PUBLIC
SAP HANA Platform 2.0 SPS 06
Document Version: 1.0 – 2021-12-03

SAP HANA HDBTable Syntax Reference

THE BEST RUN SAP

Content

1 SAP HANA HDBTable Syntax Reference

This guide explains how to use the HDBTable syntax to build design-time data-persistence models in SAP HANA Extended Application Services (SAP HANA XS). The data-persistence model is used to define the data to expose in response to client requests via HTTP, for example, from an SAPUI5-based application.

The *SAP HANA HDBTable Syntax Reference* explains the steps required to develop the design-time data-persistence model for an XS classic application using the HDBTable syntax. The information in the guide is organized as follows:

- Getting started
  On overview of the process of developing applications 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
- Creating the data-persistence model
  Detailed, step-by-step information about defining the data objects in your persistent data model (tables, views, etc.), managing the data model in the SAP HANA repository, activating the data model, managing the resulting objects in the database catalog, and consuming the data model (for example, in a client UI)

Related Information

2 Getting Started with the HDBTable Syntax

HDBTable is a language syntax 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 HDBTable, you can define a persistence model that includes objects such as tables, views, schemas, and sequences; the database objects specify what data to make accessible for consumption by applications and how. This guide takes you through the tasks required to use the HDBTable syntax to define the objects that are most often used in a data persistence model, for example:

- Create a schema
- Create a table (entity)
- Create a table type (reusable table structure)
- Create an SQL view
- Create a sequence

The *SAP HANA HDBTable Syntax Reference* also provides code examples that illustrate how to specify the various object types. This reference guide also includes the complete specification of the HDBTable syntax required for each object type.

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 HDBTable 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 HDBTable artifacts so that the newly created files can be committed to (and synchronized with) the repository.
- The owner of the schema must have SELECT privileges in the schema to be able to see the generated catalog objects.

Related Information

Setting up the Data Persistence Model in SAP HANA [page 9]

2.1   Developing Native SAP HANA Applications

In SAP HANA, native applications use the technology and services provided by the integrated SAP HANA XS platform.

The term "native application" refers to a scenario where applications are developed in the design-time environment provided by SAP HANA extended application services (SAP HANA XS) and use the integrated SAP HANA XS platform illustrated in the following graphic.

> i Note
>
> A program that consists purely of SQLScript is also considered a native SAP HANA application.

The server-centric approach to native application development envisaged for SAP HANA assumes the following high-level scenario:

- All application artifacts are stored in the SAP HANA repository
- Server-side procedural logic is defined in server-side (XS) JavaScript or SQLScript
- UI rendering occurs completely in the client (browser, mobile applications)

Each of the levels illustrated in the graphic is manifested in a particular technology and dedicated languages:
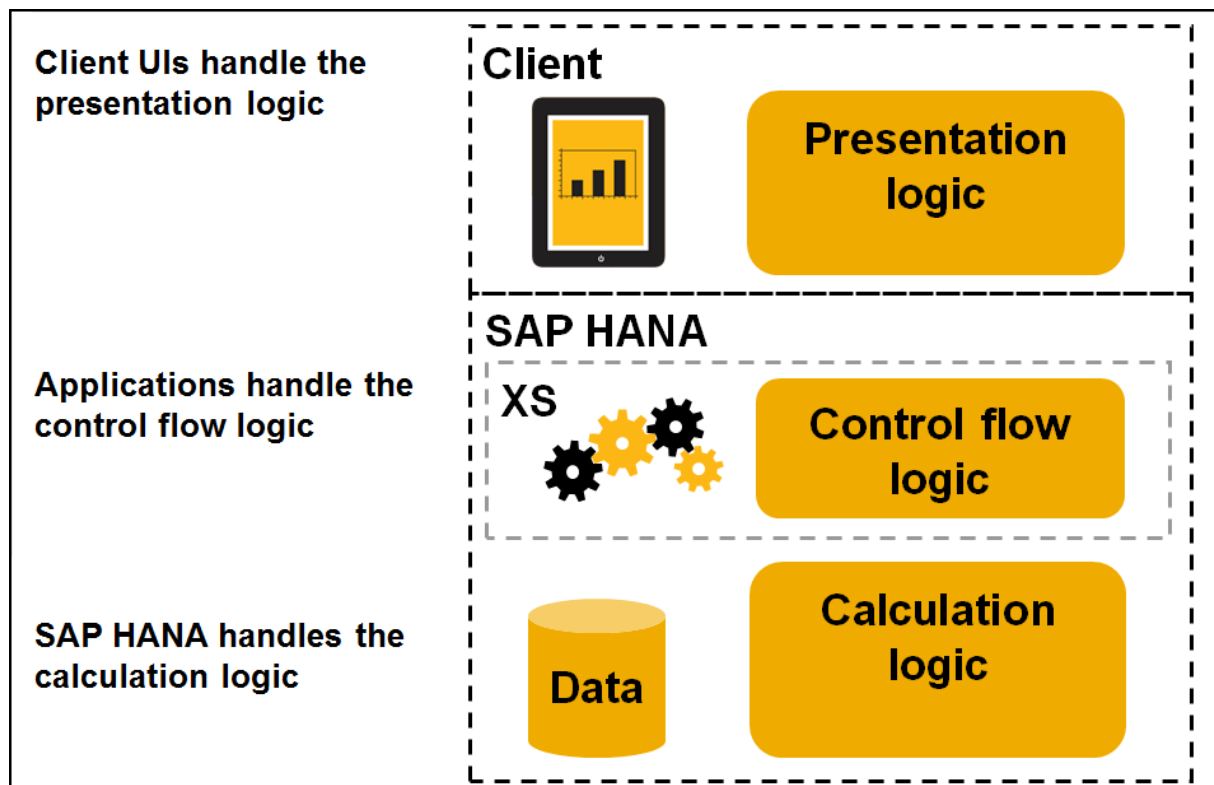
Native SAP HANA Application Development with SAP HANA XS

- Calculation Logic - data-processing technology:
  - Data:
    SQL / SQLScript, Core Data Services (CDS), DDL, HDBtable
  - SQL / SQLScript
  - Calculation Engine Functions (CE_*)

    > **i Note**
    >
    > SAP recommends you use SQL rather than the Calculation Engine functions.

  - Application Function Library (AFL)
- Control-flow logic with SAP HANA XS:
  - OData
    Validation models for OData services can be written in XS JavaScript or SQLScript
  - Server-Side JavaScript (XSJS)
    HTTP requests are implemented directly in XS JavaScript
  - XMLA
- Client UI/Front-end technology:
  - HTML5 / SAPUI5
  - Client-side JavaScript

The development scenarios for native application development are aimed at the following broadly defined audiences:

Target Development Audience for Native SAP HANA Applications

Audience	Language	Tools	Development Artifacts
Database developers	SQLScript, CDS, hdb* SAP	• SAP HANA studio • SAP HANA Web-based Workbench	Database tables, views, procedures; user-defined functions (UDF) and triggers; analytic objects; data authorization...
Application developers: • Professional (XS JS) • Casual/business	XS JavaScript, OData, SQLScript, ...	• SAP HANA studio • SAP HANA Web-based Workbench	Control-flow logic, data services, calculation logic...
UI/client developers	SAPUI5, JavaScript, ...	• SAP HANA studio • SAP HANA Web-based Workbench	UI shell, navigation, themes (look/feel), controls, events, ...

Related Information

2.2 Roles and Permissions for XS Development

An overview of the authorizations required to develop database artifacts for SAP HANA using the CDS syntax.

To enable application-developers to start building native applications that take advantage of the SAP HANA Extended Application Services (SAP HANA XS), the SAP HANA administrator must ensure that developers have access to the tools and objects that they need to perform the tasks required during the application- and database-development process.

Before you start developing applications using the features and tools provided by the SAP HANA XS, bear in mind the following prerequisites. Developers who want to build applications to run on SAP HANA XS need the following tools, accounts, and privileges:

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

> i Note
>
> The required privileges can only be granted by someone who has the necessary authorizations in SAP HANA, for example, an SAP HANA administrator.

SAP HANA XS Classic Model

To develop database artifacts for use by applications running in the SAP HANA XS classic environment, bear in mind the following prerequisites:

- Access to a running SAP HANA development system (with SAP HANA XS classic)
- A valid user account in the SAP HANA database on that system
- Access to development tools, for example, provided in:
  - SAP HANA studio
  - SAP HANA Web-based Development Workbench
- Access to the SAP HANA repository
- Access to selected run-time catalog objects

> i Note
>
> To provide access to the repository for application developers, you can use a predefined role or create your own custom role to which you assign the privileges that the application developers need to perform the everyday tasks associated with the application-development process.

To provide access to the repository from the SAP HANA studio, the EXECUTE privilege is required for SYS.REPOSITORY_REST, the database procedure through with the REST API is tunneled. To enable the activation and data preview of information views, the technical user _SYS_REPO also requires SELECT privilege on all schemas where source tables reside.

In SAP HANA, you can use roles to assign one or more privileges to a user according to the area in which the user works; the role defines the privileges the user is granted. For example, a role enables you to assign SQL

privileges, analytic privileges, system privileges, package privileges, and so on. To create and maintain artifacts in the SAP HANA repository, you can assign application-development users the following roles:

- One of the following:
  - MODELING
    The predefined MODELING role assigns wide-ranging SQL privileges, for example, on _SYS_BI and _SYS_BIC. It also assigns the analytic privilege _SYS_BI_CP_ALL, and some system privileges. If these permissions are more than your development team requires, you can create your own role with a set of privileges designed to meet the needs of the application-development team.
  - Custom DEVELOPMENT role
    A user with the appropriate authorization can create a custom DEVELOPMENT role specially for application developers. The new role would specify only those privileges an application-developer needs to perform the everyday tasks associated with application development, for example: maintaining packages in the repository, executing SQL statements, displaying data previews for views, and so on.
- PUBLIC
  This is a role that is assigned to all users by default.

Before you start using the SAP HANA Web-based Development Workbench, the SAP HANA administrator must set up a user account for you in the database and assign the required developer roles to the new user account.

> → Tip
>
> The role *sap.hana.xs.ide.roles::Developer* grants the privileges required to use **all** the tools included in the *SAP HANA Web-based Development Workbench*. However, to enable a developer to use the debugging features of the browser-based IDE, your administrator must also assign the role *sap.hana.xs.debugger::Debugger*. In addition, the section *debugger* with the parameter *enabled* and the value *true* must be added to the file `xsengine.ini` file, for example, in the SAP HANA studio *Administration* perspective.

SAP HANA XS Advanced Model

To develop database artifacts for use by applications running in the SAP HANA XS **advanced** environment, bear in mind the following prerequisites:

- Access to a running SAP HANA development system (with SAP HANA XS advanced)
- A valid user account in the SAP HANA database on that system
- Access to development tools, for example, provided in:
  - SAP Web IDE for SAP HANA
  - SAP HANA Run-time Tools (included in the SAP Web IDE for SAP HANA)

> i Note
>
> To provide access to tools and for application developers in XS advanced, you define a custom role to which you add the privileges required to perform the everyday tasks associated with the application- and database-development process. The role is then assigned to a role collection which is, in turn, assigned to the developer.

- Access to the SAP HANA XS advanced design-time workspace and repository

- Access to selected run-time catalog objects
- Access to the XS command-line interface (CLI); the XS CLI client needs to be downloaded and installed

Related Information

[Create a Design-Time Role](#)
[Assign Repository Package Privileges](#)
[SAP HANA Web-Based Development Workbench](#)

2.3 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 9]](#)
- [SAP HANA XS Advanced Model [page 10]](#)

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.

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]
[Creating Data Persistence Artifacts with CDS in XS Advanced]
[CDS Documents in XS Advanced]

2.4 The SAP HANA Developer's Information Atlas

The information **atlas** is a collection of interactive maps that are designed to help SAP HANA developers find the information they need for a particular task quickly and easily.

> → Tip
>
> The maps in this guide help you locate information about the most common development tasks; the guide is designed to help you navigate the SAP HANA documentation landscape by filtering the information available for the SAP HANA developer according to different **perspectives**, for example, the development **platform** you are working with, the development **process** you are using, or a particular developer **persona** or **scenario**.
>
> This guide is not designed to be read from end-to-end. To find the information you want, choose the perspective that best suits your development needs (for example, process, platform, persona, or scenario) and use the interactive maps and links provided for your chosen perspective to guide you to your information-related destination.
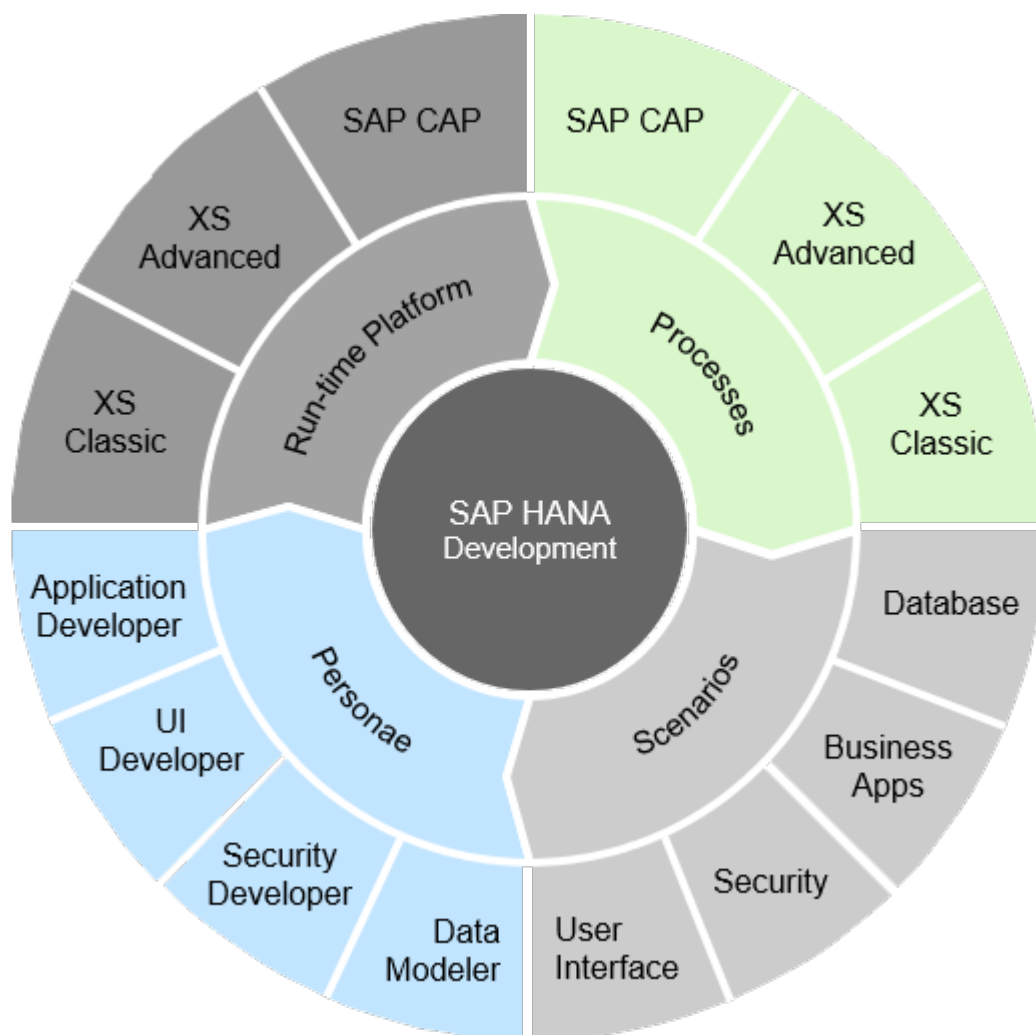
Application Development in SAP HANA

> **→ Tip**
>
> For information about where to find more maps and guides, see *Related Information* below.

Related Information

The SAP HANA Developer Center
SAP HANA Academy
Open online courses by SAP

2.4.1 SAP HANA Information Map: Development Guides

The design and organization of the SAP HANA developer library makes it easy to use the name of a guide to find the relevant information.

The SAP HANA developer library includes a selection of guides that describe the complete application-development process, from defining user roles, privileges, and data models through application setup to UI design and testing; the information available covers background and concepts, task-based tutorials, and detailed reference material.

The SAP HANA developer library also includes a selection of reference guides that describe the various languages that application developers can use to define the underlying data model (SQL, CDS), the application business logic (Java, JavaScript, Python), or the client interface (SAPIU5).

The high-level steps required to complete the process of developing an application for SAP HANA extended application services depend on the target SAP HANA XS run-time platform to which you want to deploy the application, for example, XS classic or XS advanced. The listed topics include graphics with links to much more information about each of the steps in the development process for the following application-development platforms:

- The XS Advanced Application-Development Process
- The XS Classic Application-Development Process
- The SAP Cloud Application Programming Model (CAP)

Related Information

SAP HANA Application Development Guides
SAP HANA Developer Language Reference Guides

2.4.2 SAP HANA Information Map: Development Tasks

The design and organization of the SAP HANA developer documentation library enables easy access to information according to the particular development task to be performed, for example, creating a view or procedure, or setting up an application project.

The SAP HANA developer can make use of a large number of guides that include information describing the complete application-development process. There is also a huge amount of reference material available, for example, describing the SQL and SQLScript language and syntax or the options available for coding Core Data Services (CDS).

The topics in this section include graphics that make it easy to find information about how to perform a specific task in a particular development area, for example, setting up the persistence model; creating an XS JavaScript or OData service, building client user interfaces, or managing the development life cycle. The graphics are interactive; you can use them to jump directly to the information describing the task you want to perform. The tasks are split according to the following development areas:

- Common database-development tasks
  Build data-persistence and analytic models; create procedures, synonyms, and user-defined functions (UDF), etc.
- SAP HANA Analytic-Modeling Development Tasks
  Create artifacts for use in the analytic model, for example, calculation views, analytic privileges, hierarchies, and so on.
- Common application-development tasks
  Create OData services; build Java, JavaScript, Python, or other custom-language modules; maintain application business-logic code; set up security artifacts, etc.
- Client (front-end) user-interface development tasks
  Build SAPUI5 applications; create UI views; bind data to UI event handlers; consume data and services with SAPUI5, etc.

Related Information

SAP HANA Database-Artifact Development Tasks
SAP HANA Analytic-Modeling Development Tasks
SAP HANA XS Application Artifact-Development Tasks
SAP HANA XS Client User-Interface-Artifact Development Tasks

2.4.3 SAP HANA Information Map: Development Scenarios

The SAP HANA developer documentation library enables easy access to information according to the underlying development scenario, for example, database development or coding the application business-logic.

The SAP HANA developer can make use of a large number of guides that include information describing the complete application-development process from the perspective of the development scenario, for example, database development, application development, client UI design and testing, or security considerations; the information available covers background and concepts, task-based tutorials, and detailed reference material.

> → Tip
>
> The particular scenario you select can be based on the underlying development area you are assigned to, the choice of programing language, the required development objects, or the tools you want to use.

The information in this section indicates where to find information based on the development scenario you choose, for example:

- Database development
- Application business-logic development
- Client user-interface design
- Application security configuration and maintenance

Related Information

SAP HANA Database-Development Scenarios
SAP HANA Application-Development Scenarios
SAP HANA Client User-Interface Development Scenarios
SAP HANA Application Security Scenarios

2.4.3.1 Database Development Scenarios

The focus of the database developer is primarily on the underlying 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. The following table lists some of the tasks typically performed by the database developer and indicates where to find the information that is required to perform the task.

Typical Database-Development Tasks

Task	Details	Information Source
Create tables, SQL views, sequences...	Code, syntax, ...	*SAP HANA SQLScript Reference*
		SAP HANA SQL and System Views Reference
		SAP HANA Developer Guide for SAP HANA Studio
	Packaging, activation, implementation, ...	*SAP HANA Developer Guide for SAP HANA Studio*
Create attribute, analytic, calculation views	Code, syntax, ...	*SAP HANA SQLScript Reference*
		SAP HANA Developer Guide for SAP HANA Studio

Task	Details	Information Source
	Packaging, activation, implementation, ...	*SAP HANA Developer Guide for SAP HANA Studio*
	Examples, background	*SAP HANA Modeling Guide for SAP HANA Studio*
Create/Write SQLScript procedures, UDFs, triggers...	Code, syntax, ...	*SAP HANA SQLScript Reference*
		SAP HANA SQL and System Views Reference
		SAP HANA Developer Guide for SAP HANA Studio
	Packaging, activation, implementation, ...	*SAP HANA Developer Guide for SAP HANA Studio*
Create/Use application functions	Code, syntax, ...	*SAP HANA SQLScript Reference*
		SAP HANA Business Function Library (BFL) (*)
		SAP HANA Predictive Analysis Library (PAL) (*)
		SAP HANA Developer Guide for SAP HANA Studio
	Packaging, activation, implementation, ...	*SAP HANA Developer Guide for SAP HANA Studio*

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

3 Creating the Persistence Model with HDBTable

HDBTable is a language syntax that can be used to define a design-time representation of the artifacts that comprise the persistent data models in SAP HANA.

In SAP HANA Extended Application Services (SAP HANA XS), the persistence model defines the schema, tables, and views that specify what data to make accessible and how. 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.

SAP HANA XS enables you to create database schema, tables, views, and sequences as design-time files in the repository. Repository files can be read by applications that you develop.

> **i Note**
>
> 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.

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.

As part of the process of setting up the basic persistence model for SAP HANA XS, you perform the following tasks:

Task	Description
Create a schema	Define a design-time schema and maintain the schema definition in the repository; the transportable schema has the file extension `.hdbschema`, for example, `MYSCHEMA.hdbschema`.
Create a synonym	Define a design-time synonym and maintain the synonym definition in the repository; the transportable synonym has the file extension `.hdbsynonym`, for example, `MySynonym.hdbsynonym`.
Create a table	Define a design-time table and maintain the table definition in the repository; the transportable table has the file extension `.hdbtable`, for example, `MYTABLE.hdbtable`
Create a reusable table structure	Define the structure of a database table in a design-time file in the repository; you can reuse the table-structure definition to specify the table **type** when creating a new table.
Create a view	Define a design-time view and maintain the view definition in the repository; the transportable view has the file extension `.hdbview`, for example, `MYVIEW.hdbview`
Create a sequence	Define a design-time sequence and maintain the sequence definition in the repository; the transportable sequence has the file extension `.hdbsequence`, for example, `MYSEQUENCE.hdbsequence`
Import table content	Define data-provisioning rules that enable you to import data from comma-separated values (CSV) files into SAP HANA tables using the SAP HANA XS table-import feature; the complete configuration can be included in a delivery unit and transported between SAP HANA systems.

> **i Note**
>
> On activation of a repository file, the file suffix, for example, `.hdbview`, `.hdbschema`, or `.hdbtable`, is used to determine which runtime plug-in to call during the activation process. The plug-in reads the repository file selected for activation, for example, a table, sees the object descriptions in the file, and creates the appropriate runtime object.

Related Information

3.1 Create a Schema

A schema defines the container that holds database objects such as tables, views, and stored procedures.

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.

Context

This task describes how to create a file containing a schema definition using the `hdbschema` syntax. Schema definition files are stored in the SAP HANA repository.

> **i Note**
>
> A schema generated from an `.hdbschema` artifact can also be used in the context of Core Data Services (CDS).

To create a schema 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 schema definition file.

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

   a. Right-click the folder where you want to save the schema-definition file and choose *New>Schema* in the context-sensitive popup menu.

   b. Enter or select the parent folder.

   c. Enter the name of the schema in the *File Name* field.

   > → 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.

   d. Select a template to use. Templates contain sample source code to help you.

   e. Choose *Finish* to save the new schema in the repository.

5. Define the schema name.

   To edit the schema file, in the *Project Explorer* view double-click the schema file you created in the previous step, for example, `MYSCHEMA.hdbschema`, and add the schema-definition code to the file:

   > i Note
   >
   > The following code example is provided for illustration purposes only.

   ```
   schema_name="MYSCHEMA";
   ```

6. Save the schema file.

   > 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 schema.

   a. Locate and right-click the new schema file in the *Project Explorer* view.

   b. In the context-sensitive pop-up menu, choose ▶ *Team* ▶ *Activate* ◀.

8. Grant SELECT privileges to the owner of the new schema.

   After activation in the repository, the 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 in the SAP HANA studio's *Modeler* perspective, you must grant the user the required SELECT privilege.

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

3.1.1 Schema

Relational databases contain a catalog that describes the various elements in the system. The catalog divides the database into sub-databases known as schema. A database schema enables you to logically group together objects such as tables, views, and stored procedures. Without a defined schema, you cannot write to the catalog.

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database schema as a transportable design-time file in the repository. Repository files can be read by applications that you develop.

If your application refers to the repository (design-time) version of a schema 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 schema.

If you want to define a transportable schema using the design-time `hdbschema` specifications, use the configuration schema illustrated in the following example:

```
string schema_name
```

The following example shows the contents of a valid transportable schema-definition file for a schema called `MYSCHEMA`:

```
schema_name="MYSCHEMA";
```

The schema is stored in the repository with the schema name `MYSCHEMA` as the file name and the suffix `.hdbschema`, for example, `MYSCHEMA.hdbschema`.

> **i Note**
>
> A schema generated from an `.hdbschema` artifact can also be used in the context of Core Data Services (CDS).

Schema Activation

If you want to create a schema definition as a design-time object, you must create the schema as a flat file. You save the file containing the schema definition with the suffix `.hdbschema` in the appropriate package for your application in the SAP HANA repository. You can activate the design-time objects at any point in time.

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

If you activate a schema-definition object in SAP HANA, the activation process checks if a schema with the same name already exists in the SAP HANA repository. If a schema with the specified name does not exist, the repository creates a schema with the specified name and makes _SYS_REPO the owner of the new schema.

> **i Note**
>
> The schema cannot be dropped even if the deletion of a schema object is activated.

If you define a schema in SAP HANA XS, note the following important points regarding the schema name:

- Name mapping
  The schema name must be identical to the name of the corresponding repository object.
- Naming conventions
  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.
- Name usage
  The Data Definition Language (DDL) rendered by the repository contains the schema name as a delimited identifier.

Related Information

[Create a Schema](#)

3.2 Create a Table

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database table 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 definition `MYSCHEMA.hdbschema`

Context

This task describes how to create a file containing a table definition using the `hdbtable` syntax. Table definition files are stored in the SAP HANA repository. To create a table 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 table definition file.

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

   a. Right-click the folder where you want to save the table file and choose *New* >*Database Table* in the context-sensitive popup menu.
   b. Enter or select the parent folder.
   c. Enter the name of the table in the *File Name* box.

   > → 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.

     d. Select a template to use. Templates contain sample source code to help you.

     e. Choose *Finish* to save the new table definition file.

5. Define the table.

To edit the table definition, in the *Project Explorer* view double-click the table-definition file you created in the previous step, for example, `MYTABLE.hdbtable`, and add the table-definition code to the file:

> ### i Note
>
> The following code example is provided for illustration purposes only.

```
table.schemaName = "MYSCHEMA";
table.tableType = COLUMNSTORE;
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment
= "dummy comment";},
    {name = "Col2"; sqlType = INTEGER; nullable = false;},
    {name = "Col3"; sqlType = NVARCHAR; nullable = true; length = 20;
defaultValue = "Defaultvalue";},
    {name = "Col4"; sqlType = DECIMAL; nullable = false; precision = 2; scale
= 3;}];
table.indexes =  [
    {name = "MYINDEX1"; unique = true; indexColumns = ["Col2"];},
    {name = "MYINDEX2"; unique = true; indexColumns = ["Col1", "Col4"];}];
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

6. Save the table-definition file.

> ### 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 changes in the repository.

     a. Locate and right-click the new table file in the *Project Explorer* view.

     b. In the context-sensitive pop-up menu, choose ▌ *Team* ❯ *Activate* ▌.

Related Information

3.2.1 Tables

In the SAP HANA database, as in other relational databases, a table is a set of data elements that are organized using columns and rows. A database table has a specified number of columns, defined at the time of table creation, but can have any number of rows. Database tables also typically have meta-data associated with them; the meta-data might include constraints on the table or on the values within particular columns.

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

> **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.

If your application is configured to use the design-time version of a database table in the repository rather than the runtime version in the catalog, any changes to the repository version of the table 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 table.

If you want to define a transportable table using the design-time `.hdbtable` specifications, use the configuration schema illustrated in the following example:

```
struct TableDefinition {
    string SchemaName;
    optional bool temporary;
    optional TableType tableType;
    optional bool public;
    optional TableLoggingType loggingType;
    list<ColumnDefinition> columns;
    optional list<IndexDefinition> indexes;
    optional PrimaryKeyDefinition primaryKey;
    optional string description
};
```

The following code illustrates a simple example of a design-time table definition:

```
table.schemaName = "MYSCHEMA";
table.tableType = COLUMNSTORE;
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment =
"dummy comment";},
    {name = "Col2"; sqlType = INTEGER; nullable = false;},
    {name = "Col3"; sqlType = NVARCHAR; nullable = true; length = 20;
defaultValue = "Defaultvalue";},
    {name = "Col4"; sqlType = DECIMAL; nullable = false; precision = 2; scale =
3;}];
table.indexes =  [
    {name = "MYINDEX1"; unique = true; order = DSC; indexColumns = ["Col2"];},
    {name = "MYINDEX2"; unique = true; order = DSC; indexColumns = ["Col1",
"Col4"];}];
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

If you want to create a database table as a repository file, you must create the table as a flat file and save the file containing the table dimensions with the suffix `.hdbtable`, for example, `MYTABLE.hdbtable`. The new file is located in the package hierarchy you establish in the SAP HANA repository. You can activate the repository files at any point in time.

> **i Note**
>
> On activation of a repository file, the file suffix, for example, `.hdbtable`, 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 table, parses the object descriptions in the file, and creates the appropriate runtime objects.

Security Considerations

It is important to bear in mind that an incorrectly defined table can lead to security-related problems. If the content of the table you create is used to determine the behavior of the application, for example, whether data is displayed depends on the content of a certain cell, any modification of the table content could help an attacker to obtain elevated privileges. Although you can use authorization settings to restrict the disclosure of information, data-modification issues need to be handled as follows:

- Make sure you specify the field type and define a maximum length for the field
- Avoid using generic types such as VARCHAR or BLOB.
- Keep the field length as short as possible; it is much more difficult to inject shell-code into a string that is 5 characters long than one that an can contain up to 255 characters.

Related Information

3.2.2 Table Configuration Syntax

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the `hdbtable` syntax to create a database table as a design-time file in the repository. The design-time artifact that contains the table definition must adhere to the `.hdbtable` syntax specified below.

Table Definition

The following code illustrates a simple example of a design-time table definition using the `.hdbtable` syntax.

> i Note
>
> Keywords are case-sensitive, for example, *tableType* and *loggingType*, and the schema referenced in the table definition, for example, MYSCHEMA, must already exist.

```
table.schemaName = "MYSCHEMA";
table.temporary = true;
table.tableType = COLUMNSTORE;
table.loggingType = NOLOGGING;
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment =
"dummy comment";},
    {name = "Col2"; sqlType = INTEGER; nullable = false;},
    {name = "Col3"; sqlType = NVARCHAR; nullable = true; length = 20;
defaultValue = "Defaultvalue";},
    {name = "Col4"; sqlType = DECIMAL; nullable = false; precision = 2; scale =
3;}];
table.indexes =  [
```

```
    {name = "MYINDEX1"; unique = true; order = DSC; indexColumns = ["Col2"];},
    {name = "MYINDEX2"; unique = true; order = DSC; indexType = B_TREE;
indexColumns = ["Col1", "Col4"];}];
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

Table-Definition Configuration Schema

The following example shows the configuration schema for tables defined using the `.hdbtable` syntax. Each of the entries in the table-definition configuration schema is explained in more detail in a dedicated section below:

```
struct TableDefinition {
    string SchemaName;
    optional bool temporary;
    optional TableType tableType;
    optional bool public;
    optional TableLoggingType loggingType;
    list<ColumnDefinition> columns;
    optional list<IndexDefinition> indexes;
    optional PrimaryKeyDefinition primaryKey;
    optional string description
};
```

Schema Name

To use the `.hdbtable` syntax to specify the name of the schema that contains the table you are defining, use the *schemaName* keyword. In the table definition, the *schemaName* keyword must adhere to the syntax shown in the following example.

```
table.schemaName = "MYSCHEMA";
```

Temporary

To use the `.hdbtable` syntax to specify that the table you define is temporary, use the boolean *temporary* keyword. Since data in a temporary table is session-specific, only the owner session of the temporary table is allowed to INSERT/READ/TRUNCATE the data. Temporary tables exist for the duration of the session, and data from the local temporary table is automatically dropped when the session is terminated. In the table definition, the *temporary* keyword must adhere to the syntax shown in the following example.

```
table.temporary = true;
```

Table Type

To specify the table type using the `.hdbtable` syntax, use the *tableType* keyword. In the table definition, the *TableType* keyword must adhere to the syntax shown in the following example.

```
table.tableType = [COLUMNSTORE | ROWSTORE];
```

The following configuration schema illustrates the parameters you can specify with the *tableType* keyword:

- `COLUMNSTORE`
  Column-oriented storage, where entries of a column are stored in contiguous memory locations. SAP HANA is particularly optimized for column-order storage.
- `ROWSTORE`
  Row-oriented storage, where data is stored in a table as a sequence of records

Table Logging Type

To enable logging in a table definition using the `.hdbtable` syntax, use the *tableLoggingType* keyword. In the table definition, the *tableLoggingType* keyword must adhere to the syntax shown in the following example.

```
table.tableLoggingType = [LOGGING | NOLOGGING];
```

Table Column Definition

To define the column structure and type in a table definition using the `.hdbtable` syntax, use the *columns* keyword. In the table definition, the *columns* keyword must adhere to the syntax shown in the following example.

```
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment =
"dummy comment";},
    {name = "Col2"; sqlType = INTEGER; nullable = false;},
    {name = "Col3"; sqlType = NVARCHAR; nullable = true; length = 20;
defaultValue = "Defaultvalue";},
    {name = "Col4"; sqlType = DECIMAL; nullable = false; precision = 2; scale =
3;}];
```

The following configuration schema illustrates the parameters you can specify with the *columns* keyword:

```
struct ColumnDefinition {
    string name;
    SqlDataType sqlType;
    optional bool nullable;
    optional bool unique;
    optional int32 length;
    optional int32 scale;
    optional int32 precision;
    optional string defaultValue;
    optional string comment;
};
```

SQL Data Type

To define the SQL data type for a column in a table using the `.hdbtable` syntax, use the *sqlType* keyword. In the table definition, the *sqlType* keyword must adhere to the syntax shown in the following example.

```
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment =
"dummy comment";},
    ...
    ];
```

The following configuration schema illustrates the data types you can specify with the *sqlType* keyword:

```
enum SqlDataType {
    DATE; TIME; TIMESTAMP; SECONDDATE; INTEGER; TINYINT;
    SMALLINT; BIGINT; REAL; DOUBLE; FLOAT; SMALLDECIMAL;
    DECIMAL; VARCHAR; NVARCHAR; CLOB; NCLOB;
    ALPHANUM; TEXT; SHORTTEXT; BLOB; VARBINARY;
};
```

Primary Key Definition

To define the primary key for the specified table using the `.hdbtable` syntax, use the *primaryKey* and *pkcolumns* keywords. In the table definition, the *primaryKey* and *pkcolumns* keywords must adhere to the syntax shown in the following example.

```
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

The following configuration schema illustrates the parameters you can specify with the *primaryKey* keyword:

```
struct PrimaryKeyDefinition {
    list<string> pkcolumns;
    optional IndexType indexType;
};
```

Table Index Definition

To define the index for the specified table using the `.hdbtable` syntax, use the *indexes* keyword. In the table definition, the *indexes* keyword must adhere to the syntax shown in the following example.

```
table.indexes =  [
    {name = "MYINDEX1"; unique = true; order = DSC; indexColumns = ["Col2"];},
    {name = "MYINDEX2"; unique = true; order = DSC; indexColumns = ["Col1",
"Col4"];}];
```

You can also use the optional parameter `indexType` to define the type of index, for example, `B_TREE` or `CPB_TREE`, as described in Table Index Type [page 29].

Table Index Type

To define the index type for the specified table using the `.hdbtable` syntax, use the *indexType* keyword. In the table definition, the *indexType* keyword must adhere to the syntax shown in the following example.

```
indexType = [B_TREE | CPB_TREE];
```

`B_TREE` specifies an index tree of type *B+*, which maintains sorted data that performs the insertion, deletion, and search of records. CPB_TREE stands for "Compressed Prefix B_TREE" and specifies an index tree of type *CPB+*, which is based on pkB-tree. CPB_TREE is a very small index that uses a "partial key", that is; a key that is only part of a full key in index nodes.

> **i Note**
>
> If neither the *B_TREE* nor the *CPB_TREE* type is specified in the table-definition file, SAP HANA chooses the appropriate index type based on the column data type, as follows:
>
> - CPB_TREE
>   Character string types, binary string types, decimal types, when the constraint is a composite key or a non-unique constraint
> - B_TREE
>   All column data types other than those specified for CPB_TREE

Table Index Order

To define the order of the table index using the `.hdbtable` syntax, use the *order* keyword. Insert the *order* with the desired value (for example, ascending or descending) in the index type definition; the *order* keyword must adhere to the syntax shown in the following example.

```
order = [ASC | DSC];
```

You can choose to filter the contents of the table index either in ascending (ASC) or descending (DSC) order.

Complete Table-Definition Configuration Schema

The following example shows the complete configuration schema for tables defined using the `.hdbtable` syntax.

```
enum TableType {
    COLUMNSTORE; ROWSTORE;
};
enum TableLoggingType {
    LOGGING; NOLOGGING;
};
enum IndexType {
    B_TREE; CPB_TREE;
};
enum Order {
    ASC; DSC;
```

```
};
enum SqlDataType {
    DATE; TIME; TIMESTAMP; SECONDDATE;
    INTEGER; TINYINT; SMALLINT; BIGINT;
    REAL; DOUBLE; FLOAT; SMALLDECIMAL; DECIMAL;
    VARCHAR; NVARCHAR; CLOB; NCLOB;
    ALPHANUM; TEXT; SHORTTEXT; BLOB; VARBINARY;
};
struct PrimaryKeyDefinition {
    list<string> pkcolumns;
    optional IndexType indexType;
};
struct IndexDefinition {
    string name;
    bool unique;
    optional Order order;
    optional IndexType indexType;
    list<string> indexColumns;
};
struct ColumnDefinition {
    string name;
    SqlDataType sqlType;
    optional bool nullable;
    optional bool unique;
    optional int32 length;
    optional int32 scale;
    optional int32 precision;
    optional string defaultValue;
    optional string comment;
};
struct TableDefinition {
    string schemaName;
    optional bool temporary;
    optional TableType tableType;
    optional bool public;
    optional TableLoggingType loggingType;
    list<ColumnDefinition> columns;
    optional list<IndexDefinition> indexes;
    optional PrimaryKeyDefinition primaryKey;
    optional string description;
};
TableDefinition table;
```

Related Information

3.3 Create a Reusable Table Structure

SAP HANA Extended Application Services (SAP HANA XS) enables you to define the structure of a database table in a design-time file in the repository. You can reuse the table-structure definition to specify the table **type** when creating a new table.

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 definition `MYSCHEMA.hdbschema`

Context

This task describes how to create a file containing a table-structure definition using the `hdbstructure` syntax. Table-structure definition files are stored in the SAP HANA repository with the `.hdbstructure` file extension, for example, `TableStructure.hdbstructure`. The primary use case for a design-time representation of a table structure is creating reusable type definitions for procedure interfaces. To create a table-structure 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 a folder (package) to hold the table-structure definition files.

   Browse to the folder (package) in your project workspace where you want to create the new folder (package), and perform the following steps:

   a. In the *Project Explorer* view, right-click the folder where you want to create a new folder called `Structures`, and choose ▎ *New* ❭ *Folder* ▎ in the context-sensitive popup menu.

   b. Enter a name for the new folder in the *Folder Name* box, for example, **Structures**.

   c. Choose *Finish* to create the new `Structures` folder.

5. Create the table-structure definition file.

   Browse to the `Structures` folder (package) in your project workspace and perform the following steps:

a. In the *Project Explorer* view, right-click the `Structures` folder you created in the previous step and choose ▶ *New* ▶ *File* ▶ in the context-sensitive popup menu.

b. Enter a name for the new table-structure in the *File Name* box and add the `.hdbstructure` file extension, for example, `TableStructure.hdbstructure`.

> → 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.

c. Choose *Finish* to save the new table-structure definition file.

6. Define the table structure.

To edit the table-structure definition file, in the *Project Explorer* view double-click the table file you created in the previous step, for example, `TableStructure.hdbstructure`, and add the table-structure code to the file:

> i Note
>
> The following code example is provided for illustration purposes only.

```
table.schemaName = "MYSCHEMA";
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment
= "dummy comment";},
    {name = "Col2"; sqlType = INTEGER; nullable = false;},
    {name = "Col3"; sqlType = NVARCHAR; nullable = true; length = 20;
defaultValue = "Defaultvalue";},
    {name = "Col4"; sqlType = DECIMAL; nullable = false; precision = 12;
scale = 3;}];
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

7. Save the table-structure definition file.

> 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.

8. Activate the changes in the repository.

You can activate the changes to the folder structure and the folder contents in one step.

a. In the *Project Explorer* view, locate and right-click the new folder (`Structures`) that contains the new table-structure definition file `TableStructure.hdbstructure`.

b. In the context-sensitive pop-up menu, choose ▶ *Team* ▶ *Activate* ▶.

Activating a table-definition called `TableStructure.hdbstructure` in the package `Structures` creates a new table type in SAP HANA, in the same way as the following SQL statement:

```
CREATE TABLE "MySchema"."MyTypeTable" like
"MySchema"."Structures::TableStructure"
```

9. Check that the new table-type object `Structures::TableStructure` is added to the catalog.

You can find the new table type in the *Systems* view under ▶ *Catalog* ▷ *MYSCHEMA* ▷ *Procedures* ▷ *Table Types* ▶.

a. In the *SAP HANA Development* perspective, open the *Systems* view.

b. Select the SAP HANA System where the new is located and navigate to the following node: ▶ *Catalog* ▷ *MYSCHEMA* ▷ *Procedures* ▷ *Table Types* ▶

c. Right-click the new table-structure object and choose *Open Definition* to display the specifications for the reusable table-structure in the details panel.

d. Check that the entry in the *Type* box is *Table Type*.

Related Information

3.3.1 Reusable Table Structures

A table-structure definition is a template that you can reuse as a basis for creating new tables of the same type and structure. You can reference the table structure in an SQL statement (`CREATE TABLE [...] like [...]`) or an SQLScript procedure.

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database table structure (or type) as a design-time file in the repository. All repository files including your table-structure definition can be transported to other SAP HANA systems, for example, in a delivery unit. The primary use case for a design-time representation of a table structure is creating reusable table-type definitions for procedure interfaces. However, you an also use table-type definitions in table user-defined fuctions (UDF).

If you want to define a design-time representation of a table structure with the `.hdbstructure` specifications, use the configuration schema illustrated in the following example:

```
struct TableDefinition {
    string SchemaName;
    optional bool public;
    list<ColumnDefinition> columns;
    optional PrimaryKeyDefinition primaryKey;
};
```

> ℹ Note
>
> The `.hdbstructure` syntax is a subset of the syntax used in `.hdbtable`. In a table **structure** definition, you cannot specify the table type (for example, COLUMN/ROW), define the index, or enable logging.

The following code illustrates a simple example of a design-time table-structure definition:

```
table.schemaName = "MYSCHEMA";
table.columns = [
    {name = "Col1"; sqlType = VARCHAR; nullable = false; length = 20; comment =
"dummy comment";},
```

```
    {name = "Col2"; sqlType = INTEGER; nullable = false;},
    {name = "Col3"; sqlType = NVARCHAR; nullable = true; length = 20;
defaultValue = "Defaultvalue";},
    {name = "Col4"; sqlType = DECIMAL; nullable = false; precision = 2; scale =
3;}];
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

If you want to create a database table structure as a repository file, you must create the table structure as a flat file and save the file containing the structure definition with the `.hdbstructure` file extension, for example, `TableStructure.hdbstructure`. The new file is located in the package hierarchy you establish in the SAP HANA repository. You can activate the repository files at any point in time.

> ### i Note
>
> On activation of a repository file, the file suffix 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 table structure element with the file extension `.hdbstructure`, parses the object descriptions in the file, and creates the appropriate runtime objects.

You can use the SQL command `CREATE TABLE` to create a new table based on the table structure, for example, with the `like` operator, as illustrated in the following example:

```
CREATE TABLE "MySchema"."MyTypeTable" like
"MySchema"."Structures::TableStructure"
```

Related Information

3.4   Create a Sequence

A database sequence generates a serial list of unique numbers that you can use while transforming and moving data between systems.

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 definition, for example, `MYSCHEMA.hdbschema`

Context

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database sequence as a design-time file in the repository. This task describes how to create a file containing a sequence definition using the `hdbsequence` syntax.

> **i Note**
>
> A schema generated from an `.hdbsequence` artifact can also be used in the context of Core Data Services (CDS).

To create a sequence-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 sequence definition file.

   Browse to the folder in your project workspace where you want to create the new sequence definition file and perform the following tasks:

   a. Right-click the folder where you want to save the sequence definition file and choose *New* > *Sequence Definition* in the context-sensitive popup menu.

   b. Enter or select the parent folder.

   c. Enter the name of the sequence in the *File Name* box.

      In SAP HANA, sequence-definition files require the file extension `.hdbsequence`, for example, `MySequence.hdbsequence`.

      > **→ 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.

   d. Select a template to use. Templates contain sample source code to help you.

   e. Choose *Finish* to save the new sequence in the repository.

5. Define the sequence properties.

   To edit the sequence file, in the *Project Explorer* view double-click the sequence file you created in the previous step, for example, `MYSEQUENCE.hdbsequence`, and add the sequence code to the file:

   ```
   schema= "MYSCHEMA";
   start_with= 10;
   maxvalue= 30;
   ```

```
nomaxvalue=false;
minvalue= 1;
nominvalue=true;
cycles= false;
reset_by= "SELECT T1.\"Column2\" FROM \"MYSCHEMA\".
\"com.acme.test.tables::MY_TABLE1\" AS T1 LEFT JOIN \"MYSCHEMA\".
\"com.acme.test.tables::MY_TABLE2\" AS T2 ON T1.\"Column1\" = T2.\"Column1\"";
depends_on=["com.acme.test.tables::MY_TABLE1",
"com.acme.test.tables::MY_TABLE2"];
```

6. Save the sequence-definition file.

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

Related Information

3.4.1 Sequences

A sequence is a database object that generates an automatically incremented list of numeric values according to the rules defined in the sequence specification. The sequence of numeric values is generated in an ascending or descending order at a defined increment interval, and the numbers generated by a sequence can be used by applications, for example, to identify the rows and columns of a table.

Sequences are not associated with tables; they are used by applications, which can use CURRVAL in a SQL statement to get the current value generated by a sequence and NEXTVAL to generate the next value in the defined sequence. Sequences provide an easy way to generate the unique values that applications use, for example, to identify a table row or a field. In the sequence specification, you can set options that control the start and end point of the sequence, the size of the increment size, or the minimum and maximum allowed value. You can also specify if the sequence should recycle when it reaches the maximum value specified. The relationship between sequences and tables is controlled by the application. Applications can reference a sequence object and coordinate the values across multiple rows and tables.

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database sequence as a transportable design-time file in the repository. Repository files can be read by applications that you develop.

You can use database sequences to perform the following operations:

* Generate unique, primary key values, for example, to identify the rows and columns of a table

- Coordinate keys across multiple rows or tables

The following example shows the contents of a valid sequence-definition file for a sequence called MYSEQUENCE. Note that, in this example, no increment value is defined, so the default value of 1 (ascend by 1) is assumed. To set a descending sequence of 1, set the *increment_by* value to -1.

```
schema= "TEST_DUMMY";
start_with= 10;
maxvalue= 30;
nomaxvalue=false;
minvalue= 1;
nominvalue=true;
cycles= false;
reset_by= "SELECT T1.\"Column2\" FROM \"MYSCHEMA\".
\"com.acme.test.tables::MY_TABLE1\" AS T1 LEFT JOIN \"MYSCHEMA\".
\"com.acme.test.tables::MY_TABLE2\" AS T2 ON T1.\"Column1\" = T2.\"Column1\"";
depends_on=["com.acme.test.tables::MY_TABLE1",
"com.acme.test.tables::MY_TABLE2"];
```

The sequence definition is stored in the repository with the suffix `hdbsequence`, for example, `MYSEQUENCE.hdbsequence`.

> **i Note**
>
> A schema generated from an `.hdbsequence` artifact can also be used in the context of Core Data Services (CDS).

If you activate a sequence-definition object in SAP HANA XS, the activation process checks if a sequence with the same name already exists in the SAP HANA repository. If a sequence with the specified name does not exist, the repository creates a sequence with the specified name and makes `_SYS_REPO` the owner of the new sequence.

In a sequence defined using the `.hdbsequence` syntax, the `reset_by` keyword enables you to reset the sequence using a query on any view, table or even table function. However, any dependency must be declared explicitly, for example, with the `depends_on` keyword. The target table or view specified in the `depends_on` keyword **must** be mentioned in the `SELECT` query that defines the reset condition. If the table or view specified in the dependency does not exist, the activation of the object in the repository fails.

> **i Note**
>
> On initial activation of the sequence definition, no check is performed to establish the existence of the target view (or table) in the dependency; such a check is only made on **reactivation** of the sequence definition.

Security Considerations

It is important to bear in mind that an incorrectly defined sequences can lead to security-related problems. For example, if the sequencing process becomes corrupted, it can result in data overwrite. This can happen if the index has a maximum value which rolls-over, or if a defined reset condition is triggered unexpectedly. A roll-over can be achieved by an attacker forcing data to be inserted by flooding the system with requests.

Overwriting log tables is a known practice for deleting traces. To prevent unexpected data overwrite, use the following settings:

- `cycles= false`
- Avoid using the `reset_by` feature

Related Information

3.4.2 Sequence Configuration Syntax

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the `hdbsequence` syntax to create a database sequence as a design-time file in the repository. The design-time artifact that contains the sequence definition must adhere to the `.hdbsequence` syntax specified below.

Sequence Definition

The following code illustrates a simple example of a design-time sequence definition using the `.hdbsequence` syntax.

> **i Note**
>
> Keywords are case-sensitive, for example, *maxvalue* and *start_with*, and the schema referenced in the table definition, for example, `MYSCHEMA`, must already exist.

```
schema= "MYSCHEMA";
start_with= 10;
maxvalue= 30;
nomaxvalue= false;
minvalue= 1;
nominvalue= true;
cycles= false;
reset_by= "SELECT T1.\"Column2\" FROM \"MYSCHEMA\".
\"com.acme.test.tables::MY_TABLE1\" AS T1 LEFT JOIN \"MYSCHEMA\".
\"com.acme.test.tables::MY_TABLE2\" AS T2 ON T1.\"Column1\" = T2.\"Column1\"";
depends_on= ["com.acme.test.tables::MY_TABLE1",
"com.acme.test.tables::MY_TABLE2"];
```

Sequence-Definition Configuration Schema

The following example shows the configuration schema for sequences defined using the `.hdbsequence` syntax. Each of the entries in the sequence-definition configuration schema is explained in more detail in a dedicated section below:

```
string schema;
int32 increment_by(default=1);
int32 start_with(default=-1);
optional int32 maxvalue;
bool nomaxvalue(default=false);
optional int32 minvalue;
bool nominvalue(default=false);
optional bool cycles;
optional string reset_by;
bool public(default=false);
optional string depends_on_table;
optional string depends_on_view;
optional list<string> depends_on;
```

Schema Name

To use the `.hdbsequence` syntax to specify the name of the schema that contains the sequence you are defining, use the `schema` keyword. In the sequence definition, the `schema` keyword must adhere to the syntax shown in the following example.

```
schema= "MYSCHEMA";
```

Increment Value

To use the `.hdbsequence` syntax to specify that the sequence increments by a defined value, use the `increment_by` keyword. `increment_by` specifies the amount by which the next sequence value is incremented from the last value assigned. The default increment is 1. In the sequence definition, the `increment_by` keyword must adhere to the syntax shown in the following example.

```
increment_by= 2;
```

To generate a descending sequence, specify a negative value.

> i Note
>
> An error is returned if the `increment_by` value is 0.

Start Value

To use the `.hdbsequence` syntax to specify that the sequence starts with a specific value, use the `start_with` keyword. If you do not specify a value for the `start_with` keyword, the value defined in

`minvalue` is used for ascending sequences, and value defined in `maxvalue` is used for descending sequences. In the sequence definition, the `start_with` keyword must adhere to the syntax shown in the following example.

```
start_with= 10;
```

Maximum Value

To use the `.hdbsequence` syntax to specify that the sequence stops at a specific **maximum** value, for example, 30, use the optional keyword `maxvalue`. In the sequence definition, the `maxvalue` keyword must adhere to the syntax shown in the following example.

```
maxvalue= 30;
```

> ℹ Note
>
> The maximum value (`maxvalue`) a sequence can generate must be between -4611686018427387903 and 4611686018427387902.

No Maximum Value

To use the `.hdbsequence` syntax to specify that the sequence does not stop at any specific **maximum** value, use the boolean keyword `nomaxvalue`. When the `nomaxvalue` keyword is used, the maximum value for an **ascending** sequence is 4611686018427387903 and the maximum value for a **descending** sequence is -1. In the sequence definition, the `nomaxvalue` keyword must adhere to the syntax shown in the following example.

```
nomaxvalue= true;
```

> ℹ Note
>
> Note that the default setting for `nomaxvalue` is `false`.

Minimum Value

To use the `.hdbsequence` syntax to specify that the sequence stops at a specific **minimum** value, for example, 1, use the `minvalue` keyword. In the sequence definition, the `minvalue` keyword must adhere to the syntax shown in the following example.

```
minvalue= 1;
```

> **i Note**
>
> The minimum value (`minvalue`) a sequence can generate must be between -4611686018427387903 and 4611686018427387902.

No Minimum Value

To use the `.hdbsequence` syntax to specify that the sequence does not stop at any specific **minimum** value, use the boolean keyword `nominvalue`. When the `nominvalue` keyword is used, the minimum value for an **ascending** sequence is 1 and the minimum value for a **descending** sequence is -4611686018427387903. In the sequence definition, the `nominvalue` keyword must adhere to the syntax shown in the following example.

```
nominvalue= true;
```

> **i Note**
>
> Note that the default setting `nominvalue` is `false`.

Cycles

In a sequence defined using the `.hdbsequence` syntax, the optional boolean keyword `cycles` enables you to specify whether the sequence number will be restarted after it reaches its maximum or minimum value. For example, the sequence restarts with `minvalue` after having reached `maxvalue` (where `increment_by` is greater than zero (0)) or restarts with `maxvalue` after having reached `minvalue` (where `increment_by` is less than zero (0)). In the `.hdbsequence` definition, the `cycles` keyword must adhere to the syntax shown in the following example.

```
cycles= false;
```

Reset by Query

In a sequence defined using the `.hdbsequence` syntax, the `reset_by` keyword enables you to reset the sequence using a query on any view, table or even table function. However, any dependency must be declared explicitly, for example, with the `depends_on_view` or `depends_on_table` keyword. If the table or view specified in the dependency does not exist, the activation of the sequence object in the repository fails.

In the `.hdbsequence` definition, the `reset_by` keyword must adhere to the syntax shown in the following example.

```
reset_by= "SELECT \"Col2\" FROM \"MYSCHEMA\".\"acme.com.test.tables::MY_TABLE\"
WHERE \"Col2\"='12'";
```

During a restart of the database, the system automatically executes the `reset_by` statement and the sequence value is restarted with the value determined from the `reset_by` subquery

> **i Note**
>
> If `reset_by` is not specified, the sequence value is stored persistently in the database. During the restart of the database, the next value of the sequence is generated from the saved sequence value.

Depends on

In a sequence defined using the `.hdbsequence` syntax, the optional keyword `depends_on` enables you to define a dependency to one or more specific tables or views, for example when using the `reset_by` option to specify the query to use when resetting the sequence. In the `.hdbsequence` definition, the `depends_on` keyword must adhere to the syntax shown in the following example.

```
depends_on=
["<repository.package.path>::<MY_TABLE_NAME1>","<repository.package.path>::<MY_VI
EW_NAME1>"];
```

> **i Note**
>
> The `depends_on` keyword replaces and extends the keywords `depends_on_table` and `depends_on_view`.

For example, to specify multiple tables and views with the `depends_on` keyword, use a comma-separated list enclosed in square brackets [].

```
depends_on= ["com.acme.test.tables::MY_TABLE1",
"com.acme.test.tables::MY_TABLE2", "com.acme.test.views::MY_VIEW1"];
```

The target table or view specified in the `depends_on` keyword **must** be mentioned in the `SELECT` query that defines the reset condition. On initial activation of the sequence definition, no check is performed to establish the existence of the target table or view specified in the dependency; such a check is only made during **reactivation** of the sequence definition. If one or more of the target tables or views specified in the dependency does not exist, the re-activation of the sequence object in the repository fails.

Related Information

3.5 Create an SQL View

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 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 definition, for example, `MYSCHEMA.hdbschema`

Context

This task describes how to create a file containing an SQL view definition using the `hdbview` syntax. SQL view-definition files are stored in the SAP HANA repository. To create an SQL view-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 view definition file.

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

   a. Right-click the folder where you want to save the view-definition file and choose *New* in the context-sensitive popup menu.

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

      > → 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.

c. Select a template to use. Templates contain sample source code to help you.

d. Choose *Finish* to save the new view-definition file in the repository.

5. Define the view.

   If the new view-definition file is not automatically displayed by the file-creation wizard, in the *Project Explorer* view double-click the view-definition file you created in the previous step, for example, `MYVIEW.hdbview`, and add the view definition code to the file replacing object names and paths to suit your requirements.:

   > i Note
   >
   > The following code example is provided for illustration purposes only.

   ```
   schema="MYSCHEMA";
   query="SELECT T1.\"Column2\" FROM \"MYSCHEMA\".
   \"acme.com.test.views::MY_VIEW1\" AS T1 LEFT JOIN \"MYSCHEMA\".
   \"acme.com.test.views::MY_VIEW2\" AS T2 ON T1.\"Column1\" = T2.\"Column1\"";
   depends_on=["acme.com.test.views::MY_VIEW1", "acme.com.test.views::MY_VIEW2"];
   ```

6. Save the SQL view-definition file.

   > 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 changes in the repository.

   a. Locate and right-click the new view-definition file in the *Project Explorer* view.

   b. In the context-sensitive pop-up menu, choose ▶ *Team* ❯ *Activate* ◀.

Related Information

3.5.1 SQL Views

In SQL, a view is a virtual table based on the dynamic results returned in response to an SQL statement. Every time a user queries an SQL view, the database uses the view's SQL statement to recreate the data specified in the SQL view. The data displayed in an SQL view can be extracted from one or more database tables.

An SQL view contains rows and columns, just like a real database table; the fields in an SQL view are fields from one or more real tables in the database. You can add SQL functions, for example, WHERE or JOIN statements, to a view and present the resulting data as if it were coming from one, single table.

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a database view as a 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.

The following example shows the contents of a valid transportable view-definition file for a view called MYVIEW:

```
schema="MYSCHEMA";
query="SELECT T1.\"Column2\" FROM \"MYSCHEMA\".\"acme.com.test.views::MY_VIEW1\"
AS T1 LEFT JOIN \"MYSCHEMA\".\"acme.com.test.views::MY_VIEW2\" AS T2 ON
T1.\"Column1\" = T2.\"Column1\"";
depends_on=["acme.com.test.views::MY_VIEW1", "acme.com.test.views::MY_VIEW2"];
```

If you want to create a view definition as a design-time object, you must create the view as a flat file and save the file containing the view definition with the suffix .hdbview, for example, MYVIEW.hdbview in the appropriate package in the package hierarchy established for your application in the SAP HANA repository. You can activate the design-time object at any point in time.

> → Tip
>
> On activation of a repository file, the file suffix (for example, .hdbview) is used to determine which runtime plugin to call during the activation process. The plug-in reads the repository file selected for activation, parses the object descriptions in the file, and creates the appropriate runtime objects.

In an SQL view defined using the .hdbview syntax, any dependency to another table or view must be declared explicitly, for example, with the depends_on keyword. The target view or table specified in the depends_on keyword **must** also be mentioned in the SELECT query that defines the SQL view. If one of more of the tables or views specified in the dependency does not exist, the activation of the object in the repository fails.

> i Note
>
> On initial activation of the SQL view, no check is performed to establish the existence of the target view (or table) in the depends_on dependency; such a check is only made on **reactivation** of the SQL view.

Column Names in a View

If you want to assign names to the columns in a view, use the SQL query in the .hdbview file. In this example of design-time view definition, the following names are specified for columns defined in the view:

- idea_id
- identity_id
- role_id

```
schema = "MYSCHEMA";
query = "SELECT role_join.idea_id AS idea_id, ident.member_id AS identity_id,
role_join.role_id AS role_id
            FROM \"acme.com.odin.db.iam::t_identity_group_member_transitive\" AS
ident
            INNER JOIN \"acme.com.odin.db.idea::t_idea_identity_role\" AS
role_join
```

```
              ON role_join.identity_id = ident.group_id UNION DISTINCT
        SELECT idea_id, identity_id, role_id
        FROM \"acme.com.odin.db.idea::t_idea_identity_role\"
        WITH read only";
```

Related Information

3.5.2 SQL View Configuration Syntax

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the `hdbview` syntax to create an SQL view as a design-time file in the repository. The design-time artifact that contains the SQL view definition must adhere to the `.hdbview` syntax specified below.

SQL View Definition

The following code illustrates a simple example of a design-time definition of an SQL view using the `.hdbview` syntax.

> **i Note**
>
> Keywords are case-sensitive, for example, `schema` and `query`, and the schema referenced in the table definition, for example, `MYSCHEMA`, must already exist.

```
schema="MYSCHEMA";
public=false
query="SELECT T1.\"Column2\" FROM \"MYSCHEMA\".
\"acme.com.test.tables::MY_TABLE1\" AS T1 LEFT JOIN \"MYSCHEMA\".
\"acme.com.test.views::MY_VIEW1\" AS T2 ON T1.\"Column1\" = T2.\"Column1\"";
depends_on= "acme.com.test.tables::MY_TABLE1","acme.com.test.views::MY_VIEW1";
```

SQL View Configuration Schema

The following example shows the configuration schema for an SQL view that you define using the `.hdbview` syntax. Each of the entries in the view-definition configuration schema is explained in more detail in a dedicated section below:

```
string schema;
string query;
bool public(default=true);
optional list<string> depends_on_table;
optional list<string> depends_on_view;
```

Schema Name

To use the `.hdbview` syntax to specify the name of the schema that contains the SQL view you are defining, use the `schema` keyword. In the SQL view definition, the `schema` keyword must adhere to the syntax shown in the following example.

```
schema= "MYSCHEMA";
```

query

To use the `.hdbview` syntax to specify the query that creates the SQL view you are defining, use the `query` keyword. In the SQL view definition, the `query` keyword must adhere to the syntax shown in the following example.

```
query="SELECT * FROM \"<MY_SCHEMA_NAME>\".
\"<repository.package.path>::<MY_TABLE_NAME>\"";
```

For example:

```
query="SELECT * FROM \"MY_SCHEMA\".\"com.test.tables::02_HDB_DEPARTMENT_VIEW\"";
```

public

To use the `.hdbview` syntax to specify whether or not the SQL view you are defining is publicly available, use the boolean keyword `public`. In the SQL view definition, the `public` keyword must adhere to the syntax shown in the following example.

```
public=[false|true];
```

For example:

```
public=false
```

> i Note
>
> The default value for the `public` keyword is `true`.

Depends on

In an SQL view defined using the `.hdbview` syntax, the optional keyword `depends_on` enables you to define a dependency to one or more tables or views. In the `.hdbview` definition, the `depends_on` keyword must adhere to the syntax shown in the following example.

```
depends_on=
["<repository.package.path>::<MY_TABLE_NAME1>","<repository.package.path>::<MY_VI
EW_NAME1>"];
```

> **i Note**
>
> The `depends_on` keyword replaces and extends the keywords `depends_on_table` and `depends_on_view`.

For example, to specify multiple tables and views with the `depends_on` keyword, use a comma-separated list enclosed in square brackets [].

```
depends_on= ["acme.com.test.tables::MY_TABLE1","acme.com.test.views::MY_VIEW1"];
```

The target table or view specified in the `depends_on` keyword **must** be mentioned in the `SELECT` query that defines the SQL view. On initial activation of the SQL view, no check is performed to establish the existence of the target tables or views specified in the dependency; such a check is only made during **reactivation** of the SQL view. If one or more of the target tables or views specified in the dependency does not exist, the re-activation of the SQL view object in the repository fails.

Related Information

3.6 Create a Synonym

Extended Application Services (SAP HANA XS) enables you to create a local database synonym 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.
- (SAP HANA studio only) You must have shared the project so that the newly created files can be committed to (and synchronized with) the repository.

Context

In SAP HANA, a design-time synonym artifact has the suffix `.hdbsynonym` and defines the target object by specifying an authoring schema and an object name; its activation evaluates a system's schema mapping to determine the physical schema in which the target table is expected, and creates a local synonym that points to this object.

> ! Restriction
>
> A design-time synonym cannot refer to another synonym, and you cannot define multiple synonyms in a single design-time synonym artifact. In addition, the target object specified in a design-time synonym must only exist in the catalog; it is not possible to use `.hdbsynonym` to define a synonym for a catalog object that originates from a design-time artifact.

Procedure

1. Start the SAP HANA studio.
   a. Open the *SAP HANA Development* perspective.
   b. Open the *Project Explorer* view.
2. Create the synonym definition file.

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

   To generate a synonym called `"acme.com.app1::MySynonym1"`, you must create a design-time synonym artifact called `MySynonym1.hdbsynonym` in the repository package `acme.com.app1`; the first line of the design-time synonym artifact must be specified as illustrated in the following example.

   > ⥂ Sample Code
   > ```
   > { "acme.com.app1::MySynonym1" : {...}}
   > ```

   a. Right-click the folder where you want to create the synonym-definition file and choose ▷ *New* ❯ *General* ❯ *File* ◁ in the context-sensitive popup menu.
   b. Enter the name of the new synonym-definition file in the *File Name* box and add the appropriate extension, for example, `MySynonym1.hdbsynonym`.
   c. Choose *Finish* to save the new synonym definition file.
3. Define the synonym.

   To edit the synonym definition, in the *Project Explorer* view double-click the synonym-definition file you created in the previous step, for example, `MySynonym.hdbsynonym`, and add the synonym-definition code to the new file, as illustrated in the following example.

   > i Note
   >
   > The following code example is provided for illustration purposes only.

> 📑 Sample Code

```
{ "acme.com.app1::MySynonym1" : {
      "target" : {
            "schema": "DEFAULT_SCHEMA",
            "object": "MY_ERP_TABLE_1"
                  },
      "schema": "SCHEMA_2"
      }
}
```

4. Save and activate the changes in the repository.

   a. Locate and right-click the new synonym-definition file in the *Project Explorer* view.

   b. In the context-sensitive pop-up menu, choose ▶ *Team* ▶ *Activate* ▶.

Related Information

3.6.1 Synonyms

SAP HANA Extended Application Services (SAP HANA XS) enables you to create a design-time representation of a **local** database synonym. The synonym enables you to refer to a table (for example, from a view) that only exists as a catalog object.

In SAP HANA XS, a design-time representation of a local synonym has the suffix `.hdbsynonym` that you can store in the SAP HANA repository. The syntax of the design-time synonym artifact requires you to define the target object by specifying an authoring schema and an object name. You also need to specify the schema in which to create the new synonym. On activation of a design-time synonym artifact, SAP HANA XS evaluates a system's schema mapping to determine the physical schema in which the target table is expected, and creates a local synonym in the specified schema which points to this object. You can use this type of synonym if you need to define a CDS view that refers to a table which only exists in the catalog; that is, the catalog table has no design-time representation.

> ⚠ Restriction
>
> A synonym cannot refer to another synonym, and you cannot define multiple synonyms in a single design-time synonym artifact. In addition, the target object specified in a design-time synonym must only exist in the catalog; it is not possible to define a define-time synonym for a catalog object that originates from a design-time artifact.

In the following example of a design-time synonym artifact, the table `MY_ERP_TABLE_1` resides in the schema `DEFAULT_SCHEMA`. The activation of the design-time synonym artifact illustrated in the example would generate a local synonym (`"acme.com.app1::MySynonym1"`) in the schema SCHEMA_2. Assuming that a schema-mapping table exists that maps DEFAULT_SCHEMA to the schema SAP_SCHEMA, the newly

generated synonym "SCHEMA_2"."acme.com.app1::MySynonym1" points to the run-time object "SAP_SCHEMA"."MY_ERP_TABLE_1".

> **≡, Sample Code**
>
> MySynonym1.hdbsynonym
>
> ```
> { "acme.com.app1::MySynonym1" : {
>       "target" : {
>             "schema": "DEFAULT_SCHEMA",
>             "object": "MY_ERP_TABLE_1"
>                   },
>       "schema": "SCHEMA_2"
>       }
> }
> ```

> **→ Tip**
>
> To generate a synonym called "acme.com.app1::MySynonym1", a design-time artifact called MySynonym1.hdbsynonym must exist in the repository package acme.com.app1; the first line of the design-time synonym artifact must be specified as illustrated in the example above.

Related Information

3.6.2 Synonym Configuration Syntax

A specific syntax is required to create a design-time representation of a **local** database synonym in SAP HANA Extended Application Services.

Synonym Definition

SAP HANA Extended Application Services (SAP HANA XS) enables you to use the hdbsynonym syntax to create a database synonym as a design-time file in the repository. On activation, a **local** synonym is generated in the catalog in the specified schema. The design-time artifact that contains the synonym definition must adhere to the .hdbsynonym syntax specified below.

> **i Note**
>
> The activation of the design-time synonym artifact illustrated in the following example generates a local synonym ("acme.com.app1::MySynonym1") in the schema SCHEMA_2.

≒ Sample Code

MySynonym1.hdbsynonym

```
{ "acme.com.app1::MySynonym1" : {
      "target" : {
          "schema": "DEFAULT_SCHEMA",
          "object": "MY_ERP_TABLE_1"
                },
      "schema [page 53]": "SCHEMA_2"
        }
}
```

Synonym Location

In the first line of the synonym-definition file, you must specify the absolute repository path to the package containing the synonym artifact (and the name of the synonym artifact) itself using the syntax illustrated in the following example.

≒ Code Syntax

```
{ "<full.path.to.package>::<MySynonym1>" : {...}}
```

For example, to generate a synonym called `"acme.com.app1::MySynonym1"`, you must create a design-time artifact called `MySynonym1.hdbsynonym` in the repository package `acme.com.app1`; the first line of the design-time synonym artifact must be specified as illustrated in the following example.

≒ Sample Code

```
{ "acme.com.app1::MySynonym1" : {...}}
```

target

To specify the name and location of the object for which you are defining a synonym, use the `target` keyword together with the keywords `schema` and `object`. In the synonym definition, the `target` keyword must adhere to the syntax shown in the following example.

≒ Code Syntax

```
"target" : {
          "schema": "<Name_of_schema_containing_<"object">>",
          "object": "<Name_of_target_object>"
                },
```

In the context of the `target` keyword, the following additional keywords are required:

- `schema` defines the name of the schema where the target object (defined in `object`) is located.

- `object` specifies the name of the catalog object to which the synonym applies.

> **! Restriction**
>
> The target object specified in a design-time synonym must only exist in the catalog; it is not possible to define a design-time synonym for a catalog object that originates from a design-time artifact.

schema

To specify the catalog location of the generated synonym, use the `schema` keyword. In the synonym definition, the `schema` keyword must adhere to the syntax shown in the following example.

> **≡, Code Syntax**
>
> ```
> "schema": "<Schema_location_of_generated_synonym>"
> ```

Related Information

Synonyms [page 50]
Create a Synonym [page 48]

3.7 Import Data with hdbtable 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 database 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.

Context

In this tutorial, you import data from a CSV file into a table generated from a design-time definition that uses the `.hdbtable` syntax. 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.

> → 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.

   The following files are required for a table import scenario.

   > i Note
   >
   > 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.

   ○ 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.hdbtable`, a design-time table defined using the `.hdbtable` syntax.

○ The schema definition, for example, `TISCHEMA.hdbschema`
  Specifies the name of the schema in which the target import table is created

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. Using any code editor, create or open the schema definition (`AMT.hdbschema`) file and enter the name of the schema you want to use to contain the target table.

```
schema_name="AMT";
```

4. Create or open the table definition file for the target import table (`inhabitants.hdbtable`) and enter the following lines of text; this example uses the `.hdbtable` syntax.

```
table.schemaName = "AMT";
table.tableType = COLUMNSTORE;
table.columns =
  [
    {name = "ID"; sqlType = VARCHAR; nullable = false; length = 20; comment =
""; },
    {name = "surname"; sqlType = VARCHAR; nullable = true; length = 30;
comment = ""; },
    {name = "name"; sqlType = VARCHAR; nullable = true; length = 30; comment =
""; },
    {name = "city"; sqlType = VARCHAR; nullable = true; length = 30; comment =
""; }
  ];
table.primaryKey.pkcolumns = ["ID"];
```

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 a table import configuration file.

   To create a table import configuration file, perform the following steps:

> **i Note**
>
> You can also open and use an existing table-import configuration file (for example,
> `inhabitants.hdbti`).

a. Right-click the folder where you want to save the table file and choose *New >Table Import Configuration* in the context-sensitive popup menu.

b. Enter or select the parent folder, where the table-import configuration file will reside.

c. Using the wizard, enter the name of the table-import configuration in the *File Name* field, for example, `MyTableConfiguration`.

   This creates the file `MyTableConfiguration.hdbti`.

   > **→ 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.

d. Edit the details of the new table-import configuration in the new (or existing) table-import configuration file.

   Enter the following lines of text in the table-import configuration file.

   ```
   import = [
           {
                   table = "mycompany.tests.TiTest::inhabitants";
                   schema = "AMT";
                   file = "mycompany.tests.TiTest:inhabitants.csv";
                   header = false;
           }
           ];
   ```

e. Choose *Finish* to save the table-import configuration.

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 result is that 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.

3.7.1 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](#)
[Table-Import Configuration-File Syntax](#)

3.7.2 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

3.7.3 Table-Import Configuration-File Syntax (HDBtable)

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 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="\"";
```

> **i 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;
```

> **i 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
```

3.7.4 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.

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.

THE BEST RUN **SAP**