01-01 00:00:00 through 9999-12-31 23:59:59.9999999, and a special initial value	timestamp'2011-12-31 23:59:59.7654321'	TIMESTAMP	DateTimeOffset With Precision = “7”
Boolean	Represents the concept of binary-valued logic	true, false, unknown (null)	BOOLEAN	Boolean

The following table lists all the **native** SAP HANA primitive data types that CDS supports. The information provided in this table also includes the SQL syntax required (where appropriate) as well as the equivalent SQL and EDM names for the listed types.

> **i** Note
>
> * In CDS, the name of SAP HANA data types are prefixed with the word “hana”, for example, `hana.ALPHANUM`, or `hana.SMALLINT`, or `hana.TINYINT`.

Supported Native SAP HANA Data Types

Data Type *	Description	SQL Literal Syntax	SQL Name	EDM Name
ALPHANUM	Variable-length character string with special comparison	-	ALPHANUMERIC	-
SMALLINT	Signed 16-bit integer	-32768, 32767	SMALLINT	Int16
TINYINT	Unsigned 8-bit integer	0, 255	TINYINT	Byte
REAL	32-bit binary floating-point number	-	REAL	Single
SMALLDECIMAL	64-bit decimal floating-point number	-	SMALLDECIMAL	Decimal
VARCHAR	Variable-length ASCII character string with user-definable length limit n	-	VARCHAR	String
CLOB	Large variable-length ASCII character string, no comparison	-	CLOB	String

Data Type *	Description	SQL Literal Syntax	SQL Name	EDM Name
BINARY	Byte string of fixed length n	-	BINARY	Blob
ST_POINT	0-dimensional geometry representing a single location	-	-	-
ST_GEOMETRY	Maximal supertype of the geometry type hierarchy; includes `ST_POINT`	-	-	-

The following example shows the **native** SAP HANA data types that CDS supports; the code example also illustrates the mandatory syntax.

> **i** Note
>
> Support for the geo-spatial types `ST_POINT` and `ST_GEOMETRY` is limited: these types can only be used for the definition of elements in types and entities. It is not possible to define a CDS view that selects an element based on a geo-spatial type from a CDS entity.

```
@nokey
entity SomeTypes {
  a : hana.ALPHANUM(10);
  b : hana.SMALLINT;
  c : hana.TINYINT;
  d : hana.SMALLDECIMAL;
  e : hana.REAL;
  h : hana.VARCHAR(10);
  i : hana.CLOB;
  j : hana.BINARY(10);
  k : hana.ST_POINT;
  l : hana.ST_GEOMETRY;
};
```

Related Information

Create a User-Defined Structures Type in CDS [page 62]
Create a CDS User-Defined Structure in XS Advanced [page 203]

3.6 Create an Association in CDS

Associations define relationships between entities. You create associations in a CDS entity definition, which is a design-time file in the SAP HANA repository.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have already created a development workspace and a project.
- You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema for the CDS catalog objects, for example, `MYSCHEMA`
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.

Context

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the CDS syntax to create associations between entities. The associations are defined as part of the entity definition, which are design-time files in the repository. Repository files are transportable. Activating the CDS entity creates the corresponding catalog objects in the specified schema. To create an association between CDS entities, perform the following steps:

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS entity-definition file which will contain the associations you define.

   Browse to the folder in your project workspace where you want to create the new CDS entity-definition file and perform the following steps:

   a. Right-click the folder where you want to save the entity-definition file and choose *New* > *Other...* > *Database Development* > *DDL Source File* in the context-sensitive popup menu.
   b. Enter the name of the CDS document in the *File Name* box, for example, `MyModel1`.

      > → Tip
      >
      > File extensions are important. If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically (for

example, `MyEntity1.hdbdd`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

c. Choose *Finish* to save the changes and commit the new CDS file in the repository.

5. Define the underlying CDS entities and structured types.

   If the new CDS file is not automatically displayed by the file-creation wizard, in the *Project Explorer* view double-click the CDS file you created in the previous step, for example, `MyModel1.hdbdd`, and add the code for the entity definitions and structured types to the file:

   > **i Note**
   >
   > The following code example is provided for illustration purposes only. If the schema you specify does not exist, you cannot activate the new CDS entity.

```
context MyEntity1 {
    type StreetAddress {
        name : String(80);
        number : Integer;
    };
    type CountryAddress {
        name : String(80);
        code : String(3);
    };
    entity Address {
        key id : Integer;
        street : StreetAddress;
        zipCode : Integer;
        city : String(80);
        country : CountryAddress;
        type : String(10); // home, office
    };

};
```

6. Define a one-to-**one** association between CDS entities.

   In the same entity-definition file you edited in the previous step, for example, `MyEntity.hdbdd`, add the code for the one-to-one association between the entity `Person` and the entity `Address`:

   > **i Note**
   >
   > This example does not specify cardinality or foreign keys, so the cardinality is set to the default 0..1, and the target entity's primary key (the element id) is used as foreign key.

```
entity Person
{
  key id : Integer;
  address1 : Association to Address;
  addressId : Integer;
};
```

7. Define an unmanaged association with cardinality one-to-**many** between CDS entities.

   In the same entity-definition file you edited in the previous step, for example, `MyEntity.hdbdd`, add the code for the one-to-many association between the entity `Address` and the entity `Person`. The code should look something like the following example:

```
entity Address {
   key id : Integer;
```

```
    street : StreetAddress;
    zipCode : Integer;
    city : String(80);
    country : CountryAddress;
    type : String(10); // home, office
    inhabitants : Association[*] to Person on inhabitants.addressId = id;
};
```

8. Save the CDS entity-definition file containing the new associations.

> **i Note**
>
> Saving a file in a shared project automatically commits the saved version of the file to the repository, To explicitly commit a file to the repository, right-click the file (or the project containing the file) and choose *Team* > *Commit* from the context-sensitive popup menu.

9. Activate the changes in the repository.
   a. Locate and right-click the new CDS entity-definition file in the *Project Explorer* view.
   b. In the context-sensitive pop-up menu, choose *Team* > *Activate*.

> **i Note**
>
> If you cannot activate the new CDS artifact, check that the specified schema already exists and that there are no illegal characters in the name space, for example, the hyphen (-).

Related Information

CDS Associations [page 78]
CDS Association Syntax Options [page 84]

3.6.1 CDS Associations

Associations define relationships between entities.

Associations are specified by adding an element to a source entity with an association **type** that points to a target entity, complemented by optional information defining cardinality and which keys to use.

> **i Note**
>
> CDS supports both managed and unmanaged associations.

SAP HANA Extended Application Services (SAP HANA XS) enables you to use associations in CDS entities or CDS views. The syntax for **simple** associations in a CDS document is illustrated in the following example:

```
namespace samples;
@Schema: 'MYSCHEMA'         // XS classic *only*
context SimpleAssociations {
    type StreetAddress {
        name : String(80);
        number : Integer;
```

```
    };
    type CountryAddress {
        name : String(80);
        code : String(3);
    };
    entity Address {
        key id : Integer;
        street : StreetAddress;
        zipCode : Integer;
        city : String(80);
        country : CountryAddress;
        type : String(10); // home, office
    };
    entity Person
    {
        key id : Integer;
        // address1,2,3 are to-one associations
        address1 : Association to Address;
        address2 : Association to Address { id };
        address3 : Association[1] to Address { zipCode, street, country };
        // address4,5,6 are to-many associations
        address4 : Association[0..*] to Address { zipCode };
        address5 : Association[*] to Address { street.name };
        address6 : Association[*] to Address { street.name AS streetName,
        country.name AS countryName };
    };
};
```

Cardinality in Associations

When using an association to define a relationship between entities in a CDS document, you use the **cardinality** to specify the type of relation, for example, one-to-one (to-one) or one-to-many (to-n); the relationship is with respect to both the source and the target of the association.

The target cardinality is stated in the form of `[ min .. max ]`, where `max=*` denotes infinity. If no cardinality is specified, the default cardinality setting `[ 0..1 ]` is assumed. It is possible to specify the maximum cardinality of the source of the association in the form `[ maxs, min .. max]`, too, where `maxs = *` denotes infinity.

> → Tip
>
> The information concerning the maximum cardinality is only used as a hint for optimizing the execution of the resulting JOIN.

The following examples illustrate how to express cardinality in an association definition:

```
namespace samples;
@Schema: 'MYSCHEMA'              // XS classic *only*
context AssociationCardinality {
    entity Associations {
        // To-one associations
        assoc1 : Association[0..1]    to target; // has no or one target instance
        assoc2 : Association          to target; // as assoc1, uses the default
[0..1]
        assoc3 : Association[1]       to target; // as assoc1; the default for
min is 0
        assoc4 : Association[1..1]    to target; // association has one target
instance
        // To-many associations
```

```
        assoc5 : Association[0..*]    to target{id1};
        assoc6 : Association[]        to target{id1}; // as assoc5, [] is short
for [0..*]
        assoc7 : Association[2..7]    to target{id1}; // any numbers are
possible; user provides
        assoc8 : Association[1, 0..*] to target{id1}; // additional info. about
source cardinality
    };
    // Required to make the example above work
    entity target {
        key id1 : Integer;
        key id2 : Integer;
    };
};
```

Target Entities in Associations

You use the `to` keyword in a CDS view definition to specify the target entity in an association, for example, the name of an entity defined in a CDS document. A qualified entity name is expected that refers to an existing entity. A target entity specification is mandatory; a default value is **not** assumed if no target entity is specified in an association relationship.

The entity `Address` specified as the target entity of an association could be expressed in any of the ways illustrated the following examples:

```
address1 : Association to Address;
address2 : Association to Address { id };
address3 : Association[1] to Address { zipCode, street, country };
```

Filter Conditions and Prefix Notation

When following an association (for example, in a view), it is now possible to apply a filter condition; the filter is merged into the `ON`-condition of the resulting `JOIN`. The following example shows how to get a list of customers and then filter the list according to the sales orders that are currently “open” for each customer. In the example, the infix filter is inserted after the association orders to get only those orders that satisfy the condition `[status='open']`.

Sample Code

```
view C1 as select from Customer {
  name,
  orders[status='open'].id as orderId
};
```

The association `orders` is defined in the entity definition illustrated in the following code example:

Sample Code

```
entity Customer {
   key id : Integer;
   orders : Association[*] to SalesOrder on orders.cust_id = id;
```

```
    name : String(80);
};
entity SalesOrder {
    key id : Integer;
    cust_id : Integer;
    customer: Association[1] to Customer on customer.id = cust_id;
    items : Association[*] to Item on items.order_id = id;
    status: String(20);
    date : LocalDate;
};
entity Item {
    key id : Integer;
    order_id : Integer;
    salesOrder : Association[1] to SalesOrder on salesOrder.id = order_id;
    descr : String(100);
    price : Decimal(8,2);
};
```

→ Tip

For more information about filter conditions and prefixes in CDS views, see *CDS Views* and *CDS View Syntax Options*.

Foreign Keys in Associations

For **managed** associations, the relationship between source and target entity is defined by specifying a set of elements of the target entity that are used as a foreign key. If no foreign keys are specified explicitly, the elements of the target entity's designated primary key are used. Elements of the target entity that reside inside substructures can be addressed via the respective path. If the chosen elements do not form a unique key of the target entity, the association has cardinality to-many. The following examples show how to express foreign keys in an association.

```
namespace samples;
using samples::SimpleAssociations.StreetAddress;
using samples::SimpleAssociations.CountryAddress;
using samples::SimpleAssociations.Address;
@Schema: 'MYSCHEMA'           // XS classic *only*
context ForeignKeys {
    entity Person
    {
        key id : Integer;
        // address1,2,3 are to-one associations
        address1 : Association to Address;
        address2 : Association to Address { id };
        address3 : Association[1] to Address { zipCode, street, country };
        // address4,5,6 are to-many associations
        address4 : Association[0..*] to Address { zipCode };
        address5 : Association[*] to Address { street.name };
        address6 : Association[*] to Address { street.name AS streetName,
        country.name AS countryName };
    };
    entity Header {
        key id : Integer;
        toItems : Association[*] to Item on toItems.head.id = id;
    };
    entity Item {
        key id : Integer;
        head : Association[1] to Header { id };
        // <...>
```

```
    };
};
```

- `address1`
  No foreign keys are specified: the target entity's primary key (the element `id`) is used as foreign key.
- `address2`
  Explicitly specifies the foreign key (the element `id`); this definition is similar to `address1`.
- `address3`
  The foreign key elements to be used for the association are explicitly specified, namely: `zipcode` and the structured elements `street` and `country`.
- `address4`
  Uses only `zipcode` as the foreign key. Since `zipcode` is not a unique key for entity `Address`, this association has cardinality “to-many”.
- `address5`
  Uses the subelement `name` of the structured element `street` as a foreign key. This is not a unique key and, as a result, `address4` has cardinality “to-many”.
- `address6`
  Uses the subelement `name` of both the structured elements `street` and `country` as foreign key fields. The names of the foreign key fields must be unique, so an alias is required here. The foreign key is not unique, so `address6` is a “to-many” association.

You can use foreign keys of managed associations in the definition of other associations. In the following example, the appearance of association `head` in the ON condition is allowed; the compiler recognizes that the field `head.id` is actually part of the entity `Item` and, as a result, can be obtained without following the association `head`.

Sample Code

```
entity Header {
  key id : Integer;
  toItems : Association[*] to Item on toItems.head.id = id;
};

entity Item {
  key id : Integer;
  head : Association[1] to Header { id };
  ...
};
```

Restrictions

CDS name resolution does not distinguish between CDS `associations` and CDS `entities`. If you define a CDS association with a CDS entity that has the same name as the new CDS association, CDS displays an error message and the activation of the CDS document fails.

Caution

In an CDS document, to define an association with a CDS entity of the same name, you must specify the **context** where the target entity is defined, for example, `Mycontext.Address3`.

The following code shows some examples of associations with a CDS entity that has the same (or a similar) name. Case sensitivity ("a", "A") is important; in CDS documents, `address` is not the same as `Address`. In the case of `Address2`, where the association name and the entity name are identical, the result is an error; when resolving the element names, CDS expects an entity named `Address2` but finds a CDS association with the same name instead. `MyContext.Address3` is allowed, since the target entity can be resolved due to the absolute path to its location in the CDS document.

```
context MyContext {
  entity Address  {…}
  entity Address1 {…}
  entity Address2 {…}
  entity Address3 {…}
  entity Person
  {
    key id   : Integer;
    address  : Association to Address;        // OK: "address" ≠ "Address”
    address1 : Association to Address1;       // OK: "address1" ≠ "Address1”
    Address2 : Association to Address2;       // Error: association name =
entity name
    Address3 : Association to MyContext.Address3;  //OK: full path to Address3

  };
};
```

Example: Complex (One-to-Many) Association

The following example shows a more complex association (to-many) between the entity “`Header`” and the entity “`Item`”.

```
namespace samples;
@Schema: 'MYSCHEMA'        // XS classic *only*
context ComplexAssociation {
    Entity Header {
        key PurchaseOrderId: BusinessKey;
        Items: Association [0..*] to Item on
Items.PurchaseOrderId=PurchaseOrderId;
        "History": HistoryT;
        NoteId: BusinessKey null;
        PartnerId: BusinessKey;
        Currency: CurrencyT;
        GrossAmount: AmountT;
        NetAmount: AmountT;
        TaxAmount: AmountT;
        LifecycleStatus: StatusT;
        ApprovalStatus: StatusT;
        ConfirmStatus: StatusT;
        OrderingStatus: StatusT;
        InvoicingStatus: StatusT;
    } technical configuration {
        column store;
    };
    Entity Item {
        key PurchaseOrderId: BusinessKey;
        key PurchaseOrderItem: BusinessKey;
        ToHeader: Association [1] to Header on
ToHeader.PurchaseOrderId=PurchaseOrderId;
        ProductId: BusinessKey;
        NoteId: BusinessKey null;
        Currency: CurrencyT;
        GrossAmount: AmountT;
```

```
        NetAmount: AmountT;
        TaxAmount: AmountT;
        Quantity: QuantityT;
        QuantityUnit: UnitT;
        DeliveryDate: SDate;
    } technical configuration {
        column store;
    };
    define view POView as SELECT from Header {
        Items.PurchaseOrderId as poId,
        Items.PurchaseOrderItem as poItem,
        PartnerId,
        Items.ProductId
    };
    // Missing types from the example above
    type BusinessKey: String(50);
    type HistoryT: LargeString;
    type CurrencyT: String(3);
    type AmountT: Decimal(15, 2);
    type StatusT: String(1);
    type QuantityT: Integer;
    type UnitT: String(5);
    type SDate: LocalDate;
};
```

Related Information

Create an Association in CDS [page 76]
Create a CDS Association in XS Advanced [page 217]

3.6.2 CDS Association Syntax Options

Associations define relationships between entities.

Example: Managed Associations

```
Association [ <cardinality> ]  to <targetEntity> [ <forwardLink> ]
```

Example: Unmanaged Associations

```
Association [ <cardinality> ]  to <targetEntity> <unmanagedJoin>
```

Overview

Associations are specified by adding an element to a source entity with an association **type** that points to a target entity, complemented by optional information defining cardinality and which keys to use.

> **i** Note
>
> CDS supports both managed and unmanaged associations.

SAP HANA Extended Application Services (SAP HANA XS) enables you to use associations in the definition of a CDS entity or a CDS view. When defining an association, bear in mind the following points:

- `<Cardinality>` [page 85]
  The relationship between the source and target in the association, for example, one-to-one, one-to-many, many-to-one
- `<targetEntity>` [page 87]
  The target entity for the association
- `<forwardLink>` [page 87]
  The foreign keys to use in a managed association, for example, element names in the target entity
- `<unmanagedJoin>` [page 89]
  Unmanaged associations only; the `ON` condition specifies the elements of the source and target elements and entities to use in the association

Association Cardinality

When using an association to define a relationship between entities in a CDS view; you use the **cardinality** to specify the type of relation, for example:

- one-to-one (to-one)
- one-to-many (to-n)

The relationship is with respect to both the source and the target of the association. The following code example illustrates the syntax required to define the cardinality of an association in a CDS view:

```
[  [ ( maxs | * ) , ]                  // source cardinality
   [ min .. ] ( max | * )              // target cardinality
]
```

In the most simple form, only the target cardinality is stated using the syntax `[ min .. max ]`, where `max=*` denotes infinity. Note that `[]` is short for `[ 0..* ]`. If no cardinality is specified, the default cardinality setting `[ 0..1 ]` is assumed. It is possible to specify the maximum cardinality of the source of the association in the form `[ maxs, min .. max]`, where `maxs = *` denotes infinity.

The following examples illustrate how to express cardinality in an association definition:

```
namespace samples;
@Schema: 'MYSCHEMA'              // XS classic *only*
context AssociationCardinality {
    entity Associations {
        // To-one associations
        assoc1 : Association[0..1]    to target;
        assoc2 : Association          to target;
```

```
        assoc3 : Association[1]       to target;
        assoc4 : Association[1..1]    to target; // association has one target
instance
        // To-many associations
        assoc5 : Association[0..*]    to target{id1};
        assoc6 : Association[]        to target{id1}; // as assoc4, [] is short
for [0..*]
        assoc7 : Association[2..7]    to target{id1}; // any numbers are
possible; user provides
        assoc8 : Association[1, 0..*] to target{id1}; // additional info. about
source cardinality
    };
    // Required to make the example above work
    entity target {
        key id1 : Integer;
        key id2 : Integer;
    };
};
```

The following table describes the various cardinality expressions illustrated in the example above:

Association Cardinality Syntax Examples

Association	Cardinality	Explanation
`assoc1`	[0..1]	The association has no or one target instance
`assoc2`		Like `assoc1`, this association has no or one target instance and uses the default [0..1]
`assoc3`	[1]	Like `assoc1`, this association has no or one target instance; the default for min is 0
`assoc4`	[1..1]	The association has one target instance
`assoc5`	[0..*]	The association has no, one, or multiple target instances
`assoc6`	[]	Like `assoc4`, [] is short for [0..*] (the association has no, one, or multiple target instances)
`assoc7`	[2..7]	Any numbers are possible; the user provides
`assoc8`	[1, 0..*]	The association has no, one, or multiple target instances and includes additional information about the source cardinality

When an infix filter effectively reduces the cardinality of a “to-N” association to “to-1”, this can be expressed explicitly in the filter, for example:

```
assoc[1: <cond> ]
```

Specifying the cardinality in the filter in this way enables you to use the association in the `WHERE` clause, where “to-N” associations are not normally allowed.

Sample Code

```
namespace samples;
@Schema: 'MYSCHEMA'             // XS classic *only*
context CardinalityByInfixFilter {
    entity Person {
        key id : Integer;
        name : String(100);
        address : Association[*] to Address on address.personId = id;
    };
    entity Address {
```

```
        key id : Integer;
        personId : Integer;
        type : String(20); // home, business, vacation, ...
        street : String(100);
        city : String(100);
    };
    view V as select from Person {
        name
    } where address[1: type='home'].city = 'Accra';
};
```

Association Target

You use the `to` keyword in a CDS view definition to specify the target entity in an association, for example, the name of an entity defined in a CDS document. A qualified entity name is expected that refers to an existing entity. A target entity specification is mandatory; a default value is **not** assumed if no target entity is specified in an association relationship.

```
Association[ <cardinality> ]  to <targetEntity> [ <forwardLink> ]
```

The target entity `Address` specified as the target entity of an association could be expressed as illustrated the following examples:

```
address1 : Association to Address;
address2 : Association to Address { id };
address3 : Association[1] to Address { zipCode, street, country };
```

Association Keys

In the relational model, associations are mapped to foreign-key relationships. For **managed** associations, the relation between source and target entity is defined by specifying a set of elements of the target entity that are used as a foreign key, as expressed in the `forwardLink` element of the following code example:

```
Association[ <cardinality> ]  to <targetEntity> [ <forwardLink> ]
```

The `forwardLink` element of the association could be expressed as follows:

```
<forwardLink>  =  { <foreignKeys> }
<foreignKeys>  = <targetKeyElement> [ AS <alias> ] [ , <foreignKeys> ]
<targetKeyElement> = <elementName> ( . <elementName> )*
```

If no foreign keys are specified explicitly, the elements of the target entity's designated primary key are used. Elements of the target entity that reside inside substructures can be addressed by means of the respective path. If the chosen elements do not form a unique key of the target entity, the association has cardinality to-many. The following examples show how to express foreign keys in an association.

```
entity Person
{
  key id : Integer;
  // address1,2,3 are to-one associations
```

```
  address1 : Association to Address;
  address2 : Association to Address { id };
  address3 : Association[1] to Address { zipCode, street, country };
  // address4,5,6 are to-many associations
  address4 : Association[0..*] to Address { zipCode };
  address5 : Association[*] to Address { street.name };
  address6 : Association[*] to Address { street.name AS streetName,
                                         country.name AS countryName };
};
```

Association Syntax Options

Association	Keys	Explanation
`address1`		No foreign keys are specified: the target entity's primary key (the element `id`) is used as foreign key.
`address2`	`{ id }`	Explicitly specifies the foreign key (the element `id`); this definition is identical to `address1`.
`address3`	`{ zipCode, street, country }`	The foreign key elements to be used for the association are explicitly specified, namely: `zipcode` and the structured elements `street` and `country`.
`address4`	`{ zipCode }`	Uses only `zipcode` as the foreign key. Since `zipcode` is not a unique key for entity `Address`, this association has cardinality “to-many”.
`address5`	`{ street.name }`	Uses the sub-element `name` of the structured element `street` as a foreign key. This is not a unique key and, as a result, `address4` has cardinality “to-many”.
`address6`	`{ street.name AS streetName, country.name AS countryName }`	Uses the sub-element `name` of both the structured elements `street` and `country` as foreign key fields. The names of the foreign key fields must be unique, so an alias is required here. The foreign key is not unique, so `address6` is a “to-many” association.

You can now use foreign keys of managed associations in the definition of other associations. In the following example, the compiler recognizes that the field `toCountry.cid` is part of the foreign key of the association `toLocation` and, as a result, physically present in the entity `Company`.

Sample Code

```
namespace samples;
@Schema: 'MYSCHEMA'         // XS classic *only*
context AssociationKeys {
    entity Country {
        key c_id : String(3);
        // <...>
    };
    entity Region {
        key r_id : Integer;
        key toCountry : Association[1] to Country { c_id };
        // <...>
    };
    entity Company {
        key id : Integer;
        toLocation : Association[1] to Region { r_id, toCountry.c_id };
        // <...>
    };
};
```

Unmanaged Associations

Unmanaged associations are based on existing elements of the source and target entity; no fields are generated. In the `ON` condition, only elements of the source or the target entity can be used; it is not possible to use other associations. The `ON` condition may contain any kind of expression - all expressions supported in views can also be used in the `ON` condition of an unmanaged association.

> **i** Note
>
> The names in the `ON` condition are resolved in the scope of the source entity; elements of the target entity are accessed through the association itself .

In the following example, the association `inhabitants` relates the element `id` of the source entity `Room` with the element `officeId` in the target entity `Employee`. The target element `officeId` is accessed through the name of the association itself.

```
namespace samples;
@Schema: 'MYSCHEMA'              // XS classic *only*
context UnmanagedAssociations {
    entity Employee {
        key id : Integer;
        officeId : Integer;
        // <...>
    };
    entity Room {
        key id : Integer;
        inhabitants : Association[*] to Employee on inhabitants.officeId = id;
        // <...>
    };
    entity Thing {
        key id : Integer;
        parentId : Integer;
        parent : Association[1] to Thing on parent.id = parentId;
        children : Association[*] to Thing on children.parentId = id;
        // <...>
    };
};
```

The following example defines two related **unmanaged** associations:

- `parent`
  The unmanaged association `parent` uses a cardinality of `[1]` to create a relation between the element `parentId` and the target element `id`. The target element `id` is accessed through the name of the association itself.
- `children`
  The unmanaged association `children` creates a relation between the element `id` and the target element `parentId`. The target element `parentId` is accessed through the name of the association itself.

```
entity Thing {
  key id   : Integer;
  parentId : Integer;
  parent   : Association[1] to Thing on parent.id = parentId;
  children : Association[*] to Thing on children.parentId = id;
  ...
};
```

Constants in Associations

The usage of constants is no longer restricted to annotation assignments and default values for entity elements. With SPS 11, you can use constants in the "`ON`"-condition of unmanaged associations, as illustrated in the following example:

Sample Code

```
context MyContext {
  const MyIntConst      : Integer      = 7;
  const MyStringConst   : String(10)   = 'bright';
  const MyDecConst      : Decimal(4,2) = 3.14;
  const MyDateTimeConst : UTCDateTime  = '2015-09-30 14:33';
  entity MyEntity {
    key id : Integer;
    a : Integer;
    b : String(100);
    c : Decimal(20,10);
    d : UTCDateTime;
    your : Association[1] to YourEntity on your.a - a < MyIntConst;
  };
  entity YourEntity {
    key id : Integer;
    a : Integer;
  };
  entity HerEntity {
    key id : Integer;
    t : String(20);
  };
  view MyView as select from MyEntity
           inner join HerEntity on locate (b, :MyStringConst) > 0
  {
    a + :MyIntConst as x,
    b || ' is ' || :MyStringConst as y,
    c * sin(:MyDecConst) as z
  } where d < :MyContext.MyDateTimeConst;
};
```

Related Information

Create a CDS Association in XS Advanced [page 217]
CDS Associations [page 78]

3.7 Create a View in CDS

A view is a virtual table based on the dynamic results returned in response to an SQL statement. SAP HANA Extended Application Services (SAP HANA XS) enables you to use CDS syntax to create a database view as a design-time file in the repository.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have already created a development workspace and a project.
- You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema for the CDS catalog objects, for example, `MYSCHEMA`
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.

Context

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the CDS syntax to create a database view as a design-time file in the repository. Repository files are transportable. Activating the CDS view definition creates the corresponding catalog object in the specified schema. To create a CDS view-definition file in the repository, perform the following steps:

> **i** Note
>
> The following code examples are provided for illustration purposes only.

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS-definition file which will contain the view you define in the following steps.

   Browse to the folder in your project workspace where you want to create the new CDS-definition file and perform the following steps:

   a. Right-click the folder where you want to save the view-definition file and choose *New* > *Other...* > *Database Development* > *DDL Source File* in the context-sensitive pop-up menu.

b. Enter the name of the view-definition file in the *File Name* box, for example, `MyModel2`.

> → Tip
>
> File extensions are important. If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically (for example, `MyModel2.hdbdd`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

c. Choose *Finish* to save the changes and commit the new CDS definition file in the repository.

5. Define the underlying CDS entities and structured types.

   If the new entity-definition file is not automatically displayed by the file-creation wizard, in the *Project Explorer* view double-click the entity-definition file you created in the previous step, for example, `MyModel2.hdbdd`, and add the code for the entity definitions and structured types to the file.

```
namespace com.acme.myapp1;
@Schema : 'MYSCHEMA'
context MyModel2 {
 type StreetAddress {
   name   : String(80);
   number : Integer;
 };
 type CountryAddress {
   name : String(80);
   code : String(3);
 };
 @Catalog.tableType : #COLUMN
 entity Address {
   key id  : Integer;
   street  : StreetAddress;
   zipCode : Integer;
   city    : String(80);
   country : CountryAddress;
   type    : String(10); // home, office
 };
};
```

6. Define a view as a projection of a CDS entity.

   In the same entity-definition file you edited in the previous step, for example, `MyModel2.hdbdd`, add the code for the view `AddressView` below the entity `Address` in the CDS document.

> i Note
>
> In CDS, a view is an entity without an its own persistence; it is defined as a projection of other entities.

```
view AddressView as select from Address
{
  id,
  street.name,
  street.number
};
```

7. Save the CDS-definition file containing the new view.

> i Note
>
> Saving a file in a shared project automatically commits the saved version of the file to the repository; you do not need to explicitly commit the file again.

8. Activate the changes in the repository.
   a. Locate and right-click the new CDS-definition file in the *Project Explorer* view.
   b. In the context-sensitive pop-up menu, choose *Team* > *Activate*.

      > **i Note**
      >
      > If you cannot activate the new CDS artifact, check that the specified schema already exists and that there are no illegal characters in the name space, for example, the hyphen (-).

9. Ensure access to the schema where the new CDS catalog objects are created.

   After activation in the repository, a schema object is only visible in the catalog to the _SYS_REPO user. To enable other users, for example the schema owner, to view the newly created schema and the objects it contains, you must grant the user the required SELECT privilege.

   > **i Note**
   >
   > If you already have the appropriate `SELECT` privilege, you do not need to perform this step.

   a. In the SAP HANA studio *Systems* view, right-click the SAP HANA system hosting the repository where the schema was activated and choose *SQL Console* in the context-sensitive popup menu.
   b. In the *SQL console*, execute the statement illustrated in the following example, where `<SCHEMANAME>` is the name of the newly activated schema, and `<username>` is the database user ID of the schema owner:

      ```
      call
      _SYS_REPO.GRANT_SCHEMA_PRIVILEGE_ON_ACTIVATED_CONTENT('select','<SCHEMANAME
      >','<username>');
      ```

10. Check that the new view has been successfully created.

    Views are created in the `Views` folder in the catalog.

    a. In the *SAP HANA Development* perspective, open the *Systems* view.
    b. Navigate to the catalog location where you created the new view.

       <SID> > *Catalog* > *<MYSCHEMA>* > *Views*

    c. Open a data preview for the new view `AddressView`.

       Right-click the new view `<package.path>::MyModel2.AddressView` and choose *Open Data Preview* in the pop-up menu.

Related Information

CDS Views [page 94]
CDS View Syntax Options [page 96]
Spatial Types and Functions [page 112]

3.7.1 CDS Views

A view is an entity that is not persistent; it is defined as the projection of other entities. SAP HANA Extended Application Services (SAP HANA XS) enables you to create a CDS view as a design-time file in the repository.

SAP HANA Extended Application Services (SAP HANA XS) enables you to define a view in a CDS document, which you store as design-time file in the repository. Repository files can be read by applications that you develop. In addition, all repository files including your view definition can be transported to other SAP HANA systems, for example, in a delivery unit.

If your application refers to the design-time version of a view from the repository rather than the runtime version in the catalog, for example, by using the explicit path to the repository file (with suffix), any changes to the repository version of the file are visible as soon as they are committed to the repository. There is no need to wait for the repository to activate a runtime version of the view.

To define a transportable view using the CDS-compliant view specifications, use something like the code illustrated in the following example:

```
context Views {
    VIEW AddressView AS SELECT FROM Address {
        id,
        street.name,
        street.number
    };
<...>
}
```

When a CDS document is activated, the activation process generates a corresponding catalog object for each of the artifacts defined in the document; the location in the catalog is determined by the type of object generated. For example, in SAP HANA XS classic the corresponding catalog object for a CDS view definition is generated in the following location:

<SID> > *Catalog* > <MYSCHEMA> > *Views*

Views defined in a CDS document can make use of the following SQL features:

- CDS Type definition
- Expressions and functions (for example, "`a + b as theSum`")
- Aggregates, "GROUP BY", and "HAVING" clauses
- Associations (including filters and prefixes)
- ORDER BY, CASE, UNION, JOIN, and TOP
- With Parameters
- Select Distinct
- Spatial Data (for example, "`ST_Distance`")

> → Tip
>
> For more information about the syntax required when using these SQL features in a CDS view, see *CDS View Syntax Options* in *Related Information*.

Type Definition

In a CDS view definition, you can explicitly specify the `type` of a select item, as illustrated in the following example:

Sample Code

```
type MyInteger : Integer;
entity E {
  a : MyInteger;
  b : MyInteger;
};
view V as select from E {
  a,
  a+b as s1,
  a+b as s2 : MyInteger
};
```

In the example of different `type` definitions, the following is true:

- `a,`
  Has type "`MyInteger`"
- `a+b as s1,`
  Has type "`Integer`" and any information about the user-defined `type` is lost
- `a+b as s2 : MyInteger`
  Has type "`MyInteger`", which is explicitly specified

Note

If necessary, a `CAST` function is added to the generated view in SAP HANA; this ensures that the `select` item's type in the SAP HANA view is the SAP HANA "type" corresponding to the explicitly specified CDS type.

Related Information

Create a CDS View in XS Advanced [page 232]
CDS View Syntax Options [page 96]
CDS Associations [page 78]

3.7.2 CDS View Syntax Options

SAP HANA XS includes a dedicated, CDS-compliant syntax, which you must adhere to when using a CDS document to define a view as a design-time artifact.

Example

> **i Note**
>
> The following example is intended for illustration purposes only and might contain syntactical errors. For further details about the keywords illustrated, click the links provided.

```
context views {
const x : Integer = 4;
const y : Integer = 5;
const Z : Integer = 6;
VIEW MyView1 AS SELECT FROM Employee
{
  a + b  AS theSum
};
VIEW MyView2 AS SELECT FROM Employee
{ officeId.building,
  officeId.floor,
  officeId.roomNumber,
  office.capacity,
  count(id) AS seatsTaken,
  count(id)/office.capacity as occupancyRate
} WHERE officeId.building = 1
  GROUP BY officeId.building,
           officeId.floor,
           officeId.roomNumber,
           office.capacity,
           office.type
  HAVING office.type = 'office' AND count(id)/office.capacity < 0.5;
VIEW MyView3 AS SELECT FROM Employee
{ orgUnit,
  salary
} ORDER BY salary DESC;
VIEW MyView4 AS SELECT FROM Employee {
    CASE
        WHEN a < 10 then 'small'
        WHEN 10 <= a AND a < 100 THEN 'medium'
        ELSE 'large'
    END AS size
};
VIEW MyView5 AS
  SELECT FROM E1 { a, b, c}
  UNION
  SELECT FROM E2 { z, x, y};
VIEW MyView6 AS SELECT FROM Customer {
  name,
  orders[status='open'].{ id   as orderId,
                          date as orderDate,
                          items[price>200].{ descr,
                                             price }  }
};
VIEW MyView7 as
  select from E { a, b, c}
  order by a limit 10 offset 30;
```

```
VIEW V_join as select from E join (F as X full outer join G on X.id = G.id) on
E.id = c {
  a, b, c
};
VIEW V_top as select from E TOP 10 { a, b, c};
VIEW V_dist as select from E distinct { a };
VIEW V_param with parameters PAR1: Integer, PAR2: MyUserDefinedType, PAR3: type
of E.elt
      as select from MyEntity {
        id,
        elt };
VIEW V_type as select from E {
  a,
  a+b as s1,
  a+b as s2 : MyInteger
};
view VE as select from E mixin {
  f : Association[1] to VF on f.vy = $projection.vb;
 } into {
  a as va,
  b as vb,
  f as vf
 };
VIEW SpatialView1 as select from Person {
    name,
    homeAddress.street_name   || ', ' || homeAddress.city   as home,
    officeAddress.street_name || ', ' || officeAddress.city as office,
    round( homeAddress.loc.ST_Distance(officeAddress.loc, 'meter')/1000, 1) as
distanceHomeToWork,
    round( homeAddress.loc.ST_Distance(NEW ST_POINT(8.644072, 49.292910),
'meter')/1000, 1) as distFromSAP03
};
}
```

Expressions and Functions

In a CDS view definition you can use any of the functions and expressions listed in the following example:

```
View MyView9 AS SELECT FROM SampleEntity
{
  a + b  AS theSum,
  a - b  AS theDifference,
  a * b  AS theProduct,
  a / b  AS theQuotient,
  -a     AS theUnaryMinus,
  c || d AS theConcatenation
};
```

> **Note**
>
> When expressions are used in a view element, an alias must be specified, for example, `AS theSum`.

Aggregates

In a CDS view definition, you can use the following aggregates:

- AVG
- COUNT
- MIN
- MAX
- SUM
- STDDEV
- VAR

The following example shows how to use aggregates and expressions to collect information about headcount and salary per organizational unit for all employees hired from 2011 to now.

```
VIEW MyView10 AS SELECT FROM Employee
{
  orgUnit,
  count(id)   AS headCount,
  sum(salary) AS totalSalary,
  max(salary) AS maxSalary
}
  WHERE joinDate > date'2011-01-01'
  GROUP BY orgUnit;
```

> **i** Note
>
> Expressions are not allowed in the `GROUP BY` clause.

Constants in Views

With SPS 11, you can use constants in the views, as illustrated in “MyView” at the end of the following example:

> Sample Code

```
context MyContext {
  const MyIntConst      : Integer      = 7;
  const MyStringConst   : String(10)   = 'bright';
  const MyDecConst      : Decimal(4,2) = 3.14;
  const MyDateTimeConst : UTCDateTime  = '2015-09-30 14:33';
  entity MyEntity {
    key id : Integer;
    a : Integer;
    b : String(100);
    c : Decimal(20,10);
    d : UTCDateTime;
    your : Association[1] to YourEntity on your.a - a < MyIntConst;
  };
  entity YourEntity {
    key id : Integer;
    a : Integer;
  };
  entity HerEntity {
    key id : Integer;
    t : String(20);
```

```
  };
  view MyView as select from MyEntity
           inner join HerEntity on locate (b, :MyStringConst) > 0
  {
    a + :MyIntConst as x,
    b || ' is ' || :MyStringConst as y,
    c * sin(:MyDecConst) as z
  } where d < :MyContext.MyDateTimeConst;
};
```

When constants are used in a view definition, their name must be prefixed with the scope operator "`:`". Usually names that appear in a query are resolved as alias or element names. The scope operator instructs the compiler to resolve the name outside of the query.

Sample Code

```
context NameResolution {
  const a : Integer = 4;
  const b : Integer = 5;
  const c : Integer = 6;
  entity E {
    key id : Integer;
    a : Integer;
    c : Integer;
  };
  view V as select from E {
    a    as a1,
    b,
    :a   as a2,
    E.a  as a3,
    :E,
    :E.a as a4,
    :c
  };
}
```

The following table explains how the constants used in view "V" are resolved.

Constant Declaration and Result

Constant Expression	Result	Comments
`a as a1,`	Success	"a" is resolved in the space of alias and element names, for example, element "a" of entity "E".
`b,`	Error	There is no alias and no element with name "b" in entity "E"
`:a as a2,`	Success	Scope operator ":" instructs the compiler to search for element "a" outside of the query (finds the constant "a").
`E.a as a3,`	Success	"E" is resolved in the space of alias and element names, so this matches element "a" of entity "Entity" .
`:E,`	Error	Error: no access to "E" via ":"
`:E.a as a4,`	Error	Error; no access to "E" (or any of its elements) via ":"

Constant Expression	Result	Comments
`:c`	Error	Error: there is no alias for “c”.

SELECT

In the following example of an association in a `SELECT` list, a view compiles a list of all employees; the list includes the employee's name, the capacity of the employee's office, and the color of the carpet in the office. The association follows the to-one association office from entity `Employee` to entity `Room` to collect the relevant information about the office.

```
VIEW MyView11 AS SELECT FROM Employee
{
  name.last,
  office.capacity,
  office.carpetColor
};
```

Subqueries

You can define subqueries in a CDS view, as illustrated in the following example:

! Restriction

For use in XS advanced only; subqueries are not supported in XS classic

Code Syntax

```
select from (select from F {a as x, b as y}) as Q {
  x+y as xy,
    (select from E {a} where b=Q.y) as a
} where x < all (select from E{b})
```

i Note

In a **correlated** subquery, elements of outer queries must always be addressed by means of a table alias.

WHERE

The following example shows how the syntax required in the `WHERE` clause used in a CDS view definition. In this example, the `WHERE` clause is used in an association to restrict the result set according to information located in the association's target. Further filtering of the result set can be defined with the `AND` modifier.

```
VIEW EmployeesInRoom_ABC_3_4 AS SELECT FROM Employee
{
  name.last
} WHERE officeId.building = 'ABC'
    AND officeId.floor    = 3
```

```
    AND officeId.number   = 4;
```

FROM

The following example shows the syntax required when using the FROM clause in a CDS view definition. This example shows an association that lists the license plates of all company cars.

```
VIEW CompanyCarLicensePlates AS SELECT FROM Employee.companyCar
{
  licensePlate
};
```

In the `FROM` clause, you can use the following elements:

- an entity or a view defined in the same CDS source file
- a native SAP HANA table or view that is available in the schema specified in the schema annotation (`@Schema` in the corresponding CDS document)

If a CDS view references a native SAP HANA table, the table and column names must be specified using their effective SAP HANA names.

```
create table foo (
  bar    : Integer,
  "gloo" : Integer
)
```

This means that if a table (`foo`) or its columns (`bar` and `"gloo"` were created **without** using quotation marks (""), the corresponding uppercase names for the table or columns must be used in the CDS document, as illustrated in the following example.

```
VIEW MyViewOnNative as SELECT FROM FOO
{
  BAR,
  gloo
};
```

GROUP BY

The following example shows the syntax required when using the `GROUP BY` clause in a CDS view definition. This example shows an association in a view that compiles a list of all offices that are less than 50% occupied.

```
VIEW V11 AS SELECT FROM Employee
{
  officeId.building,
  officeId.floor,
  officeId.roomNumber,
  office.capacity,
  count(id) as seatsTaken,
  count(id)/office.capacity as occupancyRate
} GROUP BY officeId.building,
           officeId.floor,
           officeId.roomNumber,
```

```
            office.capacity,
            office.type
  HAVING office.type = 'office' AND count(id)/capacity < 0.5;
```

HAVING

The following example shows the syntax required when using the `HAVING` clause in a CDS view definition. This example shows a view with an association that compiles a list of all offices that are less than 50% occupied.

```
VIEW V11 AS SELECT FROM Employee
{
  officeId.building,
  officeId.floor,
  officeId.roomNumber,
  office.capacity,
  count(id)          as seatsTaken,
  count(id)/office.capacity as occupancyRate
} GROUP BY officeId.building,
            officeId.floor,
            officeId.roomNumber,
            office.capacity,
            office.type
  HAVING office.type = 'office' AND count(id)/capacity < 0.5;
```

ORDER BY

The `ORDER BY` operator enables you to list results according to an expression or position, for example `salary`.

```
VIEW MyView3 AS SELECT FROM Employee
{
  orgUnit,
  salary
} ORDER BY salary DESC;
```

In the same way as with plain SQL, the `ASC` and `DESC` operators enable you to sort the list order as follows.

- `ASC`
  Display the result set in **ascending** order
- `DESC`
  Display the result set in **descending** order

LIMIT/OFFSET

You can use the SQL clauses `LIMIT` and `OFFSET` in a CDS query. The `LIMIT <INTEGER> [OFFSET <INTEGER>]` operator enables you to restrict the number of output records to display to a specified “limit”; the `OFFSET <INTEGER>` specifies the number of records to skip before displaying the records according to the defined `LIMIT`.

```
VIEW MyViewV AS SELECT FROM E
```

```
  { a, b, c}
  order by a limit 10 offset 30;
```

CASE

In the same way as in plain SQL, you can use the `case` expression in a CDS view definition to introduce `IF-THEN-ELSE` conditions without the need to use procedures.

```
entity MyEntity12 {
key id : Integer;
     a : Integer;
     color : String(1);
};

VIEW MyView12 AS SELECT FROM MyEntity12 {
    id,
    CASE color     // defined in MyEntity12
        WHEN 'R' THEN 'red'
        WHEN 'G' THEN 'green'
        WHEN 'B' THEN 'blue'
        ELSE 'black'
    END AS color,
    CASE
        WHEN a < 10 then 'small'
        WHEN 10 <= a AND a < 100 THEN 'medium'
        ELSE 'large'
    END AS size
};
```

In the first example of usage of the `CASE` operator, `CASE color` shows a “switched” CASE (one table column and multiple values). The second example of CASE usage shows a “conditional” CASE with multiple arbitrary conditions, possibly referring to different table columns.

UNION

Enables multiple select statements to be combined but return only one result set. `UNION` works in the same way as the SAP HANA SQL command of the same name; it selects all unique records from all select statements by removing duplicates found from different select statements.The signature of the result view is equal to the signature of the first `SELECT` in the union.

> **i** Note
>
> View `MyView5` has elements `a`, `b`, and `c`.

```
entity E1 {
  key a : Integer;
  b : String(20);
  c : LocalDate;
};
entity E2 {
  key x : String(20);
  y : LocalDate;
  z : Integer;
```

```
};
VIEW MyView5 AS
  SELECT FROM E1 { a, b, c}
  UNION
  SELECT FROM E2 { z, x, y};
```

JOIN

You can include a JOIN clause in a CDS view definition; the following JOIN types are supported:

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

The following example shows a simple join.

Sample Code

```
entity E {
  key id : Integer;
  a : Integer;
};
entity F {
  key id : Integer;
  b : Integer;
};
entity G {
  key id : Integer;
  c : Integer;
};
view V_join as select from E join (F as X full outer join G on X.id = G.id)
on E.id = c {
  a, b, c
};
```

TOP

You can use the SQL clause `TOP` in a CDS query, as illustrated in the following example:

Sample Code

```
view V_top as select from E TOP 10 { a, b, c};
```

Restriction

It is not permitted to use `TOP` in combination with the `LIMIT` clause in a CDS query.

SELECT DISTINCT

CDS now supports the `SELECT DISTINCT` semantic, which enables you to specify that only one copy of each set of duplicate records **selected** should be returned. The position of the `DISTINCT` keyword is important; it must appear directly in front of the curly brace, as illustrated in the following example:

Sample Code

```
entity E {
  key id : Integer;
  a : Integer;
};
entity F {
  key id : Integer;
  b : Integer;
};
entity G {
  key id : Integer;
  c : Integer;
};
view V_dist as select from E distinct { a };
```

With Parameters

You can define parameters for use in a CDS view; this allows you to pass additional values to modify the results of the query at run time. Parameters must be defined in the view definition before the query block, as illustrated in the following example:

Restriction

For use in XS advanced only; views with parameters are not supported in XS classic.

Sample Code

Parameters in a CDS View

```
context MyContext
{
  entity MyEntity1 {
     id: Integer;
     elt: String(100); };
  entity MyEntity2 {
     id: Integer;
     elt: String(100); };
  type MyUserDefinedType: type of E.elt;
  view MyParamView with parameters PAR1: Integer,
                                   PAR2: MyUserDefinedType,
                                   PAR3: type of E.elt
      as select from MyEntity {
        id,
        elt };
```

Note

Keywords are case insensitive.

Parameters in View Queries

Parameters can be used in a query at any position where an expression is allowed. A parameter is referred to inside a query by prefixing the parameter name either with the colon Scope operator ':' or the string “`$parameters`” .

→ Tip

If no matching parameter can be found, the scope operator “escapes” from the query and attempts to resolve the identifier outside the query.

Sample Code

Using Parameters in a View Query

```
view ExampleView with parameters PAR1: Integer,
                                 PAR2: UserDefinedType,
                                 PAR3: type of E.elt
        as select from SomeEntity
             left outer join SomeOtherEntity
               on SomeEntity.id < SomeOtherEntity.id + :PAR1
     {
       id + :PAR1 as idWithOffset,
       elt,
       :PAR1,
       $parameters.PAR3
     } where elt != $parameters.PAR2;
```

Invoking a View with Parameters

Parameters are passed to views as a comma-separated list in parentheses. Optional filter expressions must then follow the parameter list.

! Restriction

It is not allowed to use a query as value expression. Nor is it allowed to provide a parameter list in the `ON` condition of an association definition to a parameterized view. This is because the association definition establishes the relationship between the two entities but makes no assumptions about the run-time conditions. For the same reason, it is not allowed to specify filter conditions in those `ON` conditions.

The following example shows two entities `SourceEntity` and `TargetEntity` and a parameterized view `TargetWindowView`, which selects from `TargetEntity`. An association is established between `SourceEntity` and `TargetEntity`.

Sample Code

```
entity SourceEntity {
   id: Integer;
   someElementOfSourceEntity: String(100);
   toTargetViaParamView: association to TargetWindowView on
      toTargetViaParamView.targetId = id;
   };
entity TargetEntity {
   targetId: Integer;
   someElementOfTargetEntity: String(100);
   };

view TargetWindowView with parameters LOWER_LIMIT: Integer
   as select from TargetEntity {
```

```
        targetId,
        someElementOfTargetEntity
    } where targetId > :LOWER_LIMIT
      and targetId <= :LOWER_LIMIT + 10;
```

It is now possible to query `SourceEntity` in a view; it is also possible to follow the association to `TargetWindowView`, for example, by providing the required parameters, as illustrated in the following example:

Sample Code

Query a Parameterized CDS View (with Association)

```
view SourceConsumption with parameters CUSTOMER_ID: Integer
   as select from SourceEntity {
      someElementOfSourceEntity,
      toTargetViaParamView(LOWER_LIMIT:
$parameters.CUSTOMER_ID).someElementOfTargetEntity
   };
```

It is also possible to follow the association in the `FROM` clause; this provides access only to the elements of the target artifact:

Sample Code

Follow an Association in the FROM Clause

```
view ConsumptionView with parameters CUSTOMER_ID: Integer
   as select from SourceEntity.toTargetViaParamView(LOWER_LIMIT: :CUSTOMER_ID)
   {
      id,
      someElementOfTargetEntity
   };
```

You can select directly from the view with parameters, adding a free `JOIN` expression, as illustrated in the following example:

Sample Code

Select from a Parameterized View with JOIN Expression

```
view ConsumptionView with parameters CUSTOMER_ID: Integer
   as select from TargetWindowView(LOWER_LIMIT: :CUSTOMER_ID) as TWV_ALIAS
      RIGHT OUTER JOIN ... ON TWV_ALIAS.targetId ....
{
   ...
};
```

Annotations in Parameter Definitions

Parameter definitions can be annotated in the same way as any other artifact in CDS; the annotations must be prepended to the parameter name. Multiple annotations are separated either by whitespace or new-line characters.

→ Tip

To improve readability and comprehension, it is recommended to include only one annotation assignment per line.

In the following example, the view `TargetWindowView` selects from the entity `TargetEntity`; the annotation `@positiveValuesOnly` is not checked; and the `targetId` is required for the `ON` condition in the entity `SourceEntity`.

Sample Code

Annotation Assignments to Parameter Definitions in CDS Views

```
annotation remark: String(100);

view TargetWindowView with parameters
   @remark: 'This is an arbitrary annotation'
   @positiveValuesOnly: true
   LOWER_LIMIT: Integer
as select from TargetEntity
{
  targetId,
  ....
} where targetId > :LOWER_LIMIT and targetId <= :LOWER_LIMIT + 10;
```

Associations, Filters, and Prefixes

You can define an association as a view element, for example, by defining an ad-hoc association in the `mixin` clause and then adding the association to the `SELECT` list, as illustrated in the following example:

! Restriction

XS classic does not support the use of ad-hoc associations in a view's `SELECT` list.

Sample Code

Associations as View Elements

```
entity E {
  a : Integer;
  b : Integer;
 };
entity F {
  x : Integer;
  y : Integer;
 };
view VE as select from E mixin {
  f : Association[1] to VF on f.vy = $projection.vb;
 } into {
  a as va,
  b as vb,
  f as vf
 };
view VF as select from F {
  x as vx,
  y as vy
```

```
};
```

In the `ON` condition of this type of association in a view, it is necessary to use the pseudo-identifier `$projection` to specify that the following element name must be resolved in the `select` list of the view (“`VE`”) rather than in the entity (“`E`”) in the `FROM` clause

Filter Conditions

It is possible to apply a filter condition when resolving associations between entities; the filter is merged into the `ON`-condition of the resulting `JOIN`. The following example shows how to get a list of customers and then filter the list according to the sales orders that are currently “open” for each customer. In the example, the filter is inserted after the association `orders`; this ensures that the list displayed by the view only contains those orders that satisfy the condition `[status='open']`.

Sample Code

```
view C1 as select from Customer {
  name,
  orders[status='open'].id as orderId
};
```

The following example shows how to use the prefix notation to ensure that the compiler understands that there is only one association (`orders`) to resolve but with multiple elements (`id` and `date`):

Sample Code

```
view C1 as select from Customer {
  name,
  orders[status='open'].{ id   as orderId,
                          date as orderDate
                        }
};
```

Tip

Filter conditions and prefixes can be nested.

The following example shows how to use the associations `orders` and `items` in a view that displays a list of customers with open sales orders for items with a price greater than 200.

Sample Code

```
view C2 as select from Customer {
  name,
  orders[status='open'].{ id   as orderId,
                          date as orderDate,
                          items[price>200].{ descr,
                                             price
                                           }
                        }
};
```

Prefix Notation

The prefix notation can also be used without filters. The following example shows how to get a list of all customers with details of their sales orders. In this example, all uses of the association `orders` are combined

so that there is only one `JOIN` to the table `SalesOrder`. Similarly, both uses of the association `items` are combined, and there is only one `JOIN` to the table `Item`.

Sample Code

```
view C3 as select from Customer {
  name,
  orders.id          as orderId,
  orders.date        as orderDate,
  orders.items.descr as itemDescr,
  orders.items.price as itemPrice
};
```

The example above can be expressed more elegantly by combining the associations `orders` and `items` using the following prefix notation:

Sample Code

```
view C1 as select from Customer {
  name,
  orders.{ id   as orderId,
           date as orderDate,
           items. { descr as itemDescr,
                    price as itemPrice
                  }
         }
};
```

Type Definition

In a CDS view definition, you can explicitly specify the `type` of a select item, as illustrated in the following example:

! Restriction

For use in XS advanced only; assigning an explicit CDS type to an item in a `SELECT` list is not supported in XS classic.

Sample Code

```
type MyInteger : Integer;
entity E {
  a : MyInteger;
  b : MyInteger;
};
view V as select from E {
  a,
  a+b as s1,
  a+b as s2 : MyInteger
};
```

In the example of different `type` definitions, the following is true:

- `a,`
  Has type “`MyInteger`”
- `a+b as s1,`
  Has type “`Integer`” and any information about the user-defined `type` is lost
- `a+b as s2 : MyInteger`
  Has type “`MyInteger`”, which is explicitly specified

> **i Note**
>
> If necessary, a `CAST` function is added to the generated view in SAP HANA; this ensures that the `select` item's type in the SAP HANA view is the SAP HANA “type” corresponding to the explicitly specified CDS type.

Spatial Functions

The following view (`SpatialView1`) displays a list of all persons selected from the entity `Person` and uses the spatial function `ST_Distance` (*) to include information such as the distance between each person's home and business address (`distanceHomeToWork`), and the distance between their home address and the building SAP03 (`distFromSAP03`). The value for both distances is measured in kilometers, which is rounded up and displayed to one decimal point.

> **Sample Code**

```
view SpatialView1 as select from Person {
    name,
    homeAddress.street_name   || ', ' || homeAddress.city   as home,
    officeAddress.street_name || ', ' || officeAddress.city as office,
    round( homeAddress.loc.ST_Distance(officeAddress.loc, 'meter')/1000, 1)
as distanceHomeToWork,
    round( homeAddress.loc.ST_Distance(NEW ST_POINT(8.644072, 49.292910),
'meter')/1000, 1) as distFromSAP03
};
```

> **⚠ Caution**
>
> (*) For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Related Information

Create a View in CDS [page 91]
Create a View in CDS
Spatial Types and Functions [page 112]

3.7.3 Spatial Types and Functions

CDS supports the use of Geographic Information Systems (GIS) functions and element types in CDS-compliant entities and views.

Spatial data is data that describes the position, shape, and orientation of objects in a defined space; the data is represented as two-dimensional geometries in the form of points, line strings, and polygons. The following examples shows how to use the spatial function `ST_Distance` in a CDS view. The underlying spatial data used in the view is defined in a CDS entity using the type `ST_POINT`.

The following example, the CDS entity `Address` is used to store geo-spatial coordinates in element `loc` of type `ST_POINT`:

Sample Code

```
namespace samples;
@Schema: 'MYSCHEMA'
context Spatial {
    entity Person {
        key id : Integer;
        name : String(100);
        homeAddress : Association[1] to Address;
        officeAddress : Association[1] to Address;
    };
    entity Address {
        key id : Integer;
        street_number : Integer;
        street_name : String(100);
        zip : String(10);
        city : String(100);
        loc : hana.ST_POINT(4326);
    };
    view GeoView1 as select from Person {
        name,
        homeAddress.street_name || ', ' || homeAddress.city as home,
        officeAddress.street_name || ', ' || officeAddress.city as office,
        round( homeAddress.loc.ST_Distance(officeAddress.loc, 'meter')/1000,
1) as distanceHomeToWork,
        round( homeAddress.loc.ST_Distance(NEW ST_POINT(8.644072, 49.292910),
'meter')/1000, 1) as distFromSAP03
    };
};
```

The view `GeoView1` is used to display a list of all persons using the spatial function `ST_Distance` to include information such as the distance between each person's home and business address (`distanceHomeToWork`), and the distance between their home address and the building SAP03 (`distFromSAP03`). The value for both distances is measured in kilometers.

Caution

(*) For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Related Information

Create a View in CDS [page 91]
Create a View in CDS
CDS View Syntax Options [page 96]
CDS Entity Syntax Options [page 49]
CDS Primitive Data Types [page 72]

3.8 Modifications to CDS Artifacts

Changes to the definition of a CDS artifact result in changes to the corresponding catalog object. The resultant changes to the catalog object are made according to strict rules.

Reactivating a CDS document which contains changes to the original artifacts results in changes to the corresponding objects in the catalog. Before making change to the design-time definition of a CDS artifact, it is very important to understand what the consequences of the planned changes will be in the generated catalog objects.

- Removing an artifact from a CDS document [page 113]
- Changing the definition of an artifact in a CDS document [page 113]
- Modifying a catalog object generated by CDS [page 116]
- Transporting a DU that contains modified CDS documents [page 116]

Removing an Artifact from a CDS Document

If a CDS design-time artifact (for example, a table or a view) defined in an old version of a CDS document is no longer present in the new version, the corresponding runtime object is dropped from the catalog.

> **i** Note
>
> Renaming a CDS artifact results in the deletion of the artifact with the old name (with all the corresponding consequences) and the creation of a new CDS artifact with the new name.

Changing the Definition of an Artifact in a CDS Document

If a CDS design-time artifact is present in both the old and the new version of a CDS document, a check is performed to establish what, if any, changes have occurred. This applies to changes made either directly to a CDS artifact or indirectly, for example, as a result of a change to a dependent artifact. If changes have been

made to the CDS document, changes are implemented in the corresponding catalog objects according to the following rules:

- Views
  Views in the SAP HANA catalog are dropped and recreated according to the new design-time specification for the artifact in the CDS document.
- Element types
  Changing the type of an element according to the implicit conversion rules described in the SAP HANA SQL documentation (*SAP HANA SQL Data Type Conversion*). Note: For some type conversions the activation will succeed only if the data in the corresponding DB table is valid for the target type (for example the conversion of String to Integer will succeed only if the corresponding DB table column contains only numbers that match the Integer type)
- Element modifier: `Null/NOT NULL`
  Adding, removing or changing element modifiers "`Null`" and "`not null`" to make an element nullable ot not nullable respectively can lead to problems when activating the resulting artifact; the activation will succeed only if the data in the database table corresponding to the CDS entity matches the new modifier. For example, you cannot make an element not nullable, if in the corresponding column in the database table some null values exist for which there is no default value defined.
- Element modifier: Default Value
  If the default value modifier is removed, this has no effect on the existing data in the corresponding database table, and no default value will be used for any subsequently inserted record. If the default value is modified or newly added, the change will be applicable to all subsequent inserts in the corresponding database table. In addition, if the element is not nullable (irrespective of whether it was defined previously as such or within the same activation), the existing null values in the corresponding table will be replaced with the new default value.
- Element modifier: Primary Key
  You can add an element to (or remove it from) the primary key by adding or removing the "`key`" modifier.

  > **i Note**
  >
  > Adding the "`key`" modifier to an element will also make the column in the corresponding table `not nullable`. If column in the corresponding database table contains `null` values and there is no default value defined for the element, the activation of the modified CDS document will fail.

- Column or row store (`@Catalog.tableType`)
  It is possible to change the `Catalog.tableType` annotation that defines the table type, for example, to transform a table from the column store (`#COLUMN`) to row store (`#ROW`), and vice versa.
- Index types (`@Catalog.index`)
  Is is possible to change the "`Catalog.index`" annotation, as long as the modified index is valid for the corresponding CDS entity.

For changes to individual elements of a CDS entity, for example, column definitions, the same logic applies as for complete artifacts in a CDS document.

- Since the elements of a CDS entity are identified by their name, changing the order of the elements in the entity definition will have no effect; the order of the columns in the generated catalog table object remains unchanged.
- Renaming an element in a CDS entity definition is not recognized; the rename operation results in the deletion of the renamed element and the creation of a new one.
- If a new element is added to a CDS entity definition, the order of the columns in the table generated in the catalog after the change cannot be guaranteed.

> **i Note**
>
> If an existing CDS entity definition is changed, the order of the columns in the generated database tables may be different from the order of the corresponding elements in the CDS entity definition.

In the following example of a simple CDS document, the context `OuterCtx` contains a CDS entity `Entity1` and the nested context `InnerCtx`, which contains the CDS entity definition `Entity2`.

```
namespace pack;
@Schema: 'MYSCHEMA'
context OuterCtx
{
  entity Entity1
  {
    key a : Integer;
        b : String(20);
  };
  context InnerCtx
  {
    entity Entity2
    {
      key x : Integer;
          y : String(10);
          z : LocalDate;
    };
  };
};
```

To understand the effect of the changes made to this simple CDS document in the following example, it is necessary to see the changes not only from the perspective of the developer who makes the changes but also the compiler which needs to interpret them.

From the developer's perspective, the CDS entity `Entity1` has been moved from context `OuterCtx` to `InnerCtx`. From the compiler's perspective, however, the entity `pack::OuterCtx.Entity1` has disappeared and, as a result, will be deleted (and the corresponding generated table with all its content dropped), and a new entity named `pack::OuterCtx.InnerCtx.Entity1` has been defined.

```
namespace pack;
@Schema: 'MYSCHEMA'
context OuterCtx
{
  context InnerCtx
  {
    entity Entity1
    {
      key a : Integer;
          b : String(20);
    };
    entity Entity2
    {
      key x : Integer;
          q : String(10);
          z : LocalDate;
    };
  };
};
```

Similarly, renaming the element `y: String;` to `q: String;` in `Entity2` results in the deletion of column `y` and the creation of a new column `q` in the generated catalog object. As a consequence, the content of column `y` is lost.

Modifying a Catalog Object Generated from CDS

CDS does not support modifications to catalog objects generated from CDS documents. You must never modify an SAP HANA catalog object (in particular a table) that has been generated from a CDS document. The next time you activate the CDS document that contains the original CDS object definition and the corresponding catalog objects are generated, all modifications made to the catalog object are lost or activation might even fail due to inconsistencies.

Transporting a DU that Contains Modified CDS Documents

If the definition of a CDS entity has already been transported to another system, do not enforce activation of any illegal changes to this entity, for example, by means of an intermediate deletion.

Restrictions apply to changes that can be made to a CDS entity if the entity has been activated and a corresponding catalog object exists. If changes to a CDS entity on the source system produce an error during activation of the CDS document, for example, because you changed an element type in a CDS entity from `Binary` to `LocalDate`, you could theoretically delete the original CDS entity and then create a new CDS entity with the same name as the original entity but with the changed data type. However, if this change is transported to another system, where the old version of the entity already exists, the import will fail, because the information that the entity has been deleted and recreated is not available either on the target system or in the delivery unit.

Related Information

SAP HANA to CDS Data-Type Mapping [page 60]
SAP HANA SQL Data Type Conversion

3.9 Tutorial: Get Started with CDS

You can use the Data Definition Language (DDL) to define a table, which is also referred to as an “entity” in SAP HANA Core Data Services (CDS). The finished artifact is saved in the repository with the extension (suffix) `.hdbdd`, for example, `MyTable.hdbdd`.

Prerequisites

This task describes how to create a file containing a CDS entity (table definition) using DDL. Before you start this task, note the following prerequisites:

- You must have access to an SAP HANA system.

- You must have already created a development workspace and a project.
- You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.
- You must have created a schema definition `MYSCHEMA.hdbschema`.

Context

The SAP HANA studio provides a dedicated DDL editor to help you define data-related artifacts, for example, entities, or views. To create a simple database table with the name "MyTable", perform the following steps:

> → Tip
>
> File extensions are important. If you are using SAP HANA Studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically and, if appropriate, enables direct editing of the new file in the corresponding editor.

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS document that defines the entity you want to create.

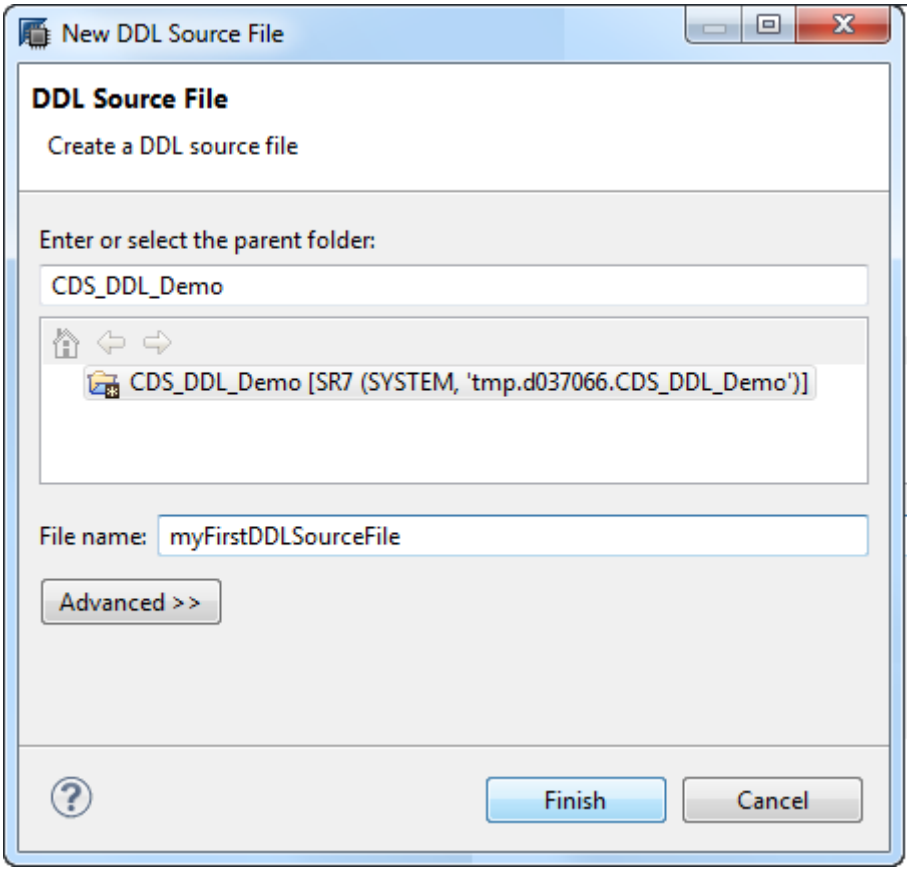
   Browse to the folder in your project workspace where you want to create the new CDS document (for example, in a project you have already created and shared) and perform the following tasks:

   a. Right-click the folder where you want to create the CDS document and choose *New* > *DDL Source File* in the context-sensitive popup menu.

   > i Note
   >
   > This menu option is only available from shared projects; projects that are linked to the SAP HANA repository.

   b. Enter the name of the entity in the *File Name* box, for example, `MyFirstCDSSourceFile`.

   > i Note
   >
   > The file extension `.hdbdd` is added automatically to the new DDL file name. The repository uses the file extension to make assumptions about the contents of repository artifacts, for example, that `.hdbdd` files contain DDL statements.



c. Choose *Finish* to save the new empty CDS document.

> **i Note**
>
> If you are using a CDS document to define a single CDS-compliant entity, the name of the CDS document must match the name of the entity defined in the CDS document, for example, with the *entity* keyword. In the example in this tutorial, you would save the entity definition "BOOK" in the CDS document `BOOK.hdbdd`.

5. Define the table entity.

   To edit the CDS document, in the *Project Explorer* view double-click the file you created in the previous step, for example, `BOOK.hdbdd`, and add the entity-definition code:

> **i Note**
>
> The CDS DDL editor automatically inserts the mandatory keywords *namespace* and *context* into any new DDL source file that you create using the *New DDL Source File* dialog. The following values are assumed:
>
> - *namespace* = `<Current Project Name>`
> - *context* = `<New DDL File Name>`
>
> The name space declared in a CDS document must match the repository package in which the object the document defines is located.

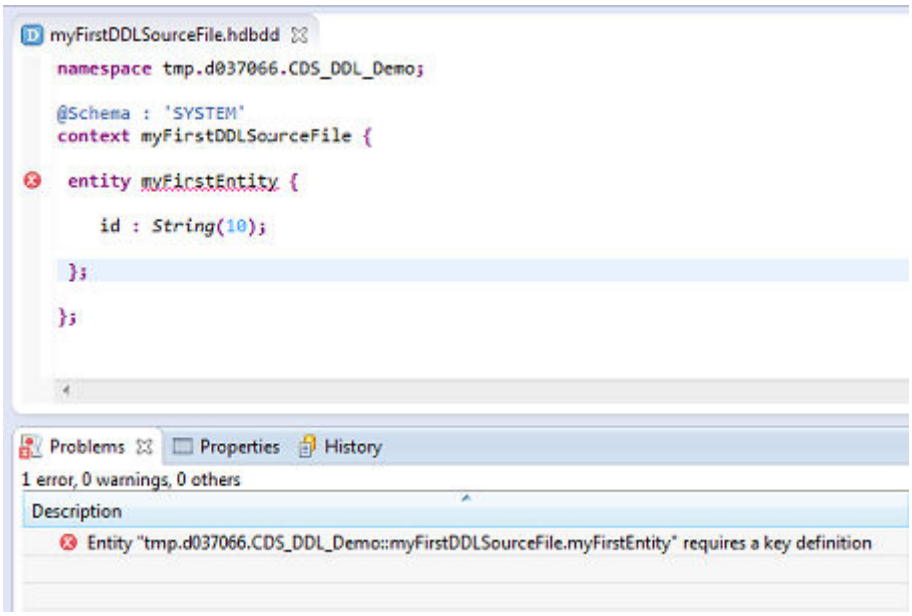
In this example, the CDS document `BOOK.hdbdd` that defines the CDS entity "BOOK" must reside in the package `mycompany.myapp1`.

```
namespace mycompany.myapp1;
@Schema : 'MYSCHEMA'
@Catalog.tableType: #COLUMN
@Catalog.index: [ { name : 'MYINDEX1', unique : true, order : #DESC,
elementNames : ['ISBN'] }  ]
entity BOOK {
    key Author    : String(100);
    key BookTitle : String(100);
        ISBN      : Integer not null;
        Publisher : String(100);
};
```

6. Save the CDS document `BOOK.hdbdd`.

> **i Note**
>
> Saving a file in a shared project automatically commits the saved version of the file to the repository, To explicitly commit a file to the repository, right-click the file (or the project containing the file) and choose *Team* > *Commit* from the context-sensitive popup menu.

7. Activate the new CDS document in the repository.
   a. In the *Project Explorer* view, locate the newly created artifact `BOOK.hdbdd`.
   b. Right-click `BOOK.hdbdd` and choose *Team* > *Activate* in the context-sensitive popup menu. The CDS/DDL editor checks the syntax of the source file code, highlights the lines where an error occurs, and provides details of the error in the *Problems* view.



The activation creates the following table in the schema MYSCHEMA, both of which are visible using the SAP HANA studio:

```
"MYSCHEMA"."mycompany.myapp1::BOOK"
```

The following public synonym is also created, which can be referenced using the standard SQL query notation:

```
"mycompany.myapp1::BOOK"
```

8. Add an entry to the BOOK entity using SQL.

```
INSERT INTO "mycompany.myapp1::BOOK" VALUES ( 'Shakespeare', 'Hamlet',
'1234567', 'Books Incorporated' );
```

9. Save and activate the modifications to the entity.
10. Check the new entry by running a simply SQL query.

```
SELECT COUNT(*) FROM "mycompany.myapp1::BOOK" WHERE Author = 'Shakespeare'
```

Related Information

Create a Schema

3.10 Import Data with CDS Table-Import

The table-import function is a data-provisioning tool that enables you to import data from comma-separated values (CSV) files into SAP HANA tables.

Prerequisites

Before you start this task, make sure that the following prerequisites are met:

- An SAP HANA database instance is available.
- The SAP HANA database client is installed and configured.
- You have a database user account set up with the roles containing sufficient privileges to perform actions in the repository, for example, add packages, add objects, and so on.
- The SAP HANA studio is installed and connected to the SAP HANA repository.
- You have a development environment including a repository workspace, a package structure for your application, and a shared project to enable you to synchronize changes to the project files in the local file system with the repository.

> **i Note**
>
> The names used in the following task are for illustration purposes only; where necessary, replace the names of schema, tables, files, and so on shown in the following examples with your own names.

Context

In this tutorial, you import data from a CSV file into a table generated from a design-time definition that uses the `.hdbdd` syntax, which complies with the Core Data Services (CDS) specifications.

> → Tip
>
> File extensions are important. If you are using SAP HANA Studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically and, if appropriate, enables direct editing of the new file in the corresponding editor.

Procedure

1. Create a root package for your table-import application.
   In SAP HANA studio, open the *SAP HANA Development* perspective and perform the following steps:
   a. In the package hierarchy displayed in the *Systems* view, right-click the package where you want to create the new package for your table-import configuration and choose *New > Package...*.
   b. Enter a name for your package, for example **`TiTest`**. You must create the new **`TiTest`** package in your own namespace, for example `mycompany.tests.TiTest`

      > i Note
      >
      > Naming conventions exist for package names, for example, a package name must not start with either a dot (.) or a hyphen (-) and cannot contain two or more consecutive dots (..). In addition, the name must not exceed 190 characters.

   a. Choose *OK* to create the new package.
2. Create a set of table-import files.

   For the purposes of this tutorial, the following files must all be created in the same package, for example, a package called `TiTest`. However, the table-import feature also allows you to use files distributed in different packages

   > → Tip
   >
   > File extensions are important. If you are using SAP HANA Studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically and, if appropriate, enables direct editing of the new file in the corresponding editor.

   - The table-import configuration file, for example, `TiConfiguration.hdbti`
     Specifies the source file containing the data values to import and the target table in SAP HANA into which the data must be inserted
   - A CSV file, for example, `myTiData.csv`
     Contains the data to be imported into the SAP HANA table during the table-import operation; values in the `.csv` file can be separated either by a comma (,) or a semi-colon (;).
   - A target table.

The target table can be either a runtime table in the catalog or a table definition, for example, a table defined using the `.hdbtable` syntax (`TiTable.hdbtable`) or the CDS-compliant `.hdbdd` syntax (`TiTable.hdbdd`).

> **i Note**
>
> In this tutorial, the target table for the table-import operation is `TiTable.hdbdd`, a design-time table defined using the CDS-compliant `.hdbdd` syntax.

- The schema named `AMT`
  Specifies the name of the schema in which the target import table resides

When all the necessary files are available, you can import data from a source file, such as a CSV file, into the desired target table.

3. If it does not already exist, create a schema named `AMT` in the catalog; the `AMT` schema is where the target table for the table-import operation resides.
4. Create or open the table-definition file for the target import table (`inhabitants.hdbdd`) and enter the following lines of text; this example uses the `.hdbdd` syntax.

> **i Note**
>
> In the CDS-compliant `.hdbdd` syntax, the *namespace* keyword denotes the path to the package containing the table-definition file.

```
namespace mycompany.tests.TiTest;

@Schema : 'AMT'
@Catalog.tableType : #COLUMN
entity inhabitants {
  key ID : Integer;
  surname : String(30);
  name : String(30);
  city : String(30);
};
```

5. Open the CSV file containing the data to import, for example, `inhabitants.csv` in a text editor and enter the values shown in the following example.

```
0,Annan,Kwesi,Accra
1,Essuman,Wiredu,Tema
2,Tetteh,Kwame,Kumasi
3,Nterful,Akye,Tarkwa
4,Acheampong,Kojo,Tamale
5,Assamoah,Adjoa,Takoradi
6,Mensah,Afua,Cape Coast
```

> **i Note**
>
> You can import data from multiple `.csv` files in a single, table-import operation. However, each `.csv` file must be specified in a separate code block (`{table= ...}`) in the table-import configuration file.

6. Create or open the table-import configuration file (`inhabitants.hdbti`) and enter the following lines of text.

```
import = [
            {
                table = "mycompany.tests.TiTest::inhabitants";
                schema = "AMT";
```

```
                file = "mycompany.tests.TiTest:inhabitants.csv";
                header = false;
            }
        ];
```

7. Deploy the table import.
   a. Select the package that you created in the first step, for example, `mycompany.tests.TiTest`.
   b. Click the alternate mouse button and choose *Commit*.
   c. Click the alternate mouse button and choose *Activate*.

   This activates all the repository objects. The data specified in the CSV file `inhabitants.csv` is imported into the SAP HANA table `inhabitants` using the data-import configuration defined in the `inhabitants.hdbti` table-import configuration file.
8. Check the contents of the runtime table `inhabitants` in the catalog.

   To ensure that the import operation completed as expected, use the SAP HANA studio to view the contents of the runtime table `inhabitants` in the catalog. You need to confirm that the correct data was imported into the correct columns.
   a. In the *SAP HANA Development* perspective, open the *Systems* view.
   b. Navigate to the catalog location where the `inhabitants` object resides, for example:

      `<SID>` > *Catalog* > *AMT* > *Tables*
   c. Open a data preview for the updated object.

      Right-click the updated object and choose *Open Data Preview* in the context-sensitive menu.

3.10.1 Data Provisioning Using Table Import

You can import data from comma-separated values (CSV) into the SAP HANA tables using the SAP HANA Extended Application Services (SAP HANA XS) table-import feature.

In SAP HANA XS, you create a table-import scenario by setting up an table-import configuration file and one or more comma-separated value (CSV) files containing the content you want to import into the specified SAP HANA table. The import-configuration file links the import operation to one or more target tables. The table definition (for example, in the form of a `.hdbdd` or `.hdbtable` file) can either be created separately or be included in the table-import scenario itself.

To use the SAP HANA XS table-import feature to import data into an SAP HANA table, you need to understand the following table-import concepts:

- Table-import configuration
  You define the table-import model in a configuration file that specifies the data fields to import and the target tables for each data field.

  > **i Note**
  >
  > The table-import file must have the `.hdbti` extension, for example, `myTableImport.hdbti`.

CSV Data File Constraints

The following constraints apply to the CSV file used as a source for the table-import feature in SAP HANA XS:

- The number of table columns must match the number of CSV columns.
- There must not be any incompatibilities between the data types of the table columns and the data types of the CSV columns.
- Overlapping data in data files is not supported.
- The target table of the import must not be modified (or appended to) outside of the data-import operation. If the table is used for storage of application data, this data may be lost during any operation to re-import or update the data.

Related Information

Table-Import Configuration [page 124]
Table-Import Configuration-File Syntax [page 126]

3.10.2 Table-Import Configuration

You can define the elements of a table-import operation in a design-time file; the configuration includes information about source data and the target table in SAP HANA.

SAP HANA Extended Application Services (SAP HANA XS) enables you to perform data-provisioning operations that you define in a design-time configuration file. The configuration file is transportable, which means you can transfer the data-provisioning between SAP HANA systems quickly and easily.

The table-import configuration enables you to specify how data from a comma-separated-value (`.csv`) file is imported into a target table in SAP HANA. The configuration specifies the source file containing the data values to import and the target table in SAP HANA into which the data must be inserted. As further options, you can specify which field delimiter to use when interpreting data in the source `.csv` file and if keys must be used to determine which columns in the target table to insert the imported data into.

> **i Note**
>
> If you use **multiple** table import configurations to import data into a **single** target table, the *keys* keyword is mandatory. This is to avoid problems relating to the overwriting or accidental deletion of existing data.

The following example of a table-import configuration shows how to define a simple import operation which inserts data from the source files `myData.csv` and `myData2.csv` into the table `myTable` in the schema `mySchema`.

```
import = [
            {
                table = "myTable";
                schema = "mySchema";
                file = "sap.ti2.demo:myData.csv";
                header = false;
                delimField = ";";
```

```
            keys = [ "GROUP_TYPE" : "BW_CUBE"];
        },
        {
            table = "sap.ti2.demo::myTable";
            file = "sap.ti2.demo:myData2.csv";
            header = false;
            delimField = ";";
            keys = [ "GROUP_TYPE" : "BW_CUBE"];
        }
    ];
```

In the table import configuration, you can specify the target table using either of the following methods:

- Public synonym (`"sap.ti2.demo::myTable"`)
  If you use the public synonym to reference a target table for the import operation, you must use either the *hdbtable* or *cdstable* keyword, for example, `hdbtable = "sap.ti2.demo::myTable";`
- Schema-qualified catalog name (`"mySchema"."MyTable"`
  If you use the schema-qualified catalog name to reference a target table for the import operation, you must use the *table* keyword in combination with the *schema* keyword, for example, `table = "myTable"; schema = "mySchema";`

> **i Note**
>
> Both the schema and the target table specified in the table-import operation must already exist. If either the specified table or the schema does not exist, SAP HANA XS displays an error message during the activation of the configuration file, for example: `Table import target table cannot be found.` or `Schema could not be resolved.`

You can also use one table-import configuration file to import data from multiple `.csv` source files. However, you must specify each import operation in a new code block introduced by the *[hdb | cds]table* keyword, as illustrated in the example above.

By default, the table-import operation assumes that data values in the `.csv` source file are separated by a comma (,). However, the table-import operation can also interpret files containing data values separated by a semi-colon (;).

- Comma (,) separated values

  ```
  ,,,BW_CUBE,,40000000,2,40000000,all
  ```

- Semi-colon (;) separated values

  ```
  ;;;BW_CUBE;;40000000;3;40000000;all
  ```

> **i Note**
>
> If the activated `.hdbti` configuration used to import data is subsequently deleted, only the data that was imported by the deleted `.hdbti` configuration is dropped from the target table. All other data including any data imported by other `.hdbti` configurations remains in the table. If the target CDS entity has no key (annotated with `@nokey`) all data that is not part of the CSV file is dropped from the table during each table-import activation.

You can use the optional keyword *keys* to specify the key range taken from the source `.csv` file for import into the target table. If keys are specified for an import in a table import configuration, multiple imports into same target table are checked for potential data collisions.

> **i Note**
>
> The configuration-file syntax does not support wildcards in the key definition; the full value of a selectable column value has to be specified.

Security Considerations

In SAP HANA XS, design-time artifacts such as tables (`.hdbtable` or `.hdbdd`) and table-import configurations (`.hdbti`) are not normally exposed to clients via HTTP. However, design-time artifacts containing comma-separated values (`.csv`) could be considered as potential artifacts to expose to users through HTTP. For this reason, it is essential to protect these exposed `.csv` artifacts by setting the appropriate application privileges; the application privileges prevents data leakage, for example, by denying access to data by users, who are not normally allowed to see all the records in such tables.

> **→ Tip**
>
> Place all the `.csv` files used to import content to into tables together in a single package and set the appropriate (restrictive) application-access permissions for that package, for example, with a dedicated `.xsaccess` file.

Related Information

Table-Import Configuration-File Syntax [page 126]

3.10.3 Table-Import Configuration-File Syntax

The design-time configuration file used to define a table-import operation requires the use of a specific syntax. The syntax comprises a series of `keyword=value` pairs.

If you use the table-import configuration syntax to define the details of the table-import operation, you can use the keywords illustrated in the following code example. The resulting design-time file must have the `.hdbti` file extension, for example, `myTableImportCfg.hdbti`.

```
import = [
   {
     table = "myTable";
     schema = "mySchema";
     file = "sap.ti2.demo:myData.csv";
     header = false;
     useHeaderNames = false;
     delimField = ";";
     delimEnclosing="\"";
     distinguishEmptyFromNull = true;
     keys = [ "GROUP_TYPE" : "BW_CUBE", "GROUP_TYPE" : "BW_DSO", "GROUP_TYPE" :
"BW_PSA"];
   }
```

```
];
```

table

In the table-import configuration, the `table`, `cdstable`, and `hdbtable` keywords enable you to specify the name of the target table into which the table-import operation must insert data. The target table you specify in the table-import configuration can be a runtime table in the **catalog** or a **design-time** table definition, for example, a table defined using either the `.hdbtable` or the `.hdbdd` (Core Data Services) syntax.

> **i Note**
>
> The target table specified in the table-import configuration must already exist. If the specified table does not exist, SAP HANA XS displays an error message during the activation of the configuration file, for example: `Table import target table cannot be found.`

Use the *table* keyword in the table-import configuration to specify the name of the target table using the qualified name for a **catalog** table.

```
table = "target_table";
schema = "mySchema";
```

> **i Note**
>
> You must also specify the name of the schema in which the target catalog table resides, for example, using the *schema* keyword.

The *hdbtable* keyword in the table-import configuration enables you to specify the name of a target table using the public synonym for a **design-time** table defined with the `.hdbtable` syntax.

```
hdbtable = "sap.ti2.demo::target_table";
```

The *cdstable* keyword in the table-import configuration enables you to specify the name of a target table using the public synonym for a **design-time** table defined with the CDS-compliant `.hdbdd` syntax.

```
cdstable = "sap.ti2.demo::target_table";
```

> **⚠ Caution**
>
> There is no explicit check if the addressed table is created using the `.hdbtable` or CDS-compliant `.hdbdd` syntax.

If the table specified with the `cdstable` or `hdbtable` keyword is not defined with the corresponding syntax, SAP HANA displays an error when you try to activate the artifact, for example,`Invalid combination of table declarations found, you may only use [cdstable | hdbtable | table].`

schema

The following code example shows the syntax required to specify a schema in a table-import configuration.

```
schema = "TI2_TESTS";
```

> **i** Note
>
> The schema specified in the table-import configuration file must already exist.

If the schema specified in a table-import configuration file does not exist, SAP HANA XS displays an error message during the activation of the configuration file, for example:

- `Schema could not be resolved.`
- `If you import into a catalog table, please provide schema.`

The *schema* is only required if you use a table's schema-qualified catalog name to reference the target table for an import operation, for example, `table = "myTable"; schema = "mySchema";`. The schema is **not** required if you use a public synonym to reference a table in a table-import configuration, for example, `hdbtable = "sap.ti2.demo::target_table";`.

file

Use the *file* keyword in the table-import configuration to specify the source file containing the data that the table-import operation imports into the target table. The source file must be a `.csv` file with the data values separated either by a comma (,) or a semi-colon (;). The file definition must also include the full package path in the SAP HANA repository.

```
file = "sap.ti2.demo:myData.csv";
```

header

Use the `header` keyword in the table-import configuration to indicate if the data contained in the specified `.csv` file includes a header line. The `header` keyword is optional, and the possible values are `true` or `false`.

```
header = false;
```

useHeaderNames

Use the `useHeaderNames` keyword in the table-import configuration to indicate if the data contained in the first line of the specified `.csv` file must be interpreted. The `useHeaderNames` keyword is optional; it is used in

combination with the`header` keyword. The `useHeaderNames` keyword is boolean: possible values are `true` or `false`.

> **i** Note
>
> The `useHeaderNames` keyword only works if `header` is **also** set to "`true`".

```
useHeaderNames = false;
```

The table-import process considers the order of the columns; if the column order specified in the `.csv`, file does not match the order used for the columns in the target table, an error occurs on activation.

delimField

Use the `delimField` keyword in the table-import configuration to specify which character is used to separate the values in the data to be imported. Currently, the table-import operation supports either the comma (,) or the semi-colon (;). The following example shows how to specify that values in the `.csv` source file are separated by a semi-colon (;).

```
delimField = ";";
```

> **i** Note
>
> By default, the table-import operation assumes that data values in the `.csv` source file are separated by a comma (,). If no delimiter field is specified in the `.hdbti` table-import configuration file, the default setting is assumed.

delimEnclosing

Use the `delimEnclosing` keyword in the table-import configuration to specify a single character that indicates both the start and end of a set of characters to be interpreted as a single value in the `.csv` file, for example "This is all one, single value". This feature enables you to include in data values in a `.CSV` file even the character defined as the field delimiter (in `delimField`), for example, a comma (,) or a semi-colon (;).

> **→** Tip
>
> If the value used to separate the data fields in your `.csv` file (for example, the comma (,)) is also used inside the data values themselves (`"This, is, a, value"`), you **must** declare and use a delimiter enclosing character and use it to enclose all data values to be imported.

The following example shows how to use the `delimEnclosing` keyword to specify the quote (") as the delimiting character that indicates both the start and the end of a value in the `.csv` file. Everything enclosed between the `delimEnclosing` characters (in this example, "") is interpreted by the import process as one, single value.

```
delimEnclosing="\"";
```

> **Note**
>
> Since the `hdbti` syntax requires us to use the quotes (“”) to specify the delimiting character, and the delimiting character in this example is, itself, also a quote ("), we need to use the backslash character (\) to escape the second quote (").

In the following example of values in a `.csv` file, we assume that `delimEnclosing"\""`, and `delimField=","`. This means that imported values in the `.csv` file are enclosed in the quote character (`"value”`) and multiple values are separated by the comma (`"value1”,"value 2”`). Any commas **inside** the quotes are interpreted as a comma and not as a field delimiter.

```
"Value 1, has a comma","Value 2 has, two, commas","Value3"
```

You can use other characters as the enclosing delimiter, too, for example, the hash (#). In the following example, we assume that `delimEnclosing="#"` and `delimField=";"`. Any semi-colons included **inside** the hash characters are interpreted as a semi-colon and not as a field delimiter.

```
#Value 1; has a semi-colon#;#Value 2 has; two; semi-colons#;#Value3#
```

distinguishEmptyFromNull

Use the `distinguishEmptyFromNull` keyword in combination with `delimEnclosing` to ensure that the table-import process correctly interprets any **empty** value in the `.CSV` file, which is enclosed with the value defined in the `delimEnclosing` keyword, for example, as an empty space. This ensures that an empty space is imported “as is” into the target table. If the empty space in incorrectly interpreted, it is imported as `NULL`.

```
distinguishEmptyFromNull = true;
```

> **Note**
>
> The default setting for `distinguishEmptyFromNull` is `false`.

If `distinguishEmptyFromNull=false` is used in combination with `delimEnclosing`, then an empty value in the `.CSV` (with or without quotes “”) is interpreted as NULL.

```
"Value1",,"",Value2
```

The table-import process would add the values shown in the example `.csv` above into the target table as follows:

```
Value1 |  NULL  |  NULL  |  Value2
```

keys

Use the *keys* keyword in the table-import configuration to specify the key range to be considered when importing the data from the `.csv` source file into the target table.

```
keys = [ "GROUP_TYPE" : "BW_CUBE", "GROUP_TYPE" : "BW_DSO", "GROUP_TYPE" :
"BW_PSA"];
```

In the example above, all the lines in the `.csv` source file where the `GROUP_TYPE` column value matches one of the given values (`BW_CUBE`, `BW_DSO`, or `BW_PSA`) are imported into the target table specified in the table-import configuration.

```
;;;BW_CUBE;;40000000;3;40000000;slave
;;;BW_DSO;;40000000;3;40000000;slave
;;;BW_PSA;;2000000000;1;2000000000;slave
```

In the following example, the `GROUP_TYPE` column is specified as empty(“”).

```
keys = [ "GROUP_TYPE" : ""];
```

All the lines in the `.csv` source file where the `GROUP_TYPE` column is empty are imported into the target table specified in the table-import configuration.

```
;;;;;40000000;2;40000000;all
```

3.10.4 Table-Import Configuration Error Messages

During the course of the activation of the table-import configuration and the table-import operation itself, SAP HANA checks for errors and displays the following information in a brief message.

Table-Import Error Messages

Message Number	Message Text	Message Reason
40200	`Invalid combination of table declarations found, you may only use [cdstable | hdbtable | table]`	The *table* keyword is specified in a table-import configuration that references a table defined using the `.hdbtable` (or `.hdbdd`) syntax. The *hdbtable* keyword is specified in a table-import configuration that references a table defined using another table-definition syntax, for example, the `.hdbdd` syntax. The *cdstable* keyword is specified in a table-import configuration that references a table defined using another table-definition syntax, for example, the `.hdbtable` syntax.
40201	`If you import into a catalog table, please provide schema`	You specified a target table with the *table* keyword but did not specify a schema with the *schema* keyword.
40202	`Schema could not be resolved`	The schema specified with the *schema* keyword does not exist or could not be found (wrong name). The public synonym for an `.hdbtable` or `.hdbdd` (CDS) table definition cannot be resolved to a catalog table.
40203	`Schema resolution error`	The schema specified with the *schema* keyword does not exist or could not be found (wrong name). The database could not complete the schema-resolution process for some reason - perhaps unrelated to the table-import configuration (`.hdbti`), for example, an inconsistent database status.

Message Number	Message Text	Message Reason
40204	`Table import target table cannot be found`	The table specified with the *table* keyword does not exist or could not be found (wrong name or wrong schema name).
40210	`Table import syntax error`	The table-import configuration file (`.hdbti`) contains one or more syntax errors.
40211	`Table import constraint checks failed`	The same key is specified in multiple table-import configurations (`.hdbti` files), which leads to overlaps in the range of data to import. If keys are specified for an import in a table-import configuration, multiple imports into the same target table are checked for potential data collisions.
40212	`Importing data into table failed`	Either duplicate keys were written (due to duplicates in the `.CSV` source file) or An (unexpected) error occurred on the SQL level.
40213	`CSV table column count mismatch`	Either the number of columns in the `.CSV` record is higher than the number of columns in the table, or The number of columns in the `.CSV` record is higher than the number of columns in its header.
40214	`Column type mismatch`	The `.CSV` file does not match the target table for either of the following reasons: 1. Data are missing for some `not-null` columns 2. Some columns specified in the `.CSV` record do not exist in the table.
40216	`Key does not match to table header`	For some key columns of the table, no data are provided.

4 Creating Data Persistence Artifacts with CDS in XS Advanced

You can use Core Data Services to define the underlying database objects that store and provide data for your application, for example, tables, views, and data types.

As part of the process of defining the database persistence model for your application, you create database design-time artifacts such as tables and views, for example using Core Data Services (CDS). You can define your data persistence model in one or more CDS documents. The syntax for the specific data artifacts is described in the following sections:

- CDS documents
- CDS entities (tables)
- CDS user-defined types
- CDS associations (between entities)
- CDS views
- CDS extensions

> **i** Note
>
> CDS and CD+S are not interchangeable. Although the Core Data **and** Services (CD+S) used by the SAP Cloud Application Programming Model (CAP) is syntactically similar to the CDS used in XS classic and XS advanced, they use different artifact types (`.hdbcds` and `.cds`, respectively) and are intended for use in different development scenarios. For more information about CD+S, see *About SAP CAP* in *Related Information* below.

Related Information

Create the Data Persistence Artifacts with CDS in XS Advanced [page 135]
Create a CDS Document (XS Advanced) [page 144]
Create a CDS Entity in XS Advanced [page 179]
Create a CDS User-Defined Structure in XS Advanced [page 203]
Create a CDS Association in XS Advanced [page 217]
Create a CDS View in XS Advanced [page 232]
Create a CDS Extension [page 254]
About CAP (SAP Cloud Application Programming Model)

4.1 Create the Data Persistence Artifacts with CDS in XS Advanced

Use Core Data Services (CDS) to define the artifacts that make up the data-persistence model.

Context

CDS artifacts are design-time definitions that are used to generate the corresponding run-time objects, when the CDS document that contains the artifact is deployed to the SAP HANA XS advanced model run time. In XS advanced, the CDS document containing the design-time definitions that you create using the CDS-compliant syntax must have the file extension `.hdbcds`, for example, `MyCDSTable.hdbcds`.

The HDI deployment tools deploy database artifacts to an HDI design-time container. Design-time database objects are typically located in the `db/` folder of the application design-time hierarchy, as illustrated in the following example:

Procedure

1. Create the infrastructure in your design-time file-system.

   Design-time database objects are typically located in the `db/src/` folder of the application design-time hierarchy, as illustrated in the following example:

   Sample Code

   ```
   AppName
   |- db/                          # Database deployment artifacts
      |- src/                      # Database artifacts: tables, views, etc.
   ```

2. Define the CDS artifacts.

   The data-persistence model can be defined in one central CDS document (for example, `myCDSmodel.hdbcds`) or split into separate CDS documents (`myCDStype.hdbcds` and `myCDSentity.hdbcds`) which describe individual artifacts, as illustrated in the following example:

   Sample Code

   ```
   AppName
   |- db/                            # Database deployment artifacts
   |  |- src/                        # Database artifacts: tables, views, etc.
   |     |- myCDSentity.hdbcds       # Database design-time table definition
   |     |- myCDSDataType.hdbcds     # Database design-time table definition
   |     \- myCDSmodel.hdbcds        # Database design-time model definition
   ```

3. Add a package descriptor for the XS advanced application.

The package descriptor (`package.json`) describes the prerequisites and dependencies that apply to the database module of an application in SAP HANA XS advanced. Add a `package.json` file to the root folder of your application's database module (`db/`).

Sample Code

```
AppName
|- db/                              # Database deployment artifacts
|   |- package.json                 # Database details/dependencies
|   |- src/                         # Database artifacts: tables, views, etc.
|       |- myCDSentity.hdbcds       # Database design-time table definition
|       |- myCDSDataType.hdbcds     # Database design-time table definition
|       \- myCDSmodel.hdbcds        # Database design-time definition
overview
```

The basic `package.json` file for your database module (`db/`) should look similar to the following example:

Sample Code

```
{
  "name": "deploy",
  "dependencies": {
     "@sap/hdi-deploy": "3.7.0"
  },
  "scripts": {
     "start": "node node_modules/@sap/hdi-deploy/deploy.js"
  }
}
```

4. Add the container-configuration files required by the SAP HANA Deployment Infrastructure (HDI).

   The following design-time artifacts are used to configure the HDI containers:

   - Container deployment configuration (`.hdiconfig`)
     Mandatory: A JSON file containing a list of the bindings between database artifact types (for example, sequence, procedure, table) and the corresponding deployment plug-in (and version).
   - Run-time container name space rules (`.hdinamespace`)
     Optional: A JSON file containing a list of design-time file suffixes and the naming rules for the corresponding runtime locations.

   Sample Code

   ```
   AppName
   |- db/                              # Database deployment artifacts
   |   |- package.json                 # Database details/dependencies
   |   |- src/                         # Database artifacts: tables, views, etc.
   |       |- .hdiconfig               # HDI build plug-in configuration
   |       |- .hdinamespace            # HDI run-time name-space configuration
   |       |- myCDSentity.hdbcds       # Database design-time table definition
   |       |- myCDSDataType.hdbcds     # Database design-time table definition
   |       \- myCDSmodel.hdbcds        # Database design-time definition
   overview
   ```

Related Information

Design-Time Database Resources in XS Advanced [page 137]
Creating the Persistence Model in Core Data Services [page 10]
Managing SAP HDI Containers and Artifacts
Create the XS Advanced Application Package Descriptor

4.1.1 Design-Time Database Resources in XS Advanced

Design-time database resources reside in the database module of your multi-target application.

The design-time representations of you database resources must be collected in the database module of your multi-target application; the database module is a folder structure that you create, for example `/db/` that includes sub folder called `src/` in you place all the source files. For database applications, the structure of the design-time resources should look something like the following example:

Sample Code

Database module in an XS Advanced Application

```
<MyAppName>
|- db/                                  # Database deployment artifacts
|   |- package.json                     # Database details/dependencies
|   \- src/                             # Database artifacts...
|       |- .hdiconfig                   # HDI build plug-in configuration
|       |- .hdinamespace                # HDI run-time name-space config...
|       |- myTable.hdbtable             # SDL DDL design-time view def.
|       |- myView.hdbview               # SQL DDL design-time table def.
|       |- myIndex.hdbindex             # SDL DDL design-time index def.
|       |- myEntity.hdbcds              # HDB CDS design-time table def.
|       |- myDataType.hdbcds            # HDB CDS design-time type def.
|       |- myDoc.hdbcds                 # HDB CDS design-time data-model
|       |- data/                        # Data source files
|          \- myData.csv                # Table-data import sources (CSV)
|       |- roles                        #
|          |- myPriv.hdbstructuredprivilege # SQL DDL privilege definition
|          \- myRole.hdbrole            # Role definition (JSON)
|- web/
|   |- xs-app.json
|   \- resources/
|- js/
|   |- start.js
|   |- package.json
|   \- src/
|- security/
|   \- xs-security.json
\- mtad.yaml
```

As illustrated in the example above, the `/db/` folder contains all your design-time database artifacts, for example: tables, views, procedures, sequences, calculation views. In addition to the design-time database artifacts, the database content must also include the following components:

- `package.json`
  A file containing details of dependencies

- `.hdiconfig`
  The HDI container configuration
- `.hdinamespace`
  The run-time name-space configuration for the deployed objects

The HDI deployment tools deploy all database artifacts located in the `db/` folder to an HDI design-time container. At the same time, the HDI tools create the corresponding run-time container and populate the container with the specified catalog objects. The deployed objects can be referenced and consumed by your business applications.

> ⚠ Caution
>
> If the deployment tools establish that a database run-time artifact has no corresponding design-time definition, the run-time artifact (for example, a table) is dropped from the catalog.

Deployment Configuration Artifacts

The following files are mandatory for the database component of your deployment. One instance of each configuration artifact is required in the `db/src/` folder, and this instance applies to all sub-folders unless another instance exists in a sub folder:

- `/db/src/.hdiconfig`
  Used to bind the database artifacts you want to deploy (determined by a file suffix, for example, `.hdbcds`) with the appropriate build plug-in. For example, to deploy a CDS data model to an HDI container, you must configure the CDS build plug-in and, in addition, any other plug-ins required.

```
"hdbcds" : {
    "plugin_name" : "com.sap.hana.di.cds",
    "plugin_version" : "2.0.19.0" },
"hdbprocedure" : {
  "plugin_name"   : "com.sap.hana.di.procedure",
  "plugin_version": "2.0.20.0" },
"hdbtable" : {
  "plugin_name"   : "com.sap.hana.di.table",
  "plugin_version": "2.0.20.0" }
```

- `/db/src/.hdinamespace`
  Defines naming rules for the run-time objects which are created in the catalog when the application is deployed

```
{
  "name"      : "com.sap.hana.db.cds",
  "subfolder" : "append"
}
```

> ℹ Note
>
> For a name-space and build-plugin configuration to take effect, the name-space and build-plugin configuration file must be deployed - in the same way as any other design-time file.

Related Information

Creating Data Persistence Artifacts with SQL DDL in XS Advanced
Creating Data Persistence Artifacts with CDS in XS Advanced [page 134]
Defining the Data Model in XS Advanced
SAP HDI Artifact Types and Build Plug-ins Reference

4.1.2 HDI Design-Time Resources and Build Plug-ins

In XS advanced, database design-time resources must be mapped to a build plug-in for deployment purposes.

In XS advanced, design-time artifacts are distinguished by means of a unique file suffix that must be mapped to an HDI build plug-in. The following example of an abbreviated HDI configuration file (`.hdiconfig`) file illustrates how the design-time artifact types `.hdbcalculationview` and `.hdbcds` are mapped to their corresponding HDI build plug-in:

Code Syntax

`.hdiconfig`

```
"hdbtable" : {
   "plugin_name" : "com.sap.hana.di.table",
   "plugin_version": "2.0.0.0"
},
"hdbview" : {
   "plugin_name" : "com.sap.hana.di.view",
   "plugin_version": "2.0.0.0"
},
"hdbcalculationview" : {
   "plugin_name" : "com.sap.hana.di.calculationview",
   "plugin_version": "2.0.0.0"
},
"hdbcds" : {
   "plugin_name" : "com.sap.hana.di.cds",
   "plugin_version": "2.0.0.0"
}
```

The following table lists in alphabetical order the design-time artifacts you can develop and deploy with the SAP HANA Deployment Infrastructure (HDI) and describes the artifact's scope. The table also indicates which build plug-in is required to ensure successful deployment of the artifact. For more information about the individual artifact types and the configuration of the corresponding build plug-in, see *Related Links* below.

Default File-Suffix to HDI Build Plug-in Mappings

Artifact Suffix	Description/Content	Build Plug-in
`csv`	Source data for a table-import operation (also `hdbtabledata`)	`com.sap.hana.di.tabledata.source`
`.? (txt, copy only)`	An arbitrary design-time resource	`com.sap.hana.di.copyonly`

Artifact Suffix	Description/Content	Build Plug-in
`hdbafllangprocedure`	Definition of a language procedure for an application function library (AFL)	`com.sap.hana.di.afllangprocedure`
`hdbanalyticprivilege`	Definition of an XML-based analytic privilege	`com.sap.hana.di.analyticprivilege`
`hdbcalculationview`	Definition of a calculation view	`com.sap.hana.di.calculationview`
`hdbcds`	A document that contains the specification of one or more database objects written with the SAP HANA Core Data Services syntax	`com.sap.hana.di.cds`
`hdbconstraint`	Transforms a design-time constraint into a constraint on database tables	`com.sap.hana.di.constraint`
`hdbdropcreatetable`	Transforms a design-time table resource into a table database object	`com.sap.hana.di.dropcreatetable`
`hdbflowgraph`	Transforms a design-time flow-graph description into a corresponding set of database procedure or task objects	`com.sap.hana.di.flowgraph`
`hdbfulltextindex`	Full text index definition	`com.sap.hana.di.fulltextindex`
`hdbfunction`	Database function definition	`com.sap.hana.di.function`
`hdbgraphworkspace`	Definition of a graph work space resource	`com.sap.hana.di.graphworkspace`
`hdbindex`	Table index definition	`com.sap.hana.di.index`

Artifact Suffix	Description/Content	Build Plug-in
`hdbmrjob`	Transform a design-time Hadoop map-reduce job resource into a virtual-function-package database object **i Note** The plug-in for the file types (`.hdbmrjob` or `.jar`) has been superseded by the Virtual Package plug-in (for `.hdbvirtualpackage*` artifacts). Use the `hdbvirtualpackage*` plug-ins for design-time Hadoop and SparkSQL JAR files, respectively.	`com.sap.hana.di.virtualfunctionpackage.hadoop`
`jar`	Optional mapping, if you want direct access to Hadoop files	`com.sap.hana.di.virtualfunctionpackage.hadoop`
`hdblibrary`	A design-time library resource	`com.sap.hana.di.library`
`hdblogicalschema`	Transforms a design-time logical-schema definition into database objects that can be consumed by synonyms etc.	`com.sap.hana.di.logicalschema`
`hdbprocedure`	Definition of a database procedure	`com.sap.hana.di.procedure`
`hdbprojectionview`	Definition of a projection view	`com.sap.hana.di.projectionview`
`hdbprojectionviewconfig`	Configuration file for a projection view	`com.sap.hana.di.projectionview.config`
`hdbpublicsynonym`	Definition of a public database synonym	`com.sap.hana.di.publicsynonym`
`hdbreptask`	Transform a design-time replication task description into a corresponding set of database procedure or task operations	`com.sap.hana.di.reptask`
`hdbresultcache`	Definition of a result cache	`com.sap.hana.di.resultcache`
`hdbrole`	Definition of a database roles	`com.sap.hana.di.role`

Artifact Suffix	Description/Content	Build Plug-in
`hdbroleconfig`	Configuration of database privileges (and other roles) to be included in a database role	`com.sap.hana.di.roleconfig`
`hdbsearchruleset`	Definition of search configurations for built-in search procedure	`com.sap.hana.di.searchruleset`
`hdbsequence`	Definition of a database sequence	`com.sap.hana.di.sequence`
`hdbsynonym`	Database synonym definition	`com.sap.hana.di.synonym`
`hdbsynonymconfig`	Configuration file for a database synonym	`com.sap.hana.di.synonym.config`
`hdbstatistics`	Statistics definition file	`com.sap.hana.di.statistics`
`hdbstructuredprivilege`	Definition of analytic or structured privileges	`com.sap.hana.di.structuredprivilege`
`hdbsysbicsynonym`	Create synonyms in the _SYS_BIC schema that refer to database objects located in the target schema of the current container	`com.sap.hana.di.sysbicsynonym`
`hdbsystemversioning`	Transforms a design-time, system-versioned table that refers to a current and history table into a system-versioned table database object	`com.sap.hana.di.systemversioning`
`hdbtable`	Table operations	`com.sap.hana.di.table`
`hdbtabledata`	Definition of a data-import operation for a database table (also `csv`)	`com.sap.hana.di.tabledata`
`hdbtabletype`	Definition of a table type	`com.sap.hana.di.tabletype`
`hdbtextconfig`	Customization of the options used for text analysis	`com.sap.hana.di.tabletype`
`hdbtextdict`	Specification of the custom entity types and entity names to be used with text analysis	`com.sap.hana.di.textdictionary`
`hdbtextrule`	Specification of the rules (patterns) for extracting complex entities and relationships using text analysis	`com.sap.hana.di.textrule`

Artifact Suffix	Description/Content	Build Plug-in
`hdbtextinclude`	Definition of the rules to be used in one or more extraction rule sets for top-level, text analysis	`com.sap.hana.di.textrule.include`
`hdbtextlexicon`	Definition of the lists of words used in one or more top-level text analysis rule sets	`com.sap.hana.di.textrule.lexicon`
`hdbtextminingconfig`	Customization of the features and options used for text mining	`com.sap.hana.di.textmining config`
`hdbtrigger`	Database trigger definition	`com.sap.hana.di.trigger`
`hdbview`	View definition file	`com.sap.hana.di.view`
`hdbvirtualfunction`	Definition of a virtual database function	`com.sap.hana.di.virtualfunction`
`hdbvirtualfunctionconfig`	Configuration file for a virtual function	`com.sap.hana.di.virtualfunction.config`
`hdbvirtualpackage`	Transform a design-time Hadoop Map Reduce Job or SparkSQL resource into a virtual-package database object	`com.sap.hana.di.virtualpackage.hadoop`
`hdbvirtualprocedure`	Definition of a virtual database procedure	`com.sap.hana.di.virtualprocedure`
`hdbvirtualprocedureconfig`	Configuration file for the virtual database procedure definition	`com.sap.hana.di.virtualprocedure.config`
`hdbvirtualtable`	Definition of a virtual table	`com.sap.hana.di.virtualtable`
`hdbvirtualtableconfig`	Virtual table configuration file	`com.sap.hana.di.virtualtable.config`
`properties` `tags`	Properties file for a table-import operation	`com.sap.hana.di.tabledata.properties`

Related Information

SAP HDI Artifact Types and Build Plug-ins Reference
The SAP HDI Container Configuration File
SAP HDI Container Configuration File Syntax

4.2 Create a CDS Document (XS Advanced)

A CDS document is a design-time source file that contains definitions of the database objects you want to create in the SAP HANA catalog.

Context

CDS documents are design-time source files that contain DDL code that describes a persistence model according to rules defined in Core Data Services. CDS documents have the file suffix `.hdbcds`. Deploying the application with the database module that contains the CDS document creates the corresponding catalog objects in the corresponding schema.

> **i** Note
>
> The examples of CDS document elements includled in this topic are incomplete `[...]`; it is intended for illustration purposes only.

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following URL:

   `https://<HANA_HOST>:53075/`

   > **→** Tip
   >
   > To display the URL for the SAP Web IDE for SAP HANA, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > ```
   > xs app webide --urls
   > ```

2. Display the application project to which you want to add a CDS document.

   In XS advanced, SAP Web IDE for SAP HANA creates an application within a context of a project. If you do not already have a project, there are a number of ways to create one, for example: by importing it, cloning it, or creating a new one from scratch.

   a. In the SAP Web IDE for SAP HANA, choose *File* > *New* > *Project from Template*.
   b. Choose the project template type.

      Currently, there is only one type of project template available, namely: *Multi-Target Application Project*. Select *Multi-Target Application Project* and choose *Next*.

c. Type a name for the new MTA project (for example, **myApp** and choose *Next* to confirm.
d. Specify details of the new MTA project and choose *Next* to confirm.
e. Create the new MTA project; choose *Finish*.

3. Create the CDS document that defines the entity you want to create.

   Database artifacts such as the ones defined in a CDS document belong in the MTA's database module.

   > → Tip
   >
   > If you do not already have a database module, right-click the root folder of your new MTA project and, in the context menu, choose *New* > *HDB Module*. Name the new database model **db**.

   a. Navigate to the `src/` folder in your application's database module `db/`.
   b. Right-click the folder `myworld/db/src/` and choose *New* > *CDS Artifact* in the context menu.
   c. Name the new CDS artifact **myCDSModel**.

      The setup Wizard adds the mandatory suffix for CDS artifacts (`.hdbcds`) to the new file name automatically.

4. Define the details of the CDS artifacts.

   In the CDS document you just created , for example, `MyCDSModel.hdbcds`, add the CDS-definition code to the file. The CDS code describes the CDS artifacts you want to add, for example, entity definitions, type definitions, view definitions. Note that the following code examples are provided for illustration purposes only.

   a. Add structured types, if required.

      Use the `type` keyword to define a type artifact in a CDS document. In this example, you add the user-defined types and structured types to the top-level entry in the CDS document, the context `MyCDSModel`.

```
context MyCDSModel {
 type BusinessKey : String(10);
 type SString  : String(40);
 type MyStruct
 {
   aNumber     : Integer;
   aText       : String(80);
   anotherText : MyString80;  // defined in a separate type
 };
<[...]>
};
```

   b. Add a new context, if required.

      Contexts enable you to group together related artifacts. A CDS document can only contain one top-level context, for example, `MyModel {};`. Any new context must be **nested** within the top-level entry in the CDS document, as shown in the following example.

```
context MyCDSModel {
 type BusinessKey : String(10);
 type SString  : String(40);
 type <[...]>
 context MasterData {
 <[...]>
 };
 context Sales {
 <[...]>
 };
 context Purchases {
```

```
  <[...]>
  };
};
```

c. Add new entities.

You can add the entities either to the top-level entry in the CDS document (in this example, the context `MyCDSModel`) or to any other context, for example, `MasterData`, `Sales`, or `Purchases`. In this example, the new entities are column-based tables in the `MasterData` context.

```
context MyCDSModel {
 type BusinessKey : String(10);
 type SString  : String(40);
 type <[...]>
 context MasterData {
    Entity Addresses {
        key  AddressId: BusinessKey;
        City: SString;
        PostalCode: BusinessKey;
        <[...]>
    };
    Entity BusinessPartner {
        key  PartnerId: BusinessKey;
        PartnerRole: String(3);
        <[...]>
    };
 };
 context Sales {
 <[...]>
 };
 context Purchases {
 <[...]>
 };
};
```

5. Save the CDS document.

Related Information

Creating the Persistence Model in Core Data Services [page 10]
CDS Documents [page 19]

4.2.1 CDS Editors in XS Advanced

The SAP Web IDE for SAP HANA provides editing tools specially designed to help you create and modify CDS documents in XS advanced.

SAP Web IDE for SAP HANA includes dedicated editors that you can use to define data-persistence objects in CDS documents using the DDL-compliant Core Data Services syntax. SAP HANA XS advanced model recognizes the `.hdbcds` file extension required for CDS object definitions and, at deployment time, calls the appropriate plug-in to parse the content defined in the CDS document and create the corresponding run-time object in the catalog. If you right-click a file with the `.hdbcds` extension in the *Project Explorer* view of your application project, SAP Web IDE for SAP HANA provides the following choice of editors in the context-sensitive menu.

- CDS Text Editor [page 11]
  View and edit DDL source code in a CDS document as text with the syntax elements highlighted for easier visual scanning.
  Right-click a CDS document: *Open With* > *Text Editor*
- CDS Graphical Editor [page 12]
  View a graphical representation of the contents of a CDS source file, with the option to edit the source code as text with the syntax elements highlighted for easier visual scanning.
  Right-click a CDS document: *Open With* > *Graphical Editor*

CDS Text Editor

SAP Web IDE for SAP HANA includes a dedicated editor that you can use to define data-persistence objects using the CDS syntax. SAP HANA recognizes the `.hdbcds` file extension required for CDS object definitions and calls the appropriate repository plug-in. If you double-click a file with the `.hdbcds` extension in the *Project Explorer* view, SAP Web IDE for SAP HANA automatically displays the selected file in the CDS text editor.

The CDS editor provides the following features:

- Syntax highlights
  The CDS DDL editor supports syntax highlighting, for example, for keywords and any assigned values. To customize the colors and fonts used in the CDS text editor, choose *Tools* > *Preferences* > *Code Editor* > *Editor Appearance* and select a theme and font size.

  > **i Note**
  >
  > The CDS DDL editor automatically inserts the keyword *namespace* into any new DDL source file that you create using the *New* > *CDS Artifact* dialog.
  >
  > The following values are assumed:
  >
  > - *namespace* = `<ProjectName>.<ApplDBModuleName>`
  > - *context* = `<NewCDSFileName>`
- Keyword completion
  The editor displays a list of DDL suggestions that could be used to complete the keyword you start to enter. To change the settings, choose *Tools* > *Code Completion* in the toolbar menu.
- Code validity
  The CDS text editor provides syntax validation, which checks for parser errors as you type. Semantic errors are only shown when you build the XS advanced application module to which the CDS artifacts belong; the errors are shown in the console tab.
- Comments
  Text that appears after a double forward slash (`//`) or between a forward slash and an asterisk (`/*...*/`) is interpreted as a comment and highlighted in the CDS editor (for example, `//this is a comment`).

CDS Graphical Editor

The CDS graphical editor provides graphical modeling tools that help you to design and create database models using standard CDS artifacts with minimal or no coding at all. You can use the CDS graphical editor to create CDS artifacts such as entities, contexts, associations, structured types, and so on.

The built-in tools provided with the CDS Graphical Editor enable you to perform the following operations:

- Create CDS files (with the extension `.hdbcds`) using a file-creation wizard.
- Create standard CDS artifacts, for example: entities, contexts, associations (to internal and external entities), structured types, scalar types, ...
- Define technical configuration properties for entities, for example: indexes, partitions, and table groupings.
- Generate the relevant CDS source code in the text editor for the corresponding database model.
- Open in the CDS graphical editor data models that were created using the CDS text editor.

> → Tip
>
> The built-in tools included with the CDS Graphical Editor are context-sensitive; right-click an element displayed in the CDS Graphical editor to display the tool options that are available.

Related Information

Getting Started with the CDS Graphical Editor [page 282]

4.2.2 CDS Documents in XS Advanced

CDS documents are design-time source files that contain DDL code that describes a persistence model according to rules defined in Core Data Services.

In general, CDS works in XS advanced (HDI) in the same way that it does in the SAP HANA XS classic Repository, which is described in the *SAP HANA Developer Guide for SAP HANA Studio* or the *SAP HANA Developer Guide for SAP HANA Web Workbench* listed in the related links below. For XS advanced, however, there are some incompatible changes and additions, which are described in the following sections:

- General overview [page 149]
- Name Space [page 150]
- `@<annotations>` [page 150]
- Entity definitions [page 152]
- Structured Types [page 155]

> ℹ Note
>
> The following example of a CDS document for XS advanced is incomplete `[...]`; it is intended for illustration purposes only.

Sample Code

```
context MyModel {
 type BusinessKey : String(10);
 type SString  : String(40);
 type MyStruct
 {
   aNumber     : Integer;
   aText       : String(80);
   anotherText : MyString80;  // defined in a separate type
 };
 table type Structure2 { ... };
 context MasterData {
    Entity Addresses {
        key  AddressId: BusinessKey;
        City: SString;
        PostalCode: BusinessKey;
        <[...]>
    } technical configuration {
      column store;
      index MyIndex1 on (a, b) asc;
      unique index MyIndex2 on (c, s) desc;
      partition by hash (id) partitions 2,
          range (a) (partition 1 <= values < 10, partition values = 10,
partition others);
      group type Foo group subtype Bar group name Wheeeeezz;
      unload priority <integer_literal>;
    };
 };
 context Purchases {
 <[...]>
 };
};
```

General Overview

In XS advanced, CDS documents must have the file suffix `.hdbcds`, for example, `MyCDSDocument.hdbcds`. Each CDS document must contain the following basic elements:

- CDS artifact definitions
  The objects that make up your persistence model, for example: contexts, entities, structured types, and views

For XS advanced applications, the CDS document does not require a `namespace` declaration, and some of the `@<Annotations>` (for example, `@Schema` or `@Catalog`) are either not required or are no longer supported. Instead, most of the features covered by the `@<Annotations>` in XS classic can now be defined in the `technical configuration` section of the entity definition or in the view definition.

→ Tip

From SAP HANA 2.0 SPS 01, it is possible to define **multiple** top-level artifacts (for example, contexts, entities, etc.) in a single CDS document. For this reason, you can choose any name for the CDS source file; there is no longer any requirement for the name of the CDS source file to be the same as the name of the top-level artifact.

Multiple top-level artifacts are now allowed in a single CDS document, as illustrated in the following example:

Sample Code

Multiple Top-Level Artifacts in a CDS Document

```
type T : Integer;
entity E {
  key id : Integer;
  a : String(20);
};
context ctx1 {
  ...
};
context ctx2 {
  ...
};
```

Name Spaces

From SPS12, the declaration of a name space in a CDS document for XS advanced usage is optional.

Note

You can only omit the name-space declaration in a CDS document if no name space is defined in the corresponding HDI container-configuration file (`.hdinamespace`).

@annotations in CDS Documents

When you are defining CDS models for XS advanced application, bear in mind that there are restrictions on the annotations that you can use. In XS advanced, the `technical configuration` section of the corresponding entity definition CDS document is used to define much of what previously was defined in annotations. This section indicates which annotations are (or are not) allowed in CDS documents in XS advanced.

Supported CDS Annotations in XS Advanced

The following table lists the annotations that **are** supported in CDS models for XS advanced:

Supported CDS Annotations in SAP HANA XS Advanced

CDS Annotation	XS Advanced (HDI)
@OData.publish	Expose a CDS context or nested context (and any artifacts the context includes) as an OData version 4 service.

Note

For more information about publishing CDS contexts as OData services, see *Related Links*.

Annotations in Parameter Definitions in CDS Views

In a CDS view, parameter definitions can be annotated in the same way as any other artifact in CDS; the annotations must be prepended to the parameter name. Multiple annotations are separated either by whitespace or new-line characters.

> → Tip
>
> To improve readability and comprehension, it is recommended to include only one annotation assignment per line.

In the following example, the view `TargetWindowView` selects from the entity `TargetEntity`; the annotation `@positiveValuesOnly` is not checked; and the `targetId` is required for the `ON` condition in the entity `SourceEntity`.

> Sample Code
>
> Annotation Assignments to Parameter Definitions in CDS Views

```
annotation remark: String(100);

view TargetWindowView with parameters
   @remark: 'This is an arbitrary annotation'
   @positiveValuesOnly: true
   LOWER_LIMIT: Integer
as select from TargetEntity
{
  targetId,
  ....
} where targetId > :LOWER_LIMIT and targetId <= :LOWER_LIMIT + 10;
```

Unsupported CDS Annotations in XS Advanced

The following table lists the annotations that are **not** supported in CDS models for XS advanced:

Unsupported CDS Annotations in SAP HANA XS Advanced

CDS Annotation	XS Advanced (HDI)
@Catalog	To specify an index or table type in XS advanced, use the `technical configuration` section of the corresponding entity definition.
@nokey	In XS advanced, it is possible to define an entity without key elements without using an annotation.
@Schema	In XS advanced, schema handling is performed automatically by the HDI container.
@GenerateTableType	In SAP HANA XS advanced, no SAP HANA table type is generated for a structured type by default. To enforce the generation of an SAP HANA table type in SAP HANA XS advanced, use the keyword `table type` in a type definition instead of the keyword `type`, for example, `table type Structure2 { ... };`

CDS Annotation	XS Advanced (HDI)
@SearchIndex	To define a search index in XS Advanced, use the `technical configuration` section of the corresponding entity definition.
@WithStructuredPrivilegeCheck	In XS advanced, you define the privilege check in the view. `view MyView as select from Foo { <select_list> } <where_groupBy_Having_OrderBy> with structured privilege check;`

Entity Definitions

The definition of an entity can contain a section called `technical configuration`, which you use to define the elements listed in the following table:

Technical Configuration Features for Entities in SAP HANA XS Versions

Configuration Element	XS Advanced
Storage type [page 152]	✓
Indexes [page 153]	✓
Full text indexes [page 153]	✓
Partitioning [page 154]	✓
Grouping [page 154]	✓
Unload priority [page 155]	✓

> ### Note
>
> The syntax in the technical configuration section is as close as possible to the corresponding clauses in the SAP HANA SQL `Create Table` statement. Each clause in the technical configuration must end with a semicolon.

Storage Type

In XS advanced, the `@Catalog.tableType` annotation is not supported; you must use the technical configuration. In the technical configuration for an entity, you can use the `store` keyword to specify the storage type ("`row`" or "`column`") for the generated table, as illustrated in the following example. If no store type is specified, a "column" store table is generated by default.

> ### Sample Code

```
entity MyEntity {
  key id : Integer;
```

```
  a : Integer;
  }
  technical configuration {
    row store;
  };
```

The specification of table **type** is split into separate components. The storage type (`row store` or `column store`) is specified in the technical configuration section of the CDS entity definition; to generate a **temporary** table, use the keyword "`temporary entity`", as illustrated in the following example:

Sample Code

```
temporary entity MyEntity2 {
  ...
} technical configuration {
  row store;
};
```

Indexes

In XS advanced, the `@Catalog.index` annotation is not supported; you must use the technical configuration. In the technical configuration for an entity, you can use the `index` and `unique index` keywords to specify the index type for the generated table. For example: "`asc`" (ascending) or "`desc`" (descending) describes the index order, and `unique` specifies that the index is unique, where no two rows of data in the indexed entity can have identical key values.

Sample Code

```
} technical configuration {
  index MyIndex1 on (a, b) asc;
  unique index MyIndex2 on (c, s) desc;
};
```

Full Text Indexes

In the technical configuration for an entity, you can use the `fulltext index` keyword to specify the full-text index type for the generated table, as illustrated in the following example.

Sample Code

```
entity MyEntity {
  key id : Integer;
  t : String(100);
  s {
    u : String(100);
  };
} technical configuration {
  fulltext index MyFTI1 on (t) <fulltext_parameter_list>;
  fuzzy search index on (s.u);
};
```

The `<fulltext_parameter_list>` is identical to the standard SAP HANA SQL syntax for `CREATE FULLTEXT INDEX`. A `fuzzy search index` in the technical configuration section of an entity definition corresponds to the `@SearchIndex` annotation in XS classic and the statement "`FUZZY SEARCH INDEX ON`"

for a table column in SAP HANA SQL. It is not possible to specify both a full-text index and a fuzzy search index for the same element.

> **! Restriction**
>
> In XS advanced, the `@SearchIndex` annotation is not supported; you must use the technical configuration to define which of the columns should be indexed for search capabilities. In XS classic, it is not possible to use both the `@SearchIndex` annotation **and** the technical configuration (for example, `fulltext index`) at the same time. In addition, the full-text parameters `CONFIGURATION` and `TEXT MINING CONFIGURATION` are not supported.

Partitioning

In the technical configuration for an entity, you can use the `partition by` clause to specify the partitioning information for the generated table, as illustrated in the following example.

> **! Restriction**
>
> The `partition by` clause is only supported in XS advanced.

> **Sample Code**

```
entity MyEntity {
  key id : Integer;
  a : Integer;
} technical configuration {
  partition by hash (id) partitions 2,
               range (a) (partition 1 <= values < 10, partition values = 10,
partition others);
};
```

The syntax in the `partition by` clause is identical to the standard SAP HANA SQL syntax for the `PARTITION BY` expression in the `HANA SQL CREATE TABLE` statement.

Grouping

In the technical configuration for an entity, you can use the `group` clause to specify the partitioning information for the generated table, as illustrated in the following example.

> **! Restriction**
>
> The `group` clause is only supported in XS advanced.

> **Sample Code**

```
entity MyEntity {
  key id : Integer;
  a : Integer;
} technical configuration {
  group type Foo group subtype Bar group name Wheeeeezz;
};
```

The syntax in the `group` clause is identical to the standard SAP HANA SQL syntax for the `GROUP OPTION` expression in the `HANA SQL CREATE TABLE` statement.

Unload Priority

In the technical configuration for an entity, you can use the `Unload Priority` clause to specify the priority for unloading the generated table from memory, as illustrated in the following example:

> **! Restriction**
>
> The `unload priority` clause is only supported in XS advanced.

> **Sample Code**

```
entity MyEntity {
  <element_list>
} technical configuration {
  unload priority <integer_literal>;
};
```

The syntax in the `unload priority` clause is identical to the standard SAP HANA SQL syntax for the `UNLOAD PRIORITY` expression in the `HANA SQL CREATE TABLE` statement.

Structured Types

In the SAP HANA XS classic model, for each structured CDS type, an SAP HANA table type is generated by default in the repository. For this reason, in XS classic, the generation of table types could be controlled explicitly by means of the `@GenerateTableType` annotation. In SAP HANA XS advanced, however, no SAP HANA table type is generated for a structured type by default.

In SAP HANA XS advanced, to enforce the generation of an SAP HANA table type you must use the keyword `table type` instead of the keyword `type`, as illustrated in the following example:

> **! Restriction**
>
> The `table type` keyword is only supported in SAP HANA XS advanced.

```
type Struc1 { ... };       // no table type is generated
table type Struc2 { ... }; // creates a corresponding table type
```

You can define structured types that do not contain any elements, for example, using the keywords `type EmptyStruct { };`. In the example, below the generated table for entity “E” contains only one column: “a”.

> **→ Tip**
>
> It is not possible to generate an SAP HANA table type for an empty structured type.

```
type EmptyStruct { };
entity E {
  a : Integer;
  s : EmptyStruct;
};
```

Related Information

Create a CDS Document (XS Advanced) [page 144]
Create the Data Persistence Artifacts with CDS in XS Advanced [page 135]
Defining OData v4 Services for XS Advanced Java Applications

4.2.3 External CDS Artifacts in XS Advanced

You can define an artifact in one CDS document by referring to an artifact that is defined in another CDS document.

The CDS syntax enables you to define a CDS artifact in one document by basing it on an "external" artifact - an artifact that is defined in a separate CDS document. Each external artifact must be explicitly declared in the source CDS document with the `using` keyword, which specifies the location of the external artifact, its name, and where appropriate its CDS context.

> → Tip
>
> The `using` declarations must be located in the header of the CDS document between the `namespace` declaration and the beginning of the top-level artifact, for example, the `context`.

The external artifact can be either a single object (for example, a type, an entity, or a view) or a context. You can also include an optional alias in the `using` declaration, for example, `ContextA.ContextA1 as ic`. The alias (`ic`) can then be used in subsequent type definitions in the source CDS document.

> ! Restriction
>
> For SAP HANA XS advanced deployments, the file name of a design-time CDS artifact must use the extension `.hdbcds`, for example, `ContextA.hdbcds` and `ContextB.hdbcds`.

```
//Filename = Pack1/Distributed/ContextB.hdbcds
namespace Pack1.Distributed;
using Pack1.Distributed::ContextA.T1;
using Pack1.Distributed::ContextA.ContextAI as ic;
using Pack1.Distributed::ContextA.ContextAI.T3 as ict3;
using Pack1.Distributed::ContextA.ContextAI.T3.a as a;  // error, is not an
artifact
context ContextB {
  type T10 {
    a : T1;               // Integer
    b : ic.T2;            // String(20)
    c : ic.T3;            // structured
    d : type of ic.T3.b;  // String(88)
    e : ict3;             // structured
    x : Pack1.Distributed::ContextA.T1;  // error, direct reference not allowed
  };
  context ContextBI {
    type T1 : String(7);  // hides the T1 coming from the first using declaration
    type T2 : T1;         // String(7)
  };
};
```

The CDS document `ContextB.hdbcds` shown above uses external artifacts (data types `T1` and `T3`) that are defined in the "target" CDS document `ContextA.hdbcds` shown below. Two `using` declarations are present

in the CDS document `ContextB.hdbcds`; one with no alias and one with an explictly specified alias (`ic`). The first `using` declaration introduces the scalar type `Pack1.Distributed::ContextA.T1`. The second `using` declaration introduces the context `Pack1.Distributed::ContextA.ContextAI` and makes it accessible by means of the explicitly specified alias `ic`.

> **i Note**
>
> If no explicit alias is specified, the last part of the fully qualified name is assumed as the alias, for example `T1`.

The `using` keyword is the only way to refer to an externally defined artifact in CDS. In the example above, the type `x` would cause an activation error; you cannot refer to an externally defined CDS artifact directly by using its fully qualilfed name in an artifact definition.

You can use the "`using`" keyword to reference the following SAP HANA artifacts in XS advanced:

- CDS DDL documents (specify which document is used)
- CDS DCL documents (CDS access policy documents)
- Calculation views (`.hdbcalculationview`)
- SAP HANA entities (`.hdbcds`)
- SAP HANA user role definitions (`.hdbrole`)

```
//Filename = Pack1/Distributed/ContextA.hdbcds
namespace Pack1.Distributed;
context ContextA {
  type T1 : Integer;
  context ContextAI {
    type T2 : String(20);
    type T3 {
      a : Integer;
      b : String(88);
    };
  };
};
```

> **i Note**
>
> Whether you use a single or multiple CDS documents to define your data-persistence model, each CDS document must contain only **one** top-level artifact, and the name of the top-level artifact must correspond to the name of the CDS document. For example, if the top-level artifact in a CDS document is `ContextA`, then the CDS document itself must be named `ContextA.hdbcds`.

Related Information

Defining the Data Model in XS Advanced
Create a CDS Document (XS Advanced) [page 144]

4.2.4 CDS Naming Conventions in XS Advanced

Rules and restrictions apply to the names of CDS documents and the package in which the CDS document resides.

The rules that apply for naming CDS documents are the same as the rules for naming the packages in which the CDS document is located. When specifying the name of a package or a CDS document (or referencing the name of an existing CDS object, for example, within a CDS document), bear in mind the following rules:

- CDS source-file name
  From SAP HANA 2.0 SPS 01, it is possible to define **multiple** top-level artifacts (for example, contexts, entities, etc.) in a single CDS document. For this reason, you can choose any name for the CDS source file; there is no longer any requirement that the name of the CDS source file must be the same as the name of a top-level artifact.
- File suffix
  The file suffix differs according to SAP HANA XS version:
  - XS classic
    `.hdbdd`, for example, `MyModel.hdbdd`.
  - XS advanced
    `.hdbcds`, for example, `MyModel.hdbcds`.
- Permitted characters
  CDS object and package names can include the following characters:
  - Lower or upper case letters (aA-zZ) and the underscore character (_)
  - Digits (0-9)
- Forbidden characters
  The following restrictions apply to the characters you can use (and their position) in the name of a CDS document or a package:
  - You cannot use either the hyphen (-) or the dot (.) in the name of a CDS document.
  - You cannot use a digit (0-9) as the first character of the name of either a CDS document or a package, for example, `2CDSobjectname.hdbdd` (XS classic) or `acme.com.1package.hdbcds` (XS advanced).
  - The CDS parser does not recognize either CDS document names or package names that consist **exclusively** of digits, for example, `1234.hdbdd` (XS classic) or `acme.com.999.hdbcds` (XS advanced).

> ⚠ Caution
>
> Although it is possible to use quotation marks (“”) to wrap a name that includes forbidden characters, as a general rule, it is recommended to follow the naming conventions for CDS documents specified here in order to avoid problems during activation in the repository.

Related Information

Create a CDS Document (XS Advanced) [page 144]
CDS Documents in XS Advanced [page 148]

4.2.5 Accessing CDS Metadata in HDI

CDS metadata is available as standard SQL monitoring views and table functions.

As of SAP HANA 1.0 SP11, any CDS catalog metadata, which was created by the CDS plug-in of the HANA Deployment Infrastructure (HDI), `com.sap.hana.di.cds`, is available as standard SQL monitoring views (CDS_*) and one table function (CDS_ARTIFACT_DEFINITION). In the same was as any SAP HANA SQL metadata, the views listing CDS metadata are available in schema SYS. The exposure of CDS metadata by means of SQL views enables consumers to combine CDS and database metadata with the SQL JOIN command.

The CDS catalog metadata enables you to reconstruct a CDS source; if the reconstructed source is processed again by the CDS compiler, it produces the same metadata even if the recreated CDS source is not identical to the original source. The result set structure of the main CDS metadata access point, for example, the table function “CDS_ARTIFACT_DEFINITION” is be minimal. If necessary, it can be joined with database catalog system views and views containing other CDS related metadata in order to enrich the result set with additional, more specific metadata.

> ℹ Note
>
> It is very important to control the run time access to the CDS metadata. Although the user has access to the CDS metadata SQL objects (in schema SYS), instance-based access control ensures that the metadata content exposed to a user is only that which the requesting user is authorized to see.

The instance based CDS metadata access control is managed at the HDI run time container schema level by means of a newly introduced schema level object privilege “SELECT CDS METADATA”. Since the HDI run time container schema is created and owned by an HDI internal technical user, the CDS metadata privilege cannot be assigned using the usual GRANT command. Instead, the granting must be performed using an built-in administrative stored procedure “GRANT_CONTAINER_SCHEMA_PRIVILEGES”; the stored procedure is provided by SAP HANA HDI in a schema which is related to the actual HDI run time container schema (for example, named `<HDI_RT_Container_Name>#DI`).

> → Tip
>
> You can also use HDI container tools to granting the privilege “SELECT CDS METADATA” to an HDI run time container schema.

Related Information

CDS Catalog Reader API for HDI [page 160]
Managing SAP HDI Containers and Artifacts

4.2.6 CDS Catalog Reader API for HDI

The new SQL-based CDS metadata access API in schema SYS is only available CDS content created for (and deployed in) the SAP HANA HDI.

From SAP HANA 1.0 SPS 11, any CDS catalog metadata that is created by the CDS plugin of the HANA Deployment Infrastructure (HDI), `com.sap.hana.di.cds`, is available as standard SQL monitoring views (`CDS_*`) and one table function (`CDS_ARTIFACT_DEFINITION`). In the same way as HANA SQL metadata, these views are available in schema SYS. The exposure of CDS catalog metadata in SQL views enables consumers to combine CDS and database metadata by means of the SQL JOIN command.

CDS Catalog Metadata Views

View Name	Description
CDS_ARTIFACT_DEFINITION	Retrieves CDS metadata for a given CDS artifact name
CDS_ARTIFACT_NAMES	Exposes all CDS artifact names and their kind for all schemas
CDS_ANNOTATION_VALUES	Exposes multiple rows, if a single CDS artifact is annotated with multiple annotation values
CDS_ANNOTATION_ASSIGNMENTS	Returns a flat list of annotation assignments
CDS_ASSOCIATIONS	Expose association metadata and definitions
CDS_ENTITIES	Expose entity (table) metadata and definitions
CDS_VIEWS	Expose view metadata and definitions

CDS_ARTIFACT_DEFINITION

`CDS_ARTIFACT_DEFINITION` is intended to be the main entry point to retrieve CDS metadata for a given CDS artifact name. If no CDS artifact name is known, the `CDS_ARTIFACT_NAMES` view can be used to retrieve all artifact names for a given schema. The other CDS_* monitoring views can be accessed individually, too, but are actually intended for use by joining with the result of CDS_ARTIFACT_DEFINITION, to enrich the result with additional, more detailed CDS metadata.

The `CDS_ARTIFACT_DEFINITION TableFunction` expects exactly two mandatory **input** parameters; the parameters specify the requested CDS artifact name in a certain schema; that is, the corresponding HDI runtime container. The input parameters are as follows:

- `SCHEMA_NAME (NVARCHAR(256))`
- `ARTIFACT_NAME (NVARCHAR(127))`

Result-Set Structure

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	The HDI runtime container schema name. The returned schema name always matches the schema name requested when calling the `TableFunction`. Since HDI allows you to deploy the same CDS artifact name into the same namespace in two different runtime container schemas, it is necessary to specify the schema name as part of the ON-condition of SQL JOINS, if additional metadata is selected from additional monitoring views.
ARTIFACT_NAME	NVARCHAR(127)	The fully qualified CDS artifact name, including the namespace, separated by a double colon "`::`", for example, `namespaceComponent1.namespaceComponent1::ContextName.EntityName`
ELEMENT_NAME	NVARCHAR(127)	CDS element name is populated with the element name, if `ARTIFACT_KIND` is "`ELEMENT`". The name can also be a path with the name components separated by a double colon "`::`", where flattened elements of (nested) structures are included.
ARTIFACT_KIND	VARCHAR(32)	The CDS artifact kind, for example, one of the following enumeration values: • ANNOTATION • ARRAY_TYPE • ASSOCIATION • ASSOCIATION_ELEMENT • CONTEXT • DERIVED_TYPE • ELEMENT • ENTITY • ENUM • STRUCTURED_TYPE • VIEW
PARENT_ARTIFACT_NAME	NVARCHAR(127)	The name of the parent CDS artifact. NULL for top-level root artifacts. **→ Tip** The parent relationship is meant in terms of structural definition; not the usage dimension.

Column Name	Type	Description
PARENT_ARTIFACT_ELEMENT	NVARCHAR(127)	The name of the parent CDS element. The PARENT_ARTIFACT_ELEMENT column has a value in the following cases: 1. Anonymous, non-primitive type definitions “behind” elements have the element as parent artifact, for example: “`elem: array of String(3);`”. The array type definition has the “`elem`” artifact as parent. The row representing the element definition refers to the anonymous array type definition in the USED_ARTIFACT_NAME column. 2. For flattened elements, non-top-level elements refer to their parent elements, which were also created as part of the flattening process.
IS_ANONYMOUS	VARCHAR(5)	Specifies whether the artifact definition is anonymous: TRUE or FALSE (Default). Artifact definitions in CDS can be anonymous in the sense that there is no identifier specified for them in the source. **i Note** “anonymous” artifact definitions are internally still supplied with a unique artifact name, which is constructed by the convention: Parent artifact name, concatenated with "." and the relative local component name. Elements are always syntactically named in the CDS source and are therefore always represented as non-anonymous, even if an enclosing structured type definition itself might be anonymous. The same is true for the foreign key elements of managed associations, which are represented with artifact kind ASSOCIATION_ELEMENT.
USED_ARTIFACT_SCHEMA	NVARCHAR(256)	The schema name of the used CDS artifact. (See also column SCHEMA_NAME). The triplet USED_ARTIFACT_SCHEMA, NAME, and KIND is similar to OBJECT_SCHEMA, NAME, and TYPE in SYS.SYNONYMS; it defines a reference to another CDS artifact definition, which is used. These three columns are only filled for usage relationships, not for nested definitions.

Column Name	Type	Description
USED_ARTIFACT_NAME	NVARCHAR(127)	The name of the used CDS artifact. If CDS or SAP HANA database (built-in) primitive types are used, the USED_ARTIFACT_NAME is filled and the kind is PRIMITIVE_TYPE. No value is given for the SCHEMA because CDS built-in types are virtual and do not exist in a actual database schema. **i Note** The list of available CDS primitive types is described in the *SAP HANA Core Data Services (CDS) Reference* available on the SAP Help Portal. CDS built-in primitive types are prefixed with namespace "`sap.cds::`". SAP HANA types that are usable but deprecated within CDS are located in the top-level context "hana" (prefixed with "hana"). If explicitly defined custom CDS artifacts are used, the USED_ARTIFACT_KIND values will be one of the following: DERIVED_TYPE, ANNOTATION, ARRAY, CONSTANT, or ENUM.
USED_ARTIFACT_KIND	VARCHAR(32)	The kind of CDS artifact used
ORDINAL_NUMBER	INTEGER	The ordinal position of the CDS artifact definition within the returned artifact tree for the requested artifact name. The ordinal number starts with 0 and is increased in pre-order manner during depth-first traversal of the requested artifact tree. **i Note** The CDS ordinal number must not be confused with the column order of tables at the database level. The user of the API can JOIN the POSITION information of the SYS.COLUMNS view to prefix the column position in the generated database object with the CDS ORDINAL_NUMBER.
SQL_DATA_TYPE_NAME	VARCHAR(16)	The SQL data type name, which is only filled for elements with built-in CDS primitive types. See also column DATA_TYPE_NAME in SYS.COLUMNS view.
TYPE_PARAM_1	INTEGER	The number of characters for character types; maximum number of digits for numeric types; number of bytes for LOB types; SRID value for GIS (geospatial) types. **i Note** Internal values for time and timestamp data type are not exposed. If such information is required, the user of the CDS metadata SQL API has to JOIN the CDS element row with the SYS.COLUMNS view.
TYPE_PARAM_2	INTEGER	The numeric types: the maximum number of digits to the right of the decimal point

Column Name	Type	Description
IS_NULLABLE	VARCHAR(5)	Specifies whether the database column generated for this CDS element is allowed to accept null values. Set to TRUE (default) or FALSE. See also column IS_NULLABLE in SYS.COLUMNS view.
IS_KEY	VARCHAR(5)	Specifies whether the column is part of the primary key, in order of the definition in the source. Set to TRUE or FALSE (default). The order of the columns in the key in given by the order of the elements in the Entity. The key information is also available for the flattened elements, for structured-type usage for elements that are part of the key.
VALUE	NVARCHAR(5000)	Default value in case of `ARTIFACT_KIND = 'ELEMENT'` and constant value in case of `ARTIFACT_KIND = 'CONSTANT'`.
AUX_ELEMENT_INFO	NVARCHAR(5000)	This column exposes the following auxiliary element related properties: • Original name path if used with "as" for foreign key element definitions of managed associations (ARTIFACT_KIND = ASSOCIATION_ELEMENT) • The period element kind in case of series entities: PERIOD or ALTERNATE_PERIOD

CDS_ARTIFACT_NAMES

This view exposes all CDS artifact names and their kind for all schemas the user has the `SELECT CDS METADATA` privilege for. It is intended for use with a "`WHERE`" condition, for example, to restrict the result set to **all** artifacts defined within one single schema (HDI runtime container) or CDS namespace.

Result-Set Structure

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	The name of the SAP HANA HDI run time container schema
ARTIFACT_NAME	NVARCHAR(127)	The fully qualified CDS artifact name, including the namespace, separated by a double colon "`::`"

Column Name	Type	Description
ARTIFACT_KIND	VARCHAR(32)	The artifact kind, for example, one of the following enumeration values: • ANNOTATION • ARRAY_TYPE • ASSOCIATION • CONTEXT • DERIVED_TYPE • ENTITY • ENUM • STRUCTURED_TYPE • VIEW

CDS_ANNOTATION_VALUES

The view `SYS_RT.CDS_ANNOTATION_VALUES` exposes multiple rows if a single CDS artifact is annotated with multiple annotation values. This allows you to select all artifact names that are annotated with a certain annotation. The values of CDS internal annotations that are prefixed with namespace "`sap.cds::`" are not exposed with this view.

Result-Set Structure

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	The name of the SAP HANA HDI run time container schema
ARTIFACT_NAME	NVARCHAR(127)	The name of the annotated CDS artifact
ELEMENT_NAME	NVARCHAR(127)	Element name, if an element is annotated
ANNOTATION_SCHEMA_NAME	NVARCHAR(256)	Schema name of annotation definition
ANNOTATION_NAME	NVARCHAR(127)	Name of the annotation definition **i Note** For built-in CDS annotations, no entry can be found in SYS.CDS_ANNOTATIONS. This is because the definitions of built-in CDS annotations are not delivered as CDS catalog content; they are defined during compiler initialization.
VALUE	NCLOB	The annotation value tree, serialized as JSON, for example: `{ "value": "<SomeAnnotationValue>" }`

CDS_ANNOTATION_ASSIGNMENTS

This view in the CDS catalog API returns a flat list of annotation assignments:

Sample Code

```
select * from SYS.CDS_ANNOTATION_ASSIGNMENTS
```

The view `SYS_RT.CDS_ANNOTATION_VALUES` exposes multiple rows if a single CDS artifact is annotated with multiple annotation values. This allows you to select all artifact names that are annotated with a certain annotation. The values of CDS internal annotations that are prefixed with namespace "`sap.cds::`" are not exposed with this view.

Result-Set Structure

Column Name	Description
SCHEMA_NAME	The schema of the annotated object
ARTIFACT_NAME	The name of the annotated CDS object
COMPONENT_NAME	The name of the annotated element (if the annotation is assigned to an element of an object)
ANNOTATION_NAME	The name of the annotation
EXTENSION_PACKAGE_NAME	The name of the extension package, if the assignment happens in an extension
FORMAT	The format of the annotation value; possible values for FORMAT are: `Enum`, `Boolean`, `String`, `Number`, `Array`, and `Other`. **Note** Values for array-like annotations are always returned as "blob"; flattening stops at arrays.
VALUE	The annotation value
VALIDATION_RESULT	Possible values for validation results are: • CORRECT A definition exists for the annotation, and the assignment matches the definition. • MISMATCHED A definition exists for the annotation, and the assignment **does not** match the definition. • UNCHECKED No definition exists for the annotation.
DEFINITION_SCHEMA_NAME	The schema of the annotation definition (if a definition exists)
DEFINITION_NAME	The name of annotation definition (if a definition exists)

The following is an example of a CDS source:

```
annotation KnownAnno {
  a : Integer;
  b : String(100);
  c : Integer;
  t : UTCTimestamp;
};
@KnownAnno : { a:1 , b:'some value' , c:'no int' , t:'2017-03-02 10:33'}
type T : Integer;
@ArrayAnno : [1,2,3]
type S {
  x : Integer;
  @UnknownAnno : { c:3 , d:'hi there' , e:true}
  y : Integer;
}
```

The following table shows the resulting annotation assignments:

> **i Note**
>
> The contents of the columns SCHEMA_NAME and DEFINITION_SCHEMA_NAME are not shown. SCHEMA_NAME always shows the name of the schema of the HDI container where the model is deployed. This is also true for DEFINITION_SCHEMA_NAME, if the annotation has a definition.

Annotation Assignments 1

ARTIFACT_NAME	COMPONENT_NAME	ANNOTATION_NAME	FORMAT
S		ArrayAnno	Array
S	y	UnknownAnno.c	Number
S	y	UnknownAnno.d	String
S	y	UnknownAnno.e	Boolean
T		KnownAnno.a	Number
T		KnownAnno.b	String
T		KnownAnno.t	String
T		KnownAnno.c	String

Annotation Assignments 2

ARTIFACT_NAME	VALUE	VALIDATION_RESULT	DEFINITION_NAME
S	[1,2,3]	UNCHECKED	
S	3	UNCHECKED	
S	hi there	UNCHECKED	
S	true	UNCHECKED	

ARTIFACT_NAME	VALUE	VALIDATION_RESULT	DEFINITION_NAME
T	1	CORRECT	KnownAnno
T	some value	CORRECT	KnownAnno
T	2017-03-02 10:33	CORRECT	KnownAnno
T	no int	MISMATCHED	KnownAnno

CDS_ASSOCIATIONS

Association definitions are already exposed as part of the `CDS_ARTIFACT_DEFINITION` result set, represented by `ARTIFACT_KIND "ASSOCIATION"`. Foreign key elements are also available with kind `ASSOCIATION_ELEMENT`, which allows then to be distinguished from "normal" elements.

The CDS association metadata can be retrieved with the following SQL SELECT statement:

Sample Code

```
SELECT * FROM CDS_ARTIFACT_DEFINITION('<SomeSchemaName>',
'<SomeCdsArtifactName>') def
LEFT OUTER JOIN SYS.CDS_ASSOCIATIONS assoc
ON def.SCHEMA_NAME = assoc.SCHEMA_NAME AND def.ARTIFACT_NAME =
assoc.ASSOCIATION_NAME
WHERE def.ARTIFACT_KIND = 'ASSOCIATION' OR def.ARTIFACT_KIND =
'ASSOCIATION_ELEMENT';
```

Result-Set Structure

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	The name of the SAP HANA HDI run time container schema
ASSOCIATION_NAME	NVARCHAR(127)	The name of the CDS entity
ASSOCIATION_KIND	NVARCHAR(32)	One of the following supported association-specific enumeration values: • UNMANAGED • FOREIGN_KEY_EXPLICIT • FOREIGN_KEY_IMPLICIT
TARGET_ARTIFACT_SCHEMA_NAME	NVARCHAR(127)	Name of the schema for the associated CDS artifact.

Column Name	Type	Description
TARGET_ARTIFACT_NAME	NVARCHAR(127)	Name of the associated CDS artifact **! Restriction** Only CDS entities are allowed as the target of an association.
TARGET_CARDINALITY_MIN	INTEGER	Minimum cardinality of association target. Default is 0.
TARGET_CARDINALITY_MAX	INTEGER	Maximum cardinality of association target. Default is 1. -1 represents unlimited.
JOIN_CONDITION	NCLOB	The join condition for unmanaged associations (ASSOCIATION_KIND = UNMANAGED)

CDS_ENTITIES

Result-Set Structure

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	The name of the SAP HANA HDI run time container schema
ENTITY_NAME	NVARCHAR(127)	The name of the CDS entity
SERIES_KIND	NVARCHAR(32)	One of the following supported series-kind enumeration values: • NO_SERIES • NOT_EQUIDISTANT, EQUIDISTANT • EQUIDISTANT_PIECEWISE

CDS_VIEWS

Result-Set Structure

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	The name of the SAP HANA HDI run time container schema
VIEW_NAME	NVARCHAR(127)	The name of the CDS view
DEFINITION	NCLOB	The view definition string. Relative names are resolved to absolute names already and constant expressions to concrete values.

Related Information

Accessing CDS Metadata in HDI [page 159]

4.2.7 CDS Contexts in XS Advanced

You can define multiple CDS-compliant elements (for example, entities or views) in a single file by assigning them to a *context*.

The following example illustrates how to assign two simple entities to a context using the CDS-compliant syntax; you store the context-definition file with a specific name and the file extension `.hdbcds`, for example, `MyContext.hdbcds`.

> → Tip
>
> From SAP HANA 2.0 SPS 01, it is possible to define **multiple** top-level artifacts (for example, contexts, entities, etc.) in a single CDS document. For this reason, you can choose any name for the CDS source file; there is no longer any requirement that the name of the CDS source file must be the same as the name of a top-level artifact.

In the example below, you must save the context definition "Books" in the CDS document `Books.hdbcds`. In XS advanced, the name space declaration is optional.

The following code example illustrates how to use the CDS syntax to define multiple design-time entities in a context named `Books`.

```
namespace com.acme.myapp1;
@OData.publish : true      //OData v4 only
context Books {
    entity Book {
        key AuthorID : String(10);
        key BookTitle : String(100);
        ISBN : Integer not null;
        Publisher : String(100);
    } technical configuration {
        column store;
        unique index MYINDEX1 on (ISBN) desc;
    };
    entity Author {
        key AuthorName : String(100);
        AuthorNationality : String(20);
        AuthorBirthday : String(100);
        AuthorAddress : String(100);
    } technical configuration {
        column store;
        unique index MYINDEX1 on (AuthorNationality) desc;
    };
};
```

Activation of the file `Books.hdbcds` containing the context and entity definitions creates the catalog objects "Book" and "Author".

> ℹ Note
>
> The namespace specified at the start of the file, for example, `com.acme.myapp1` corresponds to the location of the entity definition file (`Books.hdbcds`) in the application-package hierarchy. The namespace

is not required in XS advanced scenarios; it is optional and provided only for backwards compatibility with XS classic.

This section includes details of the following concepts and features:

- Nested Contexts [page 171]
- Name Resolution Rules [page 172]
- OData Services [page 173]

Nested Contexts

The following code example shows you how to define a nested context called `InnerCtx` in the parent context `MyContext`. The example also shows the syntax required when making a reference to a user-defined data type in the nested context, for example, (`field6 : type of InnerCtx.CtxType.b;`).

The `type of` keyword is only required if referencing an element in an entity or in a structured type; types in another context can be referenced directly, without the `type of` keyword. The nesting depth for CDS contexts is restricted by the limits imposed on the length of the database identifier for the name of the corresponding SAP HANA database artifact (for example, table, view, or type); this is currently limited to 126 characters (including delimiters).

> **i Note**
>
> The context itself does not have a corresponding artifact in the SAP HANA catalog; the context only influences the names of SAP HANA catalog artifacts that are generated from the artifacts defined in a given CDS context, for example, a table or a structured type.

```
namespace com.acme.myapp1;
context MyContext {
    // Nested contexts
    context InnerCtx {
        Entity MyEntity {
            key id : Integer;
            // <...>
        };
        Type CtxType {
            a : Integer;
            b : String(59);
        };
    };
    type MyType1 {
        field1 : Integer;
        field2 : String(40);
        field3 : Decimal(22,11);
        field4 : Binary(11);
    };
    type MyType2 {
        field1 : String(50);
        field2 : MyType1;
    };
    type MyType3 {
        field1 : UTCTimestamp;
        field2 : MyType2;
    };
    entity MyEntity1 {
        key id : Integer;
```

```
        field1 : MyType3 not null;
        field2 : String(24);
        field3 : LocalDate;
        field4 : type of field3;
        field5 : type of MyType1.field2;
        field6 : type of InnerCtx.CtxType.b; // refers to nested context
        field7 : InnerCtx.CtxType;           // more context references
    } technical configuration {
        unique index IndexA on (field1) asc;
    };
};
```

Name Resolution Rules

The sequence of definitions inside a block of CDS code (for example, `entity` or `context`) does not matter for the scope rules; a binding of an aritfact type and name is valid within the confines of the smallest block of code containing the definition, except in inner code blocks where a binding for the same identifier remains valid. This rules means that the definition of `nameX` in an inner block of code hides any definitions of `nameX` in outer code blocks.

> **i Note**
>
> An identifier may be used before its definition without the need for forward declarations.

```
context OuterCtx
{
  type MyType1 : Integer;
  type MyType2 : LocalDate;
  context InnerCtx
  {
    type Use1 : MyType1;           // is a String(20)
    type Use2 : MyType2;           // is a LocalDate
    type MyType1 : String(20);
  };
   type invalidUse : Use1;       // invalid: Use1 is not
                                 // visible outside of InnerCtx
   type validUse   : InnerCtx.Use1;   // ok
};
```

No two artifacts (including namespaces) can be defined whose absolute names are the same or are different only in case (for example, `MyArtifact` and `myartifact`), even if their artifact type is different (entity and view). When searching for artifacts, CDS makes no assumptions which artifact kinds can be expected at certain source positions; it simply searches for the artifact with the given name and performs a final check of the artifact type.

The following example demonstrates how name resolution works with multiple nested contexts, Inside context `NameB`, the local definition of `NameA` shadows the definition of the context `NameA` in the surrounding scope. This means that the definition of the identifier `NameA` is resolved to `Integer`, which does not have a sub-component `T1`. The result is an error, and the compiler does not continue the search for a “better” definition of `NameA` in the scope of an outer (parent) context.

```
context OuterCtx
{
  context NameA
  {
    type T1 : Integer;
```

```
    type T2 : String(20);
  };
  context NameB
  {
    type NameA : Integer;
    type Use   : NameA.T1;  // invalid: NameA is an Integer
    type Use2  : OuterCtx.NameA.T2;  // ok
  };
};
```

Publishing OData Services

You can use the Boolean `@OData.publish` annotation at the context level to indicate that the annotated CDS Context should be exposed (`true`) as an OData service.

! Restriction

The `@OData.publish` feature only works with OData version 4.

The `@OData.publish` annotation cannot be used to publish individual CDS entities, or CDS views. In addition, the annotation `@OData.publish` annotation cannot be used to publish a CDS context that includes a subcontext.

All contexts defined in a CDS document and annotated with `@OData.publish` are published as OData v4 services. There is no restriction on the number of CDS contexts that can be annotated as `OData.publish : true`. In the following example, the CDS artifacts `MyEntity1` and `MyEntity2` defined in the CDS context `ContextA` are exposed for consumption by OData.

‹≡› Sample Code

Publish a CDS Context (and Contents) as an OData v4 Service

```
namespace acme.com;
@OData.publish : true
context ContextA {
  MyEntity1 {
    key ID : Integer;
  };
  MyEntity2 {
    key ID : Integer;
    type Address1
  };
};
```

i Note

Although an annotated context that includes a nested (sub) context cannot be published as an OData service, it is possible to publish the individual nested contexts. However, any nested context annotated for OData publication must not include any nested contexts of its own.

In the following example, the CDS artifacts defined in the subcontexts `SubContextA1` and `SubContextA2` are exposed as OData v4 services.

Sample Code

Publish CDS Subcontexts an OData v4 Services

```
namespace acme.com;
context ContextA {
  @OData.publish : true
  context ContextA1 {
    MyEntity3 {
      key ID : Integer;
      name : String(80);
      number : Integer;
    };
    MyEntity4 {
      key ID : Integer;
      name : String(80);
    };
  };
  @OData.publish : true
  context ContextA2 {
    MyEntity5 {
      key ID : Integer;
      field1 : Address1;
    };
    type Address1 {
      a : Integer;
      b : String(40);
    };
  };
};
```

Namespaces in OData v4 Service URLs

The name of the OData service created from a CDS context in a specified name space is: `<name.space>._.<context>`. In the OData query, the name space and the context name (which is also the OData service name) are separated by the character combination "._." (dot underscore dot). If no name space is declared in the CDS document, the name space and separator "._." are omitted from the service name; the service name is the same as the context name it is derived from. For example, assuming the context `ContextA` is published as an OData service, the OData service name used in the query is as follows:

- Name space specified ("acme.com") in the CDS document (no nested contexts)
  OData Service Name: `http::/<BaseURL>/java/odata/v4/<acme.com>._.<ContextA>/$metadata`
- Name space **not** specified in the CDS document (no nested contexts)
  OData Service Name: `http::/<BaseURL>/java/odata/v4/<ContextA>/$metadata`

If the CDS document includes a nested **subcontext** that is annotated for publication as an OData v4 service, you can access the exposed artifacts defined in the nested **subcontext** directly, by specifying the complete context path (separated by a ".") in the query URL, as illustrated in the following examples:

- Subcontext `ContextA1`
  `https://<BaseURL>/java/odata/v4/<acme.com>._.<ContextA>.<ContextA1>/$metadata`
- Subcontext `ContextA2`
  `https://<BaseURL>/java/odata/v4/<acme.com>._.<ContextA>.<ContextA2>/$metadata`

Related Information

Create a CDS Document (XS Advanced) [page 144]
Defining OData v2 Services for XS Advanced JavaScript Applications
Tutorial: Create OData v4 Services for XS Advanced Java Applications

4.2.8 CDS Comment Types in XS Advanced

The Core Data Services (CDS) syntax enables you to insert comments into object definitions.

Example: Comment Formats in CDS Object Definitions

```
namespace com.acme.myapp1;

  /**
   *  multi-line comment,
   *  for doxygen-style,
   *  comments and annotations
   */
  type Type1 {
       element Fstr:       String( 5000 ); // end-of-line comment
               Flstr:      LargeString;
       /*inline comment*/  Fbin:       Binary( 4000 );
       element Flbin:      LargeBinary;
               Fint:       Integer;
       element Fint64:     Integer64;
               Ffixdec:    Decimal( 34, 34 /* another inline comment */);
       element Fdec:       DecimalFloat;
               Fflt:       BinaryFloat;
       //complete line comment element Flocdat:    LocalDate;    LocalDate
temporarily switched off
       //complete line comment        Floctim:    LocalTime;
       element Futcdatim:  UTCDateTime;
               Futctstmp:  UTCTimestamp;
  };
```

Overview

You can use the forward slash (`/`) and the asterisk (`*`) characters to add comments and general information to CDS object-definition files. The following types of comment are allowed:

- In-line comment
- End-of-line comment
- Complete-line comment
- Multi-line comment

In-line Comments

The in-line comment enables you to insert a comment into the middle of a line of code in a CDS document. To indicate the start of the in-line comment, insert a forward-slash (`/`) followed by an asterisk (`*`) before the comment text. To signal the end of the in-line comment, insert an asterisk followed by a forward-slash character (`*/`) after the comment text, as illustrated by the following example:.

```
element Flocdat:  /*comment text*/  LocalDate;
```

End-of-Line Comment

The end-of-line comment enables you to insert a comment at the end of a line of code in a CDS document. To indicate the start of the end-of-line comment, insert two forward slashes (`//`) before the comment text, as illustrated by the following example:.

```
element Flocdat:    LocalDate; // Comment text
```

Complete-Line Comment

The complete-line comment enables you to tell the parser to ignore the contents of an entire line of CDS code. The comment out a complete line, insert two backslashes (`//`) at the start of the line, as illustrated in the following example:

```
//  element Flocdat:    LocalDate;      Additional comment text
```

Multi-Line Comments

The multi-line comment enables you to insert comment text that extends over multiple lines of a CDS document. To indicate the start of the multi-line comment, insert a forward-slash (`/`) followed by an asterisk (`*`) at the start of the group of lines you want to use for an extended comment (for example, `/*`). To signal the end of the multi-line comment, insert an asterisk followed by a forward-slash character (`*/`). Each line between the start and end of the multi-line comment must start with an asterisk (`*`), as illustrated in the following example:

```
/*
*  multiline,
*  doxygen-style
*  comments and annotations
*/
```

Related Information

Create a CDS Document (XS Advanced) [page 144]

4.2.9 CDS Extensions Artifacts

Extend existing artifact definitions with properties stored in an additional, external artifact.

The CDS extension mechanism allows you to add properties to existing artifact definitions without modifying the original source files. In this way, you can split the definition of an artifact across multiple files each of which can have a different life cycle and code owner. For example, a customer can add a new element to an existing entity definition by the following statement:

‹≡› Sample Code

CDS Artifact Extension Syntax

```
extend EntityE with {
  newElement: Integer;
}
```

In the example above, the code illustrated shows how to define a new **element** inside an existing entity (`EntityE`) artifact.

i Note

The `extend` statement changes an existing artifact; it does not define any additional artifact.

It is essential to ensure that additional element definitions specified in custom extensions do not break the existing definitions of the base application. This is achieved by adapting the name-search rules and by additional checks for the `extend` statements. For the definition of these rules and checks, it is necessary to define the relationship between an `extend` statement and the artifact definitions, as well as the relationship between an `extend` statement and any additional `extend` statements.

Organization of Extensions

When you extend an SAP application, you typically add new elements to entities or views; these additional elements usually work together and can, themselves, require additional artifacts, for example, “types” used as element “types”. To facilitate the process, we define an extension package (or package for short), which is a set of extend statements, normal artifact definitions (for example, “types” which are used in an `extend` declaration), and extension relationships (also known as “dependencies”). Each CDS source file belongs to exactly one package; all the definitions in this file contribute to that one (single) package. However, a “package” typically contains contributions from multiple CDS source files.

→ Tip

It is also possible to use a package to define a clear structure for an application, even if no extensions are involved.

Package Hierarchies

The extension mechanism can be used by developers as well as SAP industry solutions, partners, and customers. A productive system is likely to have more than one package; some packages might be independent from each other; some packages might depend on other packages. With such a model, we get an acyclic directed graph, with the base application and the extension packages as nodes and the dependencies as edges. This induces a partial order on the packages with the base application as lowest package (for simplicity we also call the base application a package). There is not necessarily a single top package (here: the final customer extension).

It is essential to ensure that which package is semantically self-contained and self-explanatory; avoid defining "micro" packages which can be technically applied individually but have no independent business value.

> ! Restriction
>
> Circular dependencies between extension packages are not allowed.

Package Definition

It is necessary to specify which `extend` statements and normal artifact definitions belong to which package and, in addition, on which other packages a package depends. A package is considered to be a normal CDS artifact; it has a name, and a corresponding definition, and its use can be found in the CDS Catalog. An extension package is defined by a special CDS source file with the file suffix `.package.hdbcds`.

> i Note
>
> The full stop (.) **before** the extension-package file name is mandatory.

Related Information

Create a CDS Document (XS Advanced) [page 144]
Create a CDS Extension [page 254]
The CDS Extension Descriptor [page 260]
The CDS Extension Package Descriptor [page 267]

4.3 Create a CDS Entity in XS Advanced

Define a design-time **entity** (or table) using the Core Data Services (CDS) syntax.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have access to the SAP Web IDE for SAP HANA

  > **i** Note
  >
  > The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the tools included in the SAP Web IDE for SAP HANA.

- You must have already created a development workspace and a multi-target application (MTA) project.
- You must already have created a database module for your MTA application project.
- You must already have set up an HDI container for the CDS artifacts

  > **i** Note
  >
  > A container setup file (`.hdiconfig`) is required to define which plug-ins to use to create the corresponding catalog objects from your design-time artifacts when the multi-target application (or just the database module) is deployed.

- To view run-time objects, you must have access to the SAP HANA XS advanced run-time tools that enable you to view the contents of the catalog.

  > **i** Note
  >
  > The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the SAP HANA run-time tools.

Context

In the SAP HANA database, as in other relational databases, a CDS entity is a table with a set of data elements that are organized using columns and rows. SAP HANA Extended Application Services for XS advanced (XS advanced) enables you to use the CDS syntax to create a database entity as a design-time file. Deploying the

database module that contains the CDS entity creates the corresponding table in the corresponding schema. To create a design-time CDS entity-definition file, perform the following steps:

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following URL:

   `https://<HANA_HOST>:53075/`

   > → Tip
   >
   > To display the URL for the SAP Web IDE for SAP HANA, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > `xs app webide --urls`

2. Open the application project to which you want to add your CDS entity.
3. Create the CDS entity-definition file.

   Browse to the folder in the database module in your application's project workspace, for example, `<MyApp1>/HDB/src` where you want to create the new CDS entity-definition file and perform the following steps:

   a. Right-click the folder where you want to save the CDS entity-definition file and choose *New* > *CDS Artifact* in the context-sensitive pop-up menu.
   b. Enter the name of the entity-definition file in the *File Name* box, for example, `MyEntity`.

      > → Tip
      >
      > If you use the available setup Wizards to create your design-time artifacts, the correct file extensions is added automatically. The file extension is used to determine which plug-in to use to create the corresponding run-time object during deployment. CDS artifacts have the file extension `.hdbcds`, for example, `MyEntity.hdbcds`.

   c. Choose *Finish* to save the new CDS entity-definition file in the database module of your local project workspace.
4. Define the structure of the CDS entity.

   If the new entity-definition file is not automatically displayed by the file-creation wizard, double-click the entity-definition file you created in the previous step, for example, `MyEntity.hdbcds`, and add the entity-definition code to the file:

   > i Note
   >
   > The following code example is provided for illustration purposes only.

```
entity MyEntity {
    key Author    : String(100);
    key BookTitle : String(100);
        ISBN      : Integer not null;
        Publisher : String(100);
} technical configuration {
```

```
    column store;
    unique index MYINDEX1 on (ISBN) desc;
};
```

5. Save the CDS entity-definition file.

   Saving the definition persists the file in your local workspace; it does not create any objects in the database catalog.

6. Activate the changes in the catalog.

   To activate the new entity definition and generate a corresponding table in the catalog, use the *Build* feature.

   a. Right-click the new database module in your application project.

   b. In the context-sensitive pop-up menu, choose *Build*.

   > → Tip
   >
   > You can follow progress of the build in the console at the bottom of the CDS editor.

7. Check that the new entity has been successfully created in the catalog.

   > → Tip
   >
   > A selection of run-time tools is available in the *Database Explorer* perspective of SAP web IDE for SAP HANA at the following location:
   >
   > *Tools* > *Database Explorer*

   In XS advanced, your database run-time objects are located in the HDI container created for your multi-target application's database module; you need to locate and bind to this application-specific container to view its contents. The container name contains the name of the user logged into the SAP Web IDE for SAP HANA, the name of the database module containing the CDS design-time entities, and the string *-hdi-container*, for example:

   `<XS_UserName>`*-ctetig24[...]-*`<DB_Module>`*-hdi-container*

   To bind to the HDI container, in the SAP HANA run-time *Catalog* tool, right-click *Catalog* in the catalog list, and in the *Search HDI Containers* dialog, locate the container to which you want to bind, and choose *Bind*.

8. Delete a table (entity) created by SAP HDI (**optional**).

   If you need to delete a table created by HDI in the context of the SAP Web IDE, perform the following steps:

   a. Remove the design-time table definition from the database module of your application project in SAP Web IDE.

   b. Right-click the database module in your application project.

   c. In the context-sensitive pop-up menu, choose *Build*.

   > → Tip
   >
   > In the context of HDI, you can use the HDI container API's `DELETE` command to remove files from the HDI container's virtual file system and then use the `MAKE` command with the path to the file slated for removal specified in the `undeploy_paths` parameter. For more information about the HDI APIs, see **The HDI Container API** in *Related Information* below.

Related Information

Set up an SAP HDI Container for XS Advanced Applications
Create a CDS Document (XS Advanced) [page 144]
The HDI Container API

4.3.1 CDS Entities in XS Advanced

In the SAP HANA database, as in other relational databases, a CDS entity is a table with a set of data elements that are organized using columns and rows.

A CDS entity has a specified number of columns, defined at the time of entity creation, but can have any number of rows. Database entities also typically have meta-data associated with them; the meta-data might include constraints on the entity or on the values within particular columns. SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database entity as a design-time file. All design-time files, including your CDS entity definition, can be transported to other SAP HANA systems and, when deployed, used to generate the same catalog objects. You can define the entity using CDS-compliant DDL.

> **i** Note
>
> In XS classic, the delivery unit is the medium SAP HANA provides to enable you to assemble all your application-related repository artifacts together into an archive that can be easily exported to other systems. In XS advanced, you add your artifacts to application modules (for example, a database module); the modules are used to define and deploy a multi-target application (MTA).

The following code illustrates an example of a single design-time entity definition using CDS-compliant DDL. The name of the top-level artifact, in the following example, the entity `MyTable` must match the name of the CDS artifact. In the example below, the CDS document must be named `MyTable.hdbcds`. In XS advanced, an optional name space can be declared; it indicates the repository package in which the object the document defines is located.

> **→** Tip
>
> From SAP HANA 2.0 SPS 01, it is possible to define **multiple** top-level artifacts (for example, contexts, entities, etc.) in a single CDS document. For this reason, you can choose any name for the CDS source file; there is no longer any requirement for the name of the CDS source file to be the same as the name of the top-level artifact.

> **⁽≡⁾** Code Syntax
>
> CDS Entity definition in XS advanced (MyTable.hdbcds)

```
namespace com.acme.myapp1;
entity MyTable {
    key Author    : String(100);
    key BookTitle : String(100);
        ISBN      : Integer not null;
        Publisher : String(100);
} technical configuration {
    column store;
    unique index MYINDEX1 on (ISBN) desc;
```

```
};
```

If you want to create a CDS-compliant database entity definition as a design-time file in SAP HANA XS advanced model, you must create the entity as a flat file and save the file containing the DDL entity dimensions with the suffix `.hdbcds`, for example, `MyTable.hdbcds`. The

> **i Note**
>
> On deployment of the CDS design-time artifact, the file suffix, for example, `.hdbcds`, is used to determine which runtime plug-in to call during the activation process. The plug-in reads the repository file selected for activation, in this case a CDS-compliant entity, parses the object descriptions in the file, and creates the appropriate runtime objects.

When a CDS document is deployed (or activated), the deployment process generates a corresponding catalog object for each of the artifacts defined in the document; the location in the catalog is determined by the type of object generated. For example, the corresponding database table for a CDS entity definition is generated in the following catalog location:

Entity Element Definition

You can expand the definition of an entity element beyond the element's name and type by using element **modifiers**. For example, you can specify if an entity element is the primary key or **part** of the primary key. The following entity element modifiers are available:

- `key`
  Defines if the specified element is the **primary** key or **part** of the primary key for the specified entity.

  > **i Note**
  >
  > Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

- `null`
  Defines if an entity element can (`null`) or cannot (`not null`) have the value NULL. If neither `null` nor `not null` is specified for the element, the default value `null` applies (except for the `key` element).
- `default <literal_value>`
  Defines the default value for an entity element in the event that no value is provided during an INSERT operation. The syntax for the literals is defined in the primitive data-type specification.
- `generated always as <expression>`
  Defines a value that is computed as specified in `<expression>`, for example, `a=b`.
  You can also use the SAP HANA SQL clause `generated always as identity` in CDS entity definitions to defined a field in the database table that is present in the persistence and has a value that is computed as specified in the expression.

```
entity MyEntity {
    key   MyKey  : Integer;
    key   MyKey2 : Integer null;      // illegal combination
    key   MyKey3 : Integer default 2;
```

```
          elem2  : String(20) default 'John Doe';
          elem3  : String(20) default 'John Doe' null;
          elem4  : String default 'Jane Doe' not null;
          elem5  : Integer
          elem6  : Integer
          elem7  : Integer generated always as elem5+elem6;
};
```

Spatial Data

CDS entities support the use of spatial data types such as `hana.ST_POINT` or `hana.ST_GEOMETRY` to store geo-spatial coordinates. Spatial data is data that describes the position, shape, and orientation of objects in a defined space; the data is represented as two-dimensional geometries in the form of points, line strings, and polygons.

Multi-Store Tables, Partitions, and Extended Storage

Provided that SAP HANA Dynamic Tiering is correct installed running and extended storage is configured and available, CDS allows you to partition column entities into extended storage.

The sequence of multiple storage declarations, for example, "`using default storage`" and "`using extended storage`", is maintained against the database. Where multiple "`partition others`" clauses are defined, CDS replaces them with one "`partition others`" that is added as the last partition specification.

Sample Code

Storage Partition in CDS

```
entity Person
  {
    key id : Integer;
    ...
  }
  technical configuration {
    partitition by range(id) (
        using default storage (partitions ...)
        using extended storage (partitions...)
    ...);
  };
```

Where multi-level partitioning is defined, the first partition must not contain any "`using [ default | extended ] storage`" specification, and you can use the multi-level partition schemes “Hash-Range” and “Range-Range”.

! Restriction

Some CDS data types are not supported in selected elements of a multi-level storage scenario. For example, `sap.cds::LargeString` cannot be used in a multi-storage range expression. For more information, see *CDS Entity Syntax Options in XS Advanced* in *Related Information* below.

Related Information

Create a CDS Entity in XS Advanced [page 179]
Entity Element Modifiers [page 43]
CDS Entity Syntax Options in XS Advanced [page 190]

4.3.2 Entity Element Modifiers in XS Advanced

Element **modifiers** enable you to expand the definition of an entity element beyond the element's name and type. For example, you can specify if an entity element is the primary key or **part** of the primary key.

Example

```
entity MyEntity {
    key   MyKey  : Integer;
          elem2  : String(20) default 'John Doe';
          elem3  : String(20) default 'John Doe' null;
          elem4  : String default 'Jane Doe' not null;
};
entity MyEntity1 {
    key id : Integer;
    a  : integer;
    b  : integer;
    c  : integer generated always as a+b;
};
entity MyEntity2 {
    autoId : Integer generated [always|by default] as identity ( start with 10
increment by 2 );
    name : String(100);
};
```

key

```
key   MyKey  : Integer;
key   MyKey2 : Integer null;                  // illegal combination
key   MyKey3 : Integer default 2;
```

You can expand the definition of an entity element beyond the element's name and type by using element **modifiers**. For example, you can specify if an entity element is the primary key or **part** of the primary key. The following entity element modifiers are available:

- `key`
  Defines if the element is the **primary** key or **part** of the primary key for the specified entity. You **cannot** use the `key` modifier in the following cases:
  - In combination with a `null` modifier. The `key` element is `non null` by default because NULL cannot be used in the `key` element.

> **Note**
>
> Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

null

```
elem3  : String(20) default 'John Doe' null;
elem4  : String default 'Jane Doe' not null;
```

`null` defines if the entity element can (`null`) or cannot (`not null`) have the value NULL. If neither `null` nor `not null` is specified for the element, the default value `null` applies (except for the `key` element), which means the element **can** have the value NULL. If you use the `null` modifier, note the following points:

> **Caution**
>
> The keywords `nullable` and `not nullable` are no longer valid; they have been replaced for SPS07 with the keywords `null` and `not null`, respectively. The keywords `null` and `not null` must appear at the end of the entity element definition, for example, `field2 : Integer null;`.

- The `not null` modifier can only be added if the following is true:
  - A `default` it also defined
  - no `null` data is already in the table
- Unless the table is empty, bear in mind that when adding a new `not null` element to an existing entity, you must declare a default value because there might already be existing rows that do not accept NULL as a value for the new element.
- `null` elements with default values are permitted
- You cannot combine the element `key` with the element modifier `null`.
- The elements used for a unique index must have the `not null` property.

```
entity WithNullAndNotNull
{
  key id :  Integer;
  field1 :  Integer;
  field2 :  Integer null;  // same as field1, null is default
  field3 :  Integer not null;
};
```

default

```
default <literal_value>
```

For each scalar element of an entity, a default value can be specified. The `default` element identifier defines the default value for the element in the event that no value is provided during an INSERT operation.

> **i** Note
>
> The syntax for the literals is defined in the primitive data-type specification.

```
entity WithDefaults
{
  key id :  Integer;
  field1 :  Integer       default -42;
  field2 :  Integer64     default 9223372036854775807;
  field3 :  Decimal(5, 3) default 12.345;
  field4 :  BinaryFloat   default 123.456e-1;
  field5 :  LocalDate     default date'2013-04-29';
  field6 :  LocalTime     default time'17:04:03';
  field7 :  UTCDateTime   default timestamp'2013-05-01 01:02:03';
  field8 :  UTCTimestamp  default timestamp'2013-05-01 01:02:03';
  field9 :  Binary(32)    default x'0102030405060708090a0b0c0d0e0[...]';
  field10 : String(10)    default 'foo';
};
```

generated always as <expression>

```
entity MyEntity1 {
    key id : Integer;
    a  : integer;
    b  : integer;
    c  : integer generated always as a+b;
};
```

The SAP HANA SQL clause `generated always as <expression>` is available for use in CDS entity definitions; it specifies the expression to use to generate the column value at run time. An element that is defined with `generated always as <expression>` corresponds to a field in the database table that is present in the persistence and has a value that is computed as specified in the expression, for example, "`a+b`".

> **!** Restriction
>
> For use in XS advanced only; it is not possible to use generated calculated elements in XS classic. Please also note that the `generated always as <expression>` clause is only for use with column-based tables.

"Generated" fields and "calculated" field differ in the following way. **Generated** fields are physically present in the database table; values are computed on `INSERT` and need not be computed on `SELECT`. **Calculated** fields are not actually stored in the database table; they are computed when the element is "selected". Since the value of the **generated** field is computed on `INSERT`, the expression used to generate the value must not contain any non-deterministic functions, for example: `current_timestamp`, `current_user`, `current_schema`, and so on.

generated [always | by default] as identity

```
entity MyEntity2 {
```

```
    autoId : Integer generated always as identity ( start with 10 increment by
2 );
    name : String(100);
};
```

The SAP HANA SQL clause `generated as identity` is available for use in CDS entity definitions; it enables you to specify an identity column. An element that is defined with `generated as identity` corresponds to a field in the database table that is present in the persistence and has a value that is computed as specified in the sequence options defined in the `identity` expression, for example, `( start with 10 increment by 2 )`.

In the example illustrated here, the name of the generated column is `autoID`, the first value in the column is "10"; the `identity` expression `( start with 10 increment by 2 )` ensures that subsequent values in the column are incremented by 2, for example: 12, 14, and so on.

> **! Restriction**
>
> For use in XS advanced only; it is not possible to define an element with `IDENTITY` in XS classic. Please also note that the `generated always as identity` clause is only for use with column-based tables.

You can use either `always` or `by default` in the clause `generated as identity`, as illustrated in the examples in this section. If `always` is specified, then values are **always** generated; if `by default` is specified, then values are generated **by default**.

```
entity MyEntity2 {
    autoId : Integer generated by default as identity ( start with 10 increment
by 2 );
    name : String(100);
};
```

> **! Restriction**
>
> CDS does not support the use of reset queries, for example, `RESET BY <subquery>`.

Column Migration Behavior

The following table shows the migration strategy that is used for modifications to any given column; the information shows which actions are performed and what strategy is used to preserve content. During the migration, a comparison is performed on the column **type**, the generation **kind**, and the expression, if available. From an end-user perspective, the result of a column modification is either a preserved or new value. The aim of any modification to an entity (table) is to cause as little loss as possible.

- Change to the column **type**
  In case of a column type change, the content is converted into the new type. HANA conversion rules apply.
- Change to the expression clause
  The expression is re-evaluated in the following way:
  - "early"
    As part of the column change
  - "late"
    As part of a query

- Change to a calculated column
  A calculated column is transformed into a plain column; the new column is initialized with NULL.

Technically, columns are either dropped and added or a completely new “shadow” table is created into which the existing content is copied. The shadow table will then replace the original table.

Before column/ After row	**Plain**	**As** `<expr>`	**generated always as** `<expr>`	**generated always as identity** `<expr>`	**generated by default as identity** `<expr>`
Plain	Migrate Keep content	Drop/add Evaluate on select	Drop/add Evaluate on add	Migrate Keep content	Migrate Keep content
generated by default as identity `<expr>`	Migrate Keep content	Drop/add Evaluate on select	Drop/add Evaluate on add	Migrate Keep content	Migrate Keep content
generated always as identity `<expr>`	Migrate Keep content	Drop/add Evaluate on select	Drop/add Evaluate on add	Migrate Keep content	Migrate Keep content
generated always as `<expr>`	Drop/add NULL	Drop/add Evaluate on select	Drop/add Evaluate on add	Drop/add Keep content	Migrate Keep content
as `<expr>`	Drop/add NULL	Drop/add Evaluate on select	Drop/add Evaluate on add	Drop/add Keep content	Migrate Keep content

Related Information

Create a CDS Entity in XS Advanced [page 179]
CDS Entity Syntax Options in XS Advanced [page 190]
SAP HANA SQL and System Views Reference (CREATE TABLE)

4.3.3 CDS Entity Syntax Options in XS Advanced

The CDS syntax specifies a number of options you can use to define an **entity** (table) in a design-time artifact.

Example

> **i** Note
>
> This example is not a working example; it is intended for illustration purposes only.

```
namespace Pack1."pack-age2";
context MyContext {
  entity MyEntity1  {
    key id : Integer;
    name   : String(80);
  };
  entity MyEntity2 {
    key id : Integer;
    x      : Integer;
    y      : Integer;
    a      : Integer;
    field7 : Decimal(20,10) = power(ln(x)*sin(y), a);
  } technical configuration {
       column store;
       unique index Index1 on (x, y) desc;
       index Index2 on (x, a) desc;
       index Index3 on (y asc, a desc);
       partition by <partition_clause>;
       group <grouping_clause>;
       unload priority 0;
       no auto merge;
  };
  entity MyPartitionEntity  {
    key id : Integer;
    name   : String(80);
  } technical configuration {
       partition by range(id) (
          using default storage (partitions...)
          using extended storage (partitions...) );
  };
  entity MyEntity {
    key id : Integer;
    a : Integer;
    b : Integer;
    c : Integer;
    s {
      m : Integer;
      n : Integer;
    };
    } technical configuration {
      row store;
      index MyIndex1 on (a, b) asc;
      unique index MyIndex2 on (c, s) desc;
      fulltext index MYFTI1 on (t)
      FUZZY SEARCH INDEX off;
      };
    temporary entity MyTempEntity {
        a : Integer;
        b : String(20);
    } technical configuration {
```

```
        column store;
      };
  };
};
context MySpatialContext {
    entity Address {
      key id        : Integer;
      street_number : Integer;
      street_name   : String(100);
      zip           : String(10);
      city          : String(100);
      loc           : hana.ST_POINT(4326);
    };
};
context MySeriesContext {
    entity MySeriesEntity {
      key setId : Integer;
      t : UTCTimestamp;
      value : Decimal(10,4);
      series (
        series key (setId)
        period for series (t)
        equidistant increment by interval 0.1 second
        equidistant piecewise //increment or piecewise; not both
      )
    };
};
```

> **!** Restriction
>
> For series data, you can use either `equidistant` or `equidistant piecewise`, but not both at the same time. The example above is for illustration purposes only.

Overview

Entity definitions resemble the definition of structured types, but with the following additional features:

- Key definition [page 192]
- Calculated Fields [page 192]
- Technical Configuration [page 193]
  Includes: index, unique index, full-text index, table type (row/column), partitioning, grouping, unload priority, and auto-merge options.
- Spatial data [page 202] *
- Series Data [page 202] *

On deployment in the SAP HANA XS advanced, each entity definition in CDS is used to generate a database table.

> → Tip
>
> From SAP HANA 2.0 SPS 01, it is possible to define **multiple** top-level artifacts (for example, contexts, entities, etc.) in a single CDS document. For this reason, you can choose any name for the CDS source file; there is no longer any requirement for the name of the CDS source file to be the same as the name of the top-level artifact.

The name of the table generated for each entity definition is built according to the same rules as for table types, for example, `Pack1.Pack2::MyModel.MyContext.MyTable`. In addition, the CDS name is restricted

by the limits imposed on the length of the database identifier for the name of the corresponding SAP HANA database artifact (for example, table, view, or type); this is currently limited to 126 characters (including delimiters).

> **i** Note
>
> As of HANA 1.0 SPS 12, the name-space definition is optional.

Key Definition

```
type MyStruc2
{
  field1 : Integer;
  field2 : String(20);
};
entity MyEntity2
{
  key id  : Integer;
  name    : String(80);
  key str : MyStruc2;
};
```

Usually an entity must have a key; you use the keyword `key` to mark the respective elements. The key elements become the primary key of the generated SAP HANA table and are automatically flagged as `not null`. Key elements are also used for managed associations. Structured elements can be part of the key, too. In this case, all table fields resulting from the flattening of this structured element are part of the primary key.

Calculated Fields

The definition of an entity can contain calculated fields, as illustrated in type "`z`" the following example:

> Sample Code

```
entity MyCalcField {
  a : Integer;
  b : Integer;
  c : Integer = a + b;
  s : String(10);
  t : String(10) = upper(s);
  x : Decimal(20,10);
  y : Decimal(20,10);
  z : Decimal(20,10) = power(ln(x)*sin(y), a);
};
```

The calculation expression can contain arbitrary expressions and SQL functions. The following restrictions apply to the expression you include in a calculated field:

- The definition of a calculated field must not contain other calculated fields, associations, aggregations, or subqueries.
- A calculated field cannot be key.

- No index can be defined on a calculated field.
- A calculated field cannot be used as foreign key for a managed association.

In a query, calculated fields can be used like ordinary elements.

> **i Note**
>
> In SAP HANA tables, you can define columns with the additional configuration "`GENERATED ALWAYS AS`". These columns are physically present in the table, and all the values are stored. Although these columns behave for the most part like ordinary columns, their value is computed upon insertion rather than specified in the `INSERT` statement. This is in contrast to calculated field, for which no values are actually stored; the values are computed upon `SELECT`.

technical configuration

The definition of an entity can contain a section called `technical configuration`, which you use to define the elements listed in the following table:

- Storage type (row/column) [page 193]
- Indexes [page 194]
- Full text indexes [page 194]
- Partitioning [page 195]
- Grouping [page 200]
- Unload priority [page 201]
- Auto merge option [page 201]

> **i Note**
>
> The syntax in the technical configuration section is as close as possible to the corresponding clauses in the SAP HANA SQL `Create Table` statement. Each clause in the technical configuration must end with a semicolon (";").

Storage Type and Table Types

In the technical configuration for an entity, you can use the `store` keyword to specify the storage type ("`row`" or "`column`") for the generated table, as illustrated in the following example. If no store type is specified, a "column" store table is generated by default.

> **‹≡› Sample Code**
>
> Storage Type

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   row store;
```

```
};
```

To specify a table of type “global temporary”, use the `temporary entity` keywords in the CDS entity definition, as illustrated in the following example. To specify a “global temporary table” with the type **column**, use the `temporary entity` keywords and, in addition, define the storage type as `column`, as illustrated in the following examples.

Sample Code

Temporary Table Types

```
context MyContext1 {
    temporary entity MyEntity3 {
        ID : Integer;
        name : String(30);
    };
    temporary entity MyTempEntity {
        a : Integer;
        b : String(20);
    } technical configuration {
        column store;
    };
};
```

Indexes

In the technical configuration for an entity, you can use the `index` and `unique index` keywords to specify the index type for the generated table. For example, `unique` generates an index where no two sets of data in the indexed entity can have identical key values. You can use the keywords “`asc`” (ascending) or “`desc`” (descending) to specify the order of the index.

Sample Code

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   index MyIndex1 on (a, b) asc;
   unique index MyIndex2 on (c, s) desc;
   index MyIndex3 on (b asc, t desc);
};
```

Note

You specify the index order (ascending or descending) for individual rows or columns, for example, `(b asc, t desc, s.u desc)`.

Full text indexes

In the technical configuration for an entity, you can use the `fulltext index` keyword to specify the full-text index type for the generated table, as illustrated in the following example.

Sample Code

```
entity MyEntity {
    key id : Integer;
    a : Integer;
    b : Integer;
    t : String(100);
    s {
        u : String(100);
    };
} technical configuration {
    row store;
    index MyIndex1 on (a, b) asc;
    unique index MyIndex2 on (a, b) asc;
    fulltext index MYFTI1 on (t)
        LANGUAGE COLUMN t
        LANGUAGE DETECTION ('de', 'en')
        MIME TYPE COLUMN s.u
        FUZZY SEARCH INDEX off
        PHRASE INDEX RATIO 0.721
        SEARCH ONLY off
        FAST PREPROCESS off
        TEXT ANALYSIS off;
    fuzzy search index on (s.u);
};
```

The `fulltext index` is identical to the standard SAP HANA SQL syntax for `CREATE FULLTEXT INDEX`. A `FUZZY SEARCH INDEX` in the technical configuration section of an entity definition corresponds to the `@SearchIndex` annotation in XS classic and the statement "`FUZZY SEARCH INDEX ON`" for a table column in SAP HANA SQL. It is not possible to specify both a full-text index and a fuzzy search index for the same element.

Partitioning

In the `technical configuration` of an entity definition, you can specify the partitioning information for the generated table, as illustrated in the following example:

Sample Code

Partition Specification

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   <partition_clause>;
};
```

The code required in the `<partition_clause>` is identical to the corresponding clause in the standard `HANA SQL CREATE TABLE` statement, as illustrated in the following example:

Note

SAP HANA CDS only supports those SAP HANA partitioning features introduced up to and including SAP HANA 2.0 SPS 02.

Sample Code

Partition Specification

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
} technical configuration {
   partition by hash (id) partitions 2,
                range (a) (partition 1 <= values < 10,
                           partition values = 10,
                           partition others);
};
```

The `partition by` clause defines the rules to use to partition the corresponding table, for example, using hash, range, or round-robin partition rules.

Note

You can use the `partition by` clause to ensure any partitions added after activation of a CDS entity (for example, with an SQL statement `ALTER TABLE <MyEntity> PARTITION BY RANGE("id")`) are retained on reactivation of the original CDS entity.

Normally, any external changes to a table that was originally created by activating a CDS entity definition are lost on reactivation; the partitions defined after the entity activation are lost. With the `keeping existing layout` option, you can use the `partition by` clause to preserve an existing partitioning, if possible. It is not possible to preserve the added partitions on reactivation of the CDS entity, if the fields used in the external partition specification have changed in the entity definition in a way that no longer fits the partition.

Sample Code

Retain existing partitions on reactivation

```
entity MyEntity {
  ...
} technical configuration {
  partition by keeping existing layout;
};
```

CDS allows you to create a table fully in extended storage, as illustrated in the following example:

Sample Code

Create a table in extended storage

```
entity MyStorageEntity {
  ...
} technical configuration {
    using extended storage;
};
```

When using CDS to create a table in extended storage, all existing restrictions relating to SAP HANA extended storage tables apply and, in addition, it is not permitted to use any of the following features:

- Series Tables

- Temporary Tables (global and local)
- Elements that are "`generated by (default)`", "`generated as (identity)`"
- “Full Text” and “Fuzzy Search” indexes
- Partitioning
- The clause "`PARTITION BY KEEPING EXISTING LAYOUT`"
- Unload Priority
- Auto Merge

CDS supports the use of multi-store column-based tables in extended storage provided that SAP HANA Dynamic Tiering is installed and, in addition, the target (extended) storage is configured and available. This support for multi-store tables means that you can partition entities of storage type "`column`" into extended storage.

Sample Code

Column Tables using partitions in extended storage

```
entity MyStorageEntity {
  ...
} technical configuration {
    partitition by range(id)
       (using default storage (partitions ...)
        using extended storage (partitions...) ...);
};
```

When declaring multiple partitions, bear in mind that the sequence of "`using [ default | extended ] storage`" declarations is maintained against the database. Note, too that multiple "`partition others`" clauses are removed and replaced with a single "`partition others`" clause and added as the last partition specification.

For configurations with multi-level partitions, all existing restrictions relating to SAP HANA multi-storage tables apply and, in addition, it is not permitted to use any of the following features:

- Series Tables
- Temporary Tables (both global and local)
- Elements that are "`generated by (default)`", or "`generated as (identity)`"
- “Full Text” and “Fuzzy Search” indexes
- The clause "`USING EXTENDED STORAGE`" to create the table fully in extended storage

Range partitions can now contain the time-selection interval. However, if a time-selection interval is specified, the partition clause in the `technical configuration` section of the CDS entity definition must include and enable `WITH PARTITIONING ON ANY COLUMNS`, as illustrated in the following example:

Sample Code

Partitions with time-selection interval

```
entity MyStorageEntity {
  ...
} technical configuration {
    partition by range(column)
      (partition '00000000' is current, partition ...)
       with partitioning on any columns on;
};
```

In contrast to the SAP HANA SQL syntax, CDS does not required a time-selection interval to appear only once and at the very first position in the range specification. However, note that multiple time-selection ranges are combined into a single time range and inserted as the first range specification in the resulting SQL statement `CREATE COLUMN TABLE ... PARTITION BY ...` that is used to create the table in the database. In CDS, the following restrictions apply for time-selection partitions:

- The time-selection interval must be declared in default storage.
- The underlying type of the range expression must be a string with a minimum length of eight characters
- The value of the '`is current`' range must be an eight character sequence, for example, '00000000'
- All other interval values must be enclosed in single quotes ('`<value>`') and be eight characters long.
- Time selection cannot be combined with a YEAR/MONTH range expression
- In multi-level partitions, time selection must not appear in the first-level partition definition
- The '`WITH PARTITIONING ON ANY COLUMNS ON`' clause cannot be used in combination with '`KEEPING EXISTING LAYOUT`'.

The following table shows which CDS data types are supported in the various multi- or extended-storage contexts:

CDS Data Type Compatibility Matrix

CDS Data Type	Multi-Storage Range Expression	Multi-Storage Partition Payload	Extended Storage Table
sap.cds::String	✓	✓	✓
sap.cds::LargeString	-	✓	✓
sap.cds::Binary	✓	✓	✓
sap.cds::LargeBinary	-	✓	✓
sap.cds::Integer	✓	✓	✓
sap.cds::Integer32	✓	✓	✓
sap.cds::Integer64	✓	✓	✓
sap.cds::Decimal	✓	✓	✓
sap.cds::DecimalFloat	-	-	✓
sap.cds::BinaryFloat	-	✓	✓
sap.cds::LocalDate	✓	✓	✓
sap.cds::LocalTime	-	✓	✓
sap.cds::UTCDateTime	✓	✓	✓
sap.cds::UTCTimestamp	-	-	✓
sap.cds::Boolean	-	-	-

CDS Data Type	Multi-Storage Range Expression	Multi-Storage Partition Payload	Extended Storage Table
sap.cds::hana.ALPHANUM	-	-	-
sap.cds::hana.SMALLINT	✓	✓	✓
sap.cds::hana.TINYINT	✓	✓	✓
sap.cds::hana.SMALLDECIMAL	-	-	✓
sap.cds::hana.REAL	-	✓	✓
sap.cds::hana.CHAR	✓	✓	✓
sap.cds::hana.NCHAR	✓	✓	✓
sap.cds::hana.VARCHAR	✓	✓	✓
sap.cds::hana.CLOB	-	✓	✓
sap.cds::hana.SHORTTEXT	-	-	-
sap.cds::hana.TEXT	-	-	-
sap.cds::hana.BINTEXT	-	-	-
sap.cds::hana.BINARY	-	✓	✓
sap.cds::hana.ST_POINT	-	-	-
sap.cds::hana.ST_GEOMETRY	-	-	-

Migration Disabled

If the technical configuration of an entity contains the clause "`migration disabled`", the activation of the CDS source is only allowed if changes in the entity definition do not lead to a migration of the table. If a migration is required, the activation fails and the changes need to be made manually.

Sample Code

```
entity MyEntity {
  key id : Integer;
  name : String(100);
  value : Decimal(10,2);
} technical configuration {
  migration disabled;
};
```

If the entity defined in the previous example is changed in a way that allows CDS to adapt the corresponding table via `ALTER` statements, the activation of the modified table will succeed. This is typically the case for adding or removing elements (as long they are not "`key`" elements) or adding and removing indexes, as illustrated in the following example.

Sample Code

```
entity MyEntity {
  key id : Integer;
  name : String(120);
  name2 : String(80);
  someTime: LocalTime;
} technical configuration {
  migration disabled;
};
```

Changing an element **type** in the way illustrated in the following example is not allowed; activating the following CDS document fails because the change of type in elements "`name`" and"`name2`" mean that a migration is required, which is explicitly forbidden by the (`migration disabled` clause.

Sample Code

```
entity MyEntity {
  key id : Integer;
  name : String(120);
  name2 : String(80);
  someTime: LocalTime;
} technical configuration {
  migration disabled;
};
```

Grouping

In the `technical configuration` of an entity definition, you can specify the grouping information for the generated table, as illustrated in the following example:

Sample Code

Grouping Specification

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   <grouping_option>;
};
```

The code required in the `<grouping_option>` is identical to the corresponding clause in the standard `HANA SQL CREATE TABLE` statement, as illustrated in the following example:

Sample Code

Grouping Specification

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
```

```
} technical configuration {
   group type Foo group subtype Bar group name Wheeeeezz;
};
```

You must set the `group type`, the group `group subtype`, and the `group name`.

Unload Priority

In the `technical configuration` of an entity definition, you can specify the unload priority for the generated table, as illustrated in the following example:

Sample Code

Unload Priority Specification

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   unload priority <integer_literal>;
};
```

`unload priority` specifies the priority with which a table is unloaded from memory. The priority can be set between 0 (zero) and 9 (nine), where 0 means “cannot be unloaded” and 9 means “earliest unload”.

Auto-Merge Option

In the `technical configuration` of an entity definition, you can specify any automatic-merge options for the generated table, as illustrated in the following example:

Sample Code

Auto-Merge Option

```
entity MyEntity {
   key id : Integer;
   a : Integer;
   b : Integer;
   t : String(100);
   s {
       u : String(100);
   };
} technical configuration {
   [no] auto merge;
};
```

Note

`auto merge;` triggers an automatic delta merge; `no auto merge;` disables the automatic delta merge operation.

Spatial Types *

The following example shows how to use the spatial type `ST_POINT` in a CDS entity definition. In the example entity `Person`, each person has a home address and a business address, each of which is accessible via the corresponding associations. In the `Address` entity, the geo-spatial coordinates for each person are stored in element `loc` using the spatial type `ST_POINT` (*).

Sample Code

```
context SpatialData {
    entity Person {
        key id : Integer;
        name : String(100);
        homeAddress : Association[1] to Address;
        officeAddress : Association[1] to Address;
    };
    entity Address {
        key id : Integer;
        street_number : Integer;
        street_name : String(100);
        zip : String(10);
        city : String(100);
        loc : hana.ST_POINT(4326);
    };
    view CommuteDistance as select from Person {
        name,
        homeAddress.loc.ST_Distance(officeAddress.loc) as distance
    };
};
```

Series Data *

CDS enables you to create a table to store series data by defining an entity that includes a `series ()` clause as an table option and then defining the appropriate paremeters and options.

Note

The `period for series` must be unique and should not be affected by any shift in timestamps.

Sample Code

```
context SeriesData {
    entity MySeriesEntity1 {
        key setId : Integer;
        t : UTCTimestamp;
        value : Decimal(10,4);
        series (
            series key (setId)
            period for series (t)
            equidistant increment by interval 0.1 second
        );
    };
    entity MySeriesEntity2 {
        key setId : Integer;
        t : UTCTimestamp;
        value : Decimal(10,4);
```

```
        series (
            series key (setId)
            period for series (t)
            equidistant piecewise
        );
    };
};
```

CDS also supports the creation of a series table called `equidistant piecewise` using Formula-Encoded Timestamps (FET). This enables support for data that is not loaded in an order that ensures good compression. There is no a-priori restriction on the timestamps that are stored, but the data is expected to be well approximated as piecewise linear with some jitter. The timestamps do not have a single slope/offset throughout the table; rather, they can change within and among series in the table.

> **! Restriction**
>
> The `equidistant piecewise` specification can only be used in CDS; it cannot be used to create a table with the SQL command `CREATE TABLE`.

When a series table is defined as `equidistant piecewise`, the following restrictions apply:

1. The `period` includes one column (instant); there is no support for interval periods.
2. There is no support for `missing elements`. These could logically be defined if the period includes an interval start and end. Missing elements then occur when we have adjacent rows where the end of the `interval` **does not** equal the start of the `interval`.
3. The type of the period column must map to the one of the following types: `DATE`, `SECONDDATE`, or `TIMESTAMP`.

> **⚠ Caution**
>
> (*) For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

4.4 Create a CDS User-Defined Structure in XS Advanced

Define a design-time custom **structured type** using the Core Data Services (CDS) syntax.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have access to the SAP Web IDE for SAP HANA

> **i Note**
>
> The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the tools included in the SAP Web IDE for SAP HANA.

- You must have already a development workspace and a multi-target application (MTA) project.
- You must already have created a database module for your MTA application project.
- You must already have set up an HDI container for the CDS artifacts

  > **i** Note
  >
  > A container setup file (`.hdiconfig`) is required to define which plug-ins to use to create the corresponding catalog objects from the design-time artifacts when the multi-target application (or just the database module) is deployed.

- You must have access to the SAP HANA XS advanced run-time tools that enable you to view the contents of the catalog.

  > **i** Note
  >
  > The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the SAP HANA run-time tools.

Context

A structured type is a data type comprising a list of attributes, each of which has its own data type. SAP HANA Extended Application Services for XS advanced model (XS advanced) enables you to use the CDS syntax to create a user-defined structured type as a design-time file that is transportable. Deploying the CDS document containing the type definition creates the corresponding types in the database catalog. To create a CDS document that defines one or more structured types, perform the following steps:

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following URL:

   `https://<HANA_HOST>:53075/`

   > **→** Tip
   >
   > To display the URL for the SAP Web IDE for SAP HANA, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > `xs app webide --urls`

2. Open the application project to which you want to add a CDS-compliant, user-defined structured type.
3. Create the CDS entity-definition file.

   Browse to the folder in the database module in your project workspace, for example, `<MyApp1>/HDB/src` where you want to create the new CDS type-definition file and perform the following steps:

   a. Right-click the folder where you want to save the CDS type-definition file and choose *New* > *CDS Artifact* in the context-sensitive pop-up menu.

b. Enter the name of the entity-definition file in the *File Name* box, for example, `MyStructuredType`.

> → Tip
>
> If you use the available setup Wizards to create your design-time artifacts, the correct file extensions is added automatically. The file extension is used to determine which plug-in to use to create the corresponding run-time object during deployment. CDS artifacts have the file extension `.hdbcds`, for example, `MyStructuredType.hdbcds`.

c. Choose *Finish* to save the new CDS type-definition file in the database module of your local project workspace.

4. Define the details of the new CDS structured type.

   If the new type-definition file is not automatically displayed by the file-creation wizard, double-click the type-definition file you created in the previous step, for example, `MyStructuredType.hdbcds`, and add the type-definition code to the file:

   > i Note
   >
   > The following code example is provided for illustration purposes only.

   ```
   table type MyStructuredType
   {
       aNumber : Integer;
       someText : String(80);
       otherText : String(80);
   };
   ```

5. Save the CDS type-definition file.

   Saving the definition persists the file in your local workspace; it does not create any objects in the database catalog.

6. Activate the changes in the catalog.

   To activate the new entity definition and generate a corresponding table in the catalog, use the *Build* feature.

   a. Right-click the new database module in your application project.

   b. In the context-sensitive pop-up menu, choose *Build*.

   > → Tip
   >
   > You can follow progress of the build in the console at the bottom of the CDS editor.

7. Check that the new custom structured type has been successfully created in the catalog.

   > → Tip
   >
   > A selection of run-time tools is available in the *Database Explorer* perspective of SAP Web IDE for SAP HANA at the following location:
   >
   > *Tools* > *Database Explorer*

   In XS advanced, your database run-time objects are located in the HDI container created for your multi-target application's database module; you need to locate and bind to this application-specific container if you want to view its contents. The container name contains the name of the user logged into the SAP Web

IDE for SAP HANA, the name of the database module containing the CDS design-time entities, and the string *-hdi-container*, for example:

`<XS_UserName>-ctetig24[...]-<DB_Module>-hdi-container`

To bind to the HDI container, in the SAP HANA run-time *Catalog* tool, right-click *Catalog* in the catalog list, and in the *Search HDI Containers* dialog, locate the container to which you want to bind, and choose *Bind*.

Related Information

Set up an SAP HDI Container for XS Advanced Applications
Create the Data Persistence Artifacts with CDS in XS Advanced [page 135]

4.4.1 CDS User-Defined Data Types in XS Advanced

User-defined data types reference existing structured types (for example, user-defined) or the individual types (for example, field, type, or context) used in another data-type definition.

You can use the *type* keyword to define a new data type in CDS-compliant DDL syntax. You can define the data type in the following ways:

- Using allowed structured types (for example, user-defined)
- Referencing another data type

In the following example, the element definition `field2 : MyType1;` specifies a new element `field2` that is based on the specification in the user-defined data type `MyType1`.

> **i Note**
>
> If you are using a CDS document to define a single CDS-compliant user-defined data type, the name of the CDS document must match the name of the top-level data type defined in the CDS document. In the following example, you must save the data-type definition “MyType1” in the CDS document `MyType1.hdbcds`.

```
namespace com.acme.myapp1;
type MyType1 {
      field1 : Integer;
      field2 : String(40);
      field3 : Decimal(22,11);
      field4 : Binary(11);
    };
```

In the following example, you must save the data-type definition “MyType2” in the CDS document `MyType2.hdbcds`; the document contains a using directive pointing to the data-type “MyType1” defined in CDS document `MyType1.hdbdd`.

```
namespace com.acme.myapp1;
using com.acme.myapp1::MyType1;
type MyType2 {
    field1 : String(50);
    field2 : MyType1;
```

```
};
```

In the following example, you must save the data-type definition “MyType3” in the CDS document `MyType3.hdbcds`; the document contains a using directive pointing to the data-type “MyType2” defined in CDS document `MyType2.hdbdd`.

```
namespace com.acme.myapp1;
using com.acme.myapp1::MyType2;
type MyType3 {
    field1 : UTCTimestamp;
    field2 : MyType2;
};
```

The following code example shows how to use the `type of` keyword to define an element using the definition specified in another user-defined data-type field. For example, `field4 : type of field3;` indicates that, like `field3`, `field4` is a `LocalDate` data type.

```
namespace com.acme.myapp1;
using com.acme.myapp1::MyType1;
entity MyEntity1 {
  key id  : Integer;
  field1  : MyType3;
  field2  : String(24);
  field3  : LocalDate;
  field4  : type of field3;
  field5  : type of MyType1.field2;
  field6  : type of InnerCtx.CtxType.b;  // context reference
};
```

You can use the `type of` keyword in the following ways:

- Define a new element (`field4`) using the definition specified in another user-defined element `field3`:
  `field4 : type of field3;`
- Define a new element `field5` using the definition specified in a **field** (`field2`) that belongs to another user-defined data type (`MyType1`):
  `field5 : type of MyType1.field2;`
- Define a new element (`field6`) using an existing field (`b`) that belongs to a data type (`CtxType`) in another **context** (`InnerCtx`):
  `field6 : type of InnerCtx.CtxType.b;`

The following code example shows you how to define nested contexts (`MyContext.InnerCtx`) and refer to data types defined by a user in the specified context.

```
namespace com.acme.myapp1;
context MyContext {
   // Nested contexts
   context InnerCtx {
      Entity MyEntity {
         …
      };
      Type CtxType {
        a : Integer;
        b : String(59);
      };
   };
   type MyType1 {
      field1 : Integer;
      field2 : String(40);
      field3 : Decimal(22,11);
      field4 : Binary(11);
```

```
   };
   type MyType2 {
      field1 : String(50);
      field2 : MyType1;
   };
   type MyType3 {
      field1 : UTCTimestamp;
      field2 : MyType2;
   };
   entity MyEntity1 {
      key id  : Integer;
      field1  : MyType3 not null;
      field2  : String(24);
      field3  : LocalDate;
      field4  : type of field3;
      field5  : type of MyType1.field2;
      field6  : type of InnerCtx.CtxType.b;  // refers to nested context
      field7  : InnerCtx.CtxType;           //  more context references
   } technical configuration {
        unique index IndexA on (field1) asc;
   };
};
```

Restrictions

CDS name resolution does not distinguish between CDS `elements` and CDS `types`. If you define a CDS element based on a CDS data type that has the same name as the new CDS element, CDS displays an error message and the deployment of the objects defined in the CDS document fails.

> ⚠ Caution
>
> In an CDS document, you cannot define a CDS element using a CDS type of the same name; you must specify the **context** where the target type is defined, for example, `MyContext.doobidoo`.

The following example defines an association between a CDS element and a CDS data `type` both of which are named `doobidoo`. The result is an error when resolving the names in the CDS document; CDS expects a type named `doobidoo` but finds an CDS entity element with the same name that is **not** a type.

```
context MyContext2 {
  type doobidoo : Integer;
  entity MyEntity {
    key id : Integer;
    doobidoo : doobidoo;  // error: type expected; doobidoo is not a type
  };
};
```

The following example works, since the explicit reference to the context where the type definition is located (`MyContext.doobidoo`) enables CDS to resolve the definition target.

```
context MyContext {
  type doobidoo : Integer;
  entity MyEntity {
    key id : Integer;
    doobidoo : MyContext.doobidoo;  // OK
  };
};
```

> **i Note**
>
> To prevent name clashes between artifacts that **are** types and those that **have** a type assigned to them, make sure you keep to strict naming conventions. For example, use an **uppercase** first letter for `MyEntity`, `MyView` and `MyType`; use a lowercase first letter for elements `myElement`.

Related Information

Create a CDS User-Defined Structure in XS Advanced [page 203]

4.4.2 CDS Structured Type Definition in XS Advanced

A structured type is a data type comprising a list of attributes, each of which has its own data type. The attributes of the structured type can be defined manually in the structured type itself and reused either by another structured type or an entity.

SAP HANA Extended Application Services advanced model (XS advanced) enables you to create a database structured type as a design-time file. All design-time files including your structured-type definition can be transported to other SAP HANA systems and deployed there to create the same catalog objects as those created in the original SAP HANA system. You can define the structured type using CDS-compliant DDL.

When a CDS document is deployed as part of a database module, the deployment process generates a corresponding catalog object for each of the artifacts defined in the CDS document; the location in the catalog is determined by the type of object generated. For example, the corresponding table type for a CDS type definition is generated in the following catalog location:

`<SID>` > *Catalog* > `<SCHEMA_NAME>` > *Procedures* > *Table Types*

Structured User-Defined Types

In a structured user-defined type, you can define original types (`aNumber` in the following example) or reference existing types defined elsewhere in the same type definition or another, separate type definition (`MyString80`). If you define multiple types in a single CDS document, for example, in a parent context, each structure-type definition must be separated by a semi-colon (;).

The type `MyString80` is defined in the following CDS document:

```
namespace Package1.Package2;
type MyString80: String(80);
```

A using directive is required to resolve the reference to the data type specified in `otherText : MyString80;`, as illustrated in the following example:

```
namespace Package1.Package2;
using Package1.Package2::MyString80;  //contains definition of MyString80
```

```
type MyStruct
{
  aNumber   : Integer;
  someText  : String(80);
  otherText : MyString80;  // defined in a separate type
};
```

> **Note**
>
> If you are using a CDS document to specify a single CDS-compliant data type, the name of the CDS document (`MyStruct.hdbcds`) must match the name of the top-level data type defined in the CDS document, for example, with the *type* keyword.

Nested Structured Types

Since user-defined types can make use of other user-defined types, you can build nested structured types, as illustrated in the following example:

```
namespace Package1.Package2;
using Package1.Package2::MyString80;
using Package1.Package2::MyStruct;
@Schema: 'MYSCHEMA'
context NestedStructs {
    type MyNestedStruct
    {
        name : MyString80;
        nested : MyStruct; // defined in a separate type
    };
    type MyDeepNestedStruct
    {
        text : LargeString;
        nested : MyNestedStruct;
    };
    type MyOtherInt    : type of MyStruct.aNumber; // => Integer
    type MyOtherStruct : type of MyDeepNestedStruct.nested.nested; // => MyStruct
};
```

The example above shows how you can use the `type of` keyword to define a type based on an existing type that is already defined in another user-defined structured type.

Generated Table Types

For each structured type, a SAP HANA table type is generated, whose name is built by concatenating the following elements of the CDS document containing the structured-type definition and separating the elements by a dot delimiter (.):

- the name space (for example, `Pack1.Pack2`)
- the names of all artifacts that enclose the type (for example, `MyModel`)
- the name of the type itself (for example, `MyNestedStruct`)

```
create type "Pack1.Pack2::MyModel.MyNestedStruct" as table(
  name             nvarchar(80),
```

```
  nested.aNumber   integer,
  nested.someText  nvarchar(80),
  nested.otherText nvarchar(80)
);
```

The columns of the table type are built by flattening the elements of the type. Elements with structured types are mapped to one column per nested element, with the column names built by concatenating the element names and separating the names by dots ".".

Table types are only generated for direct structure definitions; in the following example, this would include: `MyStruct`, `MyNestedStruct`, and `MyDeepNestedStruct`. No table types are generated for **derived** types that are based on structured types; in the following example, the derived types include: `MyS`, `MyOtherInt`, `MyOtherStruct`.

Example

```
namespace Pack1."pack-age2";
context MyModel
{
  type MyInteger  : Integer;
  type MyString80 : String(80);
  type MyDecimal  : Decimal(10,2);
  type MyStruct
  {
    aNumber   : Integer;
    someText  : String(80);
    otherText : MyString80;  // defined in example above
  };
  type MyS           : MyStruct;
  type MyOtherInt    : type of MyStruct.aNumber;
  type MyOtherStruct : type of MyDeepNestedStruct.nested.nested;
  type MyNestedStruct
  {
    name   : MyString80;
    nested : MyS;
  };
  type MyDeepNestedStruct
  {
    text   : LargeString;
    nested : MyNestedStruct;
  };
};
```

Related Information

Create a CDS User-Defined Structure in XS Advanced [page 203]
CDS User-Defined Data Types in XS Advanced [page 206]

4.4.3 CDS Structured Types in XS Advanced

A structured type is a data type comprising a list of attributes, each of which has its own data type. The attributes of the structured type can be defined manually in the structured type itself and reused either by another structured type or an entity.

Example

```
namespace examples;
context StructuredTypes {
    type MyOtherInt : type of MyStruct.aNumber; // => Integer
    type MyOtherStruct : type of MyDeepNestedStruct.nested.nested; // => MyStruct
    type EmptyStruct { };
    type MyStruct
    {
        aNumber : Integer;
        aText : String(80);
        anotherText : MyString80; // defined in a separate type
    };

    entity E {
        a : Integer;
        s : EmptyStruct;
    };

    type MyString80 : String(80);
    type MyS : MyStruct;
    type MyNestedStruct
    {
        name : MyString80;
        nested : MyS;
    };
    type MyDeepNestedStruct
    {
        text : LargeString;
        nested : MyNestedStruct;
    };
};
```

type

In a structured user-defined type, you can define original types (`aNumber` in the following example) or reference existing types defined elsewhere in the same type definition or another, separate type definition, for example, `MyString80` in the following code snippet. If you define multiple types in a single CDS document, each structure definition must be separated by a semi-colon (;).

```
type MyStruct
{
    aNumber     : Integer;
    aText       : String(80);
    anotherText : MyString80;  // defined in a separate type
};
```

You can define structured types that do not contain any elements, for example, using the keywords `type EmptyStruct { };`. In the example, below the generated table for entity “E” contains only one column: “a”.

> **!** Restriction
>
> It is not possible to generate an SAP HANA table type for an empty structured type.

```
type EmptyStruct { };
entity E {
    a : Integer;
    s : EmptyStruct;
};
```

type of

You can define a type based on an existing type that is already defined in another user-defined structured type, for example, by using the `type of` keyword, as illustrated in the following example:

```
Context StructuredTypes
{
    type MyOtherInt    : type of MyStruct.aNumber;                  // => Integer
    type MyOtherStruct : type of MyDeepNestedStruct.nested.nested;  // =>
MyStruct
};
```

Related Information

Create a CDS User-Defined Structure in XS Advanced [page 203]
CDS User-Defined Data Types in XS Advanced [page 206]
CDS Structured Type Definition in XS Advanced [page 209]

4.4.4 CDS Primitive Data Types in XS Advanced

In the Data Definition Language (DDL), primitive (or core) data types are the basic building blocks that you use to define *entities* or *structure types* with DDL.

When you are specifying a design-time table (entity) or a view definition using the CDS syntax, you use data types such as *String*, *Binary*, or *Integer* to specify the type of content in the entity columns. CDS supports the use of the following primitive data types:

- DDL data types [page 72]
- Native SAP HANA data types [page 74]

The following table lists all currently supported simple DDL primitive data types. Additional information provided in this table includes the SQL syntax required as well as the equivalent SQL and EDM names for the listed types.

Supported SAP HANA DDL Primitive Types

Name	Description	SQL Literal Syntax	SQL Name	EDM Name
String (n)	Variable-length Unicode string with a specified maximum length of n=1-1333 characters (5000 for SAP HANA specific objects). Default = maximum length. String length (n) is mandatory.	'text with ‘‘quote’’'	NVARCHAR	String
LargeString	Variable length string of up to 2 GB (no comparison)	'text with ‘‘quote’’'	NCLOB	String
Binary(n)	Variable length byte string with user-defined length limit of up to 4000 bytes. Binary length (n) is mandatory.	x'01Cafe', X'01Cafe'	VARBINARY	Binary
LargeBinary	Variable length byte string of up to 2 GB (no comparison)	x'01Cafe', X'01Cafe'	BLOB	Binary
Integer	Respective container's standard signed integer. Signed 32 bit integers in 2's complement, -2**31 .. 2**31-1. Default=NULL	13, -1234567	INTEGER	Int64
Integer64	Signed 64-bit integer with a value range of -2^63 to 2^63-1. Default=NULL.	13, -1234567	BIGINT	Int64
Decimal(p, s)	Decimal number with fixed precision (p) in range of 1 to 34 and fixed scale (s) in range of 0 to p. Values for precision and scale are mandatory.	12.345, -9.876	DECIMAL(p, s)	Decimal
DecimalFloat	Decimal floating-point number (IEEE 754-2008) with 34 mantissa digits; range is roughly ±1e-6143 through ±9.99e+6144	12.345, -9.876	DECIMAL	Decimal
BinaryFloat	Binary floating-point number (IEEE 754), 8 bytes (roughly 16 decimal digits precision); range is roughly ±2.2207e-308 through ±1.7977e+308	1.2, -3.4, 5.6e+7	DOUBLE	Double
LocalDate	Local date with values ranging from 0001-01-01 through 9999-12-31	date'1234-12-31'	DATE	DateTimeOffset Combines date and time; with time zone must be converted to offset

Name	Description	SQL Literal Syntax	SQL Name	EDM Name
LocalTime	Time values (with seconds precision) and values ranging from 00:00:00 through 24:00:00	time'23:59:59', time'12:15'	TIME	Time For duration/ period of time (==xsd:duration). Use DateTimeOffset if there is a date, too.
UTCDateTime	UTC date and time (with seconds precision) and values ranging from 0001-01-01 00:00:00 through 9999-12-31 23:59:59	timestamp'2011-12-31 23:59:59'	SECONDDATE	DateTimeOffset Values ending with "Z" for UTC. Values before 1753-01-01T00:00:00 are not supported; transmitted as NULL.
UTCTimestamp	UTC date and time (with a precision of 0.1 microseconds) and values ranging from 0001-01-01 00:00:00 through 9999-12-31 23:59:59.9999999, and a special initial value	timestamp'2011-12-31 23:59:59.7654321'	TIMESTAMP	DateTimeOffset With Precision = "7"
Boolean	Represents the concept of binary-valued logic	true, false, unknown (null)	BOOLEAN	Boolean

The following table lists all the **native** SAP HANA primitive data types that CDS supports. The information provided in this table also includes the SQL syntax required (where appropriate) as well as the equivalent SQL and EDM names for the listed types.

> **i** Note
>
> * In CDS, the name of SAP HANA data types are prefixed with the word "hana", for example, `hana.ALPHANUM`, or `hana.SMALLINT`, or `hana.TINYINT`.

Supported Native SAP HANA Data Types

Data Type *	Description	SQL Literal Syntax	SQL Name	EDM Name
ALPHANUM	Variable-length character string with special comparison	-	ALPHANUMERIC	-
SMALLINT	Signed 16-bit integer	-32768, 32767	SMALLINT	Int16
TINYINT	Unsigned 8-bit integer	0, 255	TINYINT	Byte
REAL	32-bit binary floating-point number	-	REAL	Single

Data Type *	Description	SQL Literal Syntax	SQL Name	EDM Name
SMALLDECIMAL	64-bit decimal floating-point number	-	SMALLDECIMAL	Decimal
VARCHAR	Variable-length ASCII character string with user-definable length limit n	-	VARCHAR	String
CLOB	Large variable-length ASCII character string, no comparison	-	CLOB	String
BINARY	Byte string of fixed length n	-	BINARY	Blob
ST_POINT	0-dimensional geometry representing a single location	-	-	-
ST_GEOMETRY	Maximal supertype of the geometry type hierarchy; includes `ST_POINT`	-	-	-

The following example shows the **native** SAP HANA data types that CDS supports; the code example also illustrates the mandatory syntax.

> **i** Note
>
> Support for the geo-spatial types `ST_POINT` and `ST_GEOMETRY` is limited: these types can only be used for the definition of elements in types and entities. It is not possible to define a CDS view that selects an element based on a geo-spatial type from a CDS entity.

```
@nokey
entity SomeTypes {
  a : hana.ALPHANUM(10);
  b : hana.SMALLINT;
  c : hana.TINYINT;
  d : hana.SMALLDECIMAL;
  e : hana.REAL;
  h : hana.VARCHAR(10);
  i : hana.CLOB;
  j : hana.BINARY(10);
  k : hana.ST_POINT;
  l : hana.ST_GEOMETRY;
};
```

Related Information

Create a CDS User-Defined Structure in XS Advanced [page 203]

4.5 Create a CDS Association in XS Advanced

You create associations in a CDS entity definition, which is a design-time file in SAP HANA.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have access to the SAP Web IDE for SAP HANA

  > **i** Note
  >
  > The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the tools included in the SAP Web IDE for SAP HANA.

- You must have already a development workspace and a multi-target application (MTA) project.
- You must already have created a database module for your MTA project.
- You must already have set up an HDI container for the CDS artifacts

  > **i** Note
  >
  > A container setup file (`.hdiconfig`) is required to define which plug-ins to use to create the corresponding catalog objects from the design-time artifacts when the multi-target application (or just the database module) is deployed.

Context

Associations define relationships between entities (tables). SAP HANA Extended Application Services for XS advanced (XS advanced) enables you to use the CDS syntax to create associations between entities. The association is defined in the entity definiton itself. To create an association between CDS entities, perform the following steps:

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following URL:

   `https://<HANA_HOST>:53075/`

> → Tip
>
> To display the URL for the SAP Web IDE for SAP HANA, open a command shell, log on to the XS advanced run time, and run the following command:
>
> `xs app webide --urls`

2. Open the application project to which you want to add your CDS association.
3. Create the CDS entity-definition file that will contain the associations you define.

   Browse to the folder in the database module in your application's project workspace where you want to create the new CDS entity-definition file, for example, `<MyApp1>/HDB/src`, and perform the following steps:

   a. Right-click the folder where you want to save the CDS entity-definition file and choose *New* > *CDS Artifact* in the context-sensitive pop-up menu.
   b. Enter the name of the entity-definition file in the *File Name* box, for example, `MyEntity`.

      > → Tip
      >
      > If you use the available setup Wizards to create your design-time artifacts, the correct file extensions is added automatically. The file extension is used to determine which plug-in to use to create the corresponding run-time object during deployment. CDS artifacts have the file extension `.hdbcds`, for example, `MyEntity.hdbcds`.

   c. Choose *Finish* to save the new CDS entity-definition file in the database module of your local project workspace.
4. Define the underlying CDS entities and structured types.

   If the new entity-definition file is not automatically displayed by the file-creation wizard, double-click the entity-definition file you created in the previous step, for example, `MyEntity.hdbcds`, and add the entity-definition code to the file:

> i Note
>
> The following code example is provided for illustration purposes only.

```
econtext MyEntity1 {
    type StreetAddress {
        name : String(80);
        number : Integer;
    };
    type CountryAddress {
        name : String(80);
        code : String(3);
    };
    entity Address {
        key id : Integer;
        street : StreetAddress;
        zipCode : Integer;
        city : String(80);
        country : CountryAddress;
        type : String(10); // home, office
    };
    entity Person
    {
        key id : Integer;
        addressId : Integer;
    };
```

```
};
```

5. Define a one-to-**one** association between CDS entities.

   In the same entity-definition file you edited in the previous step, for example, `MyEntity.hdbcds`, add the code for the one-to-one association between the entity `Person` and the entity `Address`, as illustrated below:

   > **i** Note
   >
   > This example does not specify cardinality or foreign keys, so the cardinality is set to the default 0..1, and the target entity's primary key (the element id) is used as foreign key.

```
entity Person
  {
   key id : Integer;
   address1 : Association to Address;
   addressId : Integer;
};
```

6. Define an unmanaged association with cardinality one-to-**many** between CDS entities.

   In the same entity-definition file you edited in the previous step, for example, `MyEntity.hdbcds`, add the code for the one-to-many association between the entity `Address` and the entity `Person`. The code should look something like the bold text illustrated in the following example:

```
entity Address {
   key id : Integer;
   street : StreetAddress;
   zipCode : Integer;
   city : String(80);
   country : CountryAddress;
   type : String(10); // home, office
   inhabitants : Association[*] to Person on inhabitants.addressId = id;
};
```

7. Save the CDS document.

   Saving the definition persists the file in your local workspace; it does not create any objects in the database catalog.

8. Activate the changes in the catalog.

   To activate the new entity definition and generate a corresponding table in the catalog, use the *Build* feature.

   a. Right-click the new database module in your application project.

   b. In the context-sensitive pop-up menu, choose *Build*.

   > → Tip
   >
   > You can follow progress of the build in the console at the bottom of the CDS editor.

Related Information

Set up an SAP HDI Container for XS Advanced Applications

Create a CDS Document (XS Advanced) [page 144]

4.5.1 CDS Associations in XS Advanced

Associations define relationships between entities.

Associations are specified by adding an element to a source entity with an association **type** that points to a target entity, complemented by optional information defining cardinality and which keys to use.

> **i** Note
>
> CDS supports both managed and unmanaged associations.

SAP HANA Extended Application Services (SAP HANA XS) enables you to use associations in CDS entities or CDS views. The syntax for **simple** associations in a CDS document is illustrated in the following example:

```
namespace samples;
@Schema: 'MYSCHEMA'          // XS classic *only*
context SimpleAssociations {
    type StreetAddress {
        name : String(80);
        number : Integer;
    };
    type CountryAddress {
        name : String(80);
        code : String(3);
    };
    entity Address {
        key id : Integer;
        street : StreetAddress;
        zipCode : Integer;
        city : String(80);
        country : CountryAddress;
        type : String(10); // home, office
    };
    entity Person
    {
        key id : Integer;
        // address1,2,3 are to-one associations
        address1 : Association to Address;
        address2 : Association to Address { id };
        address3 : Association[1] to Address { zipCode, street, country };
        // address4,5,6 are to-many associations
        address4 : Association[0..*] to Address { zipCode };
        address5 : Association[*] to Address { street.name };
        address6 : Association[*] to Address { street.name AS streetName,
        country.name AS countryName };
    };
};
```

Cardinality in Associations

When using an association to define a relationship between entities in a CDS document, you use the **cardinality** to specify the type of relation, for example, one-to-one (to-one) or one-to-many (to-n); the relationship is with respect to both the source and the target of the association.

The target cardinality is stated in the form of `[ min .. max ]`, where `max=*` denotes infinity. If no cardinality is specified, the default cardinality setting `[ 0..1 ]` is assumed. It is possible to specify the maximum cardinality of the source of the association in the form `[ maxs, min .. max]`, too, where `maxs = *` denotes infinity.

> → Tip
>
> The information concerning the maximum cardinality is only used as a hint for optimizing the execution of the resulting JOIN.

The following examples illustrate how to express cardinality in an association definition:

```
namespace samples;
@Schema: 'MYSCHEMA'              // XS classic *only*
context AssociationCardinality {
    entity Associations {
        // To-one associations
        assoc1 : Association[0..1]    to target; // has no or one target instance
        assoc2 : Association          to target; // as assoc1, uses the default
[0..1]
        assoc3 : Association[1]       to target; // as assoc1; the default for
min is 0
        assoc4 : Association[1..1]    to target; // association has one target
instance
        // To-many associations
        assoc5 : Association[0..*]    to target{id1};
        assoc6 : Association[]        to target{id1}; // as assoc5, [] is short
for [0..*]
        assoc7 : Association[2..7]    to target{id1}; // any numbers are
possible; user provides
        assoc8 : Association[1, 0..*] to target{id1}; // additional info. about
source cardinality
    };
    // Required to make the example above work
    entity target {
        key id1 : Integer;
        key id2 : Integer;
    };
};
```

Target Entities in Associations

You use the `to` keyword in a CDS view definition to specify the target entity in an association, for example, the name of an entity defined in a CDS document. A qualified entity name is expected that refers to an existing entity. A target entity specification is mandatory; a default value is **not** assumed if no target entity is specified in an association relationship.

The entity `Address` specified as the target entity of an association could be expressed in any of the ways illustrated the following examples:

```
address1 : Association to Address;
address2 : Association to Address { id };
address3 : Association[1] to Address { zipCode, street, country };
```

Filter Conditions and Prefix Notation

When following an association (for example, in a view), it is now possible to apply a filter condition; the filter is merged into the `ON`-condition of the resulting `JOIN`. The following example shows how to get a list of customers and then filter the list according to the sales orders that are currently “open” for each customer. In the example, the infix filter is inserted after the association orders to get only those orders that satisfy the condition `[status='open']`.

Sample Code

```
view C1 as select from Customer {
  name,
  orders[status='open'].id as orderId
};
```

The association `orders` is defined in the entity definition illustrated in the following code example:

Sample Code

```
entity Customer {
   key id : Integer;
   orders : Association[*] to SalesOrder on orders.cust_id = id;
   name : String(80);
};
entity SalesOrder {
   key id : Integer;
   cust_id : Integer;
   customer: Association[1] to Customer on customer.id = cust_id;
   items : Association[*] to Item on items.order_id = id;
   status: String(20);
   date : LocalDate;
};
entity Item {
   key id : Integer;
   order_id : Integer;
   salesOrder : Association[1] to SalesOrder on salesOrder.id = order_id;
   descr : String(100);
   price : Decimal(8,2);
};
```

→ Tip

For more information about filter conditions and prefixes in CDS views, see *CDS Views* and *CDS View Syntax Options*.

Foreign Keys in Associations

For **managed** associations, the relationship between source and target entity is defined by specifying a set of elements of the target entity that are used as a foreign key. If no foreign keys are specified explicitly, the elements of the target entity’s designated primary key are used. Elements of the target entity that reside inside substructures can be addressed via the respective path. If the chosen elements do not form a unique key of the

target entity, the association has cardinality to-many. The following examples show how to express foreign keys in an association.

```
namespace samples;
using samples::SimpleAssociations.StreetAddress;
using samples::SimpleAssociations.CountryAddress;
using samples::SimpleAssociations.Address;
@Schema: 'MYSCHEMA'           // XS classic *only*
context ForeignKeys {
    entity Person
    {
        key id : Integer;
        // address1,2,3 are to-one associations
        address1 : Association to Address;
        address2 : Association to Address { id };
        address3 : Association[1] to Address { zipCode, street, country };
        // address4,5,6 are to-many associations
        address4 : Association[0..*] to Address { zipCode };
        address5 : Association[*] to Address { street.name };
        address6 : Association[*] to Address { street.name AS streetName,
        country.name AS countryName };
    };
    entity Header {
        key id : Integer;
        toItems : Association[*] to Item on toItems.head.id = id;
    };
    entity Item {
        key id : Integer;
        head : Association[1] to Header { id };
        // <...>
    };
};
```

- `address1`
  No foreign keys are specified: the target entity's primary key (the element `id`) is used as foreign key.
- `address2`
  Explicitly specifies the foreign key (the element `id`); this definition is similar to `address1`.
- `address3`
  The foreign key elements to be used for the association are explicitly specified, namely: `zipcode` and the structured elements `street` and `country`.
- `address4`
  Uses only `zipcode` as the foreign key. Since `zipcode` is not a unique key for entity `Address`, this association has cardinality “to-many”.
- `address5`
  Uses the subelement `name` of the structured element `street` as a foreign key. This is not a unique key and, as a result, `address4` has cardinality “to-many”.
- `address6`
  Uses the subelement `name` of both the structured elements `street` and `country` as foreign key fields. The names of the foreign key fields must be unique, so an alias is required here. The foreign key is not unique, so `address6` is a “to-many” association.

You can use foreign keys of managed associations in the definition of other associations. In the following example, the appearance of association `head` in the ON condition is allowed; the compiler recognizes that the field `head.id` is actually part of the entity `Item` and, as a result, can be obtained without following the association `head`.

Sample Code

```
entity Header {
  key id : Integer;
  toItems : Association[*] to Item on toItems.head.id = id;
};

entity Item {
  key id : Integer;
  head : Association[1] to Header { id };
  ...
};
```

Restrictions

CDS name resolution does not distinguish between CDS `associations` and CDS `entities`. If you define a CDS association with a CDS entity that has the same name as the new CDS association, CDS displays an error message and the activation of the CDS document fails.

⚠ Caution

In an CDS document, to define an association with a CDS entity of the same name, you must specify the **context** where the target entity is defined, for example, `Mycontext.Address3`.

The following code shows some examples of associations with a CDS entity that has the same (or a similar) name. Case sensitivity ("a", "A") is important; in CDS documents, `address` is not the same as `Address`. In the case of `Address2`, where the association name and the entity name are identical, the result is an error; when resolving the element names, CDS expects an entity named `Address2` but finds a CDS association with the same name instead. `MyContext.Address3` is allowed, since the target entity can be resolved due to the absolute path to its location in the CDS document.

```
context MyContext {
  entity Address  {…}
  entity Address1 {…}
  entity Address2 {…}
  entity Address3 {…}
  entity Person
  {
    key id   : Integer;
    address  : Association to Address;        // OK: "address" ≠ "Address"
    address1 : Association to Address1;       // OK: "address1" ≠ "Address1"
    Address2 : Association to Address2;       // Error: association name =
entity name
    Address3 : Association to MyContext.Address3;  //OK: full path to Address3

  };
};
```

Example: Complex (One-to-Many) Association

The following example shows a more complex association (to-many) between the entity "`Header`" and the entity "`Item`".

```
namespace samples;
@Schema: 'MYSCHEMA'        // XS classic *only*
context ComplexAssociation {
    Entity Header {
        key PurchaseOrderId: BusinessKey;
        Items: Association [0..*] to Item on
Items.PurchaseOrderId=PurchaseOrderId;
        "History": HistoryT;
        NoteId: BusinessKey null;
        PartnerId: BusinessKey;
        Currency: CurrencyT;
        GrossAmount: AmountT;
        NetAmount: AmountT;
        TaxAmount: AmountT;
        LifecycleStatus: StatusT;
        ApprovalStatus: StatusT;
        ConfirmStatus: StatusT;
        OrderingStatus: StatusT;
        InvoicingStatus: StatusT;
    } technical configuration {
        column store;
    };
    Entity Item {
        key PurchaseOrderId: BusinessKey;
        key PurchaseOrderItem: BusinessKey;
        ToHeader: Association [1] to Header on
ToHeader.PurchaseOrderId=PurchaseOrderId;
        ProductId: BusinessKey;
        NoteId: BusinessKey null;
        Currency: CurrencyT;
        GrossAmount: AmountT;
        NetAmount: AmountT;
        TaxAmount: AmountT;
        Quantity: QuantityT;
        QuantityUnit: UnitT;
        DeliveryDate: SDate;
    } technical configuration {
        column store;
    };
    define view POView as SELECT from Header {
        Items.PurchaseOrderId as poId,
        Items.PurchaseOrderItem as poItem,
        PartnerId,
        Items.ProductId
    };
    // Missing types from the example above
    type BusinessKey: String(50);
    type HistoryT: LargeString;
    type CurrencyT: String(3);
    type AmountT: Decimal(15, 2);
    type StatusT: String(1);
    type QuantityT: Integer;
    type UnitT: String(5);
    type SDate: LocalDate;
};
```

Related Information

Create a CDS Association in XS Advanced [page 217]
CDS Association Syntax Options in XS Advanced [page 226]

4.5.2 CDS Association Syntax Options in XS Advanced

Associations define relationships between entities.

Example: Managed Associations

```
Association [ <cardinality> ]  to <targetEntity> [ <forwardLink> ]
```

Example: Unmanaged Associations

```
Association [ <cardinality> ]  to <targetEntity> <unmanagedJoin>
```

Overview

Associations are specified by adding an element to a source entity with an association **type** that points to a target entity, complemented by optional information defining cardinality and which keys to use.

> **i** Note
>
> CDS supports both managed and unmanaged associations.

SAP HANA Extended Application Services (SAP HANA XS) enables you to use associations in the definition of a CDS entity or a CDS view. When defining an association, bear in mind the following points:

- `<Cardinality>` [page 227]
  The relationship between the source and target in the association, for example, one-to-one, one-to-many, many-to-one
- `<targetEntity>` [page 228]
  The target entity for the association
- `<forwardLink>` [page 229]
  The foreign keys to use in a managed association, for example, element names in the target entity
- `<unmanagedJoin>` [page 230]
  Unmanaged associations only; the `ON` condition specifies the elements of the source and target elements and entities to use in the association

Association Cardinality

When using an association to define a relationship between entities in a CDS view; you use the **cardinality** to specify the type of relation, for example:

- one-to-one (to-one)
- one-to-many (to-n)

The relationship is with respect to both the source and the target of the association. The following code example illustrates the syntax required to define the cardinality of an association in a CDS view:

```
[  [ ( maxs | * ) , ]                // source cardinality
   [ min .. ] ( max | * )            // target cardinality
]
```

In the most simple form, only the target cardinality is stated using the syntax `[ min .. max ]`, where `max=*` denotes infinity. Note that `[]` is short for `[ 0..* ]`. If no cardinality is specified, the default cardinality setting `[ 0..1 ]` is assumed. It is possible to specify the maximum cardinality of the source of the association in the form `[ maxs, min .. max]`, where `maxs = *` denotes infinity.

The following examples illustrate how to express cardinality in an association definition:

```
namespace samples;
@Schema: 'MYSCHEMA'              // XS classic *only*
context AssociationCardinality {
    entity Associations {
        // To-one associations
        assoc1 : Association[0..1]    to target;
        assoc2 : Association          to target;
        assoc3 : Association[1]       to target;
        assoc4 : Association[1..1]    to target; // association has one target
instance
        // To-many associations
        assoc5 : Association[0..*]    to target{id1};
        assoc6 : Association[]        to target{id1}; // as assoc4, [] is short
for [0..*]
        assoc7 : Association[2..7]    to target{id1}; // any numbers are
possible; user provides
        assoc8 : Association[1, 0..*] to target{id1}; // additional info. about
source cardinality
    };
    // Required to make the example above work
    entity target {
        key id1 : Integer;
        key id2 : Integer;
    };
};
```

The following table describes the various cardinality expressions illustrated in the example above:

Association Cardinality Syntax Examples

Association	Cardinality	Explanation
`assoc1`	[0..1]	The association has no or one target instance
`assoc2`		Like `assoc1`, this association has no or one target instance and uses the default [0..1]

Association	Cardinality	Explanation
`assoc3`	[1]	Like `assoc1`, this association has no or one target instance; the default for min is 0
`assoc4`	[1..1]	The association has one target instance
`assoc5`	[0..*]	The association has no, one, or multiple target instances
`assoc6`	[]	Like `assoc4`, [] is short for [0..*] (the association has no, one, or multiple target instances)
`assoc7`	[2..7]	Any numbers are possible; the user provides
`assoc8`	[1, 0..*]	The association has no, one, or multiple target instances and includes additional information about the source cardinality

When an infix filter effectively reduces the cardinality of a "to-N" association to "to-1", this can be expressed explicitly in the filter, for example:

```
assoc[1: <cond> ]
```

Specifying the cardinality in the filter in this way enables you to use the association in the `WHERE` clause, where "to-N" associations are not normally allowed.

Sample Code

```
namespace samples;
@Schema: 'MYSCHEMA'             // XS classic *only*
context CardinalityByInfixFilter {
    entity Person {
        key id : Integer;
        name : String(100);
        address : Association[*] to Address on address.personId = id;
    };
    entity Address {
        key id : Integer;
        personId : Integer;
        type : String(20); // home, business, vacation, ...
        street : String(100);
        city : String(100);
    };
    view V as select from Person {
        name
    } where address[1: type='home'].city = 'Accra';
};
```

Association Target

You use the `to` keyword in a CDS view definition to specify the target entity in an association, for example, the name of an entity defined in a CDS document. A qualified entity name is expected that refers to an existing entity. A target entity specification is mandatory; a default value is **not** assumed if no target entity is specified in an association relationship.

```
Association[ <cardinality> ]  to <targetEntity> [ <forwardLink> ]
```

The target entity `Address` specified as the target entity of an association could be expressed as illustrated the following examples:

```
address1 : Association to Address;
address2 : Association to Address { id };
address3 : Association[1] to Address { zipCode, street, country };
```

Association Keys

In the relational model, associations are mapped to foreign-key relationships. For **managed** associations, the relation between source and target entity is defined by specifying a set of elements of the target entity that are used as a foreign key, as expressed in the `forwardLink` element of the following code example:

```
Association[ <cardinality> ]  to <targetEntity> [ <forwardLink> ]
```

The `forwardLink` element of the association could be expressed as follows:

```
<forwardLink>  =  { <foreignKeys> }
<foreignKeys>  = <targetKeyElement> [ AS <alias> ] [ , <foreignKeys> ]
<targetKeyElement> = <elementName> ( . <elementName> )*
```

If no foreign keys are specified explicitly, the elements of the target entity's designated primary key are used. Elements of the target entity that reside inside substructures can be addressed by means of the respective path. If the chosen elements do not form a unique key of the target entity, the association has cardinality to-many. The following examples show how to express foreign keys in an association.

```
entity Person
{
  key id : Integer;
  // address1,2,3 are to-one associations
  address1 : Association to Address;
  address2 : Association to Address { id };
  address3 : Association[1] to Address { zipCode, street, country };
  // address4,5,6 are to-many associations
  address4 : Association[0..*] to Address { zipCode };
  address5 : Association[*] to Address { street.name };
  address6 : Association[*] to Address { street.name AS streetName,
                                         country.name AS countryName };
};
```

Association Syntax Options

Association	Keys	Explanation
`address1`		No foreign keys are specified: the target entity's primary key (the element `id`) is used as foreign key.
`address2`	`{ id }`	Explicitly specifies the foreign key (the element `id`); this definition is identical to `address1`.
`address3`	`{ zipCode, street, country }`	The foreign key elements to be used for the association are explicitly specified, namely: `zipcode` and the structured elements `street` and `country`.

Association	Keys	Explanation
`address4`	`{ zipCode }`	Uses only `zipcode` as the foreign key. Since `zipcode` is not a unique key for entity `Address`, this association has cardinality “to-many”.
`address5`	`{ street.name }`	Uses the sub-element `name` of the structured element `street` as a foreign key. This is not a unique key and, as a result, `address4` has cardinality “to-many”.
`address6`	`{ street.name AS streetName, country.name AS countryName }`	Uses the sub-element `name` of both the structured elements `street` and `country` as foreign key fields. The names of the foreign key fields must be unique, so an alias is required here. The foreign key is not unique, so `address6` is a “to-many” association.

You can now use foreign keys of managed associations in the definition of other associations. In the following example, the compiler recognizes that the field `toCountry.cid` is part of the foreign key of the association `toLocation` and, as a result, physically present in the entity `Company`.

Sample Code

```
namespace samples;
@Schema: 'MYSCHEMA'         // XS classic *only*
context AssociationKeys {
    entity Country {
        key c_id : String(3);
        // <...>
    };
    entity Region {
        key r_id : Integer;
        key toCountry : Association[1] to Country { c_id };
        // <...>
    };
    entity Company {
        key id : Integer;
        toLocation : Association[1] to Region { r_id, toCountry.c_id };
        // <...>
    };
};
```

Unmanaged Associations

Unmanaged associations are based on existing elements of the source and target entity; no fields are generated. In the `ON` condition, only elements of the source or the target entity can be used; it is not possible to use other associations. The `ON` condition may contain any kind of expression - all expressions supported in views can also be used in the `ON` condition of an unmanaged association.

Note

The names in the `ON` condition are resolved in the scope of the source entity; elements of the target entity are accessed through the association itself .

In the following example, the association `inhabitants` relates the element `id` of the source entity `Room` with the element `officeId` in the target entity `Employee`. The target element `officeId` is accessed through the name of the association itself.

```
namespace samples;
@Schema: 'MYSCHEMA'              // XS classic *only*
context UnmanagedAssociations {
    entity Employee {
        key id : Integer;
        officeId : Integer;
        // <...>
    };
    entity Room {
        key id : Integer;
        inhabitants : Association[*] to Employee on inhabitants.officeId = id;
        // <...>
    };
    entity Thing {
        key id : Integer;
        parentId : Integer;
        parent : Association[1] to Thing on parent.id = parentId;
        children : Association[*] to Thing on children.parentId = id;
        // <...>
    };
};
```

The following example defines two related **unmanaged** associations:

- `parent`
  The unmanaged association `parent` uses a cardinality of `[1]` to create a relation between the element `parentId` and the target element `id`. The target element `id` is accessed through the name of the association itself.
- `children`
  The unmanaged association `children` creates a relation between the element `id` and the target element `parentId`. The target element `parentId` is accessed through the name of the association itself.

```
entity Thing {
  key id   : Integer;
  parentId : Integer;
  parent   : Association[1] to Thing on parent.id = parentId;
  children : Association[*] to Thing on children.parentId = id;
  ...
};
```

Constants in Associations

The usage of constants is no longer restricted to annotation assignments and default values for entity elements. With SPS 11, you can use constants in the "`ON`"-condition of unmanaged associations, as illustrated in the following example:

Sample Code

```
context MyContext {
  const MyIntConst      : Integer      = 7;
  const MyStringConst   : String(10)   = 'bright';
  const MyDecConst      : Decimal(4,2) = 3.14;
  const MyDateTimeConst : UTCDateTime  = '2015-09-30 14:33';
  entity MyEntity {
    key id : Integer;
    a : Integer;
    b : String(100);
```

```
    c : Decimal(20,10);
    d : UTCDateTime;
    your : Association[1] to YourEntity on your.a - a < MyIntConst;
  };
  entity YourEntity {
    key id : Integer;
    a : Integer;
  };
  entity HerEntity {
    key id : Integer;
    t : String(20);
  };
  view MyView as select from MyEntity
           inner join HerEntity on locate (b, :MyStringConst) > 0
  {
    a + :MyIntConst as x,
    b || ' is ' || :MyStringConst as y,
    c * sin(:MyDecConst) as z
  } where d < :MyContext.MyDateTimeConst;
};
```

Related Information

Create a CDS Association in XS Advanced [page 217]
CDS Associations in XS Advanced [page 220]

4.6 Create a CDS View in XS Advanced

Define a design-time **view** using the Core Data Services (CDS) syntax.

Prerequisites

To complete this task successfully, note the following prerequisites:

- You must have access to an SAP HANA system.
- You must have access to the SAP Web IDE for SAP HANA

  > **i** Note
  >
  > The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the tools included in the SAP Web IDE for SAP HANA.

- You must have already a development workspace and a multi-target application (MTA) project.
- You must already have created a database module for your MTA application project.
- You must already have set up an HDI container for the CDS artifacts

> **i Note**
>
> A container setup file (`.hdiconfig`) is required to define which plug-ins to use to create the corresponding catalog objects from the design-time artifacts when the multi-target application (or just the database module) is deployed.

- You must have access to the SAP HANA XS advanced run-time tools that enable you to view the contents of the catalog.

  > **i Note**
  >
  > The permissions defined in the XS advanced role collection `XS_AUTHORIZATION_USER` must be assigned to the user who wants to access the SAP HANA run-time tools.

Context

A view is a virtual table based on the dynamic results returned in response to an SQL statement. SAP HANA Extended Application Services for XS advanced model (XS advanced) enables you to use CDS syntax to create a database view as a design-time file. You can use this design-time view definition to generate the corresponding catalog object when you deploy the application that contains the view-definition artifact (or just the application's database module).

> **i Note**
>
> The code examples provided are for illustration purposes only.

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following URL:

   `https://<HANA_HOST>:53075/`

   > **→ Tip**
   >
   > To display the URL for the SAP Web IDE for SAP HANA, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > ```
   > xs app webide --urls
   > ```

2. Open the application project to which you want to add your CDS entity.
3. Create the CDS document that will contain the view-definition.

   Browse to the folder in the database module in your application's project workspace, for example, `<MyApp1>/HDB/src` where you want to create the new CDS document with the view-definition file and perform the following steps:

   a. Right-click the folder where you want to save the CDS entity-definition file and choose *New* > *CDS Artifact* in the context-sensitive pop-up menu.

b. Enter the name of the view-definition file in the *File Name* box, for example, `MyViewContext`.

> → Tip
>
> If you use the available setup Wizards to create your design-time artifacts, the correct file extensions is added automatically. The file extension is used to determine which plug-in to use to create the corresponding run-time object during deployment. CDS artifacts have the file extension `.hdbcds`, for example, `MyViewContext.hdbcds`.

c. Choose *Finish* to save the new CDS view-definition file in the database module of your application's local project workspace.

4. Define the underlying CDS entities and structured types for the SQL view.

   If the new CDS document is not automatically displayed by the file-creation wizard, double-click the CDS file you created in the previous step, for example, `MyViewContext.hdbcds`, and add the code that defines the underlying table entities and structured types:

```
context MyViewContext {
    type StreetAddress {
        name : String(80);
        number : Integer;
    };
    type CountryAddress {
        name : String(80);
        code : String(3);
    };
    entity Address {
        key id : Integer;
        street : StreetAddress;
        zipCode : Integer;
        city : String(80);
        country : CountryAddress;
        type : String(10); // home, office
    } technical configuration {
        column store;
    };
};
```

5. Define an SQL view as a projection of a CDS entity.

   In the same CDS document you edited in the previous step, for example, `MyViewContext.hdbcds`, add the code for the view `AddressView` to the end of the document below the entity `Address`.

> i Note
>
> In CDS, a view is an entity without an its own persistence; it is defined as a projection of other entities.

```
view AddressView as select from Address
    {
        id,
        street.name,
        street.number
    };
};
```

6. Save the CDS document with the view-definition.

   Saving the definition persists the file in your local workspace; it does not create any objects in the database catalog.

7. Deploy the new view (and corresponding tables and types) in the catalog.

To activate the CDS artifacts defined in the CDS document and generate the corresponding objects in the catalog, use the *Build* feature.

a. Right-click the new database module in your application project.

b. In the context-sensitive pop-up menu, choose *Build*.

> → Tip
>
> You can follow the build progress in the console at the bottom of the CDS editor.

8. Check that the new table, type, and view objects have been successfully created in the catalog.

> → Tip
>
> A selection of run-time tools is available in the *Database Explorer* perspective of SAP web IDE for SAP HANA at the following location:
>
> *Tools* > *Database Explorer*

In XS advanced, your database run-time objects are located in the HDI container created for your multi-target application's database module; you need to locate and bind to this application-specific container to view its contents. The container name contains the name of the user logged into the SAP Web IDE for SAP HANA, the name of the database module containing the CDS design-time entities, and the string *-hdi-container*, for example:

`<XS_UserName>`*-ctetig24[...]-*`<DB_Module>`*-hdi-container*

To bind to the HDI container, in the SAP HANA run-time *Catalog* tool, right-click *Catalog* in the catalog list, and in the *Search HDI Containers* dialog, locate the container to which you want to bind, and choose *Bind*.

Related Information

Set up an SAP HDI Container for XS Advanced Applications
Create the Data Persistence Artifacts with CDS in XS Advanced [page 135]
Create a CDS Document (XS Advanced) [page 144]
CDS View Syntax Options [page 96]

4.6.1 CDS Views in XS Advanced

XS advanced enables you to create a CDS view as a design-time file.

A view is an entity that is not persistent; it is defined as the projection of other entities. SAP HANA Extended Application Services, advanced model, enables you to define a view in a CDS document, which you store as design-time file. Design-time files can be read by applications that you develop.

If your application refers to the design-time version of a view rather than the runtime version in the catalog, for example, by using the explicit path to the design-time file (with suffix), any changes to the design-time version of the file are visible as soon as they are committed. There is no need to wait for the activation of a runtime version of the view.

To define a transportable view using the CDS-compliant view specifications, use something like the code illustrated in the following example:

```
context Views {
    VIEW AddressView AS SELECT FROM Address {
        id,
        street.name,
        street.number
    };
<...>
}
```

When a CDS document is activated, the activation process generates a corresponding catalog object for each of the artifacts defined in the document; the location in the catalog is determined by the type of object generated. For example, in SAP HANA XS, the corresponding catalog object for a CDS view definition is generated in the following location:

Views defined in a CDS document can make use of the following SQL features:

- CDS Type definition
- Expressions and functions (for example, "`a + b as theSum`")
- Aggregates, "GROUP BY", and "HAVING" clauses
- Associations (including filters and prefixes)
- ORDER BY, CASE, UNION, JOIN, and TOP
- With Parameters
- Select Distinct
- Spatial Data (for example, "`ST_Distance`")

→ Tip

For more information about the syntax required when using these SQL features in a CDS view, see *CDS View Syntax Options* in *Related Information*.

Type Definition

In a CDS view definition, you can explicitly specify the `type` of a select item, as illustrated in the following example:

Sample Code

```
type MyInteger : Integer;
entity E {
  a : MyInteger;
  b : MyInteger;
};
view V as select from E {
  a,
  a+b as s1,
  a+b as s2 : MyInteger
};
```

In the example of different `type` definitions, the following is true:

- `a,`
  Has type "`MyInteger`"
- `a+b as s1,`
  Has type "`Integer`" and any information about the user-defined `type` is lost
- `a+b as s2 : MyInteger`
  Has type "`MyInteger`", which is explicitly specified

> **Note**
>
> If necessary, a `CAST` function is added to the generated view in SAP HANA; this ensures that the `select` item's type in the SAP HANA view is the SAP HANA "type" corresponding to the explicitly specified CDS type.

Related Information

Create a CDS View in XS Advanced [page 232]
CDS View Syntax Options in XS Advanced [page 237]
Spatial Types and Functions in XS Advanced [page 253]

4.6.2 CDS View Syntax Options in XS Advanced

SAP HANA XS includes a dedicated, CDS-compliant syntax, which you must adhere to when using a CDS document to define a view as a design-time artifact in XS advanced.

Example

> **Note**
>
> The following example is intended for illustration purposes only and might contain syntactical errors. For further details about the keywords illustrated, click the links provided.

```
context views {
const x : Integer = 4;
const y : Integer = 5;
const Z : Integer = 6;
VIEW MyView1 AS SELECT FROM Employee
{
  a + b  AS theSum
};
VIEW MyView2 AS SELECT FROM Employee
{ officeId.building,
  officeId.floor,
  officeId.roomNumber,
  office.capacity,
```

```
  count(id) AS seatsTaken,
  count(id)/office.capacity as occupancyRate
} WHERE officeId.building = 1
  GROUP BY officeId.building,
           officeId.floor,
           officeId.roomNumber,
           office.capacity,
           office.type
  HAVING office.type = 'office' AND count(id)/office.capacity < 0.5;
VIEW MyView3 AS SELECT FROM Employee
{ orgUnit,
  salary
} ORDER BY salary DESC;
VIEW MyView4 AS SELECT FROM Employee {
    CASE
        WHEN a < 10 then 'small'
        WHEN 10 <= a AND a < 100 THEN 'medium'
        ELSE 'large'
    END AS size
};
VIEW MyView5 AS
  SELECT FROM E1 { a, b, c}
  UNION
  SELECT FROM E2 { z, x, y};
VIEW MyView6 AS SELECT FROM Customer {
  name,
  orders[status='open'].{ id   as orderId,
                          date as orderDate,
                          items[price>200].{ descr,
                                             price }  }
};
VIEW MyView7 as
  select from E { a, b, c}
  order by a limit 10 offset 30;
VIEW V_join as select from E join (F as X full outer join G on X.id = G.id) on
E.id = c {
  a, b, c
};
VIEW V_top as select from E TOP 10 { a, b, c};
VIEW V_dist as select from E distinct { a };
VIEW V_param with parameters PAR1: Integer, PAR2: MyUserDefinedType, PAR3: type
of E.elt
      as select from MyEntity {
        id,
        elt };
VIEW V_type as select from E {
  a,
  a+b as s1,
  a+b as s2 : MyInteger
};
view VE as select from E mixin {
  f : Association[1] to VF on f.vy = $projection.vb;
 } into {
  a as va,
  b as vb,
  f as vf
 };
VIEW SpatialView1 as select from Person {
    name,
    homeAddress.street_name   || ', ' || homeAddress.city   as home,
    officeAddress.street_name || ', ' || officeAddress.city as office,
    round( homeAddress.loc.ST_Distance(officeAddress.loc, 'meter')/1000, 1) as
distanceHomeToWork,
    round( homeAddress.loc.ST_Distance(NEW ST_POINT(8.644072, 49.292910),
'meter')/1000, 1) as distFromSAP03
};
}
```

Expressions and Functions

In a CDS view definition you can use any of the functions and expressions listed in the following example:

```
View MyView9 AS SELECT FROM SampleEntity
{
  a + b  AS theSum,
  a - b  AS theDifference,
  a * b  AS theProduct,
  a / b  AS theQuotient,
  -a     AS theUnaryMinus,
  c || d AS theConcatenation
};
```

> **i Note**
>
> When expressions are used in a view element, an alias must be specified, for example, `AS theSum`.

Aggregates

In a CDS view definition, you can use the following aggregates:

- AVG
- COUNT
- MIN
- MAX
- SUM
- STDDEV
- VAR

The following example shows how to use aggregates and expressions to collect information about headcount and salary per organizational unit for all employees hired from 2011 to now.

```
VIEW MyView10 AS SELECT FROM Employee
{
  orgUnit,
  count(id)    AS headCount,
  sum(salary)  AS totalSalary,
  max(salary)  AS maxSalary
}
  WHERE joinDate > date'2011-01-01'
  GROUP BY orgUnit;
```

> **i Note**
>
> Expressions are not allowed in the `GROUP BY` clause.

Constants in Views

With SPS 11, you can use constants in the views, as illustrated in “MyView” at the end of the following example:

Sample Code

```
context MyContext {
  const MyIntConst      : Integer      = 7;
  const MyStringConst   : String(10)   = 'bright';
  const MyDecConst      : Decimal(4,2) = 3.14;
  const MyDateTimeConst : UTCDateTime  = '2015-09-30 14:33';
  entity MyEntity {
    key id : Integer;
    a : Integer;
    b : String(100);
    c : Decimal(20,10);
    d : UTCDateTime;
    your : Association[1] to YourEntity on your.a - a < MyIntConst;
  };
  entity YourEntity {
    key id : Integer;
    a : Integer;
  };
  entity HerEntity {
    key id : Integer;
    t : String(20);
  };
  view MyView as select from MyEntity
           inner join HerEntity on locate (b, :MyStringConst) > 0
  {
    a + :MyIntConst as x,
    b || ' is ' || :MyStringConst as y,
    c * sin(:MyDecConst) as z
  } where d < :MyContext.MyDateTimeConst;
};
```

When constants are used in a view definition, their name must be prefixed with the scope operator “:”. Usually names that appear in a query are resolved as alias or element names. The scope operator instructs the compiler to resolve the name outside of the query.

Sample Code

```
context NameResolution {
  const a : Integer = 4;
  const b : Integer = 5;
  const c : Integer = 6;
  entity E {
    key id : Integer;
    a : Integer;
    c : Integer;
  };
  view V as select from E {
    a    as a1,
    b,
    :a   as a2,
    E.a  as a3,
    :E,
    :E.a as a4,
    :c
  };
}
```

The following table explains how the constants used in view “V” are resolved.

Constant Declaration and Result

Constant Expression	Result	Comments
`a as a1,`	Success	“a” is resolved in the space of alias and element names, for example, element “a” of entity “E”.
`b,`	Error	There is no alias and no element with name “b” in entity “E”
`:a as a2,`	Success	Scope operator “:” instructs the compiler to search for element “a” outside of the query (finds the constant “a”).
`E.a as a3,`	Success	“E” is resolved in the space of alias and element names, so this matches element “a” of entity “Entity” .
`:E,`	Error	Error: no access to “E” via “:”
`:E.a as a4,`	Error	Error; no access to “E” (or any of its elements) via “:”
`:c`	Error	Error: there is no alias for “c”.

SELECT

In the following example of an association in a `SELECT` list, a view compiles a list of all employees; the list includes the employee's name, the capacity of the employee's office, and the color of the carpet in the office. The association follows the to-one association office from entity `Employee` to entity `Room` to collect the relevant information about the office.

```
VIEW MyView11 AS SELECT FROM Employee
{
  name.last,
  office.capacity,
  office.carpetColor
};
```

Subqueries

You can define subqueries in a CDS view, as illustrated in the following example:

! Restriction

For use in XS advanced only; subqueries are not supported in XS classic

Code Syntax

```
select from (select from F {a as x, b as y}) as Q {
  x+y as xy,
    (select from E {a} where b=Q.y) as a
} where x < all (select from E{b})
```

> **i Note**
>
> In a **correlated** subquery, elements of outer queries must always be addressed by means of a table alias.

WHERE

The following example shows how the syntax required in the `WHERE` clause used in a CDS view definition. In this example, the `WHERE` clause is used in an association to restrict the result set according to information located in the association's target. Further filtering of the result set can be defined with the `AND` modifier.

```
VIEW EmployeesInRoom_ABC_3_4 AS SELECT FROM Employee
{
  name.last
} WHERE officeId.building = 'ABC'
    AND officeId.floor    = 3
    AND officeId.number   = 4;
```

FROM

The following example shows the syntax required when using the FROM clause in a CDS view definition. This example shows an association that lists the license plates of all company cars.

```
VIEW CompanyCarLicensePlates AS SELECT FROM Employee.companyCar
{
  licensePlate
};
```

In the `FROM` clause, you can use the following elements:

- an entity or a view defined in the same CDS source file
- a native SAP HANA table or view that is available in the schema specified in the schema annotation (`@Schema` in the corresponding CDS document)

If a CDS view references a native SAP HANA table, the table and column names must be specified using their effective SAP HANA names.

```
create table foo (
  bar    : Integer,
  "gloo" : Integer
)
```

This means that if a table (`foo`) or its columns (`bar` and `"gloo"` were created **without** using quotation marks (""), the corresponding uppercase names for the table or columns must be used in the CDS document, as illustrated in the following example.

```
VIEW MyViewOnNative as SELECT FROM FOO
{
  BAR,
  gloo
};
```

GROUP BY

The following example shows the syntax required when using the `GROUP BY` clause in a CDS view definition. This example shows an association in a view that compiles a list of all offices that are less than 50% occupied.

```
VIEW V11 AS SELECT FROM Employee
{
  officeId.building,
  officeId.floor,
  officeId.roomNumber,
  office.capacity,
  count(id) as seatsTaken,
  count(id)/office.capacity as occupancyRate
} GROUP BY officeId.building,
           officeId.floor,
           officeId.roomNumber,
           office.capacity,
           office.type
  HAVING office.type = 'office' AND count(id)/capacity < 0.5;
```

HAVING

The following example shows the syntax required when using the `HAVING` clause in a CDS view definition. This example shows a view with an association that compiles a list of all offices that are less than 50% occupied.

```
VIEW V11 AS SELECT FROM Employee
{
  officeId.building,
  officeId.floor,
  officeId.roomNumber,
  office.capacity,
  count(id)          as seatsTaken,
  count(id)/office.capacity as occupancyRate
} GROUP BY officeId.building,
           officeId.floor,
           officeId.roomNumber,
           office.capacity,
           office.type
  HAVING office.type = 'office' AND count(id)/capacity < 0.5;
```

ORDER BY

The `ORDER BY` operator enables you to list results according to an expression or position, for example `salary`.

```
VIEW MyView3 AS SELECT FROM Employee
{
  orgUnit,
  salary
} ORDER BY salary DESC;
```

In the same way as with plain SQL, the `ASC` and `DESC` operators enable you to sort the list order as follows.

- `ASC`

Display the result set in **ascending** order

- `DESC`
  Display the result set in **descending** order

LIMIT/OFFSET

You can use the SQL clauses `LIMIT` and `OFFSET` in a CDS query. The `LIMIT <INTEGER> [OFFSET <INTEGER>]` operator enables you to restrict the number of output records to display to a specified "limit"; the `OFFSET <INTEGER>` specifies the number of records to skip before displaying the records according to the defined `LIMIT`.

```
VIEW MyViewV AS SELECT FROM E
  { a, b, c}
  order by a limit 10 offset 30;
```

CASE

In the same way as in plain SQL, you can use the `case` expression in a CDS view definition to introduce `IF-THEN-ELSE` conditions without the need to use procedures.

```
entity MyEntity12 {
key id : Integer;
     a : Integer;
     color : String(1);
};

VIEW MyView12 AS SELECT FROM MyEntity12 {
    id,
    CASE color     // defined in MyEntity12
        WHEN 'R' THEN 'red'
        WHEN 'G' THEN 'green'
        WHEN 'B' THEN 'blue'
        ELSE 'black'
    END AS color,
    CASE
        WHEN a < 10 then 'small'
        WHEN 10 <= a AND a < 100 THEN 'medium'
        ELSE 'large'
    END AS size
};
```

In the first example of usage of the `CASE` operator, `CASE color` shows a "switched" CASE (one table column and multiple values). The second example of CASE usage shows a "conditional" CASE with multiple arbitrary conditions, possibly referring to different table columns.

UNION

Enables multiple select statements to be combined but return only one result set. `UNION` works in the same way as the SAP HANA SQL command of the same name; it selects all unique records from all select statements

by removing duplicates found from different select statements.The signature of the result view is equal to the signature of the first `SELECT` in the union.

> **Note**
>
> View `MyView5` has elements `a`, `b`, and `c`.

```
entity E1 {
  key a : Integer;
  b : String(20);
  c : LocalDate;
};
entity E2 {
  key x : String(20);
  y : LocalDate;
  z : Integer;
};
VIEW MyView5 AS
  SELECT FROM E1 { a, b, c}
  UNION
  SELECT FROM E2 { z, x, y};
```

JOIN

You can include a JOIN clause in a CDS view definition; the following JOIN types are supported:

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

The following example shows a simple join.

> **Sample Code**

```
entity E {
  key id : Integer;
  a : Integer;
};
entity F {
  key id : Integer;
  b : Integer;
};
entity G {
  key id : Integer;
  c : Integer;
};
view V_join as select from E join (F as X full outer join G on X.id = G.id)
on E.id = c {
  a, b, c
};
```

TOP

You can use the SQL clause `TOP` in a CDS query, as illustrated in the following example:

Sample Code

```
view V_top as select from E TOP 10 { a, b, c};
```

! Restriction

It is not permitted to use `TOP` in combination with the `LIMIT` clause in a CDS query.

SELECT DISTINCT

CDS now supports the `SELECT DISTINCT` semantic, which enables you to specify that only one copy of each set of duplicate records **selected** should be returned. The position of the `DISTINCT` keyword is important; it must appear directly in front of the curly brace, as illustrated in the following example:

Sample Code

```
entity E {
  key id : Integer;
  a : Integer;
};
entity F {
  key id : Integer;
  b : Integer;
};
entity G {
  key id : Integer;
  c : Integer;
};
view V_dist as select from E distinct { a };
```

With Parameters

You can define parameters for use in a CDS view; this allows you to pass additional values to modify the results of the query at run time. Parameters must be defined in the view definition before the query block, as illustrated in the following example:

! Restriction

For use in XS advanced only; views with parameters are not supported in XS classic.

Sample Code

Parameters in a CDS View

```
context MyContext
{
  entity MyEntity1 {
     id: Integer;
     elt: String(100); };
  entity MyEntity2 {
     id: Integer;
     elt: String(100); };
  type MyUserDefinedType: type of E.elt;
  view MyParamView with parameters PAR1: Integer,
                                   PAR2: MyUserDefinedType,
                                   PAR3: type of E.elt
      as select from MyEntity {
        id,
        elt };
```

Note

Keywords are case insensitive.

Parameters in View Queries

Parameters can be used in a query at any position where an expression is allowed. A parameter is referred to inside a query by prefixing the parameter name either with the colon Scope operator ':' or the string “`$parameters`”.

→ Tip

If no matching parameter can be found, the scope operator “escapes” from the query and attempts to resolve the identifier outside the query.

Sample Code

Using Parameters in a View Query

```
view ExampleView with parameters PAR1: Integer,
                                 PAR2: UserDefinedType,
                                 PAR3: type of E.elt
       as select from SomeEntity
            left outer join SomeOtherEntity
              on SomeEntity.id < SomeOtherEntity.id + :PAR1
    {
      id + :PAR1 as idWithOffset,
      elt,
      :PAR1,
      $parameters.PAR3
    } where elt != $parameters.PAR2;
```

Invoking a View with Parameters

Parameters are passed to views as a comma-separated list in parentheses. Optional filter expressions must then follow the parameter list.

! Restriction

It is not allowed to use a query as value expression. Nor is it allowed to provide a parameter list in the `ON` condition of an association definition to a parameterized view. This is because the association definition establishes the relationship between the two entities but makes no assumptions about the run-time conditions. For the same reason, it is not allowed to specify filter conditions in those `ON` conditions.

The following example shows two entities `SourceEntity` and `TargetEntity` and a parameterized view `TargetWindowView`, which selects from `TargetEntity`. An association is established between `SourceEntity` and `TargetEntity`.

‘≡› Sample Code

```
entity SourceEntity {
   id: Integer;
   someElementOfSourceEntity: String(100);
   toTargetViaParamView: association to TargetWindowView on
      toTargetViaParamView.targetId = id;
   };
entity TargetEntity {
   targetId: Integer;
   someElementOfTargetEntity: String(100);
   };

view TargetWindowView with parameters LOWER_LIMIT: Integer
   as select from TargetEntity {
      targetId,
      someElementOfTargetEntity
   } where targetId > :LOWER_LIMIT
     and targetId <= :LOWER_LIMIT + 10;
```

It is now possible to query `SourceEntity` in a view; it is also possible to follow the association to `TargetWindowView`, for example, by providing the required parameters, as illustrated in the following example:

‘≡› Sample Code

Query a Parameterized CDS View (with Association)

```
view SourceConsumption with parameters CUSTOMER_ID: Integer
   as select from SourceEntity {
      someElementOfSourceEntity,
      toTargetViaParamView(LOWER_LIMIT:
$parameters.CUSTOMER_ID).someElementOfTargetEntity
   };
```

It is also possible to follow the association in the `FROM` clause; this provides access only to the elements of the target artifact:

‘≡› Sample Code

Follow an Association in the FROM Clause

```
view ConsumptionView with parameters CUSTOMER_ID: Integer
   as select from SourceEntity.toTargetViaParamView(LOWER_LIMIT: :CUSTOMER_ID)
   {
      id,
      someElementOfTargetEntity
```

```
    };
```

You can select directly from the view with parameters, adding a free `JOIN` expression, as illustrated in the following example:

Sample Code

Select from a Parameterized View with JOIN Expression

```
view ConsumptionView with parameters CUSTOMER_ID: Integer
   as select from TargetWindowView(LOWER_LIMIT: :CUSTOMER_ID) as TWV_ALIAS
      RIGHT OUTER JOIN ... ON TWV_ALIAS.targetId ....
{
   ...
};
```

Annotations in Parameter Definitions

Parameter definitions can be annotated in the same way as any other artifact in CDS; the annotations must be prepended to the parameter name. Multiple annotations are separated either by whitespace or new-line characters.

→ Tip

To improve readability and comprehension, it is recommended to include only one annotation assignment per line.

In the following example, the view `TargetWindowView` selects from the entity `TargetEntity`; the annotation `@positiveValuesOnly` is not checked; and the `targetId` is required for the `ON` condition in the entity `SourceEntity`.

Sample Code

Annotation Assignments to Parameter Definitions in CDS Views

```
annotation remark: String(100);

view TargetWindowView with parameters
   @remark: 'This is an arbitrary annotation'
   @positiveValuesOnly: true
   LOWER_LIMIT: Integer
as select from TargetEntity
{
  targetId,
  ....
} where targetId > :LOWER_LIMIT and targetId <= :LOWER_LIMIT + 10;
```

Associations, Filters, and Prefixes

You can define an association as a view element, for example, by defining an ad-hoc association in the `mixin` clause and then adding the association to the `SELECT` list, as illustrated in the following example:

> **! Restriction**
>
> XS classic does not support the use of ad-hoc associations in a view's `SELECT` list.

> **Sample Code**
>
> Associations as View Elements

```
entity E {
  a : Integer;
  b : Integer;
 };
entity F {
  x : Integer;
  y : Integer;
 };
view VE as select from E mixin {
  f : Association[1] to VF on f.vy = $projection.vb;
 } into {
  a as va,
  b as vb,
  f as vf
 };
view VF as select from F {
  x as vx,
  y as vy
 };
```

In the `ON` condition of this type of association in a view, it is necessary to use the pseudo-identifier `$projection` to specify that the following element name must be resolved in the `select` list of the view (“`VE`”) rather than in the entity (“`E`”) in the `FROM` clause

Filter Conditions

It is possible to apply a filter condition when resolving associations between entities; the filter is merged into the `ON`-condition of the resulting `JOIN`. The following example shows how to get a list of customers and then filter the list according to the sales orders that are currently “open” for each customer. In the example, the filter is inserted after the association `orders`; this ensures that the list displayed by the view only contains those orders that satisfy the condition `[status='open']`.

> **Sample Code**

```
view C1 as select from Customer {
  name,
  orders[status='open'].id as orderId
};
```

The following example shows how to use the prefix notation to ensure that the compiler understands that there is only one association (`orders`) to resolve but with multiple elements (`id` and `date`):

Sample Code

```
view C1 as select from Customer {
  name,
  orders[status='open'].{ id   as orderId,
                          date as orderDate
                        }
};
```

→ Tip

Filter conditions and prefixes can be nested.

The following example shows how to use the associations `orders` and `items` in a view that displays a list of customers with open sales orders for items with a price greater than 200.

Sample Code

```
view C2 as select from Customer {
  name,
  orders[status='open'].{ id   as orderId,
                          date as orderDate,
                          items[price>200].{ descr,
                                             price
                                           }
                        }
};
```

Prefix Notation

The prefix notation can also be used without filters. The following example shows how to get a list of all customers with details of their sales orders. In this example, all uses of the association `orders` are combined so that there is only one `JOIN` to the table `SalesOrder`. Similarly, both uses of the association `items` are combined, and there is only one `JOIN` to the table `Item`.

Sample Code

```
view C3 as select from Customer {
  name,
  orders.id          as orderId,
  orders.date        as orderDate,
  orders.items.descr as itemDescr,
  orders.items.price as itemPrice
};
```

The example above can be expressed more elegantly by combining the associations `orders` and `items` using the following prefix notation:

Sample Code

```
view C1 as select from Customer {
  name,
  orders.{ id   as orderId,
```

```
            date as orderDate,
            items. { descr as itemDescr,
                     price as itemPrice
                   }
          }
};
```

Type Definition

In a CDS view definition, you can explicitly specify the `type` of a select item, as illustrated in the following example:

> **! Restriction**
>
> For use in XS advanced only; assigning an explicit CDS type to an item in a `SELECT` list is not supported in XS classic.

> **‹≡› Sample Code**

```
type MyInteger : Integer;
entity E {
  a : MyInteger;
  b : MyInteger;
};
view V as select from E {
  a,
  a+b as s1,
  a+b as s2 : MyInteger
};
```

In the example of different `type` definitions, the following is true:

- `a,`
  Has type "`MyInteger`"
- `a+b as s1,`
  Has type "`Integer`" and any information about the user-defined `type` is lost
- `a+b as s2 : MyInteger`
  Has type "`MyInteger`", which is explicitly specified

> **i Note**
>
> If necessary, a `CAST` function is added to the generated view in SAP HANA; this ensures that the `select` item's type in the SAP HANA view is the SAP HANA "type" corresponding to the explicitly specified CDS type.

Spatial Functions

The following view (`SpatialView1`) displays a list of all persons selected from the entity `Person` and uses the spatial function `ST_Distance` (*) to include information such as the distance between each person's home

and business address (`distanceHomeToWork`), and the distance between their home address and the building SAP03 (`distFromSAP03`). The value for both distances is measured in kilometers, which is rounded up and displayed to one decimal point.

Sample Code

```
view SpatialView1 as select from Person {
    name,
    homeAddress.street_name   || ', ' || homeAddress.city   as home,
    officeAddress.street_name || ', ' || officeAddress.city as office,
    round( homeAddress.loc.ST_Distance(officeAddress.loc, 'meter')/1000, 1)
as distanceHomeToWork,
    round( homeAddress.loc.ST_Distance(NEW ST_POINT(8.644072, 49.292910),
'meter')/1000, 1) as distFromSAP03
};
```

Caution

(*) For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Related Information

Create a CDS View in XS Advanced [page 232]
CDS Views in XS Advanced [page 235]
Spatial Types and Functions in XS Advanced [page 253]

4.6.3 Spatial Types and Functions in XS Advanced

CDS supports the use of Geographic Information Systems (GIS) functions and element types in CDS-compliant entities and views.

Spatial data is data that describes the position, shape, and orientation of objects in a defined space; the data is represented as two-dimensional geometries in the form of points, line strings, and polygons. The following examples shows how to use the spatial function `ST_Distance` in a CDS view. The underlying spatial data used in the view is defined in a CDS entity using the type `ST_POINT`.

The following example, the CDS entity `Address` is used to store geo-spatial coordinates in element `loc` of type `ST_POINT`:

Sample Code

```
namespace samples;
@Schema: 'MYSCHEMA'
context Spatial {
    entity Person {
        key id : Integer;
        name : String(100);
        homeAddress : Association[1] to Address;
        officeAddress : Association[1] to Address;
```

```
    };
    entity Address {
        key id : Integer;
        street_number : Integer;
        street_name : String(100);
        zip : String(10);
        city : String(100);
        loc : hana.ST_POINT(4326);
    };
    view GeoView1 as select from Person {
        name,
        homeAddress.street_name || ', ' || homeAddress.city as home,
        officeAddress.street_name || ', ' || officeAddress.city as office,
        round( homeAddress.loc.ST_Distance(officeAddress.loc, 'meter')/1000,
1) as distanceHomeToWork,
        round( homeAddress.loc.ST_Distance(NEW ST_POINT(8.644072, 49.292910),
'meter')/1000, 1) as distFromSAP03
    };
};
```

The view `GeoView1` is used to display a list of all persons using the spatial function `ST_Distance` to include information such as the distance between each person's home and business address (`distanceHomeToWork`), and the distance between their home address and the building SAP03 (`distFromSAP03`). The value for both distances is measured in kilometers.

> ⚠ Caution
>
> (*) For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Related Information

Create a CDS View in XS Advanced [page 232]
CDS Views in XS Advanced [page 235]
CDS View Syntax Options in XS Advanced [page 237]

4.7 Create a CDS Extension

Define the artifacts required to extend an existing CDS model.

Prerequisites

- You have access to the Web-based development tools included with SAP Web IDE for SAP HANA
- Each CDS extension package must have the following elements:
  - The package descriptor (`.package.hdbcds`)

The package descriptor for a CDS extension has no name, only the suffix. Its contents must conform to the required syntax.

- The CDS extension descriptor (`myCDSExtension.hdbcds`)
  The extension descriptor's contents must conform to the required syntax.

Context

In this simple CRM scenario, the base application consists of a "type" named `Address` and an entity named `Customer`. In the first extension package, `banking`, we add a new "`type`" named `BankingAccount` and a new "`element`" named `account` to the entity `Customer`. In a further extension package named `onlineBanking` that depends on the package `banking` we add a new element to type `BankingAccount` and add a new element to `type Address`.

Sample Code

```
<myCDSBankingApp>
|- db/                                    # Database deployment artifacts
|   |- package.json
|   \- src/
|       |- .hdiconfig                     # HDI build plug-in configuration
|       |- .hdinamespace                  # HDI run-time name-space config
|       |- Address.hdbcds                 # Address type definition
|       |- CRM.hdbcds                     # Address details
|       |- banking/                       # CDS extension package
|       |   |- package.hdbcds             # CDS extension package
descriptor
|       |   \- BankingExtension.hdbcds    # CDS extension description
|       \- onlineBanking/                 # CDS extension package
|           |- package.hdbcds             # CDS extension package
descriptor
|           \- OnlineBankingExtension.hdbcds    # CDS extension description
|- web/
|- js/
|- xs-security.json
\- mtad.yaml
```

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following URL:

   `https://<HANA_HOST>:53075/`

   ### → Tip

   To display the URL for the SAP Web IDE for SAP HANA, open a command shell, log on to the XS advanced run time, and run the following command:

   ```
   xs app webide --urls
   ```

2. Display the application project to which you want to add a CDS document.

   In XS advanced, SAP Web IDE for SAP HANA creates an application within a context of a project. If you do not already have a project, there are a number of ways to create one, for example: by importing it, cloning it, or creating a new one from scratch.

   a. In the SAP Web IDE for SAP HANA, choose *File* > *New* > *Project from Template*.
   b. Choose the project template type.

      Currently, there is only one type of project template available, namely: *Multi-Target Application Project*. Select *Multi-Target Application Project* and choose *Next*.
   c. Type a name for the new MTA project (for example, **myCDSApp** and choose *Next* to confirm.
   d. Specify details of the new MTA project and choose *Next* to confirm.
   e. Create the new MTA project; choose *Finish*.

3. Navigate to the database module of the application for which you want to create CDS extensions.

   In SAP Web IDE for SAP HANA, database artifacts such as the ones defined in a CDS document belong in the MTA's database "module".

   > → Tip
   >
   > If you do not already have a database module, right-click the root folder of your new MTA project and, in the context menu, choose *New* > *HDB Module*. Name the new database model **db**.

4. Create the base database application using CDS.

   The base CDS application in `myCDSBankingApp/src/` must contain the following artifacts:

   > ℹ Note
   >
   > For the purposes of this tutorial, we are using the base CDS documents `Address.hdbcds` and `CRM.hdbcds`; an existing application would have different CDS documents. However, the `.hdinamespace` and `.hdiconfig` files would be present.

   - `.hdinamespace`
     The name-space definition to use when deploying the database application
   - `.hdiconfig`
     The list of plug-ins to use to create catalog objects when deploying the database application
   - `Address.hdbcds`
     A CDS document containing the definition of the CDS data "`type`" `Address`
   - `CRM.hdbcds`
     A CDS document named `CDM.hdbcds` which contains the definition of the CDS "`entity`" `Customer`

   a. Define the name space that applies to this CDS application.

      The name space to use for the deployment of the CDS application is defined in the configuration file `.hdinamespace`; in this case, it should look like the following example:

      > ℹ Note
      >
      > The "`append`" value ensures that the name-space rule applies to *all* subfolders in the CDS application structure.

Sample Code

```
.hdinamespace
```

```
{
  "name": "",
  "subfolder": "append"
}
```

b. Create the CDS data type `Address.hdbcds`.

Navigate to the application's database module `db/`, right-click the folder `db/src/` and choose *New* > *CDS Artifact* in the context menu.

The CDS data type definition should look like the following example:

Sample Code

```
type Address {
  zipCode : String(5);
  city : String(40);
  street : String(40);
  nr : String(10);
};
```

c. Create the CDS document `CRM.hdbcds`.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/` and choose *New* > *CDS Artifact* in the context menu.

The CDS definition for the `Customer` entity (table) should look like the following example:

Sample Code

```
using Address;
context CRM {
  entity Customer {
    name : String(40);
    address : Address;
  };
};
```

5. Create a CDS extension called “`banking`”.

The CDS extension `banking` must contain the following artifacts:

- `.package.hdbcds`
  A CDS document containing the definition of the CDS extension package `banking/`
- `BankingExtension.hdbcds`
  A CDS document containing the definition of the CDS extension `BankingExtension`

a. Create a new folder for the CDS extension `banking/`.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/` and choose *New* > *Folder* in the context menu. Name the new folder “`banking`”.

b. Create the CDS extension package descriptor `.package.hdbcds`.

> **Note**
>
> The leading dot (.) in the extension-package file name is mandatory.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/banking` and choose *New* > *CDS Artifact* in the context menu. Name the new artifact `.package.hdbcds`.

The CDS extension **package** definition should look like the following example:

> **Sample Code**

```
package banking;
```

c. Create the CDS extension descriptor `BankingExtension.hdbcds`.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/banking` and choose *New* > *CDS Artifact* in the context menu.

The CDS extension definition `BankingExtension.hdbcds` should look like the following example:

> **Sample Code**

```
namespace banking;
in package banking;
using CRM;
using CRM.Customer;
extend context CRM with {
  type BankingAccount {
    BIC : String(8);
    IBAN : String(120);
  };
};
extend entity Customer with {
  account: CRM.BankingAccount;
};
```

6. Create a CDS extension called "`onlineBanking`".

The CDS extension `onlineBanking` must contain the following artifacts:

- `.package.hdbcds`
  A CDS document containing the description of the CDS extension package `onlineBanking/`
- `BankingExtension.hdbcds`
  A CDS document containing the description of the CDS extension `OnlineBankingExtension`

a. Create a new folder for the CDS extension `onlineBanking/`.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/` and choose *New* > *Folder* in the context menu. Name the new folder "`onlineBanking`".

b. Create the CDS extension package descriptor `.package.hdbcds`.

> **Note**
>
> The leading dot (.) in the package file name is mandatory.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/onlineBanking` and choose *New* > *CDS Artifact* in the context menu. Name the new artifact `.package.hdbcds`.

The CDS extension **package** descriptor should look like the following example:

Sample Code

```
package onlineBanking depends on banking;
```

c. Create the CDS extension descriptor `BankingExtension.hdbcds`.

Navigate to the `src/` folder in your application's database module `db/`, right-click the folder `db/src/onlineBanking` and choose *New* > *CDS Artifact* in the context menu. Name the new CDS artifact `BankingExtension.hdbcds`

The CDS extension definition `BankingExtension.hdbcds` should look like the following example:

Sample Code

```
namespace onlineBanking;
in package onlineBanking;
using Address;
using CRM.BankingAccount;
extend type Address with {
  email : String(60);
};
extend type BankingAccount with {
  PIN : String(5);
};
```

7. Build the CDS application's database module.

Building a database module activates the data model and creates corresponding object in the database catalog for each artifact defined in the CDS document. In this case, the build creates all the CDS artifacts for the base CDS application as well as the artifacts defined in the two extension packages.

In SAP Web IDE for SAP H ANA, right-click the CDS application's database module (`<myCDSapp>/db`) and choose *Build* > *Build* in the context-sensitive menu.

If the builder displays the message `(Builder) Build of /<myCDSapp>/db` completed in the SAP Web IDE for SAP HANA console, the data-model was successfully activated in a SAP HANA database container, and can now be used to store and retrieve data.

Related Information

The CDS Extension Descriptor [page 260]
The CDS Extension Descriptor Syntax [page 263]
The CDS Extension Package Descriptor [page 267]
The CDS Extension Package Descriptor Syntax [page 268]

4.7.1 The CDS Extension Descriptor

Defines in a separate file the properties required to modify an existing CDS artifact definition.

The CDS extension mechanism allows you to add properties to existing artifact definitions without modifying the original source files. In this way, you can split the definition of an artifact across multiple files each of which can have a different life cycle and code owner. For example, a customer can add a new element to an existing entity definition by the following statement:

‹≡› Sample Code

CDS Artifact Extension Syntax

```
extend EntityE with {
  newElement: Integer;
}
```

In the example above, the code illustrated shows how to define a new **element** inside an existing entity (`EntityE`) artifact.

i Note

The `extend` statement changes an existing artifact; it does not define any additional artifact.

It is essential to ensure that additional element definitions specified in custom extensions do not break the existing definitions of the base application. This is achieved by adapting the name-search rules and by additional checks for the `extend` statements. For the definition of these rules and checks, it is necessary to define the relationship between an `extend` statement and the artifact definitions, as well as the relationship between an `extend` statement and any additional `extend` statements.

Organization of Extensions

When you extend an SAP application, you typically add new elements to entities or views; these additional elements usually work together and can, themselves, require additional artifacts, for example, “types” used as element “types”. To facilitate the process, we define an extension package (or package for short), which is a set of extend statements, normal artifact definitions (for example, “types” which are used in an `extend` declaration), and extension relationships (also known as “dependencies”). Each CDS source file belongs to exactly one package; all the definitions in this file contribute to that one (single) package. However, a “package” typically contains contributions from multiple CDS source files.

→ Tip

It is also possible to use a package to define a clear structure for an application, even if no extensions are involved.

Package Hierarchies

The extension mechanism can be used by developers as well as SAP industry solutions, partners, and customers. A productive system is likely to have more than one package; some packages might be independent from each other; some packages might depend on other packages. With such a model, we get an acyclic directed graph, with the base application and the extension packages as nodes and the dependencies as edges. This induces a partial order on the packages with the base application as lowest package (for simplicity we also call the base application a package). There is not necessarily a single top package (here: the final customer extension).

It is essential to ensure that which package is semantically self-contained and self-explanatory; avoid defining "micro" packages which can be technically applied individually but have no independent business value.

> ! Restriction
>
> Cyclic dependencies between extension packages are not allowed.

Package Definition

It is necessary to specify which `extend` statements and normal artifact definitions belong to which package and, in addition, on which other packages a package depends. A package is considered to be a normal CDS artifact; it has a name, and a corresponding definition, and its use can be found in the CDS Catalog. An extension package is defined by a special CDS source file with the file suffix `.package.hdbcds`.

> i Note
>
> The full stop (.) **before** the extension-package file name is mandatory.

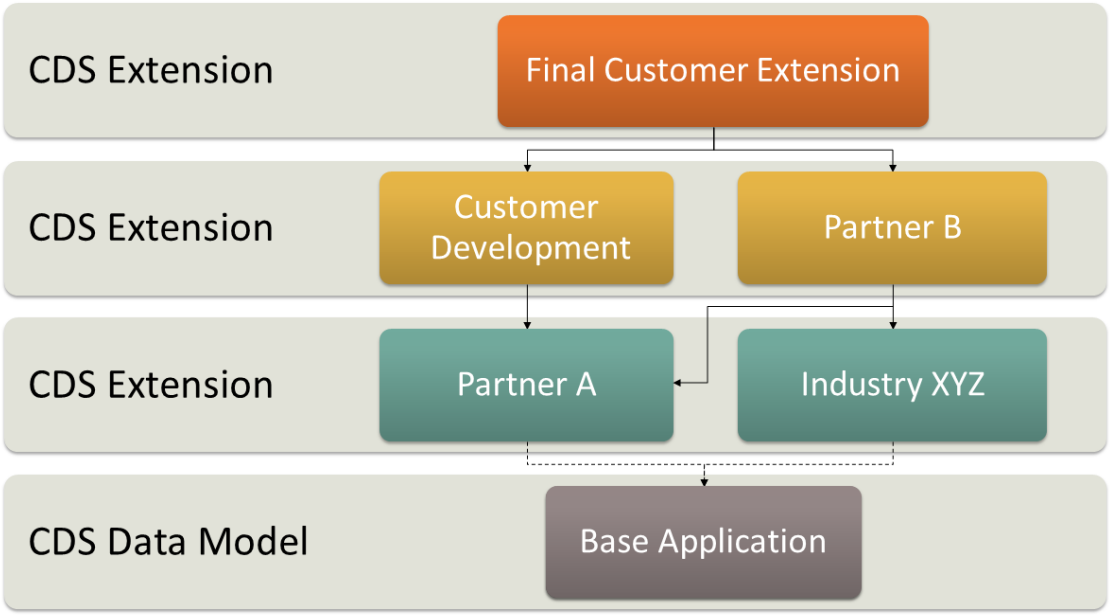
The following simple code example illustrates the syntax required for defining a CDS extension package and its dependencies:

> ‹≡› Sample Code
>
> CDS Extension Package Syntax

```
source = packageDefinition
packageDefinition = "package" packageName (
  "depends" "on" packageName ( "," packageName )* )? ";"
packageName = identifier ( "." identifier )*
```

The name of the package defined in the file must be identical to the name space that is applicable for the file (as specified in the relevant HDI container-configuration file (`.hdinamespace`).

> → Tip
>
> The base package is not explicitly defined; it contains all CDS sources that are not explicitly assigned to a package.



To define a package hierarchy according to the diagram above, the following package definition files need to be provided (the names are just an example and do not confirm to any recommended naming convention):

Package Hierarchy and Definition Files

Package Directory (Namespace)	Content of Package Definition File (.package.hdbcds)
`myapp.partners.A`	`package myapp.partners.A;`
`myapp.industries.XYZ`	`package myapp.industries.XYZ;`
`myapp.customDev`	`package myapp.customDev depends on myapp.partners.A;`
`myapp.partners.B`	`package myapp.partners.B depends on myapp.partners.A, myapp.industries.XYZ;`
`myapp.customer`	`package myapp.customer depends on myapp.customDev, myapp.partners.B;`

Package and Source-File Assignment

To ensure that all the definitions in the CDS source, both normal artifact definitions and extend statements, belong to the respective package, you must assign a CDS source file to an extension package. To assign a CDS

source to an extension package, add the statement `in package` to the beginning of the CDS source file, as illustrated in the following example:

Sample Code

CDS Extension Package Assignment

```
in package <packageName>;
```

Note

The `in package` statement must be placed at the beginning of the CDS document, for example, before (or after) the name-space declaration, if present, but in all cases **before** all other statements in the CDS document.

Related Information

Create a CDS Extension [page 254]
The CDS Extension Descriptor Syntax [page 263]
The CDS Extension Package Descriptor [page 267]

4.7.2 The CDS Extension Descriptor Syntax

The syntax required to define a CDS extension artifact.

The CDS extension mechanism allows you to add properties to existing artifact definitions without modifying the original source files. The content extension of the CDS extension descriptor must conform to the following format:

Note

The following example of a CDS document for XS advanced is incomplete; it is intended for illustration purposes only.

Sample Code

CDS Artifact Extension Syntax

```
namespace banking;
in package banking;
using CRM;
using CRM.Customer;
extend context CRM with {
  type BankingAccount {
    BIC : String(8);
    IBAN : String(120);
  };
};
extend entity Customer with {
  account: CRM.BankingAccount;
```

```
};
extend type Address with {
  email : String(60);
};
extend view MyView with {
  a,
  b as newB,
  ass[y=2].x as elemViaAssoc,
  sum(t) as aNewAggregate
};
```

! Restriction

If a CDS artifact is defined in a package, it cannot be extended in the same package. In addition, the same CDS artifact cannot be extended twice in the **same** package.

in package

Use the keywords "`in package`" to assign a CDS source document to a CDS extension package. This ensures that all the definitions in the CDS source files, both normal artifact definitions and extend statements, belong to the named package.

Sample Code

CDS Artifact Extension Syntax

```
namespace banking;
in package banking;
```

i Note

The `in package` keyword must be inserted at the beginning of the CDS document: before or after the namespace declaration, if present, but always before all other statements.

using

All artifacts that are to be extended in a CDS source document must be made available with the `using` declaration.

Sample Code

```
namespace banking;
in package banking;
using CRM;
using CRM.Customer;
```

Extending Elements in CDS Entities or Structured Types

You can use a CDS extension to add new elements to an existing CDS entity, as illustrated in the following example:

Sample Code

Extend a CDS Entity with New Elements

```
extend entity MyEntity with {
  name1 : type1;
  name2 : type2;
  ...
};
```

The SAP HANA table generated for the CDS entity contains all specified extension elements. New elements can also be added to an existing `structured type`. It does not matter whether the original type is defined by means of the keyword "`type`" or "`table type`". The following example shows how to extend a table type:

Sample Code

Extend a CDS Structured Type

```
extend type MyType with {
  name1 : type1;
  name2 : type2;
  ...
};
```

Note

For a "`table type`", the generated SAP HANA table type contains all extension elements. This kind of extension does not work for scalar types.

Extending SELECT Items in a CDS View

You can use a CDS extension to add new `SELECT` items to an existing CDS view, as illustrated in the following example:

Sample Code

Extend a CDS View

```
extend view MyView with {
  a,
  b as newB,
  ass[y=2].x as elemViaAssoc,
  sum(t) as aNewAggregate
};
```

The SAP HANA view generated for the CDS view contains all extension items added to the `SELECT` clause.

! Restriction

It is not possible to extend any part of a view definition other than the `SELECT` clause.

Adding Artifacts to a CDS Context

You can use a CDS extension to add new artifacts (for example, tables, types, or views) to an existing CDS context, as illustrated in the following example:

‹≡› Sample Code

Extend CDS Context with New Artifacts

```
extend context MyContext with {
  type T1 : Integer;
  type S1 {
    a : INteger;
    b : String;
  };
  entity E1 {
    elem1 : Integer;
    elem2 : S1;
  };
  view V1 as select from E1 {, elem1, elem2 };
};
```

Extending CDS Annotations

You can use a CDS extension to add new `@annotations` to an existing CDS artifact or element; the syntax you use to add the annotations differs according to whether you are adding them to a CDS artifact or an element, as illustrated in the following examples:

‹≡› Sample Code

Extending CDS Artifacts with Annotations

```
@MyIntegerAnnotation : 44
extend entity MyEntity;
```

‹≡› Sample Code

Extending CDS Elements with Annotations

```
extend entity MyEntity with {
  @MyIntegerAnnotation : 45
  extend baseElement;
};
```

Sample Code

Extending CDS Entities with Annotated Elements

```
extend entity MyEntity with {
  @MyIntegerAnnotation : 45
  newElement : String(88);
};
```

→ Tip

Adding associations to elements of structured types and to `SELECT` items in views works in the same way.

Extending a CDS Entity's Technical Configuration

You can use a CDS extension to add new elements to the `technical configuration` section of an existing CDS entity, as illustrated in the following example:

Sample Code

Extending the CDS Entity's Technical Configuration

```
extend entity MyEntity with technical configuration {
  partition by hash (baseElement) partitions 2;
};
```

→ Tip

You can use the same method to extend a CDS entity with additional indexes.

Related Information

The CDS Extension Descriptor [page 260]
Create a CDS Extension [page 254]
The CDS Extension Package Descriptor [page 267]
The CDS Extension Package Descriptor Syntax [page 268]

4.7.3 The CDS Extension Package Descriptor

In the context of a CDS extension scenario, it is necessary to specify which `extend` statements and normal CDS artifact definitions belong to which package and, in addition, on which other packages a CDS extension package depends.

A CDS extension package is a normal CDS artifact; it has a name, and both its definition and its use can be found in the CDS Catalog. A package is defined by a special CDS source file named `.package.hdbcds`. The syntax for defining a package and its dependencies is illustrated in the following example:

Sample Code

```
package <onlineBanking> depends on <banking>;
```

Related Information

The CDS Extension Package Descriptor Syntax [page 268]
Create a CDS Extension [page 254]

4.7.4 The CDS Extension Package Descriptor Syntax

Required syntax for the CDS extension descriptor.

A CDS extension package is defined in a CDS extension package descriptor, which is a special CDS source file named `.package.hdbcds`, as illustrated in the following example:

Note

The leading dot (.) in the file name for the CDS extension package descriptor is mandatory.

Sample Code

Extension Package Descriptor (`.package.hdbcds`)

```
package <onlineBanking> depends on <banking>;
```

In this example, the package `<onlineBanking>` depends on another CDS extension package `<banking>`, which contains extensions for a CDS base application `<myCDSapp>`.

The syntax for defining a package and its dependencies is illustrated in the following example:

Sample Code

CDS Extension **Package** Descriptor Syntax

```
source = packageDefinition
packageDefinition = "package" packageName (
    "depends" "on" packageName ( "," packageName )* )? ";"
packageName = identifier ( "." identifier )*
```

The name of the package defined in the file must be identical to the name space that is applicable for the file (as specified in the application's corresponding `.hdinamespace` file).

> **i** Note
>
> It is not necessary to explicitly define the base package for the CDS application; it contains all those CDS sources that are not explicitly assigned to a package.

packageDefinition

The syntax for defining a dependency between CDS extension packages is illustrated in the following example:

> Sample Code
>
> One-to-One Package Definition (`.package.hdbcds`)

```
package <onlineBanking> depends on <banking>;
```

A package can depend on multiple packages, as illustrated in the following example:

> Sample Code
>
> One-to-Many Package Definition (`.package.hdbcds`)

```
package <onlineBanking> depends on (<banking>, <banking1>, <[...]>);
```

The following example illustrates the syntax required in a package-dependency statement specified in the CDS extension package descriptor (`.package.hdbcds`)

> Sample Code
>
> CDS Extension **Package** Descriptor Syntax

```
source = packageDefinition
packageDefinition = "package" packageName (
    "depends" "on" packageName ( "," packageName )* )? ";"

packageName = identifier ( "." identifier )*
```

packageName

The syntax required when specifying the name of a CDS extension package in a package-dependency definition is illustrated in the following example:

> Sample Code
>
> One-to-One Package Definition (`.package.hdbcds`)

```
package <onlineBanking> depends on <banking>;
```

A package name can include a full stop (.), for example, to express a full path in a name space, as illustrated in the following example:

Sample Code

One-to-Many Package Definition (`.package.hdbcds`)

```
package <src.onlineBanking> depends on (<src.banking>);
```

The following example illustrates the syntax required when defining the name of a package in the CDS extension package descriptor (`.package.hdbcds`)

Sample Code

CDS Extension **Package** Descriptor Syntax

```
source = packageDefinition
packageDefinition = "package" packageName (
    "depends" "on" packageName ( "," packageName )* )? ";"
packageName = identifier ( "." identifier )*
```

Related Information

Create a CDS Extension [page 254]
The CDS Extension Descriptor [page 260]
The CDS Extension Descriptor Syntax [page 263]
The CDS Extension Package Descriptor [page 267]

4.8 Create a CDS Role (XS Advanced)

Set instance-based authorizations for accessing data in the SAP HANA database.

Context

CDS access-policy documents contain a set of CDS role definitions coded in the Data Control Language (DCL). You can use CDS roles to create instance-based authorizations. A role can contain CDS “aspects”, and it is also possible to use `grant` statements on CDS views and CDS roles and include the statements in the definition of the CDS roles. The database roles generated from the CDS role definition determine which data sets a user is authorized to access.

Caution

The code examples included in this document for XS advanced are sometimes syntactically incomplete; as a general rule, code examples are intended for illustration purposes only.

Example

If you provide authorization to access sales orders, you can add conditions that filter the view users have on the sales orders, for example, as follows:

- View the sales orders of all countries.
- View only the sales orders of one specific country, for example: the USA. The CDS role in the CDS access policy document contains a fixed value for “USA”.
- View only the sales orders of one specific country. The CDS role in the CDS access policy document contains a CDS aspect associated with the country value of the CDS entity called `address`.
- View only the sales order of his or her country. Here, external user information in XS advanced provides the country attribute that is used in the CDS DCL definition.

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available by default at the following default URL:

   `https://<HANA_HOST>:53075/`

   > ### → Tip
   >
   > If the SAP Web IDE for SAP HANA is running on a non-default port, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > ```
   > xs app webide --urls
   > ```

2. Open the application project to which you want to add your CDS role.
3. Open your multi-target application project, or if required, create a new project.
4. Open the CDS access policy document.

   Define the CDS roles in your CDS access policy. Save the DCL definition of your CDS roles to your local project workspace.

   > ### Sample Code
   >
   > ```
   > namespace com.sap.dcl.example;
   > using com.sap.dcl.example::<DDL_file_name> as ddl;
   >
   > AccessPolicy <DCL_file_name> {
   >   role salesOrderCountryAll {
   >     // grant all countries
   >     grant select on ddl.salesOrderView;
   >   };
   >   aspect aspCountry as
   >     select from ddl.address { country }
   >       where $user in toEmployee.loginName;
   >
   >   role salesOrderCountryUsa {
   >     // grant of a static value
   >     grant select on ddl.salesOrderView
   >         where customerCountry = 'USA';
   >   };
   >   role salesOrderCountryOwnAndUsa {
   > ```

```
    // grant based on an aspect
    grant select on ddl.salesOrderView
        where customerCountry = aspect :aspCountry;
    // grant based on an external attribute
    grant select on ddl.salesOrderView
        where customerCountry in $env.user.country;

    // grant of another role
    grant salesOrderCountryUsa;
  };
};
```

5. Check the syntax of CDS sources.

   Check the syntax of the CDS roles in the same way as you would check the syntax of CDS DDL sources.

6. Activate your CDS access policy document in the same way as you would activate DDL sources.

   After the activation of the CDS access policy document with the CDS roles, the DCL artifacts you defined are added to the SAP HANA database catalog.

   - After activation, the generated database roles are located in the *Security* section of your SAP HANA database. In this example, the naming convention of the roles is as follows:
     `<project_path>.<project>::<DCL_source_file>.<role_name>`
     `com.sap.dcl.example::dcl.salesOrderCountryAll`
     `com.sap.dcl.example::dcl.salesOrderCountryUsa`
     `com.sap.dcl.example::dcl.salesOrderCountryOwnAndUsa`

Results

The database roles are also available in the SAP HANA Web IDE for SAP HANA.

> **i** Note
>
> Roles are assigned to the technical database user of the XS advanced application. To assign roles to users, you need the `default_access_role` for the HDI container. For information, see *Related Links*).

Related Information

Create a CDS Document (XS Advanced) [page 144]
The Default Access Role for HDI Containers

4.8.1 CDS Role Syntax Options

The options available for modeling instance-based authorizations in CDS roles.

Overview

It is possible to combine `grant` statements in a role definition or use `grant` statements of multiple types. The following list shows the different ways in which `grant` statements can be used to define a CDS role:

- Using a grant on a CDS view with static values
- Using a grant on CDS aspects that associate a dynamic element of a CDS view (in a CDS DDL document)
- Using a grant on external attributes
- Using a grant of another role

CDS aspects can also be included in CDS role definitions and can be referenced in grants by using the following syntax:

```
aspect :<role_name>.<aspect_name>
```

The database roles generated from the CDS role definition determine which data set a user is authorized to access. If multiple roles are assigned to a user, the specified user has the authorizations defined in all of the assigned roles.

> ⚠ Caution
>
> The code examples included in this document for XS advanced are sometimes syntactically incomplete; as a general rule, code examples are intended for illustration purposes only.

CDS Role Using Grant with Static Values

> Sample Code

```
grant select on ddl.salesOrderView
    where customerCountry = 'USA';
```

In this CDS role, you use a `where` clause to directly refer to the `customerCountry` element defined in the CDS view called `salesOrderView`. The grant only allows a view on sales orders with the `customerCountry` value `USA`.

CDS Role Using Grant with CDS Aspects

Sample Code

```
grant select on ddl.salesOrderView
    where customerCountry = aspect :aspCountry;
```

In this CDS role, you use the previously defined CDS aspect `aspCountry`. It uses the country element of the address information that comes with the CDS entity called `address`. Since users have different addresses, this data is variable and dynamic. The `customerCountry` element defined in the CDS view called `salesOrderView` is associated with the CDS aspect `aspCountry`. This CDS role definition allows users to view the sales orders of the `customerCountry` value provided by their address, that is; the sales orders of their own country.

CDS Role Using Grant with External Attributes

Sample Code

```
grant select on ddl.salesOrderView
    where customerCountry in $env.user.country;
```

In this CDS role, you directly use external attributes that come with the user data of the XS advanced token for the database. The `customerCountry` element defined in the CDS view called `salesOrderView` is associated with the country attribute of the user's login data. Access is enabled to the external attribute using `$env.user.<external_attribute>`. This grant allows users to view the sales orders of the `customerCountry` value provided by the XS advanced token.

Note

It is also possible to use XS advanced attributes that have multiple values.

CDS Role Using Grant of Another Role

Sample Code

```
grant salesOrderCountryUsa;
```

This grant includes the already defined CDS role `salesOrderCountryUsa`. Here, the CDS role `salesOrderCountryUsa` is already defined, for example, in the same CDS access-policy document. It is also possible to define CDS roles in separate CDS DCL files. This grant with an included CDS role allows users to also view the sales orders of the USA.

> **i Note**
>
> You can also use a `grant` of native SAP HANA roles, for example, defined in design-time artifacts with the file extension `.hdbrole`. However, you must introduce the native SAP HANA role by including a `using` declaration first. Only then is it possible to integrate the native role using the `grant` statement.

Related Information

The Application Security Descriptor

4.9 Create a CDS Access-Policy Document (XS Advanced)

As an XS advanced developer, you want to set instance-based authorizations for accessing data in the SAP HANA database.

Context

CDS access-policy documents are coded in the Data Control Language (DCL). CDS access policies have the same file extension as CDS documents. In a CDS access policy document, you can create CDS roles and CDS aspects for instance-based authorizations. It is also possible to use `grant` statements on CDS views and CDS roles and include the statements in the definition of the CDS roles. The `grant` statements determine which data set a user is authorized to access.

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following default URL:

   `https://<HANA_HOST>:53075/`

   > **→ Tip**
   >
   > If the SAP Web IDE for SAP HANA is running on a non-default port, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > ```
   > xs app webide --urls
   > ```

2. Open the application project to which you want to add your CDS access policy.
3. Open your multi-target application project, or if required, create a new project.

4. Define a new CDS access-policy document.

   A CDS access-policy document is a file with a CDS file extension (`.hdbcds`). Enter a file name, for example `MyAccessPolicy` and choose *Finish*.

   Define the CDS role in your CDS access policy. Save the CDS access policy file to your local project workspace.

   > ⚠ Caution
   >
   > The code examples included in this document for XS advanced are sometimes syntactically incomplete; as a general rule, code examples are intended for illustration purposes only.

   > Sample Code

   ```
   namespace com.sap.dcl.example;
   using com.sap.dcl.example::<DDL_file_name> as ddl;

   AccessPolicy MyAccessPolicy {

     role salesOrderCountryUsa {
       grant select on ddl.salesOrderView
           where customerCountry = 'USA';
     };
   }
   ```

5. Check the syntax of CDS sources.

   Check the syntax of CDS access policies in the same way as you would check the syntax of CDS DDL sources.

6. Activate your CDS access-policy document in the same way as you would activate DDL sources.

   After the activation of the CDS DCL source files (CDS access-policy document and source file with DDL code), the DCL artifacts you defined previously (for example, roles) are added to the SAP HANA database catalog.

   - You can find the generated database role in the *Security* section of your SAP HANA database. In this example, the naming convention of the roles is as follows:
     `<project_path.project>::<DCL_source_file>.<role_name>`
     `com.sap.dcl.example::dcl.salesOrderCounryUsa`

Results

The database roles are also available in the SAP HANA Web IDE for SAP HANA.

> i Note
>
> This kind of database roles need to be assigned to the technical database user of the XS advanced application. To assign database roles to users, the XS advanced developer needs the HDI container's `default_access_role`. For more information about the default access role, see *Related Links*.

Related Information

Create a CDS Document (XS Advanced) [page 144]
The Default Access Role for HDI Containers
Configure a Database Connection

4.9.1 CDS Access Policies in XS Advanced

As a developer in XS advanced, you can set instance-based authorizations for accessing data in the SAP HANA database. You define access policies using the CDS data control language (DCL). Write CDS access policy documents using DCL code where you define CDS roles. Instance-based authorizations are based on CDS views.

Prerequisites

An XS advanced developer must have the standard developer authorization profile to assign roles to SAP HANA users.

> ⚠ Caution
>
> The code examples included in this document for XS advanced are sometimes syntactically incomplete; as a general rule, code examples are intended for illustration purposes only.

CDS Documents for Access Policies Defined with Data Control Language

If you want to create access policies using DCL, you must define the policies in CDS documents. CDS documents are CDS source files with the suffix `.hdbcds` which are located in the `db` folder of the HDI container. You use DCL code to define access-control logic for Core Data Services (CDS) views from the SAP HANA database. Definitions in DCL code enable you to filter access to data in the database based on static values, aspects, and based on external attributes.

> ⚠ Caution
>
> If you do **not** create and deploy CDS access policies for the CDS view, any user who can access the CDS view has access to all data returned by the CDS view.

CDS documents with DCL code or DDL code only differ in their top level element:

- DCL documents have `AccessPolicy` as top level element (whereas CDS documents with DDL code usually have `Context` as top level element).
- You can define CDS aspects and CDS roles within an `AccessPolicy` keyword section, as illustrated in the examples below.

When XS advanced developers compile an access-policy definition using a CDS document with DCL code, they generate corresponding database catalog artifacts, for example, roles.

Sample Code

```
namespace com.sap.dcl.example;
using com.sap.dcl.example::<DDL_file_name> as ddl;

AccessPolicy <DCL_file_name> {
  role salesOrderCountryUsa {
    grant select on ddl.salesOrderView
        where customerCountry = 'USA';
  };
}
```

In the CDS source file, the XS advanced developer must explicitly specify the structured privilege check for a CDS view, as illustrated in the following example.

Restriction

In XS advanced, the `@schema` annotation cannot be used in either CDS documents with DCL code or CDS documents with DDL code.

Source Code

```
entity salesOrder {
    key id                : Integer;
    toCustomer            : Association[0..1] to businessPartner;
    currencyCode          : String(5);
    grossAmount           : Integer;
    document              : LargeBinary;
};

  view salesOrderView as select from salesOrder
  {
    key id as id,
    toCustomer.toAddress.country as customerCountry,
    currencyCode,
    grossAmount,
    document
  } with structured privilege check;
```

Tip

In XS advanced, it is recommended to use `key` fields in CDS views when creating instance-based authorization checks.

The keys in CDS views should be defined in a similar way to the key definition for DDL entities, for the following reasons:

- Enhanced system performance
- The field types `LargeBinary` and `LargeString` must only be used in CDS views with key fields so that instance-based authorizations are possible.

Note

Only CDS **views** can be protected by a `grant` statement; CDS entities are not supported. You can use DCL for calculation views, but it is mandatory to include the calculation view in the CDS access policy, for

example, by including a `using` statement at the start of the document. Only calculation views generated outside of CDS can be protected by a `grant` statement. A definition of calculation views is not supported by DDL.

Related Information

Create a CDS Access-Policy Document (XS Advanced) [page 275]
Create a CDS Aspect (XS Advanced) [page 279]
Create a CDS Role (XS Advanced) [page 270]

4.10 Create a CDS Aspect (XS Advanced)

Use CDS aspects to create a CDS role that references external dynamic criteria.

Context

CDS aspects associate an attribute with permitted values of a user. The values are taken, for example, from a CDS entity or a CDS view, which is defined in a CDS DDL document. A `grant` statement in a role can use one or more “aspects” in a `where` clause.

> ⚠ Caution
>
> The code examples included in this document for XS advanced are sometimes syntactically incomplete; as a general rule, code examples are intended for illustration purposes only.

Procedure

1. Start the SAP HANA Web IDE for SAP HANA.

   The SAP Web IDE for SAP HANA is available at the following default URL:

   `https://<HANA_HOST>:53075/`

   > → Tip
   >
   > If the SAP Web IDE for SAP HANA is running on a non-default port, open a command shell, log on to the XS advanced run time, and run the following command:
   >
   > ```
   > xs app webide --urls
   > ```

2. Open the application project to which you want to add your CDS aspect.
3. Open your multi-target application project, or if required, create a new project.
4. Open the CDS access-policy document.

   Define the CDS aspects in your CDS access-policy document. Save the DCL definition of your CDS aspects to your local project workspace.

   In this example, you want to enable users to access only the sales orders of their own country. For this, you need the country information so that you can use it in the respective role. You define a CDS aspect that associates `country` from the `address` CDS entity with the corresponding employee's login name. This CDS aspect is then used by the corresponding CDS role.

   Sample Code

   ```
   namespace com.sap.dcl.example;
   using com.sap.dcl.example::<DDL_file_name> as ddl;

   AccessPolicy <DCL_file_name> {

     aspect aspCountry as
       select from ddl.address { country }
         where $user in toEmployee.loginName;
     role salesOrderCountryOwn {
       // grant based on an aspect
       grant select on ddl.salesOrderView
           where customerCountry = aspect :aspCountry;
     };
   }
   ```

5. Check the syntax of CDS sources.

   Check the syntax of CDS access policies in the same way as you would check the syntax of CDS DDL sources.
6. Activate your CDS access policy document.

Related Information

Create a CDS Document (XS Advanced) [page 144]
Create a CDS Role (XS Advanced) [page 270]

5 Using the Graphical Editors to Create CDS Artifacts

Create CDS design-time artifacts with the graphical CDS editor.

SAP Web IDE for SAP HANA includes dedicated editors that you can use to define data-persistence objects in CDS documents using the DDL-compliant Core Data Services syntax for SAP HANA. If you right-click a file with the `.hdbcds` extension in the *Project Explorer* view of your application project, you can choose *Open With* > *Graphical Editor* to view a graphical representation of the contents of a CDS source file, with the option to edit the source code as text with the syntax elements highlighted for easier visual scanning.

CDS Graphical Editor

The CDS graphical editor provides graphical modeling tools that help you to design and create database models using standard CDS artifacts with minimal or no coding at all. You can use the CDS graphical editor to create CDS artifacts such as entities, contexts, associations, structured types, and so on.

The built-in tools provided with the CDS Graphical Editor enable you to perform the following operations:

- Create CDS files (with the extension `.hdbcds`) using a file-creation wizard.
- Create standard CDS artifacts, for example: entities, contexts, associations (to internal and external entities), structured types, scalar types, ...
- Define technical configuration properties for entities, for example: indexes, partitions, and table groupings.
- Generate the relevant CDS source code in the text editor for the corresponding database model.
- Open in the CDS graphical editor data models that were created using the CDS text editor.

> → Tip
>
> The built-in tools included with the CDS Graphical Editor are context-sensitive; right-click an element displayed in the CDS Graphical editor to display the tool options that are available.

5.1 The Graphical CDS Editor in SAP Web IDE

Use SAP Web IDE to maintain CDS artifacts in SAP HANA.

SAP Web IDE for SAP HANA includes dedicated editors that you can use to define data-persistence objects in CDS documents using the DDL-compliant Core Data Services syntax for SAP HANA. If you right-click a file with the `.hdbcds` extension in the *Project Explorer* view of your application project, you can choose *Open With* > *Graphical Editor* to view a graphical representation of the contents of a CDS source file, with the option to edit the source code as text with the syntax elements highlighted for easier visual scanning.

Related Information

The Graphical CDS Editor in SAP Business Application Studio [page 308]

5.1.1 Getting Started with the CDS Graphical Editor

SAP Web IDE offers data modelers the graphical modeling capabilities that they require to model artifacts. These artifacts define the data persistence objects using the Core Data Services (CDS).

The CDS graphical editor in the SAP Web IDE helps model data persistence objects in CDS artifacts using the DDL-compliant CDS syntax. The tool recognizes the `.hdbcds` file extension required for CDS object definitions and calls the appropriate repository plug-in to parse the CDS artifact. If you right-click a file with the `.hdbcds` extension in the *Workspace* view, SAP Web IDE provides the following choice of editors in the context-sensitive menu.

- Text Editor
  View and edit DDL source code in a CDS document as text with the syntax elements highlighted for easier visual scanning.
- Graphical Editor
  View graphical representation of CDS artifacts, and also model CDS artifacts with graphical modeling tools.

Related Information

Create an Artifact with the CDS Graphical Editor [page 282]
Create an Association with the CDS Graphical Editor [page 288]
Create a User-Defined Structure Type with the CDS Graphical Editor [page 293]
Create an Entity with the CDS Graphical Editor [page 285]
Create a User-Defined Scalar Type with the CDS Graphical Editor [page 294]
Create a View with the CDS Graphical Editor [page 295]

5.1.1.1 Create an Artifact with the CDS Graphical Editor

A CDS artifact is a design-time source file that contains definitions of objects, which you want to create in the SAP HANA catalog. You can use graphical modeling tools to model a CDS artifact.

Context

CDS artifacts are design-time source files that contain DDL code. The code describes a persistence model according to rules defined in Core Data Services. CDS artifacts have the file suffix `.hdbcds`. Building an SAP

HANA Database Module that includes this CDS artifact, creates the corresponding catalog objects in the specified schema. To create a CDS artifact, do the following:

Procedure

1. Start SAP Web IDE in a Web browser.
2. Display the application project to which you want to add a CDS artifact view.

   SAP Web IDE creates an application within a context of a project. If you do not already have a project, there are a number of ways to create one, for example: by importing it, cloning it, or creating a new one from scratch.

   a. In the SAP Web IDE, choose *File* > *New* > *Project from Template*.
   b. Choose the project template type.

      Currently, there is only one type of project template available, namely: *Multi-Target Application Project*. Select *Multi-Target Application Project* and choose *Next*.
   c. Type a name for the new MTA project (for example, `myApp`) and choose *Next* to confirm.
   d. Specify details of the new MTA project and choose *Next* to confirm.
   e. Create the new MTA project; choose *Finish*.
3. Select the module in which you want to create the CDS artifact.
4. In the *Workspace* view, browse to the *src* folder.
5. Right-click the *src* folder and choose *New* > *HDB CDS Artifact*.
6. In the *Name* field, enter the name of the CDS artifact.
7. Choose *Create*.

   The tool opens a graphical editor that you can use to define the CDS artifact. You use the same editor to model CDS entities, contexts, associations, views, user-defined data types, and more within a CDS artifact.

   > **i Note**
   >
   > You can define your preferred CDS editor (graphical or text) in *Tools* > *Preferences* > *Default Editor* > *HANA CDS Source*.
8. (Optional) Create subcontexts.

   You can create nested subcontexts within a CDS artifact. Each of these subcontexts can contain any or multiple CDS artifacts such as entities, views, structure types, and more.

   a. In the editor toolbar, choose  (Create Context).
   b. Drop the context node on the editor.
   c. Provide a name to the context.
   d. Double-click the context node.

      You can define new entities, associations, or another context within the subcontext. Use the breadcrumb navigation in the editor toolbar to switch between contexts.
9. (Optional) Publish CDS artifact as an OData service.

   You can publish CDS artifacts at the context level as OData v4 services. You can also publish subcontexts as OData services.

In the editor toolbar, choose  (Publish as OData).

Related Information

Create an Entity with the CDS Graphical Editor [page 285]
Create an Association with the CDS Graphical Editor [page 288]
Create a User-Defined Structure Type with the CDS Graphical Editor [page 293]
Create a User-Defined Scalar Type with the CDS Graphical Editor [page 294]
Create a View with the CDS Graphical Editor [page 295]
Classification of Supported Data Types [page 284]

5.1.1.1.1 Classification of Supported Data Types

You can use the CDS graphical tools to model a single CDS artifact that comprises of multiple entities, structure types, and more. When you are defining the elements in the entities or in the structure types, define data types of these elements.

The tool supports multiple data types for defining the data type of an element. For convenience, and based on its characteristics, the data types are classified as follows:

Type	Description
Primitive	Primitive types are standard Data Definition Language (DDL) data types such as String, Binary, or Integer.
Native	Native types are the native SAP HANA data types.
Structure Type	Structure types are user defined data types that comprises of a list of elements, each of which has its own data type.
Scalar Type	Scalar types are user define scalar data types. Unlike user-defined structure types, scalar types do not comprise of any elements in its definition.
Structure Element	Structure elements help reuse the data type of an element defined in a selected structure in another data type definition.
Entity Element	Entity elements help reuse the data types of an element defined in a selected entity in another data type definition.

5.1.1.2 Create an Entity with the CDS Graphical Editor

Use the CDS graphical editor to model entities. In the SAP HANA database, a CDS entity is a table with a set of data elements that are organized using columns and rows.

Context

You model entities within a CDS artifact using the graphical modeling capabilities of the CDS graphical editor.

Procedure

1. Start SAP Web IDE in a Web browser.
2. In the *Workspace* view, select the required CDS artifact within which you want to model the entity.
3. In the context menu, choose *Open With* > *Graphical Editor*.
   The tool opens the CDS artifact in a graphical editor where you can model the entity.
4. Create an entity.
   a. In the editor toolbar, choose  (Create Entity).
   b. Drop the entity node on the editor.
   c. Provide a name for the entity.
5. Define the entity.

   You can define one or more elements for the entity.
   a. Double-click the CDS entity node.
   b. In the editor toolbar, choose + (Add) to create a new element in the entity.
   c. In the *Name* text field, you can modify the name of the element.
   d. For each element, in the *Type* dropdown list, select the required value.
   e. For each element, in the *Data Type* dropdown list, select the required value.

      If you have selected the type of data type as *Structure Element* or *Entity Element*, then in the *Data Type* dropdown list, select the required structure or entity from within the same CDS artifact. Use the *Type of Element* value help list to select the required element.
   f. Define the length and scale based on the selected data type.
   g. Define whether the specified column is a primary key or part of the primary key for the specified entity. Select the *Key* checkbox accordingly.

      > **i Note**
      >
      > Structured columns can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

   h. Define if an entity column can (null) or cannot (not null) have the value NULL. If neither null nor not null is specified for the column, the default value null applies (except for the key column). Select the *Not Null* checkbox accordingly.

i. Define the default value for an entity column in the event that no value is provided during an INSERT operation. Enter a value in the *Default* text field.

6. (Optional) Use an expression to generate the element value at runtime.

   The SAP HANA SQL clause `generated always as <expression>` is available for use in CDS entity definitions. This clause specifies the expression that the tool must use to generate the element value at runtime.

   a. Select the required element.
   b. In the *Expression* field, choose  (Element Expression Editor).
   c. In the *Expression* editor, enter the required expression.

      Use the elements and operators from the expression editor to build your expression.
   d. Choose *OK*.
   e. If you want to define the element with a generated expression, select the *Generated* checkbox.

      If you do not select the *Generated* checkbox, the tool considers the expression for the element as calculated expression. Generated elements are physically present in the database table; values are computed on `INSERT` and need not be computed on `SELECT`. Calculated elements are not actually stored in the database table; they are computed when the element is "selected". Since the value of the generated column is computed on `INSERT`, the expression used to generate the value must not contain any non deterministic functions, for example: `current_timestamp, current_user, current_schema` and more.

7. Create child element, if required.

   For any selected element in the entity, you can create one or more child elements.

   a. Select the required element.
   b. In the editor toolbar, choose  (Create child).
   c. Provide a name for the child element.
   d. Define the data type and default values for the child element.

8. Import elements, if required.

   When defining the elements in an entity, you can also do so by importing elements from other catalog tables. These catalog tables can be available in the same HDI container in which you are creating the entity or a synonym that points to catalog table in another HDI container. To reuse the definition of imported elements,

   a. In the editor toolbar, choose  (Import Element(s)).
   b. In the *Import Elements* wizard, search and select the required catalog table.

      You can use the elements and its definition from the selected catalog table to define the elements of the entity.
   c. Choose *Next*.
   d. Select the elements you want to import.
   e. Choose *Finish*.

   You can modify the definition of imported elements such as, its data type, length, scale, and more.

> **i Note**
>
> The data type of elements that you import from catalog tables, and the data type of the same elements in its design-time entity from which you generated the catalog table, need not be the same. It may vary or be the same depending on the data type defined for the elements.

9. Define additional properties.
   a. In the *Storage Type* dropdown list, select the required storage type value (row or column).

      If no storage type is specified, a "`column`" store table is generated by default.
   b. In the *Unload Priority* text field, specify the required unload priority value for the generated table.

      Unload priority specifies the priority with which a table is unloaded from memory. The priority can be set between 0 (zero) and 9 (nine), where 0 means "cannot be unloaded" and 9 means "earliest unload".
   c. Select the *Auto Merge* checkbox if you want to trigger an automatic delta merge.

      > **i Note**
      >
      > Not selecting the *Auto Merge* checkbox disables the automatic delta merge operation.
   d. If you want to specify the grouping information for the generated tables, select the *Enable Table Grouping* checkbox and specify the group name, type, and sub type.
10. (Optional) Specify an identity column.

    The SAP HANA SQL clause `generated as identity` is available for use in CDS entity definitions; if you are using an expression to generate the element value at runtime, this clause enables you to specify an identity column. An element that is defined with `generated as identity` corresponds to a field in the database table that is present in the persistence and has a value that is computed as specified in the sequence options defined in the `identity` expression, for example, `( start with 10 increment by 2 )`.
    a. In the menu bar, select the *Properties* tab.
    b. In the *Element* dropdown list, select an element from the entity that you want to use as identity column.
    c. In the *Type* dropdown list, select a value.

       You can use either *always* or *by default*. If you select *always*, then values are always generated; if you select *by default*, then values are generated by default.
    d. Provide the *Start With* and *Increment by* values.

       For example, `( start with 10 increment by 2 )`.
    e. In the *Minimum Value* and *Maximum Value* dropdown list, select the required value and provide the minimum and maximum values.
    f. In the *Cache* text field, provide a value.

       The cache value is typically the difference of maximum value and minimum value.
    g. In the *Cycle* dropdown list, select a value.

       This value helps the tool determine whether it is a cycling sequence or non cycling sequence.
11. Choose *Save* to save all your changes.
12. Build an SAP HANA Database Module.
    a. From the module context menu, choose *Build*.

Next Steps

Some business cases may require performing additional tasks on an entity such as defining associations, creating partitions, and more. After creating and defining an entity, you can perform the following additional tasks within the entity.

Requirement	Task to Perform
Define relationship between entities, or between structure types and entities.	Create an Association
Partition the entity using elements in the entity.	Create a Partition
Specify indexes on entities	Create Indexes
Create entities to efficiently store series data	Create Entities to Store Series Data

Related Information

Create an Association with the CDS Graphical Editor [page 288]
Create Indexes with the CDS Graphical Editor [page 290]
Create a Partition with the CDS Graphical Editor [page 290]
Create Entities to Store Series Data [page 291]

5.1.1.2.1 Create an Association with the CDS Graphical Editor

Associations define relationships between entities, or relationship between structure types and entities. You create associations in either a CDS entity definition or structure type definition, which are design-time files in SAP HANA.

Context

The CDS graphical editor tool in SAP Web IDE provides you graphical modeling capabilities to model associations between entities, or between structure types and entities. If you are creating associations between entities, then the associations are defined as part of the entity definition, which are design-time files in the repository. Similarly, if you are creating associations between structure types and entities, then the associations are maintained in the structure type definition, which are design-time files in the repository.

Procedure

1. In the *Workspace* view, select the required CDS artifact.
2. In the context menu, choose *Open With* > *Graphical Editor*.
3. Create associations between entities.

   If you are creating associations between entities, you need at least two entities in the CDS artifact.

   a. Select the required entity.

   b. Choose  (Association).

   c. Drag the cursor to another entity in the CDS artifact, with which you want to create associations.
4. Create associations between structure types and entities.

   If you are creating associations between a structure type and an entity, you need at least one structure type and one entity in the CDS artifact. In associations between structure type and entity, the target is an entity. This means that, the association definition is maintained in the structure type definition.

   a. Select the required structure type.

   b. Choose  .

   c. Drag the cursor to an entity in the CDS artifact, with which you want to create associations.
5. Provide association details.

   In the *Association Details* dialog box, provide further details for the association.

   a. In the *Name* text field, provide a name for the association.

   b. Select the association type.

   Select *Managed* or *Unmanaged* association type, depending on whether you want to create managed or unmanaged associations.

   > **i Note**
   >
   > For associations between structure types and entities, you can create only managed associations.

   c. If you are creating managed associations, select required elements from the structure type or entity as association keys.

   For managed associations, the relation between source and target entity is defined by specifying a set of elements of the target entity that are used as a foreign key. If no foreign keys are explicitly specified, the elements of the target entity's designated primary key are used. In the *Alias* text field, provide an alias name, if required.

   d. If you are creating unmanaged associations, in the expression editor, provide the required condition.

   Unmanaged associations are based on existing elements of the source and target entity; no fields are generated. In the `ON` condition, only elements of the source or the target entity can be used; it is not possible to use other associations. The `ON` condition may contain any kind of expression. Use the elements and operators from the editor, when defining the `ON` condition expression.
6. Define cardinality, if required.

   When using an association to define a relationship between entities in a CDS artifact, use the cardinality to specify the type of relation, for example, one-to-one (to-one) or one-to-many (to-n); the relationship is with respect to both the source and the target of the association.

   a. In the *Source Cardinality* and *Target Cardinality* dropdown lists, select the required cardinality values.

7. Choose *OK*.
8. Choose *Save*.

> **i Note**
>
> You can also create associations using a form-based editor. Double-click the required entity or structure type node in the CDS artifact, and in the Associations tab, create, and define the association.

5.1.1.2.2 Create Indexes with the CDS Graphical Editor

You can create indexes for an entity and specify an index type. The tool supports three types of indexes, plain indexes, full text indexes, and fuzzy search indexes.

Procedure

1. In the menu bar, choose the *Indexes* tab.
2. Choose + (Add) to create a new index.
3. In the *General* section, select the required index type.

   After selecting the index type, define which of the elements in the entity should be indexed. For the plain index type, you can also select the index order and use the *Unique* checkbox to define whether the index is unique (no two rows of data in the indexed entity can have identical key values). For *Full Text Index*, select the element and define the full text parameters such as the language column, mime type column, and more.

   > **i Note**
   >
   > You cannot specify both a full-text index and a fuzzy search index for the same element.
4. Choose *Save*.

5.1.1.2.3 Create a Partition with the CDS Graphical Editor

Use elements from the entity to partition the entity. You can use hash, range, or round-robin partition types to partition the entity.

Procedure

1. In the menu bar, choose the *Partitions* tab.
2. Select the *Add Primary Partition* checkbox.

3. In the *Partition Type* dropdown list, select a partition type.

> **i Note**
>
> For *Hash* or *Roundrobin* partition types, you can also define the secondary partition type. In the *Secondary Partition* section, select the required secondary partition type.

4. Choose + (Add) to add an element.
5. In the *Element* dropdown list, select an element, which you want to use to partition the data.
6. In the *Functions* dropdown list, select a value. This is applicable if any of the selected element used for partitioning the entity represents time values.
7. In the *Partitions* text field, enter the number of partitions required.

> **i Note**
>
> If you want to partition the entity based on number of servers, select the *Number of Servers* checkbox

8. Choose *Save*.

5.1.1.2.4 Create Entities to Store Series Data

Use the CDS graphical editor in SAP Web IDE to create entities that can efficiently store series data.

Procedure

1. In the menu bar, choose the *Series* tab.
2. Select the *Enable Series* checkbox.
3. In the *Series Key Element(s)* dropdown list, select one or more elements.
4. If you want to create an entity to store time series data, in the *Period for Series* dropdown list, select the required period column.

   Each row of a series table has a period of validity. The period represents the period in time that the row applies to. When the series table represents instants, then there is a single period column. When the table represents intervals, there can be one or two columns.

   > **i Note**
   >
   > The period for series must be unique and should not be affected by any shift in timestamps. If you want to create an entity to store non time series data, select the *Null* checkbox.

5. Select the required *Equidistant Type*.

Type	Description
Not Equidistant	If you want to create an entity to store arbitrary time values without any regular pattern.
Equidistant	If you are creating an entity to store time values with regular pattern. For equidistant series, provide the *Increment By* value that defines the distance between adjacent elements in an equidistant series. The equidistant series tables offer improved compression compared to non equidistant tables.
Equidistant Precisewise	If you want to create an entity to store series data that retains a degree of regularity without being equidistant across the entire dataset. Equidistant piecewise data consists of regions where equal increments exist between successive points, but these increments are not always consistent for all regions in the table. Different series or different parts of a series may have different intervals. For equidistant precisewise series, you use only one period column, but you can specify one or more *Alternate Period for Series*. These columns can be used to record the time for each row, or represent the end of an interval associated with each row. Good compression can be achieved for these columns when successive values within the series differ by a constant amount.

6. (Optional) Provide minimum and maximum values.

   You can define equidistant and non equidistant series tables with a minimum and/or maximum value. These values are used to verify that loaded data corresponds to the range definition. The values in *Period for Series* column must satisfy the minimum values and the maximum value constraint.

   a. In the *Min Value* dropdown list, select the *Min Value* menu option and provide the required minimum value.
   b. In the *Max Value* dropdown list, select the *Max Value* menu option and provide the required maximum value.

> **i Note**
>
> If the *Period for Series* column does not have a lower limit and upper limit, you can use the *No Min Value* and *No Max Value* respectively.

5.1.1.3 Create a User-Defined Structure Type with the CDS Graphical Editor

A structured type is a user-defined data type comprising a list of attributes, each of which has its own data type. You create a user-defined structured type as a design-time file in the SAP HANA repository.

Context

Use the CDS graphical editor to create a user-defined structured type as a design-time file in the repository. The attributes of the structured type can be defined manually in the structured type itself and reused by another structured type (or by scalar type, or by an entity). You create a user-defined structure types within a CDS artifact.

Procedure

1. Start SAP Web IDE in a Web browser.
2. In the *Workspace* view, select the required CDS artifact within which you want to model the entity.
3. In the context menu, choose *Open With* > *Graphical Editor*.
   The tool opens the CDS artifact in a graphical editor where you can create the structure type.
4. Create a structured type.
   a. In the editor toolbar, select (Create Structure).
   b. Drop the structure node on the editor.
   c. Provide a name to the structure.
5. Define the elements.
   a. Double-click the structure type node.
   b. In the editor toolbar, choose + (Add) to create a new element in the structure.
   c. In the *Name* field, you can modify the name of the element.
   d. For each element, in the *Type* dropdown list, select the required value.
   e. For each element, in the *Data Type* dropdown list, select the required value.

      If you have selected the type of data type as *Structure Element* or *Entity Element*, then in the *Data Type* dropdown list, select the required structure or entity in the CDS artifact. Use the *Type of Element* value help list to select the required element.
   f. Define the length and scale based on the selected data type.
6. Create child element, if required.

   For any element in the structure, you can also create one or more child elements.
   a. Select the required element.
   b. In the editor toolbar, choose (Create child).
   c. Define the data type for the child element.

> **i Note**
>
> When you create a child element, the tool automatically changes the type of data type of its parent element to *Inline*.

7. Import elements, if required.

   When defining the elements in an entity, you can also do so by importing elements from other catalog tables. These catalog tables can be available in the same HDI container in which you are creating the entity or a synonym that points to catalog table in another HDI container. To reuse the definition of imported elements,

   a. In the editor toolbar, choose (Import Element(s)).
   b. In the *Import Elements* wizard, search and select the required catalog table.

      You can use the elements and its definition from the selected catalog table to define the elements of the structure type.
   c. Choose *Next*.
   d. Select the elements you want to import.
   e. Choose *Finish*.

   You can modify the definition of imported elements such as its data type, length, scale, and more.

8. In the editor toolbar, choose (Navigate Back).
9. Choose *Save*.

5.1.1.4 Create a User-Defined Scalar Type with the CDS Graphical Editor

Scalar types are user-defined scalar data types that does not comprise of any elements in its definition. You create a user-defined structured type as a design-time file in the SAP HANA repository.

Context

Use the CDS graphical editor to create a user-defined structured type as a design-time file in the repository. You create a user-defined scalar data types within a CDS artifact. After creating a scalar type, you can reuse it when defining data types of elements in other structure types, scalar types, or entities.

Procedure

1. Start SAP Web IDE in a Web browser.
2. In the *Workspace* view, select the required CDS artifact within which you want to model the entity.
3. In the context menu, choose *Open With* > *Graphical Editor*.

   The tool opens the CDS artifact in a graphical editor where you can create the scalar type.

4. Create a scalar type.

    a. In the editor toolbar, select  (Create Scalar type).
    b. Drop the scalar type node on the editor.
    c. Provide a name to the scalar type.
5. Define the scalar type.
    a. Double-click the scalar type node.
    b. For each element, in the *Type* dropdown list, select the required value.

        The data types are classified as primitive data types, native SAP HANA data types, user-defined data types (structure types and scalar types), entity element and structure element. Based on your requirement you can select a value from any of the preceding data type classifications.
    c. In the *Data Type* dropdown list, select the required value.

        If you have selected the type of data type as *Structure Element* or *Entity Element*, then in the *Data Type* dropdown list, select the required structure or entity in the CDS artifact. Use the *Type of Element* value help list to select the required element.
    d. Define the length and scale based on the selected data type.
6. In the editor toolbar, choose  (Navigate Back).

5.1.1.5 Create a View with the CDS Graphical Editor

Use the graphical modeling capabilities of the CDS graphical editor tool in SAP Web IDE to create a design-time CDS view.

Context

A view is a virtual table based on the dynamic results returned in response to an SQL statement. You use the graphical modeling tools to create a view with the CDS artifact. After creating the CDS view, use this design-time view definition to generate the corresponding catalog object when you deploy the application that contains the view-definition artifact (or just the application's database module).

Procedure

1. Start SAP Web IDE in a Web browser.
2. In the *Workspace* view, select the required CDS artifact within which you want to model a CDS view.
3. In the context menu, choose *Open With* > *Graphical Editor*.

    The tool opens the CDS artifact in a graphical editor where you can create and define the CDS view.
4. Create a CDS view.

a. In the editor toolbar, select  (Create View).
b. Drop the view node on the editor.
c. Provide a name to the view.

5. Add local CDS artifacts as data sources.

   You can add local CDS artifacts, active CDS artifacts, or both as a data source in a view node. Local CDS artifacts are those entities that exist within the same artifact in which you are creating the CDS view. Active CDS artifacts are those artifacts that are already built and activated.

   a. Double-click the view node.
   b. If you want to add a local CDS artifact, in the editor toolbar, select + (Add Local Objects).
   c. Select the required entities that you want to add as data sources.
   d. Choose *OK*.

6. Add active CDS artifacts as data sources.

   Active CDS artifacts are those artifacts that are already built and activated.

   a. Double-click the view node.
   b. If you want to add an active CDS artifact, in the editor toolbar, select  (Add Active Objects).
   c. Choose *Finish*.
   d. Select the required menu option.

   > **i Note**
   >
   > If you have more than one data source in the CDS view, either create a union, or a join between the data sources.

7. (Optional) Using elements from associated entities.

   You can model a CDS view with entities, which are defined with associations to other entities. In such cases, you can use the elements from the associated entities in the CDS view definition. For using elements from the associated entities, it is not necessary to use associated entities as data sources in the CDS view.

   a. Select the required entity.
   b. Select the association column in the entity.
   c. In the *Select Element(s)* dialog box, select the required elements from the associated entity.

      You can use the elements from the associated entities in the CDS view definition. For example, as output columns, to define SQL clauses, perform SQL `GROUP BY` or `ORDER BY` operations, and more.
   d. (Optional) Define a filter expression.

      In the *Filter Expression* editor, provide a filter expression rule that defines how the elements from the associated entities are used.

      When following an association (for example, in a view), it is possible to apply a filter condition; the filter is merged into the `ON`-condition of the resulting `JOIN`. The following example shows how to get a list of customers and then filter the list according to the sales orders that are currently “open” for each customer. In the example, the infix filter is inserted after the association orders to get only those orders that satisfy the condition `status='open'`.

      > **Sample Code**
      >
      ```
      view C1 as select from Customer {
        name,
        orders[status='open'].{id as orderId, date as orderDate}
      ```

```
};
```

Use the elements from the associated entities and the operators provided in the dialog box to build your filter expression.

e. Choose *OK*.

8. (Optional) Using elements from structure types.

   You can model a CDS view with an entity that has a structure type. In such cases, you can use elements from the structure types in the CDS view definition.

   a. Select the required entity.
   b. Select the structure type element in the entity.
   c. In the *Select Element(s)* dialog box, select the required elements from the structure type.

      You can use the elements from the structure types in the CDS view definition. For example, as output columns, to define SQL clauses, perform SQL `GROUP BY` or `ORDER BY` operations, and more.

   d. Choose *OK*.

9. Define output columns.

   a. In the CDS view editor, select the required data source.
   b. Select the columns from the data source that you want to add to the output.
   c. In the context menu, choose *Add to Output*.
   d. In the *Select Element(s)* dialog box, provide an alias name to the output column, if required.
   e. Choose *OK*.

      The tool displays the output columns of the view under the *Columns* section (in the details pane) of the editor.

10. (Optional) Group by result set.

    Use one or more columns from the data source to perform a SQL `GROUP BY` operation.

    a. Select the data source.
    b. Select the required columns.
    c. In the context menu, choose *Add to Group By* to perform a SQL `GROUP BY` operation with the selected columns.

       You can also use an output column to perform a SQL `GROUP BY` operation. In the *Columns* section, select an output column, and in the context menu, choose *Add to Group By*.

11. (Optional) Order by result set.

    Use one or more output columns from the view to perform a SQL `ORDER BY` operation.

    a. In the details pane, expand the *Columns* section.
    b. Select the required columns.
    c. In the context menu, choose *Add to Order By* to perform a SQL `ORDER BY` operation with the selected output columns.

12. (Optional) Limit and offset the result set.

    You can use the SQL `LIMIT` clause and the SQL `OFFSET` clause in a CDS view. In the details pane, specify the required limit value and offset value as integers.

    a. In the *Limit* text field, provide the required limit value.

       Use the `LIMIT` clause to restrict the number of result sets to a specified limit.

    b. In the *Offset* text field, provide an integer value required offset value.

Use the `OFFSET` clause to specify the number of records to skip before displaying the results sets defined by the `LIMIT` SQL clause.

13. (Optional) Create Subqueries.

    You can model CDS views with nested SQL queries (subqueries) to obtain the desired output. Subqueries help model CDS views for complex business scenarios.

    a. Double-click the CDS view.

    b. In the editor toolbar, choose  (Sub Query).

    c. Drop the subquery node on the editor.

    d. Provide a name to the subquery node.

    e. Double click the subquery node.

    f. Define the subquery.

    Defining a subquery is similar to defining a view. Within a subquery, you can only creating entities, other subqueries, creating union of entities, creating union of subqueries, and creating union of an entity and a subquery.

    > → Tip
    >
    > You can use the breadcrumb navigation in the editor toolbar to switch between subqueries and the target CDS view.

14. Choose *Save* to save all your changes.

15. Build an SAP HANA Database Module.

    The build process uses the design-time database artifacts to generate the corresponding actual objects in the database catalog. The activation process generates a corresponding catalog object for each of the artifacts defined within another artifact; the location in the catalog is determined by the type of object generated.

    a. From the module context menu, choose *Build*.

Next Steps

Modeling CDS views for complex business scenarios could include layers of calculation logic. In such cases, it may require to perform certain additional tasks to obtain the desired output.

The following table lists some important additional tasks that you can perform to enrich the CDS view.

Requirement	Task to Perform
If you want to query data from two data sources and combine records from both the data sources based on a join condition.	Create Joins
If you want to combine the results of two more data sources.	Create Unions
If you want to create new output columns and calculate its values at runtime using an expression.	Create Calculated Columns
If you want to define relationships between CDS views.	Create Associations for CDS Views with CDS Graphical Editor

Related Information

Create SQL WHERE Clause [page 299]
Create SQL HAVING Clause [page 299]
Create Calculated Columns [page 300]
Create Joins [page 300]
Create Unions [page 302]
Creating Synonyms [page 305]
Create Associations for CDS Views with CDS Graphical Editor [page 303]

5.1.1.5.1 Create SQL WHERE Clause

Define SQL `WHERE` clause in a view to filter and retrieve records from the output of a data source based on a specified condition.

Procedure

1. Open the required CDS artifact.
2. Select the CDS view in which you want to define the SQL `WHERE` clause.
3. Double-click the CDS view.
4. In the *Where* section, choose + (Add Where Clause).
5. In the *Where Clause Editor*, define the SQL `WHERE` clause condition using the required elements and operators.
6. Choose *OK*.
7. Choose *Save*.

5.1.1.5.2 Create SQL HAVING Clause

Define SQL `HAVING` clause in a view to retrieve records from the output of a data source, only when the aggregate values satisfy a defined condition.

Procedure

1. Open the required CDS artifact.
2. Select the CDS view in which you want to define the SQL `HAVING` clause.

> **Note**
>
> You can create SQL `HAVING` clause only if you are using one or more output columns to perform a SQL `GROUP BY` operation on the selected view.

3. Double-click the CDS view.
4. In the *Having* section, choose + (Add Having Clause).
5. In the *Having Clause Editor*, define the SQL `HAVING` clause condition using the required elements and operators.
6. Choose *OK*.
7. Choose *Save*.

5.1.1.5.3 Create Calculated Columns

Create new output columns for a view and calculate the column values at runtime based on the result of an expression. You can use other column values, functions, or constants when defining the expression.

Procedure

1. Open the required CDS artifact.
2. Select the CDS view in which you want to create calculated columns.
3. Double-click the CDS view.
4. In the details pane, select the *Columns* section.
5. Choose + (Add Calculated Column).
6. Provide a valid expression.

   In the *Add Calculated Column* dialog box, provide the expression that defines the calculated column.
   a. In the *Name* text field, provide a name for the calculated column.
   b. In the expression editor, provide a valid expression using the required elements and operators.
   c. Choose *OK*.

5.1.1.5.4 Create Joins

Joins help query data from two or more data sources. It helps to limit the number of records, or to combine records from both the data sources, so that they appear as one record in the query results.

Context

After modeling a CDS view, you can include a `JOIN` clause in the CDS view definition.

Procedure

1. Open the required CDS artifact.
2. Select the CDS view in which you want to create a join for data sources.
3. Double-click the CDS view.

   > **i Note**
   >
   > You can create a join only if the CDS view has two or more data sources.

4. Select an entity.
5. Choose  (Join).
6. Drag the cursor to the required data source.
7. Define the join properties.

   In the *Join Definition* dialog box, define the join properties and the join condition.

   a. In the *Type* dropdown list, select the required join type.
   b. In the expression editor, specify a valid join condition using the necessary element and operators.

      Using proposed join condition.

      The tool analyzes the data in the participating entities and proposes a join condition. Based on your requirement, you can create your own join condition or use the join condition that the tool proposes. In the *Join Definition* dialog box, choose *Propose Condition* to use the join condition that the tool proposes.

      > **i Note**
      >
      > The tool does not propose any join condition for cross joins.

   c. Choose *OK*.

Related Information

Supported Join Types [page 301]

5.1.1.5.4.1 Supported Join Types

When creating a join between two data sources, you specify the join type. The following table lists the supported join types in CDS graphical editor.

Join Type	Description
Inner	This join type returns all rows when there is at least one match in both the data sources.

Join Type	Description
Left Outer	This join type returns all rows from the left data source, and the matched rows from the right data source.
Right Outer	This join type returns all rows from the right data source, and the matched rows from the left data source.
Full Outer	This join type displays results from both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.
Cross Joins	This join type displays results of all possible combinations of rows from the two tables. Cross join is also called a cross product or Cartesian product.

5.1.1.5.5 Create Unions

Creating unions help combine the result sets of two or more data sources.

Context

After modeling a CDS view using the graphical modeling tools, you can include a `UNION` clause in the CDS view definition.

Procedure

1. Open the required CDS artifact.
2. Select the CDS view in which you want to perform the union operation.
3. Double-click the CDS view.

   > **i Note**
   >
   > You can create a union only if the CDS view has two or more data sources.

4. Select a data source.
5. Choose  (Union).
6. Drag the cursor to the required data source.

   This operation creates a *ResultSet* node for each of the data sources.
7. Add more data sources to the union.

   You can perform union operation on multiple data sources (more than two data sources). If you have already created a union of two data sources, then to create a union of these two data sources with the third data source, you use the ResultSet node.

a. Select a *ResultSet* node.

   You cannot select the *DefaultResultSet* node to union records of multiple data sources.

b. Choose  (Union).

c. Drag the cursor to the required data source.

8. Add columns to the union output.

   The columns you add to the *DefaultResultSet* node are the output columns of the union.

   a. Select the data source.

   b. Select the columns in the data source you want to add to the output.

   c. In the context menu, choose *Add to Output*.

> **i Note**
>
> Union operation combine result sets of two or more SELECT statements. It is necessary that each result set (SELECT statements) has the same number of columns, have similar data types, and also the columns in each result sets are in same order.

5.1.1.5.6 Create Associations for CDS Views with CDS Graphical Editor

Associations define relationships between CDS views, and you create association in the CDS view definition.

Context

In a CDS view definition, associations can be used to gather information from the specified target entities. You can use the CDS graphical editor to create associations only if the views are simple CDS views. This means that, the CDS views involved in the association must contain only a single entity as a data source.

Procedure

1. In the *Workspace* view, select the required CDS artifact.
2. In the context menu, choose *Open With* > *Graphical Editor*.
3. Create associations between views.
   a. Select the required CDS view.
   b. Choose  (Association).
   c. Drag the cursor to another CDS view in the CDS artifact with which you want to create the association.
4. Provide association details.
   a. In the *Association Details* dialog box, provide further details for the association.

b. In the *Name* text field, provide a name for the association.
c. (Optional) If you want the tool to propose a condition for the association, choose *Propose Condition*.
d. In the expression editor, provide the required `ON` condition.

   In the `ON` condition, only elements of the source or the target CDS view can be used; it is not possible to use other associations. The `ON` condition may contain any kind of expression. Use the elements and operators from the dialog box, when defining the `ON` condition expression.

5. Define cardinality, if required.
   a. When using an association to define a relationship between CDS views in a CDS artifact, use the cardinality to specify the type of relation, for example, one-to-one (to-one) or one-to-many (to-n); the relationship is with respect to both the source and the target of the association.
   b. In the *Source Cardinality* and *Target Cardinality* dropdown lists, select the required cardinality values.
6. Choose *OK*.
7. Choose *Save*.

5.1.1.6 Preview Output of CDS Views and Entities

After modeling and activating a CDS view or an entity, you can preview the output data using the SAP Database Explorer.

Context

You can preview output data of CDS views or entities in simple in tabular format, or you can preview output data in graphical representations, such as bar graphs, area graphs, and pie charts.

You can also export and download the output data to `.csv` files. The Database Explorer also allows you to preview the SQL query that the tool generates for an activated CDS artifact.

Procedure

1. Start SAP Web IDE in a Web browser.
2. In the *Workspace* view, select the required CDS artifact to preview the output of its views or entities.
3. In the context menu, choose *Open With* > *Graphical Editor*.
   The tool opens the CDS artifact in a graphical editor.
4. In the editor, select the required view or entity and choose  (Open Data Preview).
5. Apply filters.
   a. For CDS views, if you want to apply filters on columns and view the filtered output data, choose  (Add Filter).
   b. Choose *Add Filters*.

c. Choose a column and define filter conditions.

6. Export output data, if required.

   If you want to export the raw data output to a .csv file,

   a. In the toolbar, choose  (Download).
   b. In the *Delimiter* dropdown list, select the required delimiter that the tool must use to separate the values in the .csv file.
   c. Choose *Download*.

7. View SQL query for the entity or view, if required.

   a. If you want to view the SQL query that the tool generates for the activated CDS artifact, in the toolbar, choose *SQL*.
   b. In you want to view and execute the SQL query in SQL editor, choose 

(Edit SQL Statement in SQL Console).

8. Preview output in graphical representations.

   The tool supports bar graph, area graph, pie chart and table charts, and other graphical representations to preview the output of a CDS view or entity.

   a. In the menu bar, choose *Analysis*.
   b. Configure the axis values by dragging and dropping the required columns to the *Label Axis* and the *Value Axis*.

      The tool displays the output data in graphical representation. Select the required chart icons in the menu to view the output in different graphical representation.

   c. In the menu bar, choose  (Display Settings) to toggle legends and values.

Related Information

Create a View with the CDS Graphical Editor [page 295]
Create an Entity with the CDS Graphical Editor [page 285]

5.1.2 Creating Synonyms

Synonym are design time files in the repository that provides data independence. After creating a synonym, you can bind this synonym with tables in different schemas.

- `hdbsynonym` - A synonym declaration (with an optional default configuration).
- `hdbsynonymconfig` - An explicit configuration of the synonym's target.

This section describes creating the above two synonym definition files without writing any synonym definition code and the prerequisites to creating such a database synonym.

Related Information

Create a Database Synonym [page 306]

5.1.2.1 Create a Database Synonym

Use the synonym editor in SAP Web IDE to create and define synonyms without writing any synonym-definition code.

Prerequisites

- You have correctly configured all services including the user provided service in the mta.yaml file. This allows using the user provided service within the project. A sample mta.yaml configuration is shown below.

  Sample Code

```
ID: DwProject
version: 1.2.3
modules:
 - name: DwDbModule
   type: hdb
   requires:
     - name: DwHdiService
       properties:
         TARGET_CONTAINER: ~{hdi-service-name}
     - name: CrossSchemaService
resources:
 - name: DwHdiService
   type: com.sap.xs.hdi-container
   properties:
     hdi-service-name: ${service-name}
 - name: CrossSchemaService
   type: org.cloudfoundry.existing-service
   parameters:
     service-name : CROSS_SCHEMA_SERVICE_SYSTEM_ERP
```

- You have created a grantor file (.hdbgrants) that specifies the necessary privileges to access external tables.

  Sample Code

```
{
  "CROSS_SCHEMA_ACCESS_SERVICE_ERP": {
    "object_owner" : {
      "schema_privileges":[
        {
          "reference":"ERP",
          "privileges_with_grant_option":["SELECT"]
        }

      ]
    },
     "application_user" : {
```

```
      "object_privileges":[
        {
          "schema":"ERP",
          "name": "TABLE",
          "privileges":["SELECT"]
        }

      ]
    }
  }
}
```

Procedure

1. Start SAP Web IDE in a browser.
2. Display the application project to which you want to create a database synonym.

   SAP Web IDE creates an application within a context of a project. If you do not already have a project, there are a number of ways to create one, for example: by importing it, cloning it, or creating a new one from scratch.

   a. In the SAP Web IDE, choose *File* > *New* > *Project from Template*.
   b. Choose the project template type.

      Currently, there is only one type of project template available, namely: *Multi-Target Application Project*. Select *Multi-Target Application Project* and choose *Next*.
   c. Type a name for the new MTA project (for example, **myApp** and choose *Next* to confirm).
   d. Specify details of the new MTA project and choose *Next* to confirm.
   e. Create the new MTA project; choose *Finish*.
3. Select the HDB module in which you want to create synonyms.
4. Browse to the *src* folder.
5. Create a `.hdbsynonym` file.

   a. In the context menu of the *src* folder, choose *New* > *File*.
   b. Provide a name for the synonym declaration file.

      > **i Note**
      >
      > You must add `.hdbsynonym` as a suffix to the file name.

   c. Choose *OK*.

      The tool opens a new synonym editor where you can declare all the synonyms that you require.
   d. Choose +.
   e. In the *Find Data Sources* dialog box, search the required data source for which you want to declare a synonym.

      You can declare multiple synonyms in a single file.

      > **i Note**
      >
      > You can also directly enter the name of the data source in the *Object Name* text field.

f. Choose *Save*.

6. Build an HDB module.

   The build process uses the design-time database artifacts to generate the corresponding actual objects in the database catalog.

   a. From the module context menu, choose *Build* > *Build*.

5.2 The Graphical CDS Editor in SAP Business Application Studio

Use SAP Business Application Studio to maintain CDS artifacts in SAP HANA.

SAP Business Application Studio includes dedicated editors that you can use to define data-persistence objects in CDS documents using the DDL-compliant Core Data Services syntax for SAP HANA. If you right-click a file with the `.hdbcds` extension in the *Project Explorer* view of your application project, you can choose *Open With* > *Graphical Editor* to view a graphical representation of the contents of a CDS source file, with the option to edit the source code as text with the syntax elements highlighted for easier visual scanning.

Related Information

The Graphical CDS Editor in SAP Web IDE [page 281]

5.2.1 Getting Started wth the CDS Graphical Editor in SAP Business Application Studio

SAP Business Application Studio provides data modelers with the graphical modeling capabilities that they require to model artifacts. These artifacts define the data persistence objects using SAP HANA Core Data Services (HDB CDS).

The CDS graphical editor in SAP Business Application Studio enables you to model data persistence objects in SAP HANA CDS artifacts using the DDL-compliant SAP HANA CDS syntax. The tool recognizes the `.hdbcds` file extension required for SAP HANA CDS object definitions and calls the appropriate SAP HDI deployment plug-in to parse the SAP HANA CDS artifact. If you right-click a file with the `.hdbcds` extension in the *Workspace* view, SAP Business Application Studio provides the following choice of editors in the context-sensitive menu:

- Text Editor
  View and edit DDL source code in an SAP HANA CDS document as text with the syntax elements highlighted for easier visual scanning.
- Graphical Editor
  View graphical representation of SAP HANA CDS artifacts, and also model SAP HANA CDS artifacts with graphical modeling tools.

Related Information

Set up a Development Space for SAP HANA CDS in SAP Business Application Studio [page 309]
Create a New Business Application Project for SAP HANA CDS Development in HaaS [page 311]

5.2.1.1 Set up a Development Space for SAP HANA CDS in SAP Business Application Studio

Create a development space that includes tools that enable application development in SAP HANA as a Service (HaaS).

Prerequisites

- You have access to SAP Business Application Studio.

Context

In SAP Business Application Studio, the developer is provided with one or more development spaces (dev spaces). As a developer, you can chose which tools will be installed in your dev spaces by selecting the suitable extension pack along with any additional extensions that you want or need. The dev space is an isolated development environment providing a pseudo-local development experience. Among other tools, it provides terminal access to the file system so you can run various commands, you can test-run your application in the dev space itself without deploying to the target run-time environment (Cloud Foundry), almost as if the you were working on your own desktop.

This tutorial shows how to create and set up a dev space in SAP Business Application Studio, which you can then use to develop and deploy SAP HANA database applications.

Procedure

1. Log in to SAP Business Application Studio.
2. Create a new development space.
   a. Choose *Create Dev Space*.
   b. In the *Create a New Dev Space* Wizard, choose the business scenario that best suits the applications you want to create, for example, *SAP HANA Native Application*.
   c. Check the list of predefined SAP extensions.

      The SAP HANA Native Application dev space includes a set of basic predefined extensions. Check that the list includes all the extensions you need.

d. Select additional extensions if necessary.

   Check the list of *Additional SAP Extensions* to see if it includes any other tools you might need, for example: *SAP HANA Performance Tools*, etc.

   > → Tip
   >
   > If you are not sure which extensions you need, bear in mind that you can add extensions later after creating a dev space, for example, using the *[Edit]* tool in the list of dev spaces as described in the next steps.

e. Provide a name for the new space, for example, "**`HaaSDEV`**"

   > → Tip
   >
   > The name must contain only alphanumeric characters; special characters are not permitted.

f. Choose *Create Dev Space*.

   The new dev space *HaaSDEV* is displayed in the dev space manager and automatically started.

3. Open the new dev space.

   In the list of dev space names, monitor the status of the new *HaaSDEV* dev space. When it displays the status *RUNNING*, click the dev space name *HaaSDEV* to open it in a new workspace.

   > → Tip
   >
   > On startup, SAP Business Application Studio displays a list of available development spaces in the dev space manager, but all dev spaces are in the status *STOPPED*. Since you can only open a dev space whose status is *RUNNING*, you have to start it manually. Choose ▷ *(Run)* to start the development space that contains the application project you want to work on.

4. Connect the dev space to the Cloud Foundry organization and space where you want to develop and deploy your business application.

   By default, SAP Business Application Studio logs on to the most recently used Cloud Foundry target and displays the connection details in the status bar, for example, *Targeting Cloud Foundry MyOrg/MySpace*. If no target has yet been set, the status bar displays the message *The Organization and space in CF have not been set*. To set the organization and space for your dev space, you need to log on to Cloud Foundry and specify the target to use, as described in the following steps:

   a. Log in to Cloud Foundry.

      Click the message *The Organization and space in CF have not been set* in the status bar to open the command palette, where the *CF: Log in to cloud foundry* Wizard prompts you for the information required to log on to Cloud Foundry. Alternatively, you can use the *Cloud Foundry: Targets* tool in the tool-selection pane to display a list of known Cloud Foundry targets; create a new target; or set the default target organization and space to use for your dev space.

      > i Note
      >
      > To log on to Cloud Foundry, you need to know the Cloud Foundry API end point, for example, `https://api.cf.sap.hana.<etc...>`. You will also need to provide a CF user name (or e-mail address) and a corresponding password. If two-factor authentication is active, bear in mind that an additional one-time authentication token is required.

b. Set the Cloud Foundry organization.
c. Set the Cloud Foundry space.

5. Customize a dev space.

   You can add extensions to (or remove them from) an existing dev space, for example, using the *(Edit)* tool in the list of dev spaces.

   > **i** Note
   >
   > The dev space you want to modify must be *STOPPED*; it is not possible to edit a dev space whose status is *RUNNING* or *STARTING*.

   a. Display the list of dev spaces you have created.
   b. Check that the dev space you want to edit is *STOPPED*; if it is not, then stop it.
   c. Choose the *Edit* button for the dev space you want to modify.
   d. Select the extensions you want to add or remove and choose *Save Changes*.
   e. Restart the modified dev space.

5.2.1.2 Create a New Business Application Project for SAP HANA CDS Development in HaaS

Set up a new project for your SAP HANA CDS application in SAP Business Application Studio.

Prerequisites

- You are logged in to SAP Business Application Studio.
- You have created a development space (dev space) in SAP Business Application Studio (for example, the one that you set up in the previous tutorial). For instructions, see *Related Information* below.

Context

This tutorial shows how to use the templates provided by SAP Business Application Studio to set up a project for your new SAP HANA database application.

Procedure

1. Start the *Create Project from Template* Wizard.
   - Open the command palette:
     Either press `Crtl` + `Shift` + `P`

Or choose *View* > *Find Command...*
Or just press F1

- In the command palette, type **Project** and choose *Create Project from Template*.

> **→ Tip**
>
> Alternatively, in the *Welcome* tab, choose *Start from Template* > *Create a new project*.

2. Create a new project for your business application.
   a. In the *New Project from Template* Wizard, choose *SAP HANA Database Project* and then choose *Start*.
   b. In the *Add Basic Information* pane, type the name of the new project (**MyHaaSCDSapp**) and choose *Next*.
   c. In the *Set Basic Properties* pane, accept the default settings and choose *Next*.
   d. In the *Set Database Information* pane, use the default settings and choose *Next*.

      You can use the following default settings:

      - *Namespace* (empty)
      - *Schema Name* (empty)
      - *SAP HANA Database Version* * (HANA Service)
      - *Bind the database module to a Cloud Foundry service instance?* (Yes)

   e. In the *Bind to HDI Container Service* pane, provide the requested information.

      You need to make the following choices:

      - *Create a new HDI service Instance? [Yes/No]*
        When creating the new project, automatically create a service instance to use for application deployment.
      - *Use the default database instance of the selected Cloud Foundry space?*
        When creating the new project, connect to the underlying database automatically and use the database for all future deployments.

      > **i Note**
      >
      > It is necessary to provide the credentials required to log on to the Cloud Foundry end point that provides the service instance to which you want to bind the database module of your new application project. After you log on to Cloud Foundry, you can choose the target organization, space, and service instance.

      - *Enter e-mail address* (Type the e-mail address of a registered Cloud Foundry user)
      - *Enter Password* (Type the Cloud Foundry user's password)
        Press Enter or click *Login ->* to connect to Cloud Foundry.

        > **i Note**
        >
        > If two-factor authentication (2FA) is configured in your landscape, you might need to provide an additional single-use token.

      After you log in to Cloud Foundry, use the following options to specify the target Cloud Foundry space where the service instance is running:

      - *Choose Cloud Foundry Org* * (Select from the drop-down list)
      - *Choose Cloud Foundry Space* * (Select from the drop-down list)

- *Choose Cloud Foundry Service* * (Create a new service instance)
- *Please enter a unique and non-existing service-instance name** (default, for example, "<ProjectName>-hdidb-ws-<WorkspaceName>")

> **i Note**
>
> It is recommended to accept the default service-instance name displayed, but you can also choose an existing service instance from the drop-down list provided.

Choose *Finish* to generate the new project called *MyHaaSCDSapp* in the chosen Cloud Foundry organization and space.

3. Add the new project to a new workspace; choose *Open in New Workspace*.

   SAP Business Application Studio opens the explorer and displays the new project.

4. Check the contents of the new project:

   Expand *MyHaaSCDSapp* > *db* > *src* and look for `.hdiconfig` which provides a list of all supported database artifacts and a corresponding deployment HDI plug-in.

   In the *SAP HANA Project Explorer*, check the following details:

   1. The new project has been added to your workspace.
   2. In the *SAP HANA Project Explorer,* the UI element *Database Connections* contains the entry *hdi_db*.
   3. The UI elements *Database Connections* and *hdi_db* are both green (enabled).

   > **→ Tip**
   >
   > The status enabled indicates that the database module is bound to the deploy service and the SAP HDI container that is used to store design-time and run-time artifacts.

5. Set up the SAP HANA Database Explorer.

   The SAP HANA Database Explorer enables you to view both the design-time artifacts and the corresponding run-time objects that you deploy to the SAP HDI container assigned to the SAP HANA database application.

   a. Open the command palette.
   b. Locate and run *SAP HANA: Open Database Explorer*

      You might need to provide some login details, for example, an access code.
   c. Add an HDI container to the SAP HANA Database Explorer.

   > **→ Tip**
   >
   > At this point, no containers are available for the new application project. You can complete this step in the next tutorial which shows you how to create a new database application project.

Related Information

Set up a Development Space for SAP HANA CDS in SAP Business Application Studio [page 309]

5.2.1.3 Create an Artifact with the CDS Graphical Editor for SAP Business Application Studio

Use graphical modeling tools to model a CDS artifact, which is a design-time source file that contains definitions of objects that you want to create in the SAP HANA catalog.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.

Context

CDS artifacts are design-time source files that contain DDL code. The code describes a persistence model according to rules defined in Core Data Services. CDS artifacts have the file suffix `.hdbcds`. Building an SAP HANA Database Module that includes this CDS artifact, creates the corresponding catalog objects in the specified schema. To create a CDS artifact, do the following:

Procedure

1. Start SAP Business Application Studio.
2. Display the application project to which you want to add a CDS artifact.

   SAP Web IDE creates an application within a context of a project. If you do not already have a project, see *Related Information* below for instructions on how to create one.
3. Select the database module in which you want to create the CDS artifact.

   In the *Workspace* view, browse to the project's database source folder *db/src*
4. Create the new SAP HANA CDS artifact.
   a. Open the command palette.
      - Press `Crtl` + `Shift` + `P` or
      - Press `F1` or
      - Choose *View* > *Find Command...*
   b. Locate and run the command *SAP HANA: Create HANA Database Artifact*.

In the command palette, type **`data`** and choose *SAP HANA: Create HANA Database Artifact* in the list of commands displayed.

c. Select the database artifact **type**.

In the *Artifact Type* selection box, type **`hdbcds`** and choose *SAP HANA Core Data Services "CDS" document (hdbcds)* in the list that appears.

d. Provide a name for the new SAP HANA CDS artifact.

The appropriate file suffix (`.hdbcds`) is appended to the name by the artifact-creation Wizard.

> → Tip
>
> The file extension is mandatory; it is used to determine which HDI plug-in to call when creating the corresponding run-time object during application build or deployment. SAP HANA CDS artifacts have the file extension `.hdbcds`, for example, `myCDSmodel.hdbcds`.

e. Choose *Create*.

SAP Business Application Studio opens a graphical editor that you can use to define the CDS artifact. You can use the same editor to model CDS entities, contexts, associations, views, user-defined data types, and more within a CDS artifact.

5. (Optional) Create subcontexts.

You can create nested subcontexts within a CDS artifact. Each of these subcontexts can contain any or multiple CDS artifacts such as entities, views, structure types, and more.

a. In the editor toolbar, choose  (Create Context).
b. Drop the context node on the editor.
c. Provide a name for the context.
d. Double-click the context node.

You can define new entities, associations, or another context within the subcontext. Use the breadcrumb navigation in the editor toolbar to switch between contexts.

6. (Optional) Publish CDS artifact as an OData service.

You can publish CDS artifacts at the context level as OData v4 services. You can also publish subcontexts as OData services.

Select the CDS context you want to publish as an OData service and choose  (Publish as OData).

Related Information

Create a New Business Application Project for SAP HANA CDS Development in HaaS [page 311]
Classification of Data Types Supported by SAP HANA CDS Artifacts [page 316]

5.2.1.3.1 Classification of Data Types Supported by SAP HANA CDS Artifacts

You can use the CDS graphical tools to model a single CDS artifact comprised of multiple entities, structure types, and more.

The tool supports multiple data types for defining the data type of an element. For convenience, and based on its characteristics, the data types are classified as follows:

Type	Description
Primitive	Primitive types are standard Data Definition Language (DDL) data types such as String, Binary, or Integer.
Native	Native types are the native SAP HANA data types.
Structure Type	Structure types are user defined data types that comprises of a list of elements, each of which has its own data type.
Scalar Type	Scalar types are user define scalar data types. Unlike user-defined structure types, scalar types do not comprise of any elements in its definition.
Structure Element	Structure elements help reuse the data type of an element defined in a selected structure in another data type definition.
Entity Element	Entity elements help reuse the data types of an element defined in a selected entity in another data type definition.

Related Information

Create an Artifact with the CDS Graphical Editor for SAP Business Application Studio [page 314]

5.2.1.4 Create an Entity with the CDS Graphical Editor in SAP Business Application Studio

Use the CDS graphical editor to model entities. In the SAP HANA database, a CDS entity is a table with a set of data elements that are organized using columns and rows.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

> **Note**
>
> SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.

Context

You model entities within a CDS artifact using the graphical modeling capabilities of the CDS graphical editor.

Procedure

1. Start SAP Business Application Studio.
2. In the *Workspace* view, select the required CDS artifact within which you want to create and model the new entity.
3. In the context menu, choose *Open With* > *Graphical Editor*.

   The tool opens the CDS artifact in a graphical editor where you can model the entity.
4. Create an entity.
   a. In the editor toolbar, choose (Create Entity).
   b. Drop the entity node on the editor.
   c. Provide a name for the entity.
5. Define the entity.

   You can define one or more elements for the entity.
   a. Double-click the CDS entity node.
   b. In the editor toolbar, choose + (Add) to create a new element in the entity.
   c. In the *Name* text field, you can modify the name of the element.
   d. For each element, in the *Type* dropdown list, select the required value.
   e. For each element, in the *Data Type* dropdown list, select the required value.

      If you have selected the type of data type as *Structure Element* or *Entity Element*, then in the *Data Type* dropdown list, select the required structure or entity from within the same CDS artifact. Use the *Type of Element* value help list to select the required element.
   f. Define the length and scale based on the selected data type.
   g. Define whether the specified column is a primary key or part of the primary key for the specified entity. Select the *Key* checkbox accordingly.

      > **Note**
      >
      > Structured columns can be part of the key, too. In this case, all table fields resulting from the flattening of this structured field are part of the primary key.

h. Define if an entity column can (null) or cannot (not null) have the value NULL. If neither null nor not null is specified for the column, the default value null applies (except for the key column). Select the *Not Null* checkbox accordingly.

i. Define the default value for an entity column in the event that no value is provided during an INSERT operation. Enter a value in the *Default* text field.

6. (Optional) Use an expression to generate the element value at runtime.

   The SAP HANA SQL clause `generated always as <expression>` is available for use in CDS entity definitions. This clause specifies the expression that the tool must use to generate the element value at runtime.

   a. Select the required element.

   b. In the *Expression* field, choose  (Element Expression Editor).

   c. In the *Expression* editor, enter the required expression.

      Use the elements and operators from the expression editor to build your expression.

   d. Choose *OK*.

   e. If you want to define the element with a generated expression, select the *Generated* checkbox.

      If you do not select the *Generated* checkbox, the tool considers the expression for the element as calculated expression. Generated elements are physically present in the database table; values are computed on `INSERT` and need not be computed on `SELECT`. Calculated elements are not actually stored in the database table; they are computed when the element is "selected". Since the value of the generated column is computed on `INSERT`, the expression used to generate the value must not contain any non deterministic functions, for example: `current_timestamp, current_user, current_schema` and more.

7. Create a child element, if required.

   For any selected element in the entity, you can create one or more child elements.

   a. Select the required element.

   b. In the editor toolbar, choose  (Create child).

   c. Provide a name for the child element.

   d. Define the data type and default values for the child element.

8. Import elements, if required.

   When defining the elements in an entity, you can also do so by importing elements from other catalog tables. These catalog tables can be available in the same HDI container in which you are creating the entity or a synonym that points to catalog table in another HDI container. To reuse the definition of imported elements, perform the following steps:

   a. In the editor toolbar, choose  (Import Element(s)).

   b. In the *Import Elements* wizard, search and select the required catalog table.

      You can use the elements and its definition from the selected catalog table to define the elements of the entity.

   c. Choose *Next*.

   d. Select the elements you want to import.

   e. Choose *Finish*.

   You can modify the definition of imported elements such as, its data type, length, scale, and more.

> **Note**
>
> The data type of elements that you import from catalog tables, and the data type of the same elements in its design-time entity from which you generated the catalog table, need not be the same. It may vary or be the same depending on the data type defined for the elements.

9. Define additional properties.
   a. In the *Storage Type* dropdown list, select the required storage type value (row or column).

      If no storage type is specified, a "`column`" store table is generated by default.
   b. In the *Unload Priority* text field, specify the required unload priority value for the generated table.

      Unload priority specifies the priority with which a table is unloaded from memory. The priority can be set between 0 (zero) and 9 (nine), where 0 means "cannot be unloaded" and 9 means "earliest unload".
   c. Select the *Auto Merge* checkbox if you want to trigger an automatic delta merge.

      > **Note**
      >
      > Not selecting the *Auto Merge* checkbox disables the automatic delta merge operation.
   d. If you want to specify the grouping information for the generated tables, select the *Enable Table Grouping* checkbox and specify the group name, type, and sub type.
10. (Optional) Specify an identity column.

    The SAP HANA SQL clause `generated as identity` is available for use in CDS entity definitions; if you are using an expression to generate the element value at runtime, this clause enables you to specify an identity column. An element that is defined with `generated as identity` corresponds to a field in the database table that is present in the persistence and has a value that is computed as specified in the sequence options defined in the `identity` expression, for example, `( start with 10 increment by 2 )`.
    a. In the menu bar, select the *Properties* tab.
    b. In the *Element* dropdown list, select an element from the entity that you want to use as identity column.
    c. In the *Type* dropdown list, select a value.

       You can use either *always* or *by default*. If you select *always*, then values are always generated; if you select *by default*, then values are generated by default.
    d. Provide the *Start With* and *Increment by* values.

       For example, `( start with 10 increment by 2 )`.
    e. In the *Minimum Value* and *Maximum Value* dropdown list, select the required value and provide the minimum and maximum values.
    f. In the *Cache* text field, provide a value.

       The cache value is typically the difference of maximum value and minimum value.
    g. In the *Cycle* dropdown list, select a value.

       This value helps the tool determine whether it is a cycling sequence or non cycling sequence.
11. Choose *Save* to save all your changes.

Next Steps

Some business cases may require performing additional tasks on an entity such as defining associations, creating partitions, and more. After creating and defining an entity, you can perform the following additional tasks within the entity.

Requirement	Task to Perform
Define relationship between entities, or between structure types and entities.	Create an Association
Partition the entity using elements in the entity.	Create a Partition
Specify indexes on entities	Create Indexes
Create entities to efficiently store series data	Create Entities to Store Series Data

Related Information

Create an Association with the CDS Graphical Editor in SAP Business Application Studio [page 320]
Create Indexes with the CDS Graphical Editor in SAP Business Application Studio [page 322]
Create a Partition with the CDS Graphical Editor in SAP Business Application Studio [page 323]
Create Entities to Store Series Data in SAP Business Application Studio [page 325]

5.2.1.4.1 Create an Association with the CDS Graphical Editor in SAP Business Application Studio

Associations define relationships between entities, or relationship between structure types and entities. You create associations in either a CDS entity definition or structure type definition, which are design-time files in SAP HANA.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.

- You have already created the CDS entity that you want to add associations to.

Context

The CDS graphical editor tool in SAP Business Application Studio provides you graphical modeling capabilities to model associations between entities, or between structure types and entities. If you are creating associations between entities, then the associations are defined as part of the entity definition, which are design-time files in the repository. Similarly, if you are creating associations between structure types and entities, then the associations are maintained in the structure type definition, which are design-time files in the repository.

Procedure

1. In the *Workspace* view, select the required CDS artifact.
2. In the context menu, choose *Open With* > *Graphical Editor*.
3. Create associations between entities.

   If you are creating associations between entities, you need at least two entities in the CDS artifact.
   a. Select the required entity.
   b. Choose  (Association).
   c. Drag the cursor to another entity in the CDS artifact, with which you want to create associations.
4. Create associations between structure types and entities.

   If you are creating associations between a structure type and an entity, you need at least one structure type and one entity in the CDS artifact. In associations between structure type and entity, the target is an entity. This means that, the association definition is maintained in the structure type definition.
   a. Select the required structure type.
   b. Choose  (Association).
   c. Drag the cursor to an entity in the CDS artifact, with which you want to create associations.
5. Provide association details.

   In the *Association Details* dialog box, provide further details for the association.
   a. In the *Name* text field, provide a name for the association.
   b. Select the association type.

      Select *Managed* or *Unmanaged* association type, depending on whether you want to create managed or unmanaged associations.

      > **i Note**
      >
      > For associations between structure types and entities, you can create only managed associations.

   c. If you are creating managed associations, select required elements from the structure type or entity as association keys.

For managed associations, the relation between source and target entity is defined by specifying a set of elements of the target entity that are used as a foreign key. If no foreign keys are explicitly specified, the elements of the target entity's designated primary key are used. In the *Alias* text field, provide an alias name, if required.

d. If you are creating unmanaged associations, in the expression editor, provide the required condition.

Unmanaged associations are based on existing elements of the source and target entity; no fields are generated. In the `ON` condition, only elements of the source or the target entity can be used; it is not possible to use other associations. The `ON` condition may contain any kind of expression. Use the elements and operators from the editor, when defining the `ON` condition expression.

6. Define cardinality, if required.

When using an association to define a relationship between entities in a CDS artifact, use the cardinality to specify the type of relation, for example, one-to-one (to-one) or one-to-many (to-n); the relationship is with respect to both the source and the target of the association.

a. In the *Source Cardinality* and *Target Cardinality* dropdown lists, select the required cardinality values.

7. Choose *OK*.
8. Choose *Save*.

> **i Note**
>
> You can also create associations using a form-based editor. Double-click the required entity or structure type node in the CDS artifact, and in the Associations tab, create, and define the association.

Related Information

Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]

5.2.1.4.2 Create Indexes with the CDS Graphical Editor in SAP Business Application Studio

You can create indexes for an entity and specify an index type. The tool supports three types of indexes, plain indexes, full text indexes, and fuzzy search indexes.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

> **i Note**
>
> SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created the CDS entity for which you want to create an index.

Procedure

1. In the *Workspace* view, select the required CDS artifact.
2. In the context menu, choose *Open With* > *Graphical Editor*.
3. In the menu bar, choose the *Indexes* tab.
4. Choose *[+]* (Add) to create a new index.
5. In the *General* section, select the required index type.

   After selecting the index type, define which of the elements in the entity should be indexed. For the plain index type, you can also select the index order and use the *Unique* checkbox to define whether the index is unique (no two rows of data in the indexed entity can have identical key values). For *Full Text Index*, select the element and define the full text parameters such as the language column, mime type column, and more.

   > **i Note**
   >
   > You cannot specify both a full-text index and a fuzzy search index for the same element.

6. Choose *Save*.

Related Information

Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]

5.2.1.4.3 Create a Partition with the CDS Graphical Editor in SAP Business Application Studio

Use elements from the entity to partition the entity. You can use hash, range, or round-robin partition types to partition the entity.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i Note**
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created the CDS entity to which you want to add a partition.

Procedure

1. In the SAP Business Application Studio *Workspace* view, right-click the required CDS artifact.
2. In the context menu, choose *Open With* > *Graphical Editor*.
3. Double-click the CDS entity node where you want to create a partition.
4. In the menu bar, choose the *Partitions* tab.
5. Select the *Add Primary Partition* checkbox.
6. In the *Partition Type* dropdown list, select a partition type.

   > **i Note**
   >
   > For *Hash* or *Roundrobin* partition types, you can also define the secondary partition type. In the *Secondary Partition* section, select the required secondary partition type.

7. Choose + (Add) to add an element.
8. In the *Element* dropdown list, select an element, which you want to use to partition the data.
9. In the *Functions* dropdown list, select a value. This is applicable if any of the selected element used for partitioning the entity represents time values.
10. In the *Partitions* text field, enter the number of partitions required.

    > **i Note**
    >
    > If you want to partition the entity based on number of servers, select the *Number of Servers* checkbox

11. Choose *Save*.

Related Information

Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]

5.2.1.4.4 Create Entities to Store Series Data in SAP Business Application Studio

Use the CDS graphical editor in SAP Business Application Studio to create entities that can store series data.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i Note**
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created the CDS entity that you want to use to store series data.

Procedure

1. In the SAP Business Application Studio *Workspace* view, right-click the required CDS artifact.
2. In the context menu, choose *Open With* > *Graphical Editor*.
3. Double-click the CDS entity node where you want to store series data.
4. In the menu bar, choose the *Series* tab.
5. Select the *Enable Series* checkbox.
6. In the *Series Key Element(s)* dropdown list, select one or more elements.
7. If you want to create an entity to store time series data, in the *Period for Series* dropdown list, select the required period column.

   Each row of a series table has a period of validity. The period represents the period in time that the row applies to. When the series table represents instants, then there is a single period column. When the table represents intervals, there can be one or two columns.

   > **i Note**
   >
   > The period for series must be unique and should not be affected by any shift in timestamps. If you want to create an entity to store non time series data, select the *Null* checkbox.

8. Select the required *Equidistant Type*.

Type	Description
Not Equidistant	If you want to create an entity to store arbitrary time values without any regular pattern.

Type	Description
Equidistant	If you are creating an entity to store time values with regular pattern. For equidistant series, provide the *Increment By* value that defines the distance between adjacent elements in an equidistant series. The equidistant series tables offer improved compression compared to non equidistant tables.
Equidistant Precisewise	If you want to create an entity to store series data that retains a degree of regularity without being equidistant across the entire dataset. Equidistant piecewise data consists of regions where equal increments exist between successive points, but these increments are not always consistent for all regions in the table. Different series or different parts of a series may have different intervals. For equidistant precisewise series, you use only one period column, but you can specify one or more *Alternate Period for Series*. These columns can be used to record the time for each row, or represent the end of an interval associated with each row. Good compression can be achieved for these columns when successive values within the series differ by a constant amount.

9. (Optional) Provide minimum and maximum values.

   You can define equidistant and non equidistant series tables with a minimum and/or maximum value. These values are used to verify that loaded data corresponds to the range definition. The values in *Period for Series* column must satisfy the minimum values and the maximum value constraint.

   a. In the *Min Value* dropdown list, select the *Min Value* menu option and provide the required minimum value.
   b. In the *Max Value* dropdown list, select the *Max Value* menu option and provide the required maximum value.

> **i Note**
>
> If the *Period for Series* column does not have a lower limit and upper limit, you can use the *No Min Value* and *No Max Value* respectively.

Related Information

Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]

5.2.1.5 Create a User-Defined Structure Type with the CDS Graphical Editor in SAP Business Application Studio

A structured type is a user-defined data type comprising a list of attributes, each of which has its own data type.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.

Context

Use the CDS graphical editor in SAP Business Application Studio to create a user-defined structured type as a design-time file in SAP HANA. The attributes of the structured type can be defined manually in the structured type itself and reused by another structured type (or by scalar type, or by an entity). You create a user-defined structure types within a CDS artifact.

Procedure

1. Start SAP Business Application Studio in a Web browser.
2. In the project *Workspace* view, right-click the CDS artifact within which you want to model the entity.
3. In the context menu, choose *Open With* > *Graphical Editor*.

   The tool opens the CDS artifact in a graphical editor where you can create the structured type.
4. Create a structured type.
   a. In the editor toolbar, select  (Create Structure).
   b. Drop the structure node on the editor.
   c. Provide a name for the structure.
5. Define the elements of the structured type.
   a. Double-click the structure type node.
   b. In the editor toolbar, choose *[+]* (Add) to create a new element in the structure.

c. In the *Name* field, you can modify the name of the element.
d. For each element, in the *Type* dropdown list, select the required value.
e. For each element, in the *Data Type* dropdown list, select the required value.

   If you have selected the type of data type as *Structure Element* or *Entity Element*, then in the *Data Type* dropdown list, select the required structure or entity in the CDS artifact. Use the *Type of Element* value help list to select the required element.
f. Define the length and scale based on the selected data type.

6. Create a child element, if required.

   For any element in the structure, you can also create one or more child elements.

   a. Select the required element.
   b. In the editor toolbar, choose (Create child).
   c. Define the data type for the child element.

   > **i Note**
   >
   > When you create a child element, the tool automatically changes the type of data type of its parent element to *Inline*.

7. Import elements, if required.

   When defining the elements in an entity, you can also do so by importing elements from other catalog tables. These catalog tables can be available in the same HDI container in which you are creating the entity or a synonym that points to catalog table in another HDI container. To reuse the definition of imported elements,

   a. In the editor toolbar, choose (Import Element(s)).
   b. In the *Import Elements* wizard, search and select the required catalog table.

      You can use the elements and its definition from the selected catalog table to define the elements of the structure type.
   c. Choose *Next*.
   d. Select the elements you want to import.
   e. Choose *Finish*.

   You can modify the definition of imported elements such as its data type, length, scale, and more.

8. In the editor toolbar, choose (Navigate Back).
9. Choose *Save*.

Related Information

Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]
Create an Artifact with the CDS Graphical Editor for SAP Business Application Studio [page 314]
Getting Started wth the CDS Graphical Editor in SAP Business Application Studio [page 308]

5.2.1.6 Create a User-Defined Scalar Type with the CDS Graphical Editor in SAP Business Application Studio

Scalar types are user-defined scalar data types that does not comprise of any elements in its definition.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.

Context

Use the CDS graphical editor in SAP Business Application Studio to create a user-defined structured type as a design-time file in SAP HANA CDS. You create a user-defined scalar data types within a CDS artifact. After creating a scalar type, you can reuse it when defining data types of elements in other structure types, scalar types, or entities.

Procedure

1. Start SAP Business Application Studio in a Web browser.
2. In the project *Workspace* view, select and right-click the CDS artifact within which you want to model the entity.
3. In the context menu, choose *Open With* > *Graphical Editor*.

   SAP Business Application Studio opens the CDS artifact in a graphical editor where you can create the scalar type.
4. Create a scalar type.

   a. In the editor toolbar, select  (Create Scalar type).
   b. Drop the scalar type node on the editor.
   c. Provide a name to the scalar type.
5. Define the scalar type.

   a. Double-click the scalar type node.

b. For each element, in the *Type* dropdown list, select the required value.

   The data types are classified as primitive data types, native SAP HANA data types, user-defined data types (structure types and scalar types), entity element and structure element. Based on your requirement you can select a value from any of the preceding data type classifications.

c. In the *Data Type* dropdown list, select the required value.

   If you have selected the type of data type as *Structure Element* or *Entity Element*, then in the *Data Type* dropdown list, select the required structure or entity in the CDS artifact. Use the *Type of Element* value help list to select the required element.

d. Define the length and scale based on the selected data type.

6. In the editor toolbar, choose (Navigate Back).

Related Information

Create an Artifact with the CDS Graphical Editor for SAP Business Application Studio [page 314]
Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]
Create a User-Defined Structure Type with the CDS Graphical Editor in SAP Business Application Studio [page 327]

5.2.1.7 Create a View with the CDS Graphical Editor in SAP Business Application Studio

Use the modeling capabilities of the CDS graphical editor in SAP Business Application Studio to create a design-time SAP HANA CDS view.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i Note**
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an applicaition project that contains a database module.
- You have already created a CDS artifact.

Context

A view is a virtual table based on the dynamic results returned in response to an SQL statement. You use the graphical modeling tools to create a view with the CDS artifact. After creating the CDS view, use this design-

time view definition to generate the corresponding catalog object when you deploy the application that contains the view-definition artifact (or just the application's database module).

Procedure

1. Start SAP Business Application Studio in a Web browser.
2. In the project *Workspace* view, right-click the CDS artifact within which you want to model a CDS view.
3. In the context menu, choose *Open With* > *Graphical Editor*.

   SAP Business Application Studio opens the CDS artifact in a graphical editor where you can create and define the CDS view.
4. Create a CDS view.

   a. In the editor toolbar, select  (Create View).
   b. Drop the view node on the editor.
   c. Provide a name to the view.
5. Add local CDS artifacts as data sources.

   You can add local CDS artifacts, active CDS artifacts, or both as a data source in a view node. Local CDS artifacts are those entities that exist within the same artifact in which you are creating the CDS view. Active CDS artifacts are those artifacts that are already built and activated.

   a. Double-click the view node.
   b. If you want to add a local CDS artifact, in the editor toolbar, select + (Add Local Objects).
   c. Select the required entities that you want to add as data sources.
   d. Choose *OK*.

   > → Tip
   >
   > The active artifacts could also be synonyms pointing to artifacts in other HDI containers.
6. Add active CDS artifacts as data sources.

   Active CDS artifacts are those artifacts that are already built and activated.

   a. Double-click the view node.
   b. If you want to add an active CDS artifact, in the editor toolbar, select  (Add Active Objects).
   c. In the *Find Data Source* dialog box, search and add activated artifacts from within the same HDI container, or search and add synonyms pointing to artifacts in other HDI containers.

      The active artifacts include built and activated entities (catalog tables), CDS views, calculation views, and SQL views.

      > i Note
      >
      > You cannot use calculation views with input parameters as data sources in CDS views.

   d. In the *Find Data Source* dialog box, search and add activated artifacts from within the same HDI container.

      The active artifacts include built and activated entities (catalog tables), CDS views, calculation views, and SQL views.

> **i Note**
>
> You cannot use calculation views with input parameters as data sources in CDS views.

e. Choose *Finish*.

f. Select the required menu option.

> **i Note**
>
> If you have more than one data source in the CDS view, either create a union, or a join between the data sources.

> **→ Tip**
>
> The active artifacts could also be synonyms pointing to artifacts in other HDI containers.

7. (Optional) Using elements from associated entities.

   You can model a CDS view with entities, which are defined with associations to other entities. In such cases, you can use the elements from the associated entities in the CDS view definition. For using elements from the associated entities, it is not necessary to use associated entities as data sources in the CDS view.

   a. Select the required entity.

   b. Select the association column in the entity.

   c. In the *Select Element(s)* dialog box, select the required elements from the associated entity.

      You can use the elements from the associated entities in the CDS view definition. For example, as output columns, to define SQL clauses, perform SQL `GROUP BY` or `ORDER BY` operations, and more.

   d. (Optional) Define a filter expression.

      In the *Filter Expression* editor, provide a filter expression rule that defines how the elements from the associated entities are used.

      When following an association (for example, in a view), it is possible to apply a filter condition; the filter is merged into the `ON`-condition of the resulting `JOIN`. The following example shows how to get a list of customers and then filter the list according to the sales orders that are currently “open” for each customer. In the example, the infix filter is inserted after the association orders to get only those orders that satisfy the condition `status='open'`.

      > **‹≡› Sample Code**
      >
      > ```
      > view C1 as select from Customer {
      >   name,
      >   orders[status='open'].{id as orderId, date as orderDate}
      > };
      > ```

      Use the elements from the associated entities and the operators provided in the dialog box to build your filter expression.

   e. Choose *OK*.

8. (Optional) Using elements from structure types.

   You can model a CDS view with an entity that has a structure type. In such cases, you can use elements from the structure types in the CDS view definition.

   a. Select the required entity.

   b. Select the structure type element in the entity.

c. In the *Select Element(s)* dialog box, select the required elements from the structure type.

   You can use the elements from the structure types in the CDS view definition. For example, as output columns, to define SQL clauses, perform SQL `GROUP BY` or `ORDER BY` operations, and more.

d. Choose *OK*.

9. Define output columns.
   a. In the CDS view editor, select the required data source.
   b. Select the columns from the data source that you want to add to the output.
   c. In the context menu, choose *Add to Output*.
   d. In the *Select Element(s)* dialog box, provide an alias name to the output column, if required.
   e. Choose *OK*.

      The tool displays the output columns of the view under the *Columns* section (in the details pane) of the editor.

10. (Optional) Group by result set.

    Use one or more columns from the data source to perform a SQL `GROUP BY` operation.

    a. Select the data source.
    b. Select the required columns.
    c. In the context menu, choose *Add to Group By* to perform a SQL `GROUP BY` operation with the selected columns.

       You can also use an output column to perform a SQL `GROUP BY` operation. In the *Columns* section, select an output column, and in the context menu, choose *Add to Group By*.

11. (Optional) Order by result set.

    Use one or more output columns from the view to perform a SQL `ORDER BY` operation.

    a. In the details pane, expand the *Columns* section.
    b. Select the required columns.
    c. In the context menu, choose *Add to Order By* to perform a SQL `ORDER BY` operation with the selected output columns.

12. (Optional) Limit and offset the result set.

    You can use the SQL `LIMIT` clause and the SQL `OFFSET` clause in a CDS view. In the details pane, specify the required limit value and offset value as integers.

    a. In the *Limit* text field, provide the required limit value.

       Use the `LIMIT` clause to restrict the number of result sets to a specified limit.

    b. In the *Offset* text field, provide an integer value required offset value.

       Use the `OFFSET` clause to specify the number of records to skip before displaying the results sets defined by the `LIMIT` SQL clause.

13. (Optional) Create Subqueries.

    You can model CDS views with nested SQL queries (subqueries) to obtain the desired output. Subqueries help model CDS views for complex business scenarios.

    a. Double-click the CDS view.
    b. In the editor toolbar, choose  (Sub Query).
    c. Drop the subquery node on the editor.
    d. Provide a name to the subquery node.
    e. Double click the subquery node.
    f. Define the subquery.

Defining a subquery is similar to defining a view. Within a subquery, you can only creating entities, other subqueries, creating union of entities, creating union of subqueries, and creating union of an entity and a subquery.

> → Tip
>
> You can use the breadcrumb navigation in the editor toolbar to switch between subqueries and the target CDS view.

14. Choose *Save* to save all your changes.
15. Build an SAP HANA Database Module.

    The build process uses the design-time database artifacts to generate the corresponding actual objects in the database catalog. The activation process generates a corresponding catalog object for each of the artifacts defined within another artifact; the location in the catalog is determined by the type of object generated.

    a. From the module context menu, choose *Build*.

Next Steps

Modeling CDS views for complex business scenarios could include layers of calculation logic. In such cases, it may require to perform certain additional tasks to obtain the desired output.

The following table lists some important additional tasks that you can perform to enrich the CDS view.

Requirement	Task to Perform
If you want to query data from two data sources and combine records from both the data sources based on a join condition.	Create Joins
If you want to combine the results of two more data sources.	Create Unions
If you want to create new output columns and calculate its values at runtime using an expression.	Create Calculated Columns
If you want to define relationships between CDS views.	Create Associations for CDS Views with CDS Graphical Editor
If you want to create CDS views with parameters that enable you to pass additional values to modify the results of the CDS view at runtime.	Create Parameters

Related Information

Getting Started wth the CDS Graphical Editor in SAP Business Application Studio [page 308]
Create an SQL WHERE Clause for an SAP HANA CDS View [page 335]
Create Calculated Columns for an SAP HANA CDS View in SAP Business Application Studio [page 337]
Create Joins for an SAP HANA CDS View in SAP Business Application Studio [page 338]
Create Unions for an SAP HANA CDS View in SAP Business Application Studio [page 341]

Create Associations for SAP HANA CDS Views in SAP Business Application Studio [page 342]
Create Parameters for a SAP HANA CDS View in SAP Business Application Studio [page 344]

5.2.1.7.1 Create an SQL WHERE Clause for an SAP HANA CDS View

Define SQL `WHERE` clause in a view to filter and retrieve records from the output of a data source based on a specified condition.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

Use the graphical CDS editor in SAP Business Application Studio to define an SQL `WHERE` clause in an SAP HANA CDS view to filter and retrieve records from the output of a data source based on a specified condition.

Procedure

1. Open the required CDS artifact in the graphical CDS editor.
   a. In the project *Workspace* view, right-click the CDS artifact within which you want to model a CDS view.
   b. In the context menu, choose *Open With* > *Graphical Editor*.
2. Double-click the CDS view in which you want to define the SQL `WHERE` clause.
3. In the *Where* section, choose + (Add Where Clause).
4. In the *Where Clause Editor*, define the SQL `WHERE` clause condition using the required elements and operators.
   You can also use parameters in the expression.
5. Choose *OK*.
6. Choose *Save*.

Related Information

Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]
Create an SQL HAVING Clause for an SAP HANA CDS View in SAP Business Application Studio [page 336]
Create Calculated Columns for an SAP HANA CDS View in SAP Business Application Studio [page 337]

5.2.1.7.2 Create an SQL HAVING Clause for an SAP HANA CDS View in SAP Business Application Studio

Define SQL `HAVING` clause in a view to retrieve records from the output of a data source, only when the aggregate values satisfy a defined condition.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i Note**
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

Use the graphical CDS editor in SAP Business Application Studio to define an SQL `HAVING` clause in an SAP HANA CDS view to retrieve records from the output of a data source, only when the aggregate values satisfy a defined condition.

Procedure

1. Open the required CDS artifact.
   a. In the project *Workspace* view, right-click the CDS artifact within which you want to model a CDS view.
   b. In the context menu, choose *Open With* > *Graphical Editor*.
2. Double-click the CDS view in which you want to define the SQL `HAVING` clause.

> **Note**
>
> You can create SQL `HAVING` clause only if you are using one or more output columns to perform a SQL `GROUP BY` operation on the selected view.

3. In the *Having* section, choose + (Add Having Clause).
4. In the *Having Clause Editor*, define the SQL `HAVING` clause condition using the required elements and operators.

   You can also use parameters in the `HAVING` clause condition.
5. Choose *OK*.
6. Choose *Save*.

Related Information

Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

5.2.1.7.3 Create Calculated Columns for an SAP HANA CDS View in SAP Business Application Studio

Create new output columns for a view and calculate the column values at run time based on the result of an expression.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **Note**
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.
- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

Use the graphical CDS editor in SAP Business Application Studio to define new output columns for a view and calculate the column values at run time based on the result of an expression.

→ Tip

You can use other column values, functions, or constants when defining the expression.

Procedure

1. Open the required CDS artifact.
   a. In the project *Workspace* view, right-click the CDS artifact within which you want to model a CDS view.
   b. In the context menu, choose *Open With* > *Graphical Editor*.
2. Double-click the CDS view in which you want to create calculated columns.
3. In the details pane, select the *Columns* section.
4. Choose *[+]* (Add Calculated Column).
5. Provide a valid expression.
   In the *Add Calculated Column* dialog box, provide the expression that defines the calculated column.
   a. In the *Name* text field, provide a name for the calculated column.
   b. In the expression editor, provide a valid expression using the required elements and operators.
      You can also use parameters in the expression.
   c. Choose *OK*.

Related Information

Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

5.2.1.7.4 Create Joins for an SAP HANA CDS View in SAP Business Application Studio

Create joins in your SAP HANA CDS view to query data from two or more data sources.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

> ℹ Note
>
> SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

You can use the graphical CDS editor to create joins in your SAP HANA CDS view to query data from two or more data sources. You include a `JOIN` clause in the CDS view definition. Joins help query data from two or more data sources.

> → Tip
>
> It helps to limit the number of records or to combine records from both the data sources so that they appear as one record in the query results.

Procedure

1. Open the required CDS artifact.
   a. In the project *Workspace* view, right-click the CDS artifact within which you want to model a CDS view.
   b. In the context menu, choose *Open With* > *Graphical Editor*.
2. Double-click the CDS view in which you want to create a join for data sources.

   > i Note
   >
   > You can create a join only if the CDS view has two or more data sources.
3. Select an entity.
4. Choose  (Join).
5. Drag the cursor to the required data source.
6. Define the join properties.

   In the *Join Definition* dialog box, define the join properties and the join condition.
   a. In the *Type* drop-down list, select the required join type.
   b. In the expression editor, specify a valid join condition using the necessary element and operators.

      Using proposed join condition.

      The tool analyzes the data in the participating entities and proposes a join condition. Based on your requirement, you can create your own join condition or use the join condition that the tool proposes. In the *Join Definition* dialog box, choose *Propose Condition* to use the join condition that the tool proposes.

      > i Note
      >
      > The tool does not propose any join condition for cross joins.

c. Choose *OK*.

Related Information

Join Types Supported by SAP HANA CDS Artifacts [page 340]
Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

5.2.1.7.4.1 Join Types Supported by SAP HANA CDS Artifacts

A list of the join types supported by SAP HANA CDS artifacts.

You can use the CDS graphical editor included in SAP Business Application Studio to create entities, views, and so on. When creating a join between two data sources, you need to specify the join type. The following table lists the types of joins that you can use for SAP HANA CDS modelling in the SAP HANA CDS graphical editor

Join Type	Description
Inner	This join type returns all rows when there is at least one match in both the data sources.
Left Outer	This join type returns all rows from the left data source, and the matched rows from the right data source.
Right Outer	This join type returns all rows from the right data source, and the matched rows from the left data source.
Full Outer	This join type displays results from both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.
Cross Joins	This join type displays results of all possible combinations of rows from the two tables. Cross join is also called a cross product or Cartesian product.

Related Information

Create Joins for an SAP HANA CDS View in SAP Business Application Studio [page 338]

5.2.1.7.5 Create Unions for an SAP HANA CDS View in SAP Business Application Studio

Creating unions help combine the result sets of two or more data sources.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i Note**
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

After modeling a CDS view using the graphical modeling tools, you can include a `UNION` clause in the CDS view definition.

Procedure

1. Open the required CDS artifact.
2. Select the CDS view in which you want to perform the union operation.
3. Double-click the CDS view.

   > **i Note**
   >
   > You can create a union only if the CDS view has two or more data sources.

4. Select a data source.
5. Choose  (Union).
6. Drag the cursor to the required data source.

   This operation creates a *ResultSet* node for each of the data sources.
7. Add more data sources to the union.

   You can perform union operation on multiple data sources (more than two data sources). If you have already created a union of two data sources, then to create a union of these two data sources with the third data source, you use the *ResultSet* node.

a. Select a *ResultSet* node.

   You cannot select the *DefaultResultSet* node to union records of multiple data sources.

b. Choose  (Union).

c. Drag the cursor to the required data source.

8. Add columns to the union output.

   The columns you add to the *DefaultResultSet* node are the output columns of the union.

   a. Select the data source.

   b. Select the columns in the data source you want to add to the output.

   c. In the context menu, choose *Add to Output*.

> **i Note**
>
> Union operation combine result sets of two or more SELECT statements. It is necessary that each result set (SELECT statements) has the same number of columns, have similar data types, and also the columns in each result sets are in same order.

Related Information

Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

5.2.1.7.6 Create Associations for SAP HANA CDS Views in SAP Business Application Studio

Associations define relationships between CDS views, and you create association in the SAP HANA CDS view definition.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

> **i Note**
>
> SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

In a CDS view definition, associations can be used to gather information from the specified target entities. You can use the CDS graphical editor in SAP Business Application Studio to create associations only if the views are **simple** SAP HANA CDS views. The CDS views involved in the association must contain only a single entity as a data source.

Procedure

1. In the *Workspace* view, open the CDS artifact for which you want to define some associations between views.
   a. In the project *Workspace* view, right-click the CDS artifact within which you want to model a CDS view.
   b. In the context menu, choose *Open With* > *Graphical Editor*.
2. Create associations between views.
   a. Select the required CDS view.
   b. Choose  (Association).
   c. Drag the cursor to another CDS view in the CDS artifact with which you want to create the association.
3. Provide association details.
   a. In the *Association Details* dialog box, provide further details for the association.
   b. In the *Name* text field, provide a name for the association.
   c. (Optional) If you want the tool to propose a condition for the association, choose *Propose Condition*.
   d. In the expression editor, provide the required `ON` condition.

      In the `ON` condition, only elements of the source or the target CDS view can be used; it is not possible to use other associations. The `ON` condition may contain any kind of expression. Use the elements and operators from the dialog box, when defining the `ON` condition expression.
4. Define cardinality, if required.
   a. When using an association to define a relationship between CDS views in a CDS artifact, use the cardinality to specify the type of relation, for example, one-to-one (to-one) or one-to-many (to-n); the relationship is with respect to both the source and the target of the association.
   b. In the *Source Cardinality* and *Target Cardinality* dropdown lists, select the required cardinality values.
5. Choose *OK*.
6. Choose *Save*.

Related Information

Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

5.2.1.7.7 Create Parameters for a SAP HANA CDS View in SAP Business Application Studio

Use the CDS graphical editor to define CDS views with parameters that modify the results of the CDS view at runtime.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS view definition.

Context

You can use the CDS graphical editor in SAP Business Application Studio to define SAP HANA CDS views that include parameters. The parameters enable you to pass additional values to modify the results of the CDS view at run time. After creating a parameter, you can use the parameter in the CDS view in the following ways:

- As an output column
- In calculated column expressions
- In the SQL `WHERE` clause
- In the SQL `Group By` clause
- In joins, unions, and more...

Procedure

1. In the *Workspace* view, open the CDS artifact for which you want to define some parameters.
   a. In the project *Workspace* view, right-click the CDS artifact that contains the CDS view you want to modify.
   b. In the context menu, choose *Open With* > *Graphical Editor*.
2. Select the CDS view in which you want to create parameters.
3. Double-click the CDS view.
4. In the details pane, choose + in the *Parameters* section.
5. Define the parameter.

   In the *Parameter Properties* dialog box, define the parameter.

a. In the *Name* text field, provide a name for the parameter.
b. In the *Category* dropdown list, select the required category.
c. In the *Type* dropdown list, select the required data type.

   If you have selected the type of data type as *Structure Element* or *Entity Element*, in the *Type* dropdown list, select the required structure or entity from within the same CDS artifact. Use the *Type Of* value help list to select the required element.
d. Define the *Length* and *Scale* based on the selected data type.
e. Choose *OK*.

6. If you want to add the parameter as an output column of the CDS view, right-click the parameter and choose *Add As Output Column*.

Related Information

Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

5.2.1.8 Preview Output of CDS Views and Entities in SAP Business Application Studio

After modeling and activating a CDS view or an entity, you can preview the output data using the SAP Database Explorer.

Prerequisites

- You have access to SAP Business Application Studio.
- You have a development workspace linked to an SAP HANA Service database.

  > **i** Note
  >
  > SAP HANA CDS is not compatible with SAP HANA Cloud.

- You have an application project that contains a database module.
- You have already created a CDS artifact that contains a CDS entity (or view) definition.

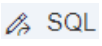
Context

In SAP Business Application Studio's Graphical CDS Editor, you can preview output data of CDS views or entities in simple in tabular format, or you can preview output data in graphical representations, such as bar graphs, area graphs, and pie charts. You can also export and download the output data to `.csv` files. The Database Explorer also allows you to preview the SQL query that the tool generates for an activated CDS artifact.

Procedure

1. Start SAP Business Application Studio in a Web browser.
2. In the project *Workspace* view, right-click a CDS artifact containing the entities and views whose data you want to preview.
3. In the context menu, choose *Open With* > *Graphical Editor*.

   SAP Business Application Studio opens the CDS artifact in a graphical editor.
4. In the editor, select the required view or entity and choose  (Open Data Preview).
5. Apply filters.
   a. For CDS views, if you want to apply filters on columns and view the filtered output data, choose  (Add Filter).
   b. Choose *Add Filters*.
   c. Choose a column and define filter conditions.
6. Export output data, if required.

   If you want to export the raw data output to a .csv file,
   a. In the toolbar, choose  (Download).
   b. In the *Delimiter* dropdown list, select the required delimiter that the tool must use to separate the values in the .csv file.
   c. Choose *Download*.
7. View SQL query for the entity or view, if required.
   a. If you want to view the SQL query that the tool generates for the activated CDS artifact, in the toolbar, choose *SQL*.
   b. In you want to view and execute the SQL query in SQL editor, choose  (Edit SQL Statement in SQL Console).
8. Preview output in graphical representations.

   The tool supports bar graph, area graph, pie chart and table charts, and other graphical representations to preview the output of a CDS view or entity.
   a. In the menu bar, choose *Analysis*.
   b. Configure the axis values by dragging and dropping the required columns to the *Label Axis* and the *Value Axis*.

      The tool displays the output data in graphical representation. Select the required chart icons in the menu to view the output in different graphical representation.
   c. In the menu bar, choose  (Display Settings) to toggle legends and values.

Related Information

Create an Entity with the CDS Graphical Editor in SAP Business Application Studio [page 316]
Create a View with the CDS Graphical Editor in SAP Business Application Studio [page 330]

Important Disclaimer for Features in SAP HANA

For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

THE BEST RUN SAP