



PUBLIC

SAP HANA Platform 2.0 SPS 08

Document Version: 1.0 – 2024-11-20

SAP HANA Developer Guide

For SAP HANA Studio

Content

1	SAP HANA Developer Guide for SAP HANA Studio.	8
2	Introduction to SAP HANA Development.	9
2.1	SAP HANA Architecture.	10
	SAP HANA In-Memory Database.	11
	SAP HANA Database Architecture.	13
	SAP HANA Extended Application Services.	14
	SAP HANA-Based Applications.	16
2.2	Developer Information Map for XS Classic.	17
2.3	Developer Scenarios.	18
	Developing Native SAP HANA Applications.	20
	Developing Non-Native SAP HANA Applications.	24
3	Getting Started.	29
3.1	Prerequisites.	30
3.2	SAP HANA Studio.	30
	The SAP HANA Development Perspective.	31
3.3	SAP HANA XS Application Descriptors.	35
3.4	SAP HANA Projects.	35
3.5	Tutorials.	36
	Tutorial: My First SAP HANA Application.	37
4	Setting Up Your Application.	54
4.1	Roles and Permissions for XS Development.	55
4.2	Maintaining Delivery Units.	57
	Maintain the Delivery-Unit Vendor ID.	59
	Create a Delivery Unit.	60
4.3	Using SAP HANA Projects.	62
	Maintain a Repository Workspace.	63
	Create a Project for SAP HANA XS.	65
	Share an SAP HANA XS Project.	68
	Import an SAP HANA XS Project.	69
4.4	Maintaining Repository Packages.	70
	Define the Repository Package Hierarchy.	71
	Assign Repository Package Privileges.	76
	Create a Repository Package.	79
	Delete a Repository Package.	80
4.5	Creating the Application Descriptors.	81

	Create an Application Descriptor File.	83
	Enable Access to SAP HANA XS Application Packages.	85
	Create an SAP HANA XS Application Privileges File.	100
4.6	Maintaining Application Security.	105
	Set up Application Security.	105
	Set up Application Authentication.	108
4.7	Maintaining HTTP Destinations.	112
	Tutorial: Create an HTTP Destination.	114
	Tutorial: Extend an HTTP Destination.	126
	Tutorial: Create an OAuth Configuration Package.	130
4.8	Maintaining Application Artifacts.	144
	Design-Time Application Artifacts.	144
	Studio-Based SAP HANA Development Tools.	147
5	Setting up the Data Persistence Model in SAP HANA.	152
5.1	Creating Data Persistence Artifacts with CDS in XS Classic.	155
	CDS Editors.	156
	Create a CDS Document.	159
	Create an Entity in CDS.	183
	Migrate an Entity from hdbtable to CDS (hdbdd).	201
	Create a User-Defined Structured Type in CDS.	206
	Create an Association in CDS.	220
	Create a View in CDS.	235
	Modifications to CDS Artifacts.	257
	Tutorial: Get Started with CDS.	260
	Import Data with CDS Table-Import.	264
5.2	Creating the Persistence Model with HDBTable.	277
	Create a Schema.	278
	Create a Table.	282
	Create a Reusable Table Structure.	291
	Create a Sequence.	294
	Create an SQL View.	302
	Create a Synonym.	308
	Import Data with hdbtable Table-Import.	313
6	Setting Up the Analytic Model.	327
6.1	Setting Up the Modeling Environment.	327
	Set Modeler Preferences.	327
	Set Keyboard Shortcuts.	328
6.2	Creating Views.	329
	Attributes and Measures.	329
	First Steps to View Creation.	330

	Create Attribute Views.	333
	Native HANA Models.	339
	Create Graphical Calculation Views.	343
	Create Script-Based Calculation Views.	355
	Activating Objects.	359
	Description Mapping.	362
	Import BW Objects.	362
	Group Related Measures	365
6.3	Additional Functionality for Information Views.	365
	Create Level Hierarchies.	366
	Create Parent-Child Hierarchies.	368
	Input Parameters.	372
	Assign Variables.	373
	Using Currency and Unit of Measure Conversions.	373
	Manage Information Views with Missing Objects.	374
6.4	Working with Views.	375
	Manage Editor Layout.	375
	Validate Models.	376
	Maintain Search Attributes.	376
	Data Preview Editor.	377
	Using Functions in Expressions.	377
	Resolving Conflicts in Modeler Objects.	378
6.5	Create Decision Tables.	381
	Changing the Layout of a Decision Table.	383
	Using Parameters in a Decision Table.	385
	Using Calculated Attributes in Decision Tables.	386
6.6	Generate Object Documentation.	387
7	Developing Procedures.	388
7.1	SQLScript Security Considerations.	389
7.2	Create and Edit Procedures.	391
	Define and Use Table Types in Procedures.	393
	Tutorial: Create an SQLScript Procedure that Uses Imperative Logic.	395
7.3	Create Scalar and Table User-Defined Functions.	400
	Tutorial: Create a Scalar User-Defined Function.	400
	Tutorial: Create a Table User-Defined Function.	405
7.4	Create Procedure Templates.	410
	Create Procedure Template Instances.	411
	Update Procedure Templates and Instances.	413
	Delete Procedure Templates and Instances.	413
7.5	Debugging Procedures.	414
	Setup Debugger Privileges.	414

	Debug Design-Time and Catalog Procedures.	415
	Debug an External Session.	418
7.6	Developing Procedures in the Modeler Editor.	420
7.7	Transforming Data Using SAP HANA Application Function Modeler.	421
	Converting deprecated AFL Models (AFLPMML objects).	424
	Setting up the SAP HANA Application Function Modeler.	425
	Flowgraphs.	427
	Modeling a flowgraph.	429
	Tutorial: Creating a Runtime Procedure using Application Function Modeler (AFM).	465
	Node palette flowgraphs.	469
8	Defining Web-based Data Access in XS Classic.	474
8.1	Data Access with OData in SAP HANA XS.	474
	OData in SAP HANA XS.	475
	Define the Data an OData Service Exposes.	476
	OData Service Definitions.	477
	Create an OData Service Definition.	480
	Tutorial: Use the SAP HANA OData Interface.	482
	OData Service-Definition Examples.	486
	OData Service Definition Language Syntax (XS Advanced).	509
	OData Service Definition: SQL-EDM Type Mapping (XS Advanced).	511
	OData Security Considerations.	513
	OData Batch Requests (XS Advanced).	513
8.2	Using the SAP HANA REST API.	516
	SAP HANA REST Info API.	517
	SAP HANA REST File API.	518
	SAP HANA REST Change-Tracking API.	523
	SAP HANA REST Metadata API.	524
	SAP HANA REST Transfer API.	526
	SAP HANA REST Workspace API.	527
9	Writing Server-Side JavaScript Code.	529
9.1	Data Access with JavaScript in SAP HANA XS.	529
9.2	Using Server-Side JavaScript in SAP HANA XS.	530
	Tutorial: Write Server-Side JavaScript Application Code.	530
9.3	Using Server-Side JavaScript Libraries.	546
	Import Server-Side JavaScript Libraries.	547
	Write Server-Side JavaScript Libraries.	549
9.4	Using the Server-Side JavaScript APIs.	550
	Tutorial: Use the XSJS Outbound API.	562
	Tutorial: Call an XS Procedure with Table-Value Arguments.	566
	Tutorial: Query a CDS Entity using XS Data Services.	571

	Tutorial: Update a CDS Entity Using XS Data Services.	575
9.5	Creating Custom XS SQL Connections.	578
	Create an XS SQL Connection Configuration.	580
9.6	Setting the Connection Language in SAP HANA XS.	587
9.7	Scheduling XS Jobs.	589
	Tutorial: Schedule an XS Job.	590
	Add or Delete a Job Schedule during Runtime.	599
9.8	Tracing Server-Side JavaScript.	600
	Trace Server-Side JavaScript Applications.	601
	View XS JavaScript Application Trace Files.	603
9.9	Debugging Server-Side JavaScript.	603
	Create a Debug Configuration.	607
	Execute XS JavaScript Debugging.	608
	Troubleshoot Server-Side JavaScript Debugging.	613
9.10	Testing XS JavaScript Applications.	615
	Automated Tests with XSTest in SAP HANA.	616
	Application Development Testing Roles.	617
	Test an SAP HANA XS Application with XSTest.	617
	Testing JavaScript with XSTest.	631
10	Building UIs.	638
10.1	Building User Interfaces with SAPUI5 for SAP HANA.	638
10.2	Consuming Data and Services with SAPUI5 for SAP HANA.	639
10.3	SAPUI5 for SAP HANA Development Tutorials.	640
	Tutorial: Create a Hello-World SAP UI5 Application.	641
	Tutorial: Consume an XSJS Service from SAPUI5.	645
	Tutorial: Consume an OData Service from SAPUI5.	651
	Tutorial: Consume an OData Service with the CREATE Option.	658
	Tutorial: Create and Translate Text Bundles for SAPUI5 Applications.	665
10.4	Using UI Integration Services.	669
	Creating Content for Application Sites.	670
	SAP Fiori Launchpad Sites.	682
	Creating a Standard Site.	690
	Configuring the SAP HANA Home Page.	690
11	Setting Up Roles and Privileges.	692
11.1	Create a Design-Time Role.	693
	Database Roles.	699
	Privileges.	710
11.2	Creating Analytic Privileges.	733
	Create Classical XML-based Analytic Privileges.	734
	Create SQL Analytic Privileges.	737

	Analytic Privileges.	739
12	SAP HANA Application Lifecycle Management in XS Classic.	761
13	SAP HANA Database Client Interfaces.	764
14	Migrating XS Classic Applications to XS Advanced.	766
14.1	Migrating XS Classic Applications to XS Advanced Model with the XS Advanced Migration Assistant.	766
	The XS Advanced Application-Migration Process.	768
	The XS Advanced Migration Assistant.	770
	The XS Application Migration Report.	771
	Legacy Object Types not Supported in XS Advanced.	773
15	Important Disclaimer for Features in SAP HANA.	775

1 SAP HANA Developer Guide for SAP HANA Studio

This guide explains how to build applications using SAP HANA, including how to model data, how to write procedures, and how to build application logic in SAP HANA Extended Application Services, classic model.

The *SAP HANA Developer Guide for SAP HANA Studio* explains the steps required to develop, build, and deploy applications that run in the SAP HANA XS classic model run-time environment using the tools provided with SAP HANA Studio. It also describes the technical structure of applications that can be deployed to the XS classic run-time platform. The information in the guide is organized as follows:

📘 Note

The SAP HANA Studio tooling has been deprecated along with SAP HANA Extended Application Services, Classic Model (XS Classic). For more information about alternative platforms and tools that are available for developing applications on SAP HANA, for example, XS advanced and SAP Business Application Studio, see SAP Note [2465027](#) 📄 or *Related Information* below.

- **Introduction and overview**
A high-level overview of the basic capabilities and architecture of SAP HANA XS classic model. This section also includes an information map, which is designed to help you navigate the library of information currently available for SAP HANA developers.
- **Getting started**
A collection of tutorials which are designed to demonstrate how to build and deploy a simple SAP HANA-based application on SAP HANA XS classic model quickly and easily
- **The development process**
Step-by-step information that shows in detail how to develop the various elements that make up an XS classic application. The information provided uses tasks and tutorials to explain how to develop the SAP HANA development objects. Where appropriate, you can also find background information that explains the context of the task, and reference information that provides the detail you need to adapt the task-based examples to suit the requirements of your application environment.
- **Reference sources**
A selection of reference guides that support the XS classic application development process, for example: information about application life-cycle management including transport packages, and details of the client interfaces you can use to connect applications to SAP HANA.

Related Information

[SAP Note 2465027](#) 📄

[SAP Note 3310638](#) 📄

[SAP HANA Developer Guide for XS Advanced Model \(SAP Business App Studio\)](#)

[What Is SAP Business Application Studio? \(SAP Business Application Studio Guide\)](#)

2 Introduction to SAP HANA Development

The SAP HANA developer guides present a developer's view of SAP HANA®.

The SAP HANA developer guides explain not only how to use the SAP HANA development tools to create comprehensive analytical models but also how to build applications with SAP HANA's programmatic interfaces and integrated development environment. The information in this guide focuses on the development of native code that runs inside SAP HANA.

This guide is organized as follows:

- Introduction and overview
 - SAP HANA architecture
Describes the basic capabilities and architecture of SAP HANA
 - SAP HANA developer information map
Information in graphical and textual form that is designed to help you navigate the library of information currently available for SAP HANA developers and find the information you need quickly and easily. The information provided enables access from different perspectives, for example: by SAP HANA guide, by development scenario, or by development task
 - SAP HANA development scenarios
Describes the main developer scenarios for which you can use SAP HANA to develop applications. The information focuses on native development scenarios, for example, applications based on SAP HANA XS JavaScript and XS OData services, but also provides a brief overview of the development of non-native applications (for example, using JDBC, ODBC, or ODBO connections to SAP HANA).
- Getting started
A collection of tutorials which are designed to demonstrate how to build a simple SAP HANA-based application quickly and easily, including how to use the SAP HANA studio tools and work with the SAP HANA repository
- The development process
Most of the remaining chapters use tasks and tutorials to explain how to develop the SAP HANA development objects that you can include in your SAP HANA application. Where appropriate, you can also find background information which explains the context of the task and reference information that provides the detail you need to adapt the task-based information to suit the requirements of your application environment.
Some of the tutorials in this guide refer to models that are included in the demonstration content provided with the SAP HANA Interactive Education (SHINE) delivery unit (DU). The SHINE DU is available for download in the SAP Software Download Center.

Note

Access to the SAP Software Download Center is only available to SAP customers and requires logon credentials.

Audience

This guide is aimed at people performing the following developer roles:

- Database developers
Often a business/data analyst or database expert, the database developer is concerned with the definition of the data model and schemas that will be used in SAP HANA, the specification and definition of tables, views, primary keys, indexes, partitions and other aspects of the layout and inter-relationship of the data in SAP HANA.
The database developer is also concerned with designing and defining authorization and access control, through the specification of privileges, roles and users.
- Application programmers
The programmer is concerned with building SAP HANA applications, which could take many forms but are designed based on the model-view-controller architecture. Programmers develop the code for the following component layers:
 - Views
Running inside a browser or on a mobile device
 - Controller
Typically running in the context of an application server
 - Model
Interacting closely with the data model and performing queries. Using embedded procedures or libraries, the model can be developed to run within the SAP HANA data engine.
- Client UI developers
The user-interface (UI) client developer designs and creates client applications which bind business logic (from the application developer) to controls, events, and views in the client application user interface. In this way, data exposed by the database developer can be viewed in the client application's UI.

Related Information

[SAP HANA Architecture \[page 10\]](#)

[The SAP HANA Developer's Information Atlas](#)

[Developer Scenarios \[page 18\]](#)

2.1 SAP HANA Architecture

SAP HANA is an in-memory data platform that can be deployed on premise or on demand. At its core, it is an innovative in-memory relational database management system.

SAP HANA can make full use of the capabilities of current hardware to increase application performance, reduce cost of ownership, and enable new scenarios and applications that were not previously possible. With SAP HANA, you can build applications that integrate the business control logic and the database layer with unprecedented performance. As a developer, one of the key questions is how you can minimize data movements. The more you can do directly on the data in memory next to the CPUs, the better the application will perform. This is the key to development on the SAP HANA data platform.

2.1.1 SAP HANA In-Memory Database

SAP HANA runs on multi-core CPUs with fast communication between processor cores, and containing terabytes of main memory. With SAP HANA, all data is available in main memory, which avoids the performance penalty of disk I/O. Either disk or solid-state drives are still required for permanent persistency in the event of a power failure or some other catastrophe. This does not slow down performance, however, because the required backup operations to disk can take place asynchronously as a background task.

2.1.1.1 Columnar Data Storage

A database table is conceptually a two-dimensional data structure organized in rows and columns. Computer memory, in contrast, is organized as a linear structure. A table can be represented in row-order or column-order. A row-oriented organization stores a table as a sequence of records. Conversely, in column storage the entries of a column are stored in contiguous memory locations. SAP HANA supports both, but is particularly optimized for column-order storage.

Table			Row Store		Column Store	
Country	Product	Sales		US	Country	US
US	Alpha	3.000	Row 1	Alpha		US
				3.000		JP
US	Beta	1.250	Row 2	US		UK
				Beta	Product	Alpha
JP	Alpha	700		1.250		Beta
			Row 3	JP		Alpha
UK	Alpha	450		Alpha		3.000
			Row 4	700	Sales	1.250
				UK		700
				Alpha		450
				450		

Columnar data storage allows highly efficient compression. If a column is sorted, often there are repeated adjacent values. SAP HANA employs highly efficient compression methods, such as run-length encoding, cluster coding and dictionary coding. With dictionary encoding, columns are stored as sequences of bit-coded integers. That means that a check for equality can be executed on the integers; for example, during scans or join operations. This is much faster than comparing, for example, string values.

Columnar storage, in many cases, eliminates the need for additional index structures. Storing data in columns is functionally similar to having a built-in index for each column. The column scanning speed of the in-memory column store and the compression mechanisms – especially dictionary compression – allow read operations with very high performance. In many cases, it is not required to have additional indexes. Eliminating additional indexes reduces complexity and eliminates the effort of defining and maintaining metadata.

2.1.1.2 Parallel Processing

SAP HANA was designed to perform its basic calculations, such as analytic joins, scans and aggregations in parallel. Often it uses hundreds of cores at the same time, fully utilizing the available computing resources of distributed systems.

With columnar data, operations on single columns, such as searching or aggregations, can be implemented as loops over an array stored in contiguous memory locations. Such an operation has high spatial locality and can

Compressed data can be loaded into the CPU cache faster. This is because the limiting factor is the data transport between memory and CPU cache, and so the performance gain exceeds the additional computing time needed for decompression.

The diagram illustrates a data distribution and processing flow. It shows how data from two initial cores (Core 1 and Core 2) is processed into columns (Col A and Col B), which are then grouped and processed by a final set of cores (Core 3 and Core 4).

Core 1 Data (Col A):

Col A
1000032
67867968
2345
89886757
234123
21
2342343
78787
9999993
13427777
23423
123123123
1212
2009

Core 2 Data (Col B):

Col B
4545
76
6347264
435
3434
1252
342455
3333333
8789
4523523
6767312
789976
20002
2346098

Core 3 Data (Col C):

Col C
2500
21
78675
3432423
89089
562356

Core 4 Data (Col C):

Col C
123
56743
342564
4523523
1343414
33129089

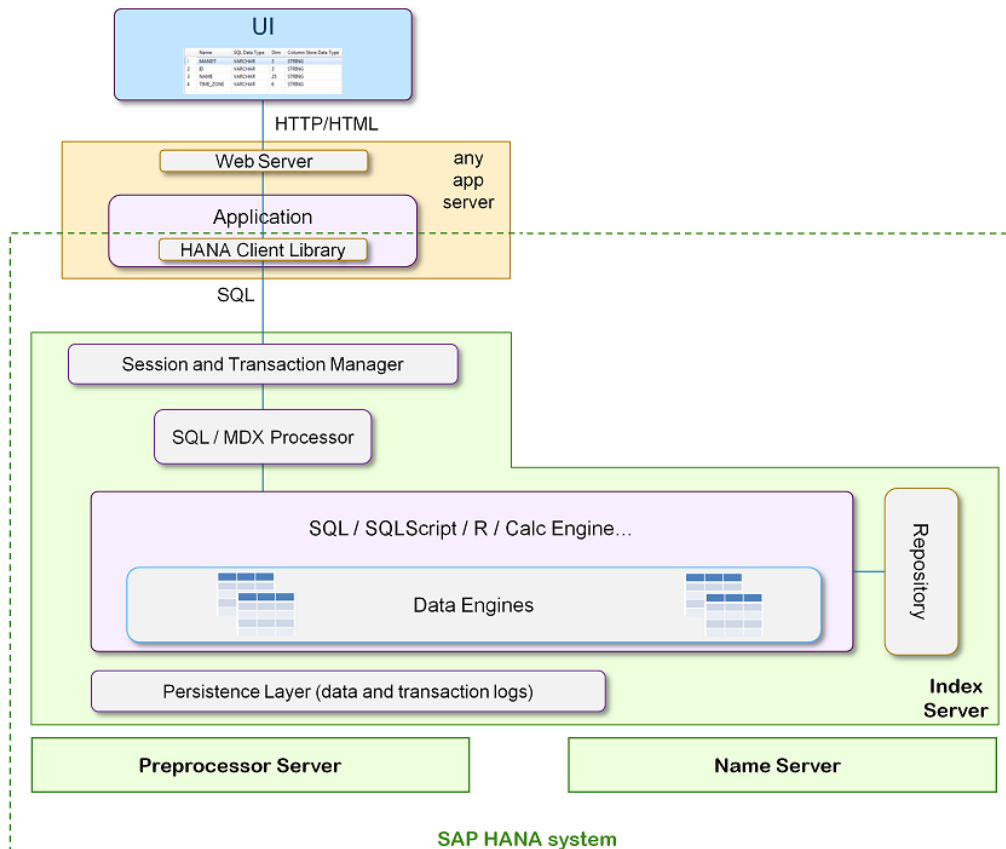
The diagram shows that the data from Core 1 and Core 2 is processed by Core 3 and Core 4. The data from Core 1 and Core 2 is processed by Core 3, and the data from Core 3 and Core 4 is processed by Core 4.

Traditional business applications often use materialized aggregates to increase performance. These aggregates are computed and stored either after each write operation on the aggregated data, or at scheduled times. Read operations read the materialized aggregates instead of computing them each time they are required.

12 PUBLIC

2.1.2 SAP HANA Database Architecture

A running SAP HANA system consists of multiple communicating processes (services). The following shows the main SAP HANA database services in a classical application context.



SAP HANA Database High-Level Architecture

Such traditional database applications use well-defined interfaces (for example, ODBC and JDBC) to communicate with the database management system functioning as a data source, usually over a network connection. Often running in the context of an application server, these traditional applications use Structured Query Language (SQL) to manage and query the data stored in the database.

The main SAP HANA database management component is known as the index server, which contains the actual data stores and the engines for processing the data. The index server processes incoming SQL or MDX statements in the context of authenticated sessions and transactions.

The SAP HANA database has its own scripting language named SQLScript. SQLScript embeds data-intensive application logic into the database. Classical applications tend to offload only very limited functionality into the database using SQL. This results in extensive copying of data from and to the database, and in programs that slowly iterate over huge data loops and are hard to optimize and parallelize. SQLScript is based on side-effect free functions that operate on tables using SQL queries for set processing, and is therefore parallelizable over multiple processors.

In addition to SQLScript, SAP HANA supports a framework for the installation of specialized and optimized functional libraries, which are tightly integrated with different data engines of the index server. Two of these

functional libraries are the SAP HANA Business Function Library (BFL) and the SAP HANA Predictive Analytics Library (PAL). BFL and PAL functions can be called directly from within SQLScript.

SAP HANA also supports the development of programs written in the R language.

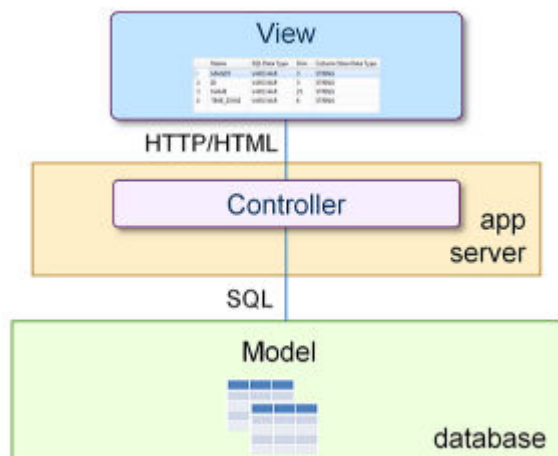
SQL and SQLScript are implemented using a common infrastructure of built-in data engine functions that have access to various meta definitions, such as definitions of relational tables, columns, views, and indexes, and definitions of SQLScript procedures. This metadata is stored in one common catalog.

The database persistence layer is responsible for durability and atomicity of transactions. It ensures that the database can be restored to the most recent committed state after a restart and that transactions are either completely executed or completely undone.

The index server uses the preprocessor server for analyzing text data and extracting the information on which the text search capabilities are based. The name server owns the information about the topology of SAP HANA system. In a distributed system, the name server knows where the components are running and which data is located on which server.

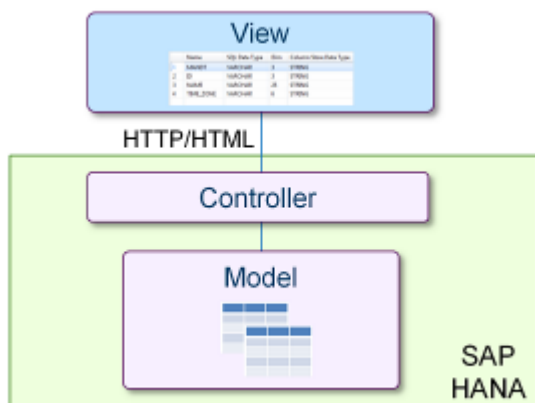
2.1.3 SAP HANA Extended Application Services

Traditional database applications use interfaces such as ODBC and JDBC with SQL to manage and query their data. The following illustrates such applications using the common Model-View-Controller (MVC) development architecture.



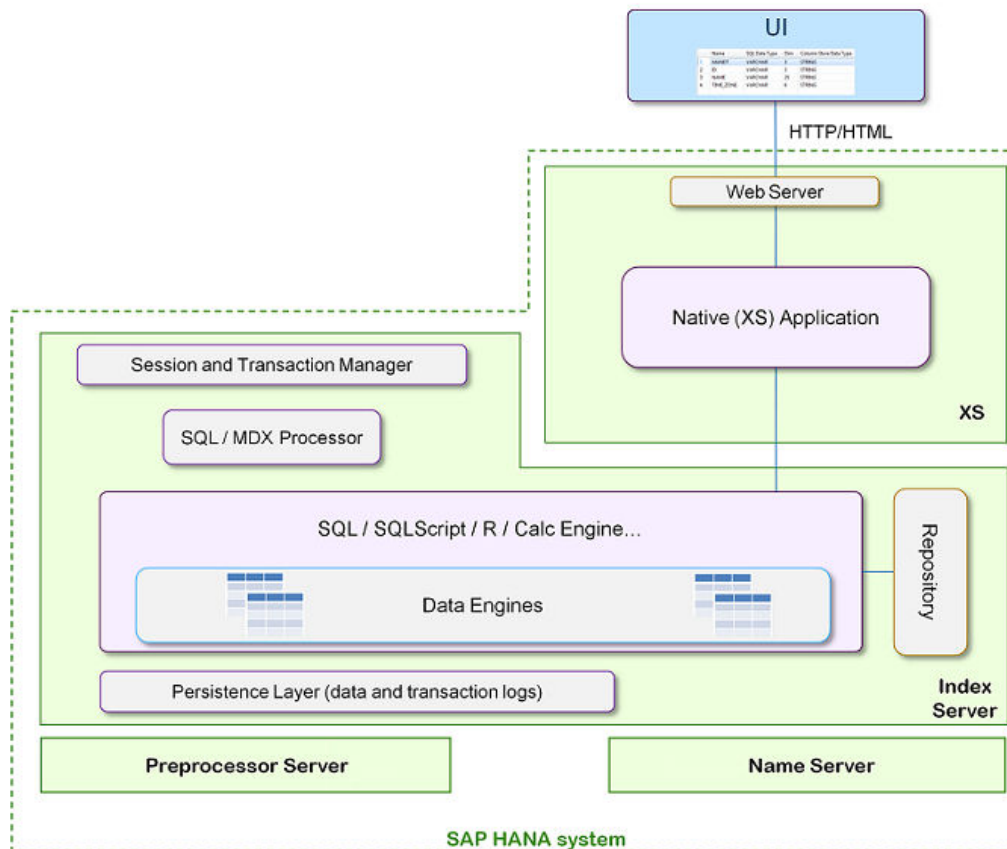
SAP HANA greatly extends the traditional database server role. SAP HANA functions as a comprehensive platform for the development and execution of native data-intensive applications that run efficiently in SAP HANA, taking advantage of its in-memory architecture and parallel execution capabilities.

By restructuring your application in this way, not only do you gain from the increased performance due to the integration with the data source, you can effectively eliminate the overhead of the middle-tier between the user-interface (the view) and the data-intensive control logic, as shown in the following figure.



In support of this data-integrated application paradigm, SAP HANA Extended Application Services provides a comprehensive set of embedded services that provide end-to-end support for Web-based applications. This includes a lightweight web server, configurable OData support, server-side JS execution and, of course, full access to SQL and SQLScript.

These SAP HANA Extended Application Services are provided by the SAP HANA XS server, which provides lightweight application services that are fully integrated into SAP HANA. It allows clients to access the SAP HANA system via HTTP. Controller applications can run completely natively on SAP HANA, without the need for an additional external application server. The following shows the SAP HANA XS server as part of the SAP HANA system.



The application services can be used to expose the database data model, with its tables, views and database procedures, to clients. This can be done in a declarative way using OData services or by writing native application-specific code that runs in the SAP HANA context. Also, you can use SAP HANA XS to build dynamic HTML5 UI applications.

In addition to exposing the data model, SAP HANA XS also hosts system services that are part of the SAP HANA system. The search service is an example of such a system application. No data is stored in the SAP HANA XS server itself. To read tables or views, to modify data or to execute SQLScript database procedures and calculations, it connects to the index server (or servers, in case of a distributed system).

Note

From SPS 11, SAP HANA includes an additional run-time environment for application development: SAP HANA extended application services (XS), advanced model. SAP HANA XS advanced model represents an evolution of the application server architecture within SAP HANA by building upon the strengths (and expanding the scope) of SAP HANA extended application services (XS), classic model. SAP recommends that customers and partners who want to develop new applications use SAP HANA XS advanced model. If you want to migrate existing XS classic applications to run in the new XS advanced run-time environment, SAP recommends that you first check the features available with the installed version of XS advanced; if the XS advanced features match the requirements of the XS classic application you want to migrate, then you can start the migration process.

Related Information

[SAPUI5 Demo Kit \(version 1.28\)](#)

[SAP HANA Search Developer Guide](#)

[SAP Note 1779803](#) 

2.1.4 SAP HANA-Based Applications

The possibility to run application-specific code in SAP HANA raises the question: What kind of logic should run where? Clearly, data-intensive and model-based calculations must be close to the data and, therefore, need to be executed in the index server, for instance, using SQLScript or the code of the specialized functional libraries.

The presentation (view) logic runs on the client – for example, as an HTML5 application in a Web browser or on a mobile device.

Native application-specific code, supported by SAP HANA Extended Application Services, can be used to provide a thin layer between the clients on one side, and the views, tables and procedures in the index server on the other side. Typical applications contain, for example, control flow logic based on request parameters, invoke views and stored procedures in the index server, and transform the results to the response format expected by the client.

The communication between the SAP HANA XS server and index server is optimized for high performance. However, performance is not the only reason why the SAP HANA XS server was integrated into SAP HANA. It also leads to simplified administration and a better development experience.

The SAP HANA XS server completes SAP HANA to make it a comprehensive development platform. With the SAP HANA XS server, developers can write SAP HANA-based applications that cover all server-side aspects, such as tables and database views, database procedures, server-side control logic, integration with external systems, and provisioning of HTTP-based services. The integration of the SAP HANA XS server into the SAP HANA system also helps to reduce cost of ownership, as all servers are installed, operated and updated as one system.

2.2 Developer Information Map for XS Classic

Find the information you need in the library of user and reference documentation currently available for SAP HANA development projects.

The development environment for SAP HANA supports a wide variety of application-development scenarios. For example, database developers need to be able to build a persistence model or design an analytic model; professional developers want to build enterprise-ready applications; business experts with a development background might like to build a simple server-side, line-of-business application; and application developers need to be able to design and build a client user interface (UI) that displays the data exposed by the data model and business logic. It is also essential to set up the development environment correctly and securely and ensure the efficient management of the various phases of the development lifecycle.

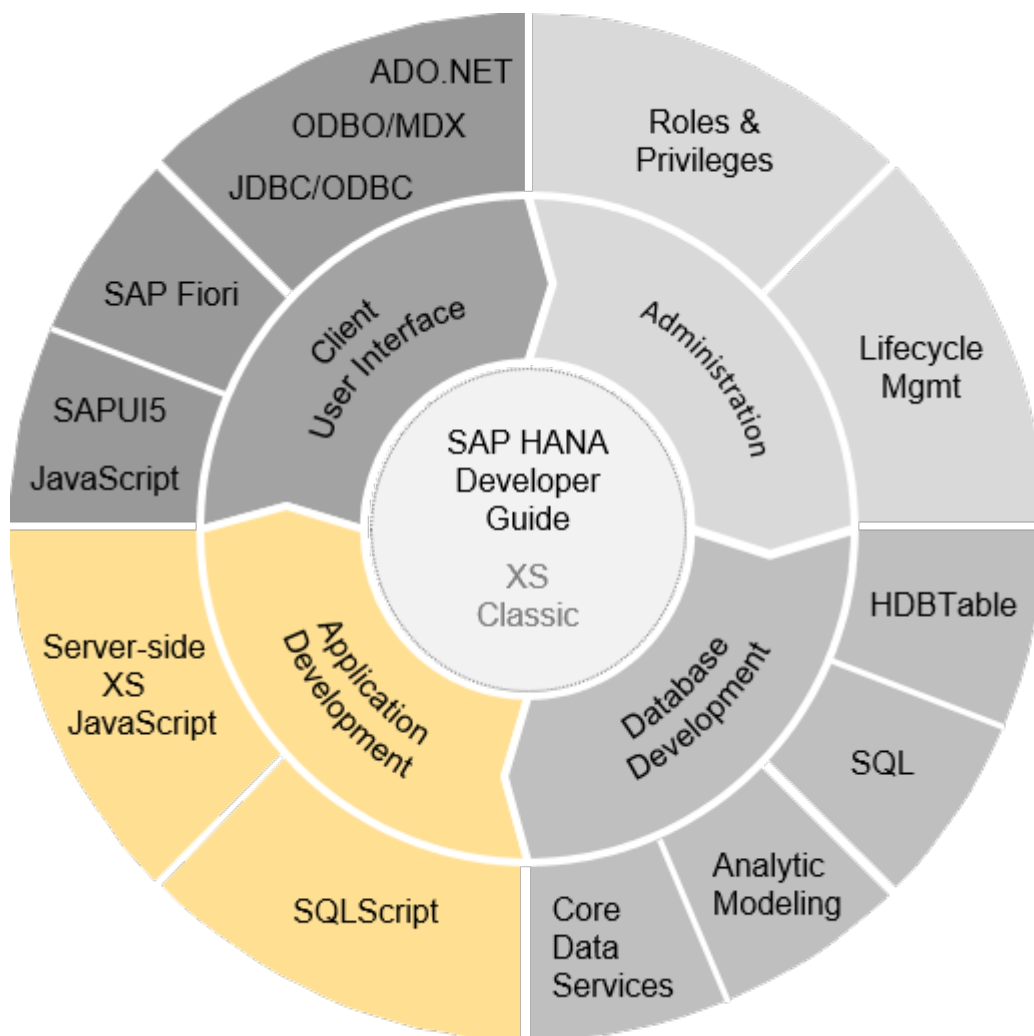
→ Tip

For more information about which guides are available in the *SAP HANA Platform* library, use the link to the SAP Help Portal in *Related Information* below. For help navigating the library, see the *SAP HANA Developer Information Map*, which is available on the SAP Help Portal.

The following image provides an overview of where to find the essential information sources for anyone planning to develop applications in SAP HANA Extended Application Services **classic** model.

→ Tip

For more information about where to find the guides and details of the individual development tasks and scenarios that each guide describes, see *Developer Information Map* in *Related Information* below.



Application Development in SAP HANA XS Classic Model

Related Information

[The SAP HANA Developer's Information Atlas](#)

2.3 Developer Scenarios

The possibility to run application specific code in SAP HANA creates several possibilities for developing SAP HANA based applications, representing various integration scenarios, and corresponding development processes.

Application developers can choose between the following scenarios when designing and building applications that access an SAP HANA data model:

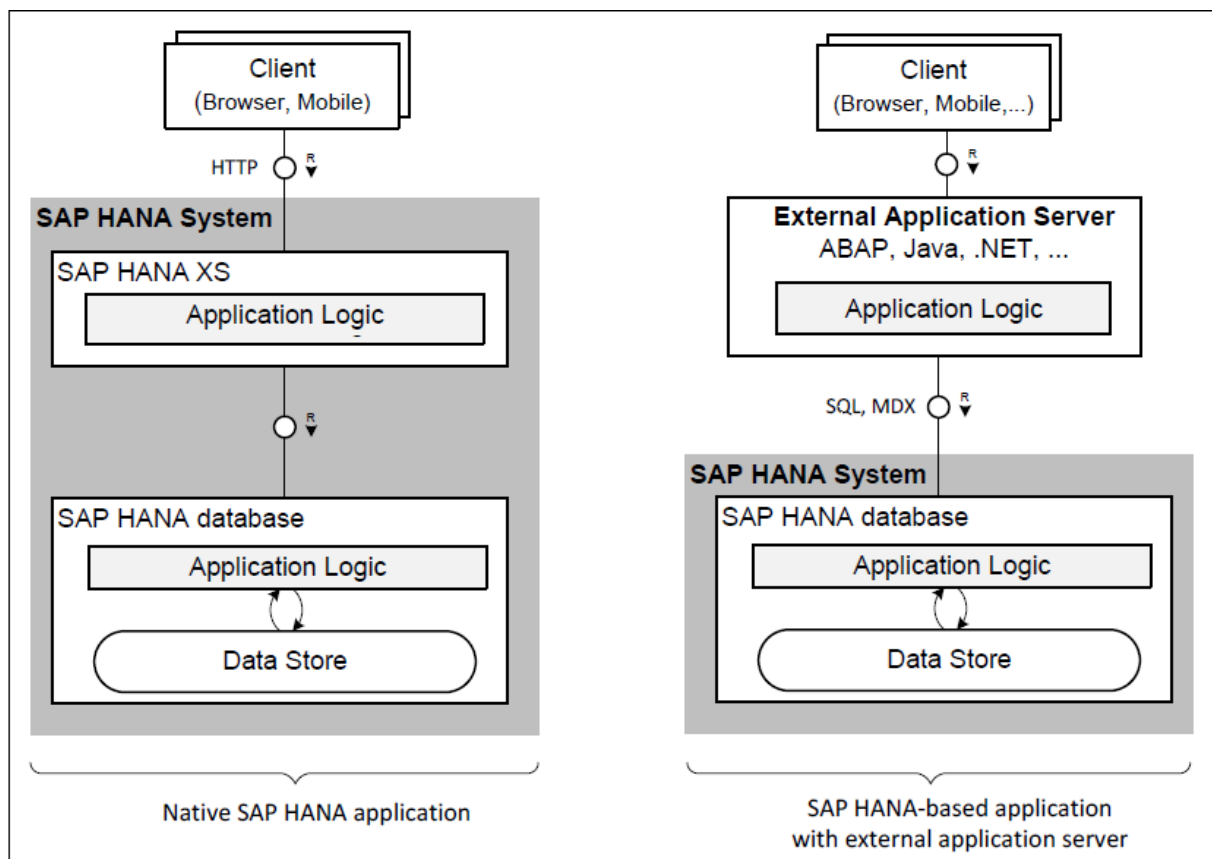
- **Native Application Development**
Native applications are developed and run in SAP HANA, for example, using just SQLScript or the extended application services provided by the SAP HANA XS platform (or both)

- Non-native Application Development

Non-native applications are developed in a separate, external environment (for example, ABAP or Java) and connected to SAP HANA by means of an external application server and a client connection: ADBC, JDBC, ODBC, or ODBO. These more traditional scenarios only use SQL and native SQLScript procedures.

Note

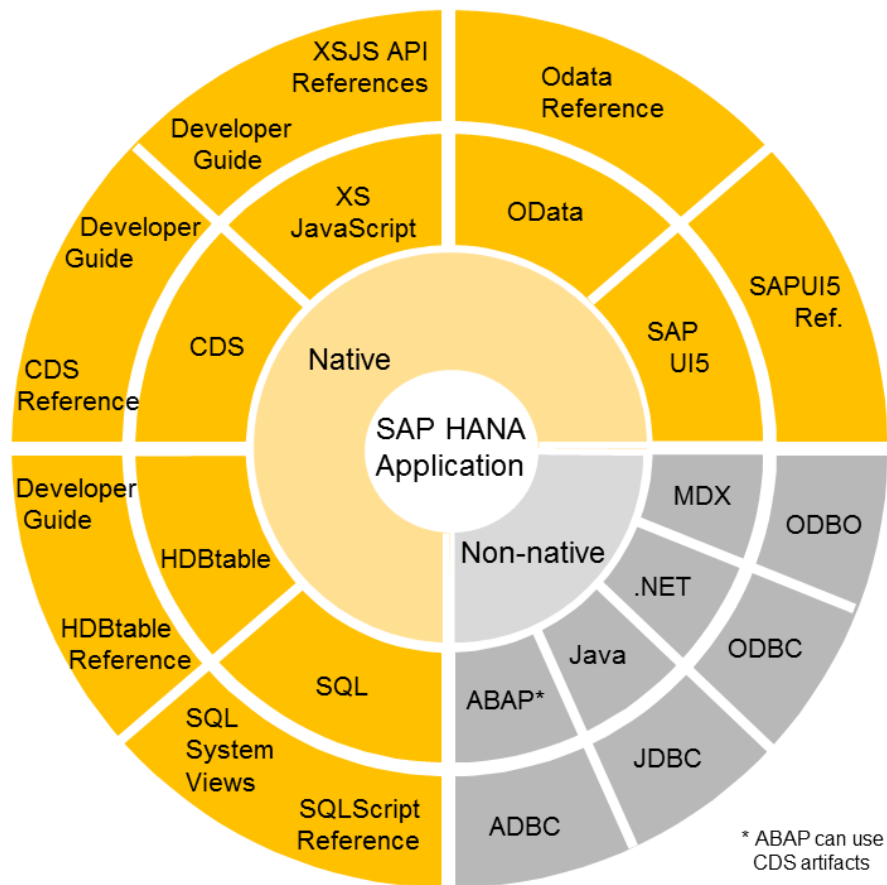
If you are using the SAP HANA Client version >2.16, and you want to use the ODBO interface, you need to use the ODBO driver provided in the SAP HANA Client for EXCEL 2.0, which is available for download from the SAP Support Portal. For more information, see *Related Information* below.



Native and Non-Native SAP HANA Application Architecture

The following diagram shows the scope of the languages and the environment you use in the various phases of the process of developing applications that harness the power of SAP HANA. For example, if you are developing native SAP HANA applications you can use CDS, HDBtable, or SQLScript to create design-time representations of objects that make up your data persistence model; you can use server-side JavaScript (XSJS) or OData services to build the application's business logic; and you can use SAPUI5 to build client user interfaces that are bound to the XSJS or OData services.

If you are developing non-native SAP HANA applications, you can choose between any of the languages that can connect by means of the client interfaces that SAP HANA supports, for example, ABAP (via ADBC) or Java (JDBC).



SAP HANA Applications and Development Languages

Related Information

[SAP Support Portal \(SAP HANA Client for EXCEL 2.0\)](#)

[SAP Note 3537501](#)

2.3.1 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.

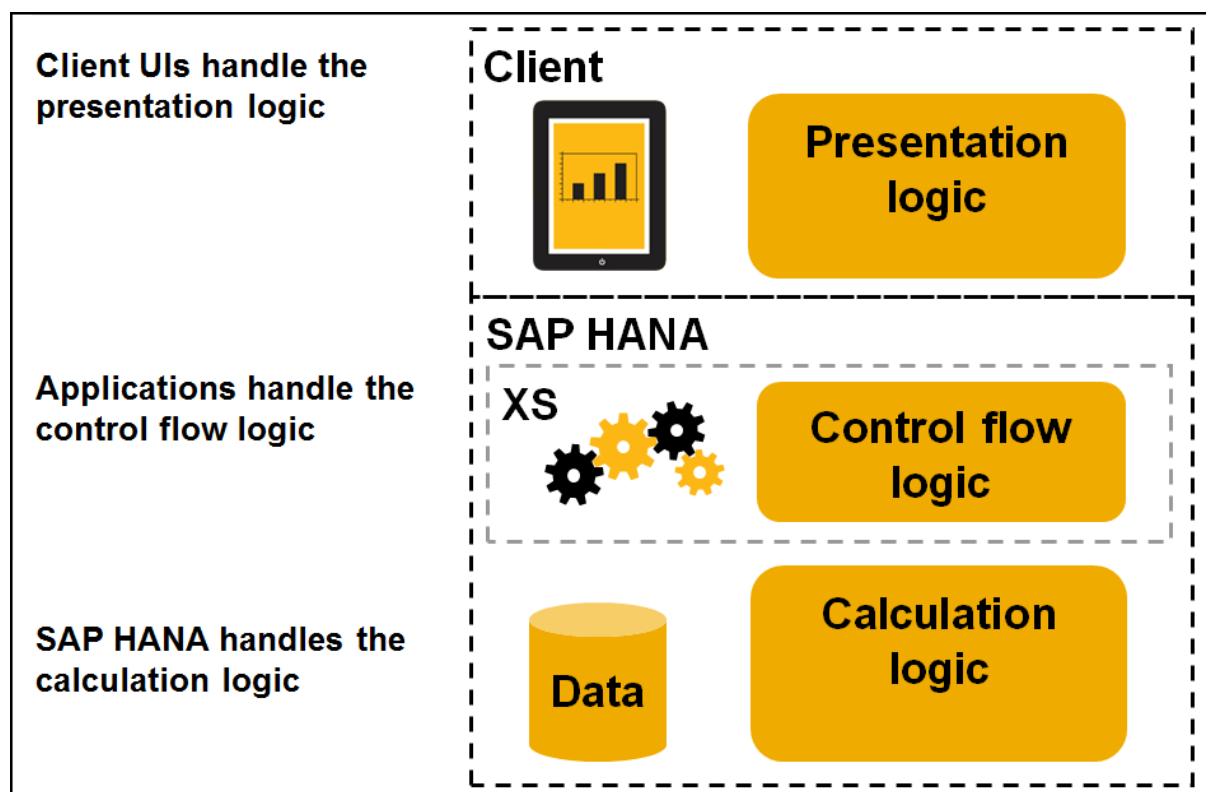
Note

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_*)

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
- 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

[Database Development Scenarios \[page 22\]](#)

[Professional Application Development Scenarios \[page 23\]](#)

[UI Client-Application Development Scenarios \[page 24\]](#)

2.3.1.1 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, ...	<i>SAP HANA SQLScript Reference</i> <i>SAP HANA SQL and System Views Reference</i> <i>SAP HANA Developer Guide for SAP HANA Studio</i>

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

⚠ Caution

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

2.3.1.2 Professional Application Development Scenarios

The primary focus of the professional application developer is to create applications.

The professional application developer creates server-side applications that define the business logic required to serve client requests, for example, for data created and exposed by the database developer. The following

table lists some of the tasks typically performed by the professional application developer and indicates where to find the information that is required to perform the task.

Typical Application-Development Tasks

Task	Details	Information Source
Create an XSJS service: - Extract data from SAP HANA - Control application response - Bind to a UI control/event	Context, examples, libraries, debugging, implementation, ...	<i>SAP HANA Developer Guide for SAP HANA Studio</i> (XS classic)
	Function code, syntax...	<i>SAP HANA XS JavaScript Reference</i> (XS classic)
	SQL code, syntax, ...	<i>SAP HANA SQLScript Reference</i>
	UI controls, events...	SAPUI5 Demo Kit (version 1.28)
Create an OData service (for example, to bind a UI control/event to existing data tables or views)	Context, service syntax, examples, libraries, debugging, implementation, ...	<i>SAP HANA Developer Guide for SAP HANA Studio</i>
	Query options, syntax...	OData Reference ➔
	UI controls, events...	SAPUI5 Demo Kit (version 1.28)

2.3.1.3 UI Client-Application Development Scenarios

Developers can build client applications to display a SAP HANA data model exposed by SAP HANA XS services.

The user-interface (UI) developer designs and creates client applications which bind business logic to controls, events, and views in the client application user interface. The UI developer can use SAPUI5 (based on HTML5) or client-side JavaScript to build the client applications. In a UI client development scenario, a developer performs (amongst others) the tasks listed in the following table, which also indicates where to find the information required to perform the task.

Typical UI-Client Development Tasks

Task	Details	Information Source
Create an SAPUI5 application to display SAP HANA data exposed by an XSJS/OData service	Context, service code/syntax, packaging, activation ...	<i>SAP HANA Developer Guide for SAP HANA Studio</i> (XS Classic)
	UI controls, events...	SAPUI5 Demo Kit (version 1.28)
Build the graphical user interface of an SAPUI5 application using UI services (widgets)	Context, tools ...	<i>Developer Guide for SAP HANA Studio</i> (XS Classic)
	UI controls, events...	SAPUI5 Demo Kit (version 1.28)

2.3.2 Developing Non-Native SAP HANA Applications

In SAP HANA, non-native applications do use the technology and services provided by the integrated SAP HANA XS platform; they run in an external application server.

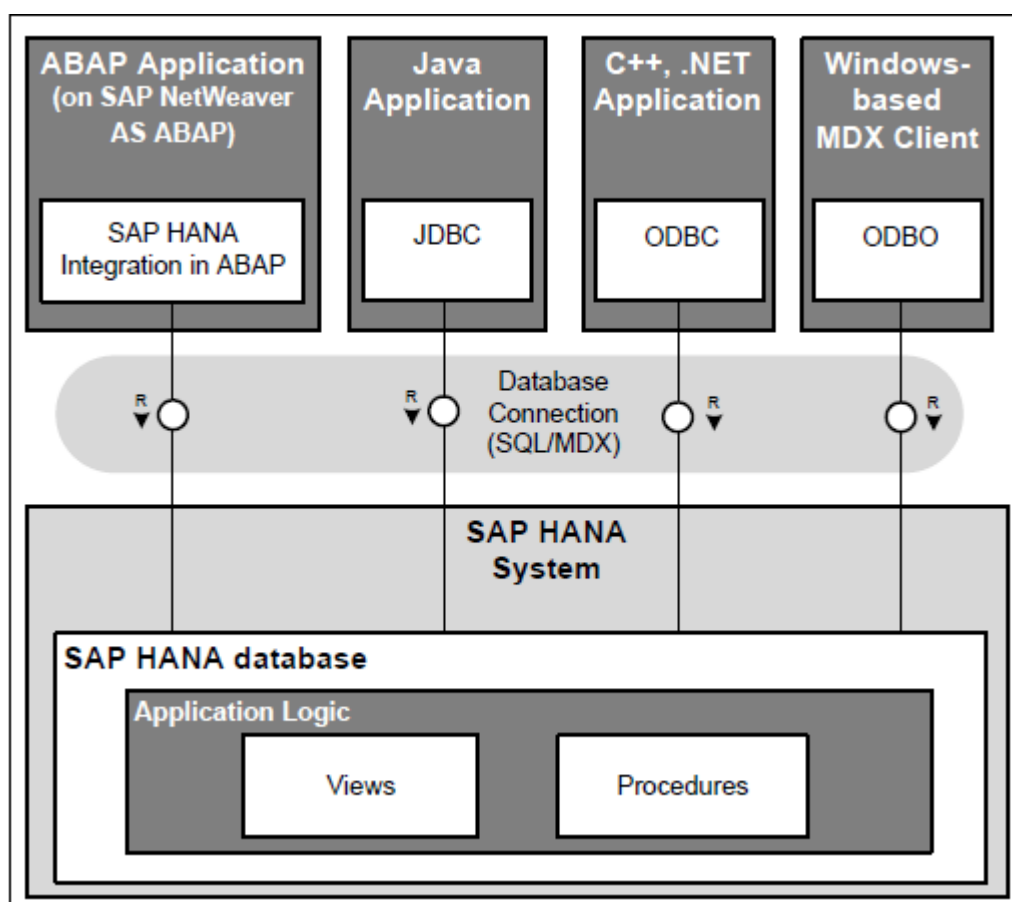
The term “non-native application” refers to a scenario where you develop applications in an environment outside of SAP HANA, for example, SAP NetWeaver (ABAP or Java). The non-native application logic runs in an external application server which accesses the SAP HANA data model (for example, tables and analytic

views) by means of a standard client interface such as JDBC, ODBC, or ODBO using SQL and native SQLScript procedures.

Note

Technically, it is also possible for non-native front-end applications to connect to the SAP HANA database directly via SQL or MDX, for example when SAP HANA is used as a data source for Microsoft Excel. However, it is not recommended to use such an approach for SAP business applications. If you are using the SAP HANA Client version ≥ 2.16 , and you want to use the ODBO interface, you need to use the ODBO driver provided in the SAP HANA Client for EXCEL 2.0, which is available for download from the SAP Support Portal. For more information, see *Related Information* below.

The following figure shows how you use the client interfaces to connect your non-native SAP HANA application to an SAP HANA data model.



Non-native SAP HANA Application Architecture

Related Information

[ABAP Client Interface \[page 26\]](#)

[The JDBC Client Interface \[page 26\]](#)

[ODBC Client Interface \[page 27\]](#)

[ODBO Client Interface \[page 27\]](#)

[SAP Support Portal \(SAP HANA Client for EXCEL 2.0\)](#)

2.3.2.1 ABAP Client Interface

ABAP database connectivity (ADBC) provides the benefits of a native SQL connection by means of `EXEC SQL`. ADBC is basically a series of `CL_SQL*` classes, which simplify and abstract the `EXEC SQL` blocks.

You can build a custom ABAP application that runs in an external application environment but connects directly to an SAP HANA data model using the client ADBC interface. Support for external ABAP applications includes dedicated Eclipse-based tools, external views (ABAP Dictionary objects that can be accessed like a normal dictionary view), and ABAP managed database procedures (ABAP dictionary objects that enable you to map procedure parameters to the ABAP parameters).

Note

It is possible to make use of native data-persistence objects in your ABAP application, for example, design-time data-persistence objects specified using the Core Data Services (CDS) syntax.

To build an ABAP application that accesses an SAP HANA data model, you need to perform the following high-level steps

1. Write an ABAP application in your own development environment, for example using the ABAP tools-integration in Eclipse.
2. Connect the ABAP development environment to SAP HANA using the ADBC interface; the ABAP environment can be either:
 - An ABAP application server
 - Your development machine
3. Run the ABAP application to connect to a SAP HANA data model.

2.3.2.2 The JDBC Client Interface

Java Database Connectivity (JDBC) is a Java-based application programming interface (API) which includes a set of functions that enable Java applications to access a data model in a database. The SAP HANA client includes a dedicated JDBC interface.

You can build a custom Java application that runs in an external application environment but connects directly to an SAP HANA data model using the client JDBC interface. To build a Java application that accesses an SAP HANA data model, you need to perform the following high-level steps:

1. Write a Java application in your own development environment.
2. Connect the Java development environment to SAP HANA using the JDBC client interface; the Java environment can be either:
 - A Java application server
 - Your development machine
3. Run the Java application to connect to an SAP HANA data model.

Related Information

[SAP HANA Client Interface Programming Reference](#)

2.3.2.3 ODBC Client Interface

Open Database Connectivity (ODBC) is a standard application programming interface (API) that provides a set of functions that enable applications to access a data model in a database. The SAP HANA client includes a dedicated ODBC interface.

You can build a custom .NET application (using C++, C#, Visual Basic and so on) that runs in an external application environment but connects directly to an SAP HANA data model using the client ODBC interface. To build an .NET application that accesses an SAP HANA data model, you need to perform the following high-level steps:

1. Install the client ODBC interface on your development machine.
2. Write a .NET application in your development environment.
3. Connect the .NET application to SAP HANA using the ODBC interface.
4. Run the .NET application to connect to an SAP HANA data model.

Note

The SAP HANA data provider for Microsoft ADO.NET is installed as part of the SAP HANA client installation.

Related Information

[SAP HANA Client Interface Programming Reference](#)

2.3.2.4 ODBO Client Interface

OLE database for OLAP (ODBO) is a standard application programming interface (API) that enables Windows clients to exchange data with an OLAP server. The SAP HANA client includes an ODBO driver which applications can use to connect to the database and execute MDX statements

You can build a Windows-based client application that runs in an external application environment but connects directly to an SAP HANA data model, for example, to run queries with multidimensional expressions (MDX) using the native SAP HANA MDX interface. To build an MDX application that accesses a SAP HANA data model, you need to perform the following high-level steps:

1. Install the client ODBO interface on your development machine.

Note

If you are using the SAP HANA Client version ≥ 2.16 , and you want to use the ODBO interface, you need to use the ODBO driver provided in the SAP HANA Client for EXCEL 2.0, which is available for download from the SAP Support Portal. For more information, see *Related Information* below.

2. Write an application that uses multi-dimensional expressions (MDX) in your own development environment.
3. Connect the application to SAP HANA using the ODBO interface.
4. Run the Windows-based MDX application to connect to an SAP HANA data model.

Related Information

[SAP HANA Client Interface Programming Reference](#)

[SAP Support Portal \(SAP HANA Client for EXCEL 2.0\)](#) 

[SAP Note 3537501](#) 

2.3.2.5 SAP HANA Data Provider for Microsoft ADO.NET

SAP HANA includes a data provider that enables applications using Microsoft .NET to connect to the SAP HANA database.

You can build a custom .NET application (for example, using C++, C#, or Visual Basic) that runs in an external application environment but connects directly to an SAP HANA data model using the SAP HANA data provider for Microsoft ADO.NET. The SAP HANA data provider for Microsoft ADO.NET is installed as part of the SAP HANA client installation. To build a .NET application that accesses an SAP HANA data model, you need to perform the following high-level steps:

1. Install the SAP HANA data provider for Microsoft ADO.NET on your development machine.
2. Write a .NET application in your development environment, for example, using Visual Studio.
3. Connect the .NET application to SAP HANA using the client interface included with the SAP HANA data provider for Microsoft ADO.NET.
4. Run the .NET application to connect to an SAP HANA data model.

You can use the SAP HANA data provider for Microsoft ADO.NET to develop Microsoft .NET applications with Microsoft Visual Studio by including both a reference to the data provider and a line in your source code referencing the data-provider classes.

Related Information

[SAP HANA Client Interface Programming Reference](#)

3 Getting Started

To understand which tools SAP HANA Extended Application Services (SAP HANA XS) provides to enable you to start developing native applications, you need to run through the process of building a small application, for example, in the form of a “Hello World” application.

As part of the getting-started process, you go through the following steps:

- **Prerequisites**
A short list of the tools and permissions required to start working with the SAP HANA application-development tools.
- **Workspaces and projects SAP HANA projects**
If you are using the SAP HANA studio, you must create a shared project, which you use to group all your application-related artifacts and synchronize any changes with the repository workspace.

ⓘ Note

If you are using the SAP HANA Web-based Development Workbench, you do not need to create a project or a repository workspace.

- **Creating application descriptors**
Each native SAP HANA application requires descriptor files. The application descriptors are the core files that you use to describe an application's framework within SAP HANA XS, for example: to mark the root point from which content can be served, which content is to be exposed, or who has access to the content.
- **Tutorials**
A selection of “Hello World” tutorials are used to demonstrate the application-development process in SAP HANA XS and show you how to produce a simple application quickly and easily. Some of the tutorials in this guide refer to models that are included in the demonstration content provided with the SAP HANA Interactive Education (SHINE) delivery unit (DU). The SHINE DU is available for download in the SAP Software Download Center.

ⓘ Note

Access to the SAP Software Download Center is only available to SAP customers and requires logon credentials.

Related Information

[Prerequisites \[page 30\]](#)

3.1 Prerequisites

To start working with the tools provided to enable application development on SAP HANA Extended Application Services (SAP HANA XS), it is necessary to ensure that the developers have the required software and access permissions.

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:

Note

The following can only be provided by someone who has the required authorizations in SAP HANA, for example, an SAP HANA administrator.

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA developer tools, for example: SAP HANA studio or the SAP HANA Web-based Development Workbench.

Note

To provide access to the SAP HANA repository from the SAP HANA studio, the EXECUTE privilege is required for SYS.REPOSITORY_REST, the database procedure through which the REST API is tunneled.

- Access to the SAP HANA repository
- Access to selected run-time catalog objects
- Some of the tutorials in this guide refer to models that are included in the demonstration content provided with the SAP HANA Interactive Education (SHINE) delivery unit (DU). The SHINE DU is available for download in the SAP Software Download Center.

Note

Access to the SAP Software Download Center is only available to SAP customers and requires logon credentials.

3.2 SAP HANA Studio

The SAP HANA studio is an Eclipse-based development and administration tool for working with SAP HANA. You use the SAP HANA studio to develop native applications that can take advantage of the benefits provided by SAP HANA Extended Application Services (SAP HANA XS).

One of the most important features of the Eclipse-based environment is the *perspective*. SAP HANA provides a number of dedicated perspectives that are aimed at the application developer. As an application developer, you frequently use the following perspectives:

- The *SAP HANA Development* perspective

Provides views and menu options that enable you to perform all the tasks relating to application development on SAP HANA XS, for example: to manage application-development projects, display content of application packages, and browse the SAP HANA repository. You can also define your data-persistence model here by using design-time artifacts to define tables, views, sequences, and schemas.

- The *Debug* perspective
Provides views and menu options that help you test your applications, for example: to view the source code, monitor or modify variables, and set break points.
- The *Modeler* perspective
Provides views and menu options that enable you to define your analytic model, for example, attribute, analytic, and calculation views of SAP HANA data.
- The *Team Synchronizing* perspective
Provides views and menu options that enable you to synchronize artifacts between your local file system and the SAP HANA Repository.
- The *Administration Console* perspective
Provides views that enable you to perform administrative tasks on SAP HANA instances.

Note

SAP HANA Studio has been deprecated, as described in SAP Note [2465027](#). Patched versions are available for download, but no new features are being added.

Related Information

[Download SAP HANA Studio 2 \(Support Packages & Patches\)](#)

3.2.1 The SAP HANA Development Perspective

SAP HANA studio's *SAP HANA Development Perspective* includes a selection of programming tools that developers can use to build applications in SAP HANA. You can customize the perspective to include your own favorite tools, too.

The *SAP HANA Development perspective* is where you will do most of your programming work, for example:

- Creating and sharing projects
- Creating and modifying development objects
- Managing development object versions
- Committing development objects to the SAP HANA repository

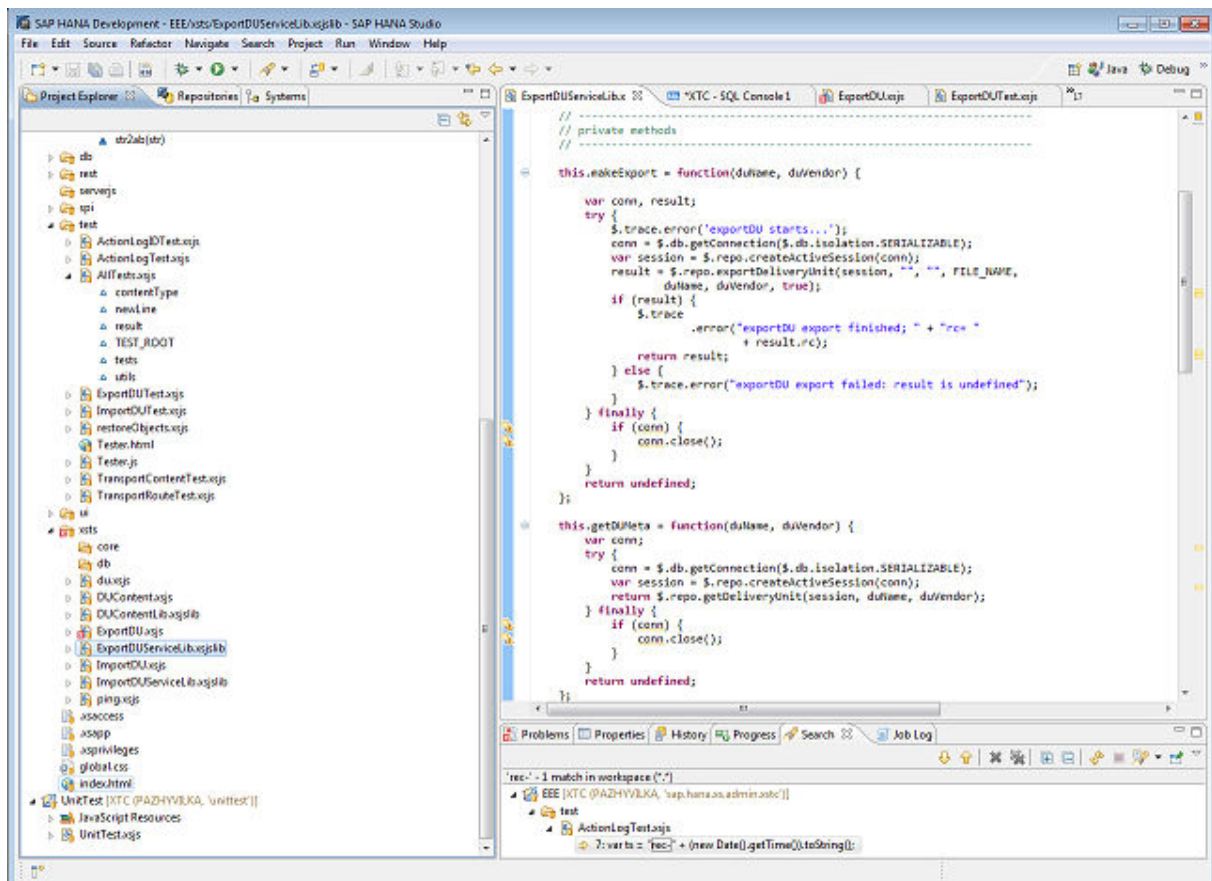
Note

By default, saving a file automatically commits the saved version of the file to the Repository.

- Activating development objects in the SAP HANA repository

The SAP HANA Development perspective contains the following main work areas:

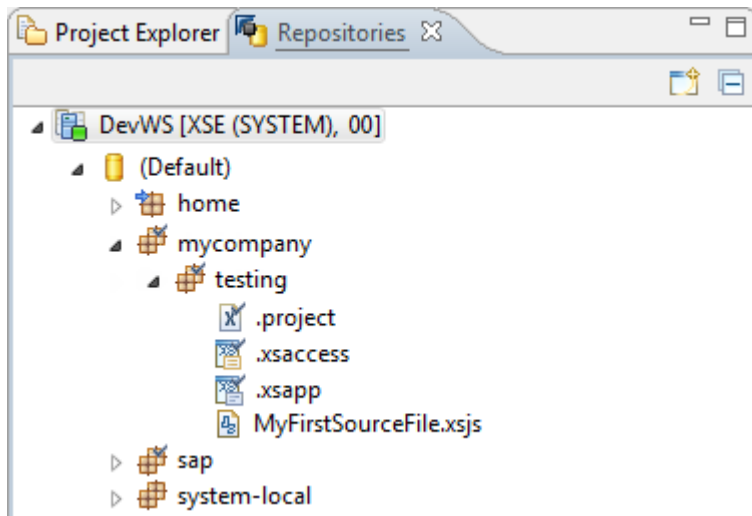
- **Explorers/Browsers**
Selected views enable you to browse your development artifacts: the objects on your workstation, and the objects in the repository of the SAP HANA system you are working with.
- **Editors**
Specialized editors enable you to work with different types of development objects, for example, application-configuration files, JavaScript source files, SQLScript files.



3.2.1.1 The Repositories View

You can browse and perform actions on the contents of the SAP HANA Repository on a specific SAP HANA system.

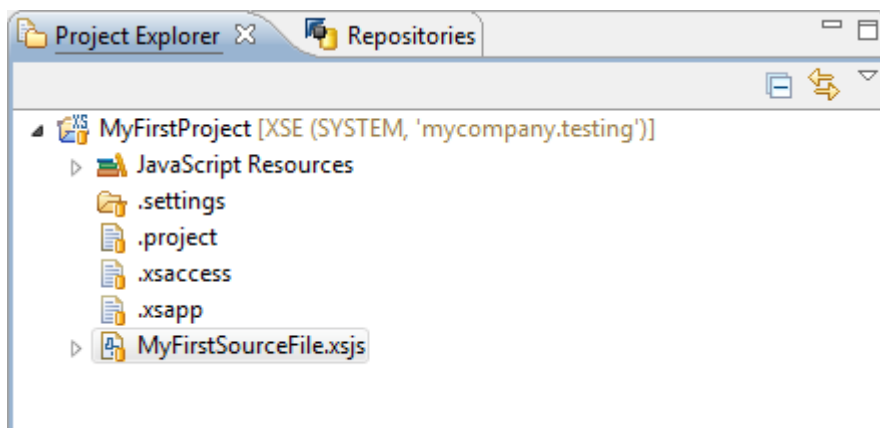
The [Repositories](#) view displays the contents of the repository on a specific SAP HANA system. You can navigate the package hierarchy and check out project files from the SAP HANA Repository; the checked out files are downloaded to the workspace on your local file system, where you can work on them and modify them as required.



The [Repositories](#) view is a list of repository workspaces that you have created for development purposed on various SAP HANA systems. Generally, you create a workspace, check out files from the repository, and then do most of your development work in the [Project Explorer](#). However, with more recent versions of SAP HANA, you can use the [Repositories](#) view to perform actions directly on repository objects in multiple workspaces, for example: edit objects, activate objects, and manage object versions - all without the need to set up a project. The [Repositories](#) view also provides direct access to lifecycle-management tools.

3.2.1.2 The Project Explorer View

The [Project Explorer](#) view is the most commonly used element of the [SAP HANA Development](#) perspective; it shows you the development files located in the repository workspace you create on your workstation. You use the [Project Explorer](#) view to create and modify development files. Using context-sensitive menus, you can also commit the development files to the SAP HANA repository and activate them. Bear in mind that saving a file in shared project commits the saved version of the file to the repository automatically.



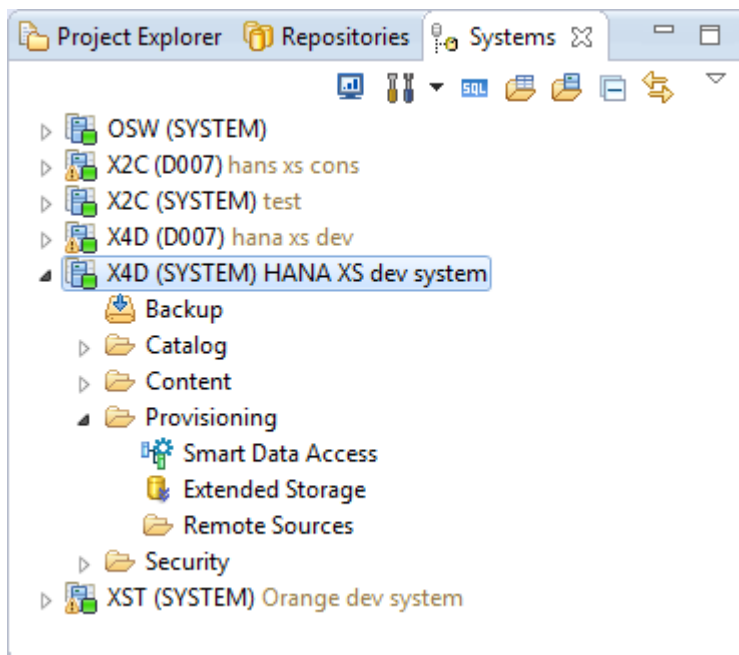
→ Tip

Files with names that begin with the period (.), for example, `.xsapp`, are sometimes not visible in the *Project Explorer*. To enable the display of all files in the *Project Explorer* view, use the [Customize View](#) [Available Customization](#) option and clear all check boxes.

3.2.1.3 The Systems View

The *Systems* view is one of the basic organizational elements included with the *Development* perspective.

You can use the *Systems* view to display the contents of the SAP HANA database that is hosting your development project artifacts. The *Systems* view of the SAP HANA database shows both activated objects (objects with a runtime instance) and the design-time objects you create but have not yet activated.



The *Systems* view is divided into the following main sections:

- **Security**
Contains the roles and users defined for this system.
- **Catalog**
Contains the database objects that have been activated, for example, from design-time objects or from SQL DDL statements. The objects are divided into schemas, which is a way to organize activated database objects.
- **Provisioning**
Contains administrator tools for configuring smart data access, data provisioning, and remote data sources
- **Content**
Contains design-time database objects, both those that have been activated and those not activated. If you want to see other development objects, use the *Repositories* view.

3.3 SAP HANA XS Application Descriptors

Each application that you want to develop and deploy on SAP HANA Extended Application Services (SAP HANA XS) required so-called “application descriptor” files. The application descriptors describe an application's framework within SAP HANA XS.

The framework defined by the SAP HANA XS application descriptors includes the root point in the package hierarchy where content is to be served to client requests. When defining the application framework, you also have to specify whether the application is permitted to expose data to client requests, what (if any) authentication method is required to access application content, and (optionally) what if any privileges are required to perform actions on the packages and package content that are exposed.

- The application descriptor
The core file that you use to describe an application's framework within SAP HANA XS. The package that contains the application descriptor file becomes the root path of the resources exposed to client requests by the application you develop.
- The application-access file
The configuration file you use to specify who or what is authorized to access the content exposed by an SAP HANA XS application package and what content they are allowed to see. For example, you use the application-access file to specify the following:
 - The application content that can be exposed to client requests
 - The authentication method used to enable access to package content, for example, form-based, basic, or none at all.

3.4 SAP HANA Projects

In SAP HANA, a project groups together all the artifacts you need for a specific part of the application-development environment.

Before you can start the application-development workflow, you must create a project, which you use to group together all your application-related artifacts. However, a project requires a repository workspace, which enables you to synchronize changes in local files with changes in the SAP HANA repository. You can create the workspace before or during the project-creation step. As part of the project-creation process, you perform the following tasks:

1. Add a development system
2. Create a development workspace.

The place where you work on development objects is called a repository *workspace*. The workspace is the link between the SAP HANA repository and your local file system. When you check out a package from the repository, SAP HANA copies the contents of the package hierarchy to your workspace. To ensure that the changes you make to project-related files are visible to other team members, you must commit the artifacts back into the repository and activate them.

Note

By default, saving the file automatically commits the saved version of the file to the repository.

3. Create a project

You use the project to collect all your application-related artifacts in one convenient place. Shared projects enable multiple people to work on the same files at the same time.

Note

Files checked out of the repository are not locked; conflicts resulting from concurrent changes to the same file must be resolved manually, using the [Merge](#) tools provided in the context-sensitive [Team](#) menu.

4. Share a project

Sharing a project establishes a link between project-specific files in your development workspace and the SAP HANA repository. A shared project ensures that changes you make to project-related files in your development workspace are synchronized with the SAP HANA repository and, as a result, visible to other team members. Shared projects are available for import by other members of the application-development team.

3.5 Tutorials

Tutorials are a good way to understand quickly what is required to write a simple native application for SAP HANA XS.

In this section you can use the following tutorials to help you understand the basic steps you need to perform when developing native SAP HANA XS applications:

- **Hello OData**
A simple application that enables you to test the SAP HANA OData interface by exposing an OData collection for analysis and display in a client application.
- **Hello World in server-side JavaScript (XSJS)**
A simple application written in server-side JavaScript which displays the words “Hello World” in a Web browser along with a string extracted from a table in the SAP HANA database.

Note

The namespace `sap` in the SAP HANA repository is restricted. Place the new packages and application artifacts that you create during the tutorials in your own namespace, for example, `com.acme`, or use the `system.local` area for testing.

Related Information

[Tutorial: Use the SAP HANA OData Interface \[page 482\]](#)

[Tutorial: My First SAP HANA Application \[page 37\]](#)

3.5.1 Tutorial: My First SAP HANA Application

This topic describes the steps required to develop a simple application that runs natively in SAP HANA.

Context

This tutorial shows you how to use the SAP HANA studio to develop a functional SAP HANA application. Although it is simple, the tutorial demonstrates the development process that you can apply to all types of application-development scenarios.

The tutorial shows how to create a simple SAP HANA application. The application uses server-side JavaScript code to retrieve data from SAP HANA by executing SQL statements in the SAP HANA database. The retrieved data is displayed in a Web browser. During the tutorial, you use tools provided in the SAP HANA studio to perform the following tasks:

- **Connect to an SAP HANA system**
Add (and connect to) an SAP HANA system, which hosts the repository where development objects are stored
- **Create a repository workspace**
Create a development workspace which enables you to synchronize the development artifacts in your local file system with the repository hosted on the SAP HANA system you connect to.
- **Create and share a project**
Add a project which you can use to hold the application-development artifacts in a convenient central location.
Sharing the project makes the contents of the new project available to other members of the application-development team by linking the local project to the SAP HANA repository. In this way, you can manage object versions and synchronize changes to development objects.
- **Write server-side JavaScript code**
Use JavaScript code to extract data from the SAP HANA database in response to a client request; the code will include SQLScript to perform the data extraction.
- **Display data**
Display data extracted from the SAP HANA database in a Web browser.

Related Information

[Tutorial: Add an SAP HANA System \[page 38\]](#)

[Tutorial: Add a Repository Workspace \[page 41\]](#)

[Tutorial: Add an Application Project \[page 43\]](#)

[Tutorial: Write Server-Side JavaScript \[page 48\]](#)

[Tutorial: Retrieve Data from SAP HANA \[page 52\]](#)

3.5.1.1 Tutorial: Add an SAP HANA System

Application-development artifacts are stored and managed in the SAP HANA repository. To connect to an SAP HANA repository, you must add the system to SAP HANA studio.

Prerequisites

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA studio

Note

To provide access to the SAP HANA repository from the SAP HANA studio, the EXECUTE privilege is required for SYS.REPOSITORY_REST, the database procedure through which the REST API is tunneled.

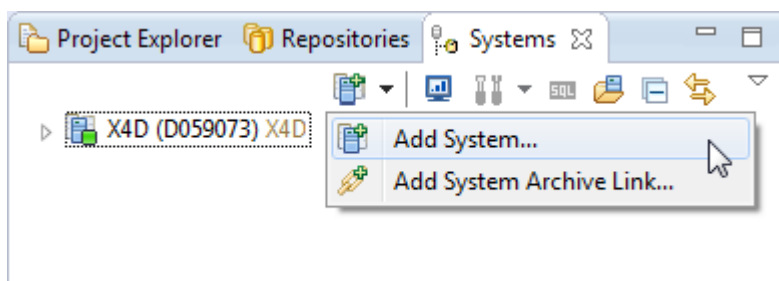
- Access to the SAP HANA Repository

Context

You must add a connection to the SAP HANA system hosting the repository that stores the application-development artifacts you will be working with.

Procedure

1. Open SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. In the *Systems* view, click [+] *Add System...* and choose *Add System...*.



4. Type the details of the SAP HANA system in the following fields:
 - Host Name:
The name of the server hosting the SAP HANA database instance, for example, **dev.host.acme.com**
If you are adding a tenant database in a multi-database system, you can specify either the fully qualified domain name (FQDN) of the system hosting the tenant database or the virtual host name for

the tenant database. Every tenant database requires a virtual host name so that the system's internal SAP Web Dispatcher can forward HTTP requests to the XS server of the correct database.

→ Tip

If you do not enter the virtual host name for the tenant database here, you must specify it explicitly as the XS server host in the system properties. You can do this after you have finished adding the system. In the [Systems](#) view by right-clicking the system whose properties you want to modify and choosing [Properties](#) > [XS Properties](#).

- Instance Number
SAP HANA instance number on that server, for example, **00**
- Description
A display name for the system you are adding. When you start working with a lot of systems, you will need to label and recognize the systems in the SAP HANA studio. Enter **Development System**.

System

Specify System

Specify the host name and instance number of the system.

Host Name:

Instance Number:

Database Mode:

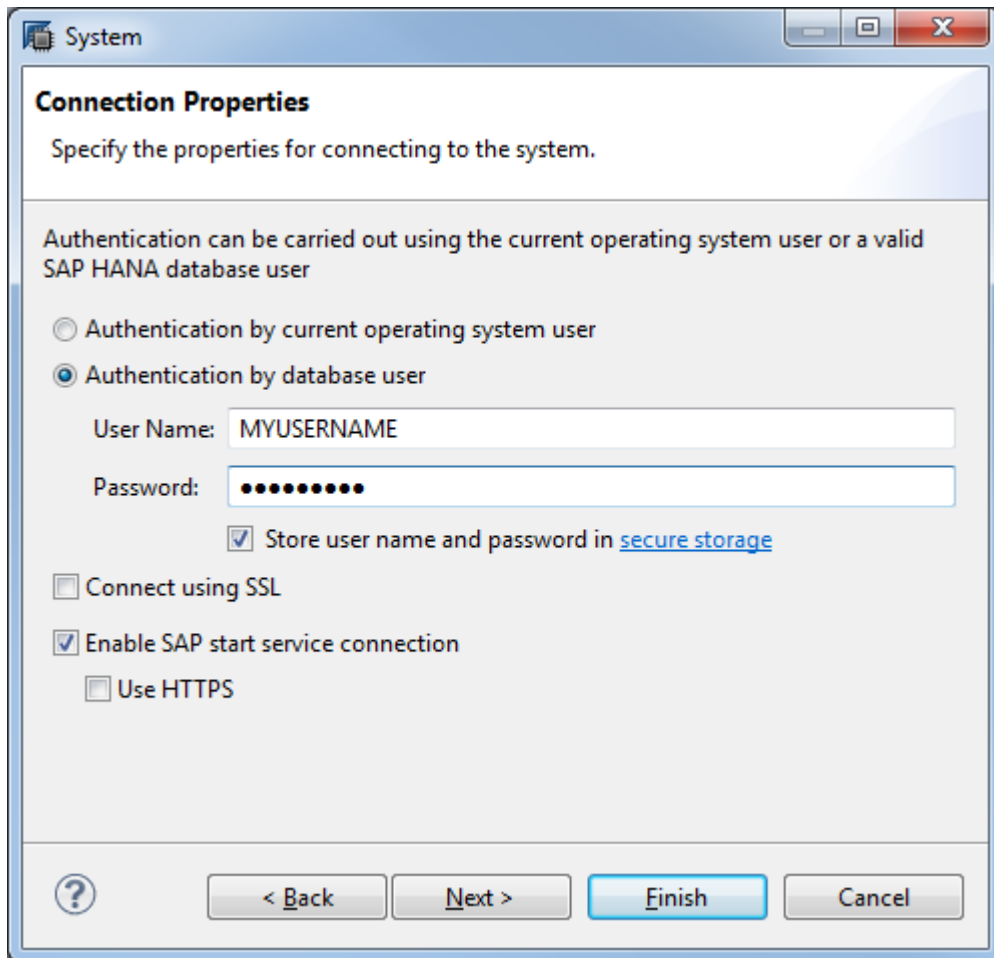
- ☒ Single-database mode
- ☐ Multiple-database mode
- ☐ Tenant database
Database Name:
- ☐ System database

Description:

Locale:

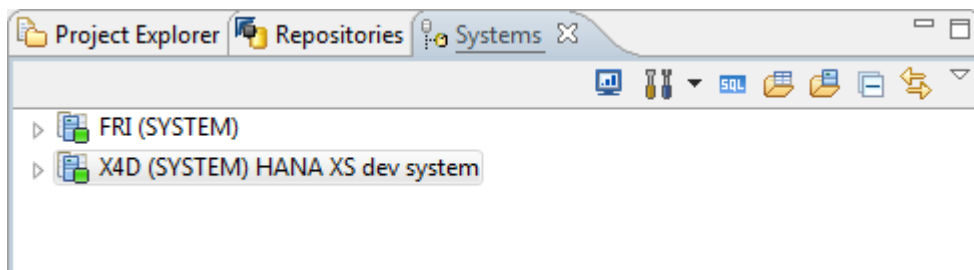
Folder:

5. Select [Next](#).
6. Enter a user name and password for the connection, and select [Finish](#).



Results

After adding the system, you will see the system in the *Systems* view.



3.5.1.2 Tutorial: Add a Repository Workspace

The place where you work on development objects is called a repository workspace. The workspace is the link between the SAP HANA repository and your local file system.

Prerequisites

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA studio
- Access to the SAP HANA Repository

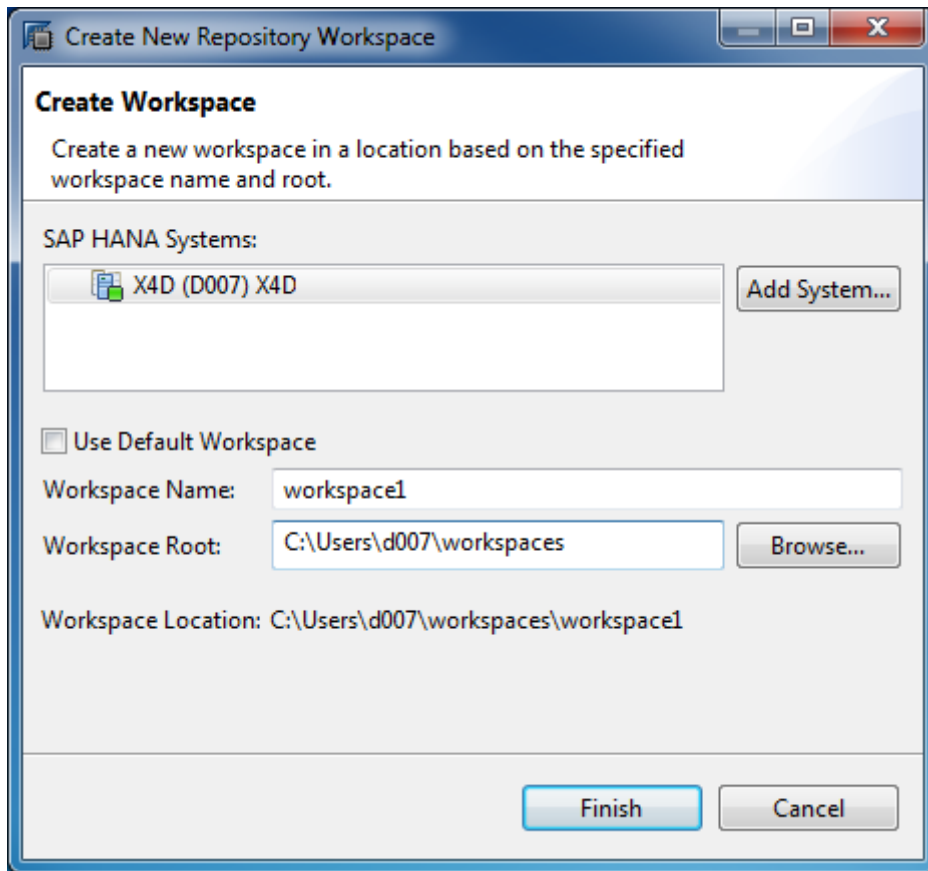
Context

After you add the SAP HANA system hosting the repository that stores your application-development files, you must specify a repository workspace, which is the location in your file system where you save and work on the development files.

To create a repository workspace, perform the following steps:

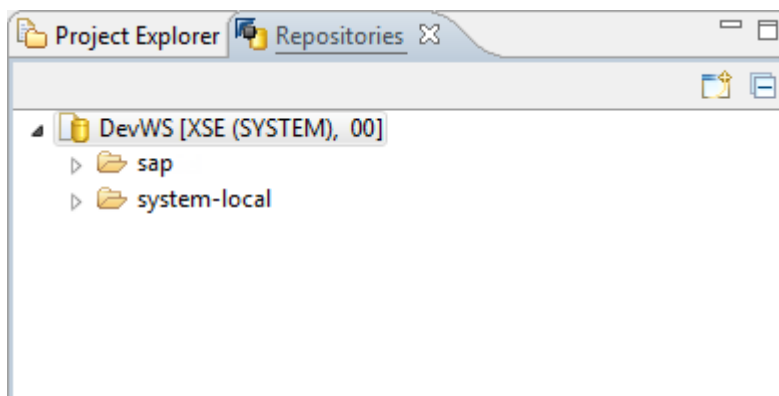
Procedure

1. Open SAP HANA studio.
2. In the *SAP HANA Development* perspective, open the *Repositories* view.
3. In the *Repositories* view, choose **File > New > Repository Workspace**.
4. You must provide the following information:
 - SAP HANA system
The name of the SAP HANA system hosting the repository that you want to synchronize your workspace with; choose the same system you just added for this tutorial.
 - Workspace Name
If a default repository workspace exists, uncheck the *Default workspace* option and enter a workspace name; the workspace name can be anything you like, for example, **DevWS**.
A folder with the name you type is created below the *Workspace Root*.
 - Workspace root
The *Workspace Root* is a folder that contains the workspace you create in this step. The *Workspace Root* can be anywhere on your local file system. For this tutorial, create a folder at **C:\users\<PATH>\workspaces** and make this the *Workspace Root*.

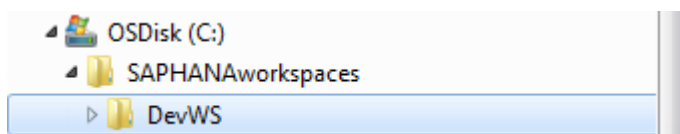


5. Click *Finish*.

In the *Repositories* view, you see your workspace, which enables you to browse the repository of the system tied to this workspace. The repository packages are displayed as folders.



At the same time, a folder will be added to your file system to hold all your development files.



6. Remove a repository workspace.

If it is necessary to remove a workspace, you can choose between multiple deletion options; the option you choose determines what is removed, from where (local file system or remote repository), and what, if anything, is retained.

- a. Open the [SAP HANA Development](#) perspective.
- b. Choose the [Repositories](#) view and expand the repository node containing the workspace you want to remove.
- c. Right-click the workspace you want to remove.
- d. Choose the workspace-deletion mode.

The following modes apply when you delete a workspace in SAP HANA studio:

Workspace Deletion Modes

Workspace Deletion Mode	Description
Delete	Remove workspace; delete all workspace-related local files; delete related changes to remote (repository) data.
Remove from client (keep remote changes)	Remove workspace from local client system; delete all local workspace-related files; retain changes to remote (repository) data.
Disconnect local from remote (keep changes)	Keep the workspace but remove the workspace label from the list of workspaces displayed in the Repositories view. The connection to the disconnected workspace can be reestablished at any time with the option Import Local Repository Workspaces .

3.5.1.3 Tutorial: Add an Application Project

You use the project to collect all the development artifacts relating to a particular part of an application in one convenient place.

Prerequisites

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA studio
- Access to an SAP HANA Repository workspace

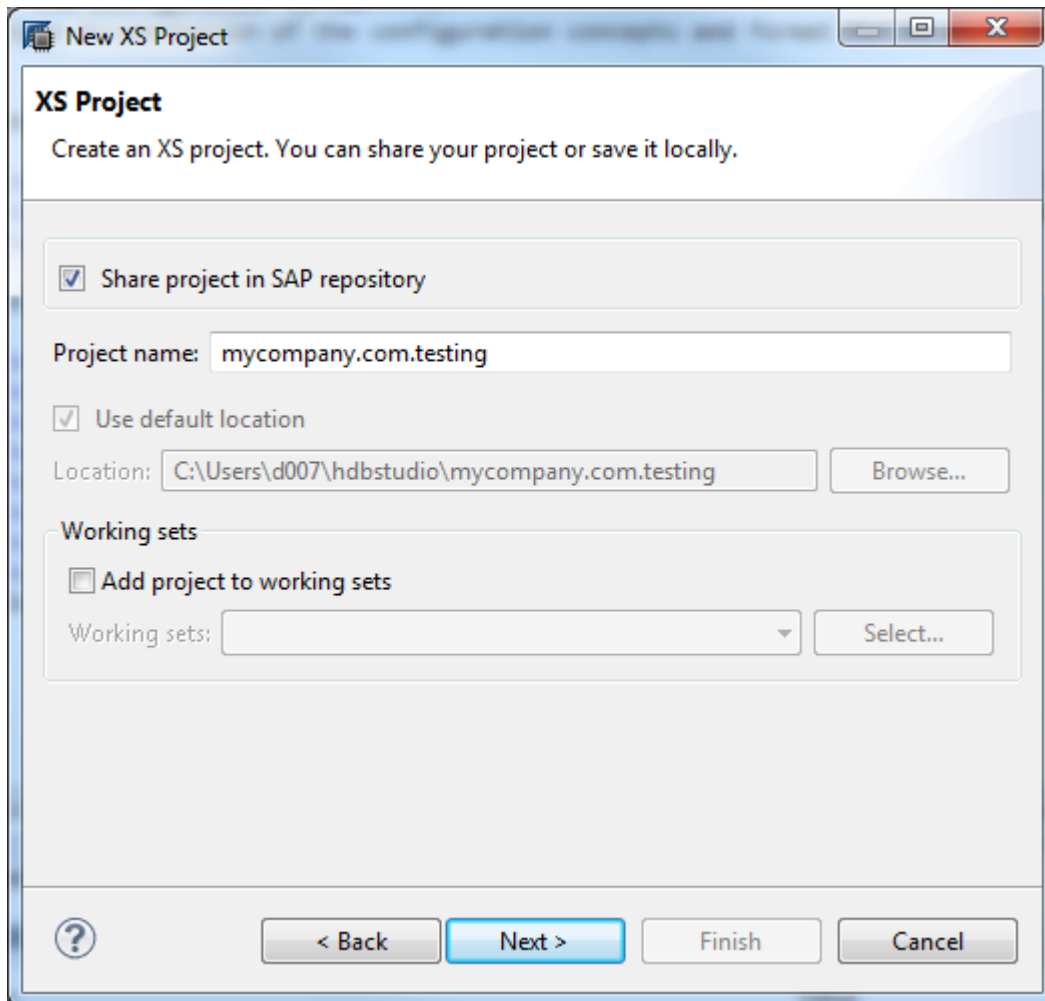
Context

After you set up a development environment for the chosen SAP HANA system, you can add a project to contain all the development objects you want to create as part of the application-development process.

There are a variety of project types for different types of development objects. Generally, a project type ensures that only the necessary libraries are imported to enable you to work with development objects that are specific to a project type. In this tutorial, you create an [XS Project](#).

Procedure

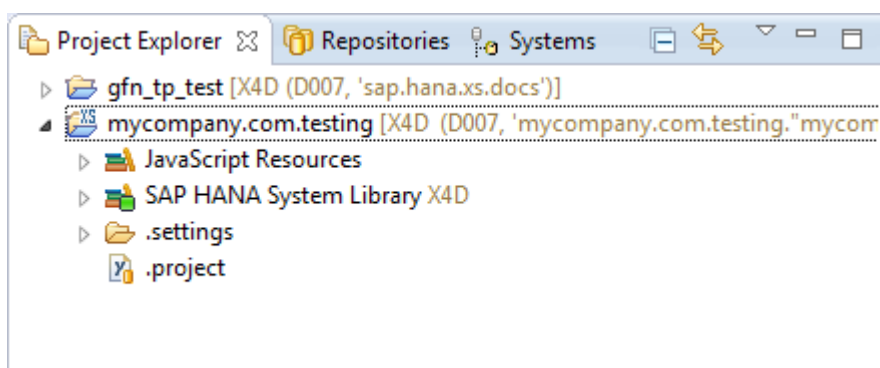
1. Open SAP HANA studio.
2. From the *File* menu in SAP HANA studio, choose ► *New* ► *Project* ►.
3. In the *New Project* dialog, under ► *SAP HANA* ► *Application Development* ►, select *XS Project*, and choose *Next*.
4. Enter the following details for the new project:
 - Project name
Enter: **mycompany.com.testing**
Since a project name must be unique within the same Eclipse workspace, a good convention is to use the fully qualified package name as the project name.
 - Project location
You can keep this as the default SAP HANA studio (Repository) workspace. To save the project in an alternative location from the recommended default, you must first disable the option *Share project in SAP repository*. You can share the new project manually later. Sharing a project enables continuous synchronization with the SAP HANA repository.
 - Working sets (optional)
A working set is a concept similar to favorites in a Web browser, which contain the objects you work on most frequently.



5. Choose *Finish*.

Results

The *Project Explorer* view in the *SAP HANA Development* perspective displays the new project. The system information in brackets [*X4D (D007)...*] to the right of the project node name in the *Project Explorer* view indicates that the project has been shared; shared projects are regularly synchronized with the Repository hosted on the SAP HANA system you are connected to.



Note

If you disabled the option *Share project in SAP repository* when you created the project, you must share the new project manually.

Related Information

[Tutorial: Share an Application Project \[page 46\]](#)

3.5.1.4 Tutorial: Share an Application Project

Sharing a project establishes a link between project-specific files in your development workspace and the repository hosted by the SAP HANA system you are connected to.

Prerequisites

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA studio
- Access to an SAP HANA Repository workspace
- An existing SAP HANA project

Context

Sharing a project associates the project with your repository workspace and synchronizes the project with the repository hosted on the SAP HANA system you are connected to. By default, a project is automatically shared at the same time as it is created; the option to disable the auto-share operation is available in the project-creation wizard.

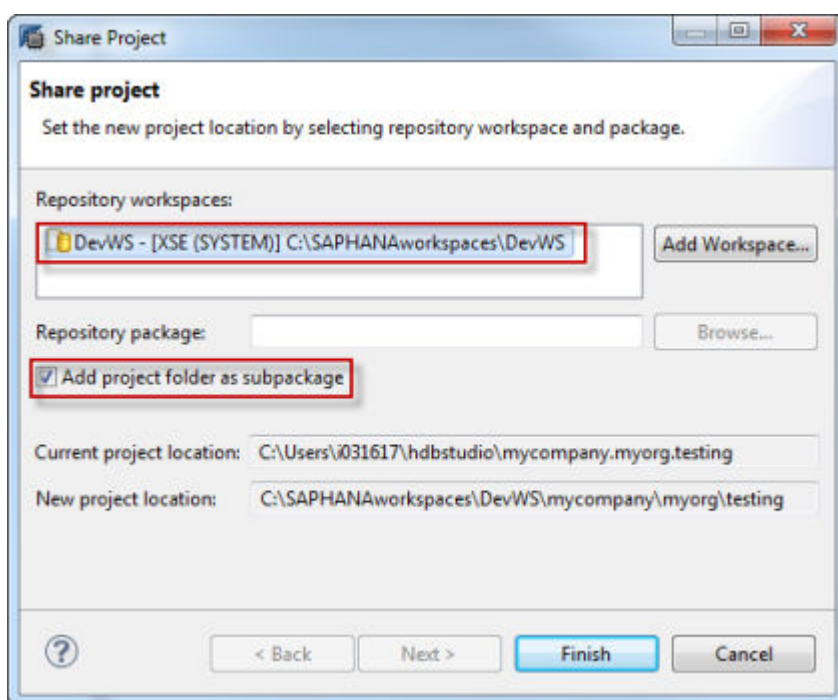
Note

Manually sharing a project is necessary only if you disabled the option *Share project in SAP repository* when you created the project or chose to explicitly **unshare** the project after you created it.

If you need to manually share a project, perform the following steps:

Procedure

1. Start SAP HANA studio and open the *SAP HANA Development* perspective.
2. In the *Project Explorer* view, right-click the project you want to share, and choose ► *Team* ► *Share Project* ► in the context-sensitive popup menu to display the *Share Project* dialog.



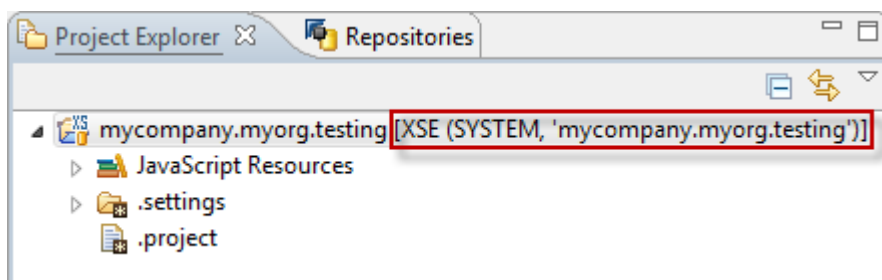
Since you only have one workspace, the wizard selects it for you automatically. If you have more than one workspace, you must choose the workspace to host the shared project.

The dialog also shows the *Current project location* (the current location of your project, in the repository workspace), and the *New project location* (where your project will be copied so it can be associated with the repository workspace).

Also, since *Add project folder as subpackage* is checked, subpackages will be created based on the name of your project.

3. Choose *Finish*.

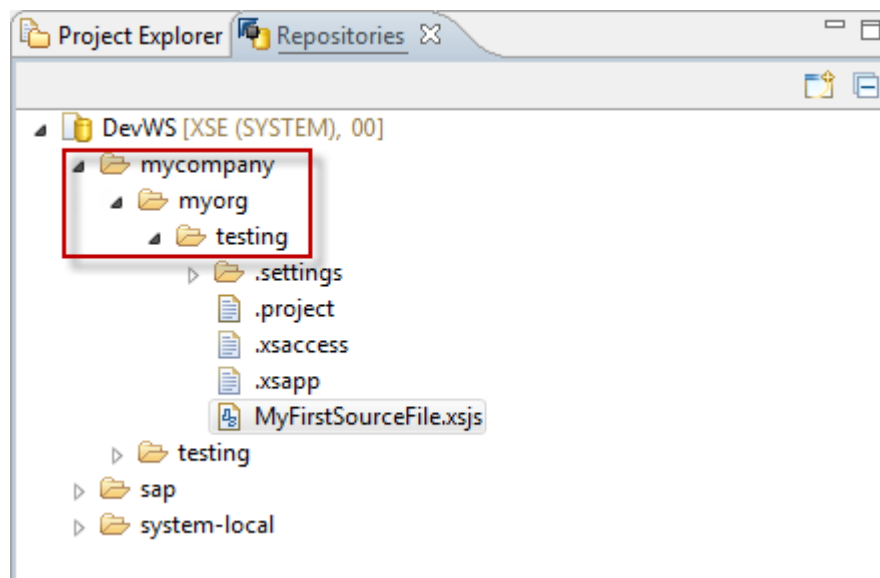
The shared project is displayed in the *Project Explorer* view associated with your workspace.



The `.project` file is shown with an asterisk , which indicates that the file has changed but has yet to be committed to the repository.

4. Right-click the `.project` file, and select **Team** > **Commit**  from the context-sensitive popup menu to add your project and its files to the repository. The `.project` file is now displayed with a diamond icon, , indicating that the latest version of the file on your workstation has been committed to the SAP HANA repository.

In addition, the [Repositories](#) view shows that a new hierarchy of packages has been created based on the name of your project, `mycompany.myorg.testing`.



3.5.1.5 Tutorial: Write Server-Side JavaScript

SAP HANA Extended Application Services (SAP HANA XS) supports server-side application programming in JavaScript. In this step we add some simple JavaScript code that generates a page which displays the words *Hello, world!*.

Prerequisites

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA studio
- Access to a shared project in the SAP HANA Repository where you can create the artifacts required for this tutorial.

Context

As part of this server-side JavaScript tutorial, you create the following files:

- `MyFirstSourceFile.xsjs`
This contains your server-side JavaScript code.
- `.xsapp`
This marks the root point in the application's package hierarchy from which content can be exposed via HTTP. You still need to explicitly expose the content and assign access controls.
- `.xsaccess`
Expose your content, meaning it can be accessed via HTTP, and assign access controls, for example, to manage who can access content and how.

→ Tip

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 enables direct editing of the file in the appropriate editor.

Procedure

1. Open SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. In the *Project Explorer* view, right-click your XS project, and choose **New > Other** in the context-sensitive popup menu.
4. In the *Select a Wizard* dialog, choose **SAP HANA > Application Development > XS JavaScript File**.
5. In the *New XS JavaScript File* dialog, enter **MyFirstSourceFile.xsjs** in *File name* text box.

→ Tip

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 enables direct editing of the file in the appropriate editor. You can also select a template to use. Templates contain sample source code to help you.

6. Choose *Finish*.
7. In the `MyFirstSourceFile.xsjs` file, enter the following code and save the file:

ⓘ Note

By default, saving the file automatically commits the saved version of the file to the repository.

```
$.response.contentType = "text/html";  
$.response.setBody( "Hello, World !");
```

The example code shows how to use the SAP HANA XS JavaScript API's `response` object to write HTML. By typing `$.` you have access to the API's objects.

8. Check that the application descriptor files are present in the root package of your new XS JavaScript application.

The application descriptors (`.xsapp` and `.xsaccess`) are mandatory and describe the framework in which an SAP HANA XS application runs. The `.xsapp` file indicates the root point in the package hierarchy where content is to be served to client requests; the `.xsaccess` file defines who has access to the exposed content and how.

Note

By default, the project-creation Wizard creates the application descriptors automatically. If they are not present, you will see a 404 error message in the Web Browser when you call the XS JavaScript service.

If you need to create the application descriptors manually, perform the following steps:

- a. Add a blank file called `.xsapp` (no name, just a file extension) to the root package of your XS JavaScript application.

To add an `.xsapp` file, right-click the project to which you want to add the new file, select **New** **> Other > SAP HANA > Application Development > XS Application Descriptor File** from the context-sensitive popup menu, and choose **Next**.

→ Tip

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.

- b. Add a file called `.xsaccess` (no name, just a file extension) to the root package of your XS JavaScript application, and copy the following code into the new `.xsaccess` file:

To add a `.xsaccess` manually, right-click the project to which you want to add the file, select **New** **> Other > SAP HANA > Application Development > XS Application Access File** from the context-sensitive popup menu, and choose **Next**.

→ Tip

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, provides a working template, and, if appropriate, enables direct editing of the file.

```
{
  "exposed" : true,
  "authentication" :
    [
      { "method" : "Form" }
    ],
  "prevent_xsrf" : true
}
```

This code exposes the application content via HTTP, specifies form-based logon as the default authentication method for the corresponding SAP HANA application, and helps protect your application from cross-site request-forgery (XSRF) attacks.

→ Tip

You define the user-authentication method for a SAP HANA application in the application's runtime configuration, for example, using the *SAP HANA XS Administration Tool*. For the purposes of this tutorial, you do not need to change the runtime configuration.

9. Activate the new files in the SAP HANA repository.

Activating a file makes the file available to other project members. Right-click the new files (or the folder/package containing the files) and select **Team > Activate** from the context-sensitive popup menu.

The activate operation publishes your work and creates the corresponding catalog objects; you can now test it.

Results

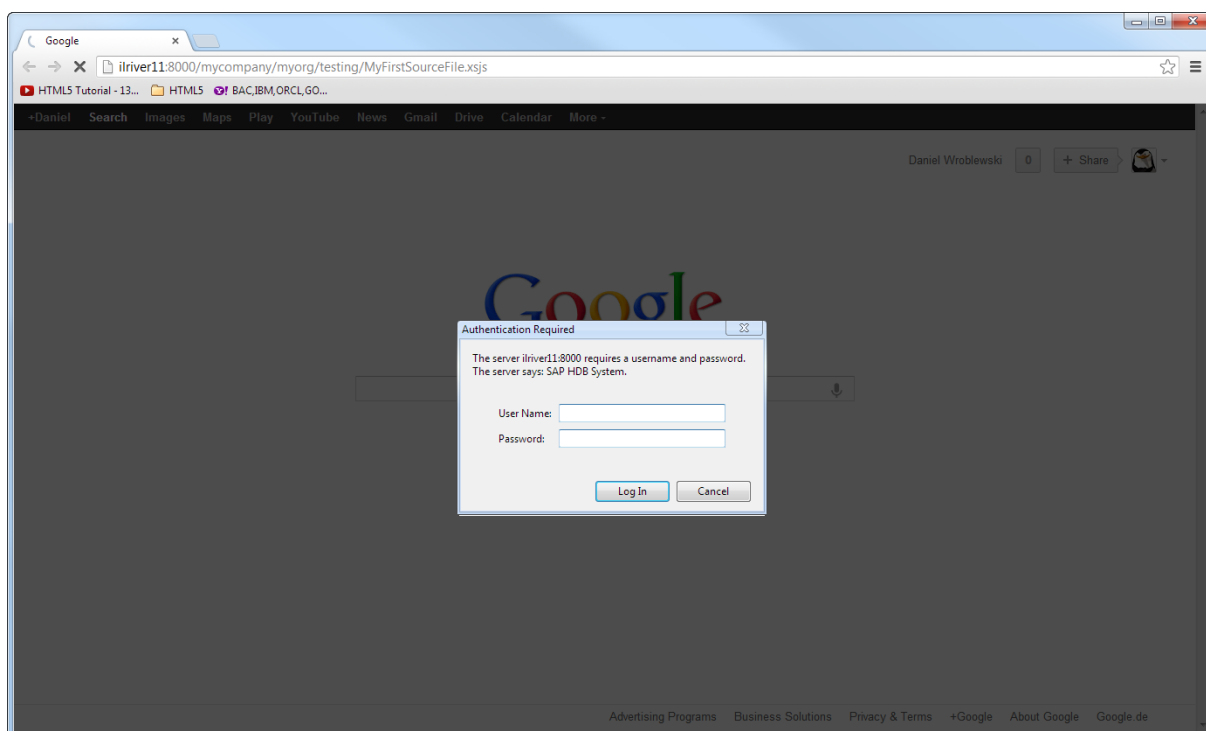
To access your JavaScript application, open a Web browser and enter the following URL, replacing **<myServer>** with the name of the server hosting your SAP HANA instance, and where appropriate the path to the server-side JavaScript source file:

```
http://<myServer>:8000/mycompany/myorg/testing/MyFirstSourceFile.xsjs
```

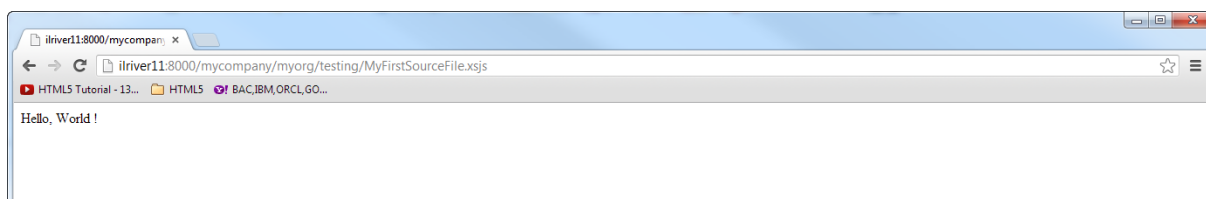
Note

For standard HTTP access, the port number is 80<SAPHANA_ID>, where **<SAPHANA_ID>** is two digits representing your SAP HANA instance number. For example, if your SAP HANA instance is 00, then the port number to use is 8000.

If everything works as expected, you should see the following result:



After logging in with your SAP HANA user name and password, the following page should be displayed:



3.5.1.6 Tutorial: Retrieve Data from SAP HANA

The final step of the data display tutorial is to extract data from the database and display it in a Web Browser.

Prerequisites

- Access to a running SAP HANA development system (with SAP HANA XS)
- A valid user account in the SAP HANA database on that system
- Access to SAP HANA studio
- Access to the shared project in the SAP HANA Repository which contains the artifacts used in this tutorial.

Context

To extract data from the database we use our JavaScript code to open a connection to the database and then prepare and run an SQL statement. The results are added to the response which is displayed in the Web browser. You use the following SQL statement to extract data from the database:

```
select * from DUMMY
```

The SQL statement returns one row with one field called *DUMMY*, whose value is *X*.

Procedure

1. Open SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. In the Project Explorer view, located the server-side JavaScript file `MyFirstSourceFile.xsjs` and open it in the embedded JavaScript editor.
4. In `MyFirstSourceFile.xsjs`, replace your existing code with the code in the following example.

```
$.response.contentType = "text/html";  
var output = "Hello, World !";
```

```

var conn = $.db.getConnection();
var pstmt = conn.prepareStatement( "select * from DUMMY" );
var rs = pstmt.executeQuery();
if (!rs.next()) {
    $.response.setBody( "Failed to retrieve data" );
    $.response.status = $.net.http.INTERNAL_SERVER_ERROR;
} else {
    output = output + "This is the response from my SQL: " + rs.getString(1);
}
rs.close();
pstmt.close();
conn.close();
$.response.setBody(output);

```

5. Save the file `MyFirstSourceFile.xsjs`.

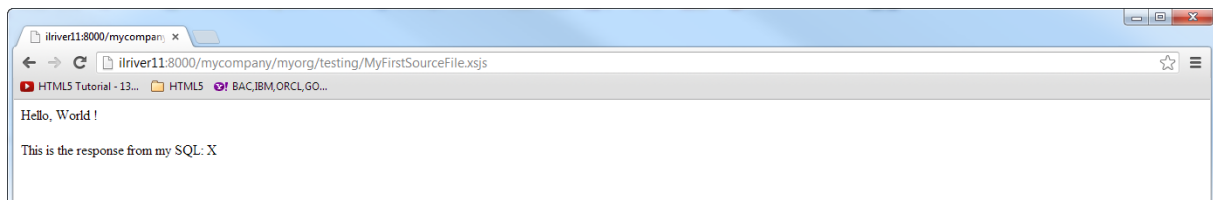
Note

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.

6. Activate the file `MyFirstSourceFile.xsjs` by right-clicking the file and choosing **Team > Activate**.

Results

In your browser, refresh the page. If everything works as expected, you should see the following page:



4 Setting Up Your Application

In SAP HANA Extended Application Services (SAP HANA XS), the design-time artifacts that make up your application are stored in the repository like files in a file system. You first choose a root folder for your application-development activities, and within this folder you create additional subfolders to organize the applications and the application content according to your own requirements.

Note

For the latest information about the availability of features for SAP HANA Extended Application Services (SAP HANA XS) and related development tools, see [1779803](#) .

As part of the application-development process, you typically need to perform the tasks described in the following list. Each of the tasks in more detail is described in its own section:

Application Setup Steps

Step	Action	Notes
1	Check roles and permissions	Before you start developing applications using the features and tools provided by the SAP HANA XS, developers who want to build applications to run on SAP HANA XS need to be granted access to development tools, SAP HANA systems, database accounts, and so on.
2	Set up delivery units	To create and manage delivery units, for example, using the SAP HANA Application Lifecycle Management , you must set the identity of the vendor with whom the delivery units are associated. To avoid conflicts with applications from SAP or other providers, we recommend that you use the DNS name of your company as the name of your root application-development folder, for example, <code>com.acme</code> .
3	Set up an SAP HANA project	In SAP HANA, projects enable you to group together all the artifacts you need for a specific part of the application-development environment. To create a project, you must first create a repository workspace, a directory structure to store files on your PC.
4	Maintain repository packages	To perform the high-level tasks that typically occur during the process of maintaining repository packages, you need to be familiar with the concepts of packages and package hierarchies, which you use to manage the artifacts in your applications.
5	Maintain application descriptors	The framework defined by the application descriptors includes the root point in the package hierarchy where content is to be served to client requests; it also defines if the application is permitted to expose data to client requests and what kind of access to the data is allowed.
6	Maintain application security	As part of the application-development process, you must decide how to grant access to the applications you develop. For example, you must specify which (if any) authentication method is used to grant access to content exposed by an application, and what content is visible.

Related Information

[Roles and Permissions for XS Development \[page 55\]](#)

[Maintaining Delivery Units \[page 57\]](#)

[Using SAP HANA Projects \[page 62\]](#)

[Maintaining Repository Packages \[page 70\]](#)

[Creating the Application Descriptors \[page 81\]](#)

[Set up Application Security \[page 105\]](#)

4.1 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 \[page 55\]](#)
- [SAP HANA XS Advanced Model \[page 56\]](#)

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

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)

📌 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 \[page 693\]](#)

[Assign Repository Package Privileges \[page 76\]](#)

[SAP HANA Web-Based Development Workbench](#)

4.2 Maintaining Delivery Units

A delivery unit (DU) is a collection of packages that are to be transported together. You assign all the packages belonging to your application to the same DU to ensure that they are transported consistently together within your system landscape. Each DU has a unique identity.

Prerequisites

To maintain delivery units with the SAP HANA Application Lifecycle Management, you must ensure the following prerequisites are met:

- You have access to an SAP HANA system.
- You have been assigned the privileges granted by a role based on the SAP HANA `sap.hana.xs.lm.roles::Administrator` user role template.
- A vendor ID (repository namespace) is already defined.

Context

The identity of a delivery unit consists of two parts: a vendor name and a delivery-unit name. The combined ID ensures that delivery units from different vendors are easy to distinguish and follows a pattern that SAP uses for all kinds of software components.

To create and manage delivery units you first need to maintain the identity of the vendor, with whom the delivery units are associated, and in whose namespace the packages that make up the delivery unit are stored. As part of the vendor ID maintenance process, you must perform the following tasks:

Procedure

1. Understand delivery units.
You must be familiar with the conventions that exist for delivery-unit names and understand the phases of the delivery-unit lifecycle.
2. Maintain details of the vendor ID associated with a delivery unit.
Delivery units are located in the namespace associated with the vendor who creates them and who manages the delivery-unit's lifecycle.
3. Create a delivery unit.
Create a transportable "container" to hold the repository packages in application.
4. Assign packages to a delivery unit.
Add to a delivery unit the repository packages that make up your application.
5. Export a delivery unit.
You can export the contents of a delivery unit from the SAP HANA Repository to a compressed Zip archive, which you can download to a client file system.
6. Import a delivery unit.
You can import the contents of a delivery unit into the SAP HANA Repository, for example, from a compressed Zip archive, which you upload from a client file system.

Related Information

[Maintain the Delivery-Unit Vendor ID \[page 59\]](#)

[Create a Delivery Unit \[page 60\]](#)

[Export a Delivery Unit](#)

[Import a Delivery Unit](#)

4.2.1 Maintain the Delivery-Unit Vendor ID

In SAP HANA, the [vendor ID](#) is used primarily to define the identity of the company developing a software component that it plans to ship for use with SAP HANA, for example, "sap.com". To create a delivery unit, it is a prerequisite to maintain a vendor ID in your system.

Prerequisites

To set the vendor ID, you must ensure the following prerequisites are met:

- You have access to an SAP HANA system.
- You have been assigned the privileges granted by a role based on the SAP HANA XS `sap.hana.xs.lm.roles::Administrator` user role template.

Context

Before creating your own first delivery unit, you must set the identity of the vendor in the development system's configuration. To maintain details of the delivery-unit vendor ID, perform the following steps:

Procedure

1. Start the SAP HANA Application Lifecycle Management.

The SAP HANA Application Lifecycle Management is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/lm`

ⓘ Note

To start the SAP HANA Application Lifecycle Management, you must use the logon credentials of an existing database user, who has the appropriate user role assigned.

2. Choose the [SETTINGS](#) tab.
3. Maintain details of the vendor ID.

In the [SETTINGS](#) tab, perform the following steps:

- a. Choose [Change Vendor](#).
 - b. In the [Set Vendor](#) dialog, enter the name of the new vendor, for example, `mycompany.com`.
 - c. Choose [OK](#) to save the changes.
- The new vendor ID appears in the [Vendor](#) box.

ⓘ Note

The vendor ID is required to create a delivery unit.

Related Information

[SAP HANA Application Lifecycle Management](#)

4.2.2 Create a Delivery Unit

A delivery unit (DU) is a group of transportable packages that contain objects used for content delivery. You can use the SAP HANA Application Lifecycle Management to create a DU for your application content or your software component.

Prerequisites

To create a delivery unit with the SAP HANA Application Lifecycle Management, you must ensure the following prerequisites are met:

- You have access to an SAP HANA system.
- You have the privileges granted by a role based on the SAP HANA `sap.hana.xs.lm.roles::Administrator` user role template.
- The vendor ID is defined for the DU; the vendor ID defines the repository namespace in which the new DU resides.

Context

You use a DU to transport the design-time objects that are stored in the SAP HANA repository between two systems, for example, from a development system to a consolidation system. To create a new delivery unit using the SAP HANA application lifecycle management, perform the following steps.

Procedure

1. Open SAP HANA Application Lifecycle Management.
SAP HANA Application Lifecycle Management is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/lm`
2. Choose the *PRODUCTS* tab.
3. Choose the *Delivery Units* tab.
4. Choose *Create*.
The *New Delivery Unit* dialog box appears.
5. Enter details for the new DU.
When entering details, note the following points:

- **Name**
The field is mandatory and you must follow strict naming conventions, for example, use capital letters.
- **Vendor**
This field is mandatory. However, you cannot enter a vendor here; the box is populated by the value you enter when defining the vendor in the [SETTINGS](#) tab.
- **Version**
Version numbers must take the form “#.#.#”, for example, **1.0.5**, where:
 - **1** = the DU version number
 - **0** = the support package version (if required)
 - **5** = the patch version (if required)

📌 Note

The numbers you enter here refer to the application component that you are developing; the numbers do not refer to the patch or service-pack level deployed on the SAP HANA server.

6. Choose [Create](#).
The new delivery unit is added to the SAP HANA repository in the namespace specified by the vendor ID and the application path.
7. Check the status bar at the bottom of the browser window for error messages. Choose the message link to display the message text.

Results

You have created a delivery unit.

Related Information

[SAP HANA Application Lifecycle Management](#)
[SAP HANA Change Recording](#)
[Enable SAP HANA Change Recording](#)

4.2.2.1 SAP HANA Delivery Unit Naming Conventions

The delivery unit (DU) is the vehicle that SAP HANA application lifecycle management uses to ship software components from SAP (or a partner) to a customer. The DU is also the container you use to transport application content in your system landscape. In SAP HANA, the name of a DU must adhere to conventions and guidelines.

If you create a delivery unit, the name of the new delivery unit must adhere to the following conventions

- A delivery-unit name must contain only capital letters (A-Z), digits (0-9), and underscores (_).
- The name must start with a letter.

- The maximum length of a delivery-unit name must not exceed 30 characters

ⓘ Note

The naming conventions for packages in a delivery unit differ from the naming conventions that apply to the delivery unit itself. For example, the maximum length of a package name is not restricted to 30 characters; however, it must be less than 190 characters (including the namespace hierarchy).

4.3 Using SAP HANA Projects

Projects group together all the artifacts you need for a specific part of the application-development environment.

Context

Before you can start the application-development workflow, you must create a project, which you use to group together all your application-related artifacts. However, a project requires a repository workspace, which enables you to synchronize changes in local files with changes in the repository. You can create the workspace before or during the project-creation step. As part of the project-creation process, you perform the following tasks:

Procedure

1. Create a development workspace.

The workspace is the link between the SAP HANA repository and your local filesystem, where you work on project-related objects.

2. Create a project.

Create a new project for a particular application or package; you can use the project to collect in a convenient place all your application-related artifacts.

3. Share a project.

Sharing a project enables you to ensure that changes you make to project-related files are visible to other team members and applications. Shared projects are available for import by other members of the application-development team.

ⓘ Note

Files checked out of the repository are not locked; conflicts resulting from concurrent changes to the same file must be resolved manually, using the [Merge](#) tools provided in the context-sensitive [Team](#) menu.

4. Import a project.

Import a project (and its associated artifacts) that has been shared by another member of the application-development team.

Related Information

[Maintain a Repository Workspace \[page 63\]](#)

[Create a Project for SAP HANA XS \[page 65\]](#)

[Share an SAP HANA XS Project \[page 68\]](#)

[Import an SAP HANA XS Project \[page 69\]](#)

4.3.1 Maintain a Repository Workspace

A workspace is a local directory that you map to all (or part) of a package hierarchy in the SAP HANA repository. When you check out a package from the repository, SAP HANA copies the contents of the package hierarchy to your workspace, where you can work on the files.

Context

Before you can start work on the development of the application, you need to set up a workspace, where you store checked-out copies of your application's source-code files. To ensure that only the owner of data can access the data stored in a workspace, a workspace must be created in the owner's home directory. In addition, it is recommended that users encrypt the data on their hard drives using an encryption tool.

To create a new workspace in the SAP HANA studio, perform the following steps:

Procedure

1. Open the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Choose the *Repositories* view.
4. Choose *Create Workspace...*

The *Create Workspace...* button is located in the top right-hand corner of the *Repositories* view.

5. Specify the workspace details. In the *Create New Repository Workspace* dialog, enter the following information and choose *Finish*:
 - a. Specify the SAP HANA system, for which you want to create a new workspace.
 - b. Enter a workspace name, for example the name of the SAP HANA system where the repository is located. To avoid the potential for confusion, it is recommended to associate one workspace with one repository.

- c. Specify where the workspace root directory should be located on your local file system, for example:
`C:\users\username\workspaces`

The new workspace is displayed in the [Repositories](#) view.

Note

Although the packages and objects in the chosen repository are visible in the [Repositories](#) view, you cannot open or work on the objects here. To work on objects, you must create a project and use the [Project Explorer](#) view.

The [Repositories](#) view displays the status of a workspace as follows:

UI Icon	Explanation
Yellow database icon	An inactive workspace exists in the SAP HANA repository
Yellow database icon with a blue check mark	An inactive workspace has been imported to your local file system (and the contents checked out from the SAP HANA repository)

6. Remove a repository workspace.

If it is necessary to remove a workspace, you can choose between multiple deletion options; the option you choose determines what is removed, from where (local file system or remote repository), and what, if anything, is retained.

- a. Open the [SAP HANA Development](#) perspective.
- b. Choose the [Repositories](#) view and expand the repository node containing the workspace you want to remove.
- c. Right-click the workspace you want to remove.
- d. Choose the workspace-deletion mode.

The following modes apply when you delete a workspace in SAP HANA studio:

Workspace Deletion Modes

Workspace Deletion Mode	Description
Delete	Remove workspace; delete all workspace-related local files; delete related changes to remote (repository) data.
Remove from client (keep remote changes)	Remove workspace from local client system; delete all local workspace-related files; retain changes to remote (repository) data.
Disconnect local from remote (keep changes)	Keep the workspace but remove the workspace label from the list of workspaces displayed in the Repositories view. The connection to the disconnected workspace can be reestablished at any time with the option Import Local Repository Workspaces .

4.3.1.1 SAP HANA Repository Workspaces

The place where you work on project-related objects is called a repository **workspace**. A workspace is an environment that maps a local directory to all (or part) of a package hierarchy in the SAP HANA repository.

In SAP HANA studio, the repository tools enable you to browse the entire hierarchy of design-time objects stored in the repository. However, when you check a package out of the repository, SAP HANA copies the contents of the package hierarchy to your workspace, where you can work on the files in your local file system.

Note

Before you can create a workspace you must maintain connection information in the SAP HANA database user store.

To start development work with SAP HANA studio, for example, to checkout the contents of a package, you must create a repository **workspace**. The workspace contains a system folder with metadata and package folders for the repository content. The file-system folders and their subfolders reflect the package hierarchy in the repository; the repository client ensures that changes are synchronized.

In the SAP HANA studio, the *Repositories* view displays the status of a workspace as follows:

- Yellow database icon
An inactive workspace exists in the SAP HANA repository
- Yellow database icon with a blue check mark
An inactive workspace has been imported to your local file system (and the contents checked out from the SAP HANA repository)

4.3.2 Create a Project for SAP HANA XS

Before you can start the application-development workflow, you must create a project, which you use to group all your application-related artifacts.

Context

Projects group together all the artifacts you need for a specific part of your application-development environment. A basic project contains folders and files. More advanced projects are used for builds, version management, sharing, and the organization and maintenance of resources.

To create a new project in the SAP HANA studio, perform the following steps:

Procedure

1. Open the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.

3. Choose the *Project Explorer* view.
4. Choose **File > New > Project...** or right-click the white space in the *Project Explorer* view and choose *New > Project...* in the popup menu.

The type of project you create determines the details you have to provide in the *New Project* wizard that appears. Choose **SAP HANA > Application Development > XS Project**.

- a. Enter the following details for the new XS project:

- **Shared project**
This is the default setting. Sharing a project enables continuous synchronization between your local-file system workspace and the SAP HANA repository. If you choose not to share the project at this point, you can share the new project manually later.
- **Project name**
Enter a project name that describes what the project is about, for example: **XS_JavaScript** or **XS_SAPUI5**. Since a project name must be unique within the same Eclipse workspace, it is recommended to use the fully qualified package name as the project name.
- **Project location**
You can save the project in the default location, which is the SAP HANA studio (Repository) workspace. To save the project in an alternative location from the recommended default, first disable the option *Share project in SAP repository*.
You can share the new project manually later. Sharing a project enables continuous synchronization with the SAP HANA repository.
- **Working sets**
A working set is a concept similar to favorites in a Web browser, which contain the objects you work on most frequently.
- **Repository workspace and package**
For a shared project, you can set the project location by selecting a repository workspace and package.
- **Common objects**
For a shared project, you can include some commonly used objects in your project. Some of these will provide you with a basic template to begin with.
- **Access objects**
For a shared project, the access objects are checked by default. However, if either an *.xsaccess* file or an *.xsapp* file already exists in the folder you have chosen to create the new project, the corresponding option is automatically unchecked and greyed out.

- b. Click *Finish* to create the new project.

All the objects included are activated automatically when the project is created. The new project is displayed in the *Project Explorer* view.

Note

- If there is an error during activation of one of the project objects, none of the objects will be automatically activated. You can manually correct the error and then manually activate the objects.
- The contents of the project depend on the type of project you create. For example, a general project is empty immediately after creation; a JavaScript project contains all the resource files associated with a JavaScript project, such as libraries and build-environment artifacts.

4.3.2.1 SAP HANA Studio Projects

Before you can start the application-development workflow, you must create a project, which you use to group all your application-related artifacts.

Projects group together all the artifacts you need for a specific part of the application-development environment. A basic project contains folders and files. More advanced projects are used for builds, version management, sharing, and the organization and maintenance of resources.

Projects enable multiple people to work on the same files at the same time. You can use SAP HANA studio to perform the following project-related actions in the repository:

- [Checkout](#) folders and files from the repository
- [Commit](#) changes to the repository
- [Activate](#) the committed changes
- [Revert](#) inactive changes to the previously saved version

Note

Files checked out of the repository are not locked; conflicts resulting from concurrent changes to the same file must be resolved manually, using the [Merge](#) tools provided in the context-sensitive [Team](#) menu.

By committing project-related files to the repository and activating them, you enable team members to see the latest changes. The commit operation detects all changes in packages that you configure SAP HANA studio tool to track and writes the detected changes back to the repository. The repository client tools also support synchronization with changes on the server, including conflict detection and merging of change. All workspace-related repository actions are available as context-sensitive menu options in SAP HANA studio. For example, if you right click a repository object at the top of the package hierarchy in the *Project Explorer* in SAP HANA studio, you can commit and activate **all** changed objects within the selected hierarchy.

Note

If you create a new project using SAP HANA studio, you can assign the new project to an existing workspace.

You can share and unshare projects. Sharing a project associates it with a particular package in the repository linked to a particular workspace. The act of sharing the project sets up a link between the workspace and the repository and enables you to track and synchronize local changes with the versions of the objects stored in the repository. When a project is shared, it becomes available to other people with authorization to access to the repository, for example, colleagues in an application-development team. Team members can import a shared project and see and work on the same files as the creator of the project.

Note

Always unshare a project before deleting it.

In the SAP HANA studio you can create a project at any package level, which enables a fine level of control of the artifacts that may (or may not) be exposed by sharing the project.

4.3.3 Share an SAP HANA XS Project

Before you can start working on files associated with a new project, you must share the project; sharing a project enables you to track and synchronize local changes with the repository.

Context

When you share a project, you set up a connection to the SAP HANA repository associated with a particular SAP HANA instance. Sharing the project enables you to ensure that changes you make to project-related files are visible to other team members and applications. Other developers can import a shared project and work on the same files.

Note

Use the *Project Explorer* view in the SAP HANA studio to check if a project is shared. In addition to the project name, a shared project displays the SAP HANA system ID of the repository where the shared artifacts are located, an SAP HANA user name, and the path to the repository package to which the shared project is assigned, for example. "XSJS_myproject [SID (dbusername, 'sap.hana.xs.app1')]."

To share a project in the SAP HANA studio, perform the following steps:

Procedure

1. Open the SAP HANA studio
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Share the project.

Right-click the project you want to share and choose **Team > Share Project...** in the pop-up menu.

5. Select the repository *type*.

The *Share Project* dialog displays a list of all available repository types; choose *SAP HANA Repository* and choose *Next*.

6. Select the repository workspace where the project should be located.
7. Specify the package that you want to associate the shared project with.

The *Share Project* dialog displays the suggested location for the shared project in the *New Project location* screen area. The default location is the name of the workspace with the name of the project you want to share. Choose *Browse...* to locate the package you want to associate the shared project with. The selected package is displayed in the *Path to package* text box.

Note

The *Keep project folder* option appends the name of the project you are sharing to the name of the workspace in which you are sharing the project and creates a new package with the name of the shared project under the workspace location displayed. Use this option only if you want to create multiple

projects for a selected package, for example, if you are creating a root project in your root application package.

8. Click *Finish* to complete the project-sharing procedure.
9. Add new files as required

At this point you can start adding project-specific files to the shared project. These artifacts can then be committed to the repository, where they reside as inactive objects until they are activated, for example, using the **Team > Activate** option in the context-sensitive menus available in the *Project Explorer* view.

Note

The *Project Explorer* view decorates the file icons to indicate the current state of the repository files, for example: local (not yet committed), committed (inactive), and active (available for use by others).

10. Make the project available for import, for example, so that others can join it and make changes to project content.

The project-sharing procedure creates some artifacts (for example, the `.project` file) that must be committed to the repository and activated so that other team members can import the project more easily into their workspace. The `.project` file is used in several dialogs to populate the list of available projects.

Note

Use the *Repositories* view to import projects (and checkout project content).

Related Information

[Import an SAP HANA XS Project \[page 69\]](#)

4.3.4 Import an SAP HANA XS Project

Before you can start the application-development workflow, you must either create a new project and share it (with the repository), or import a shared project from the repository into your workspace. Importing a project enables you to track and synchronize local changes with the colleagues working on the objects in the imported project.

Context

To import an existing project from the repository into your workspace, perform the following steps.

Procedure

1. Open the SAP HANA studio
2. Open the *SAP HANA Development* perspective.
3. Choose the *Repositories* view.
4. Right-click the package where the project you want to import is located and choose *Checkout and Import Projects...* in the popup menu.

Projects can be assigned to a package at any level of the package hierarchy. If you know where the project is located, browse to the package first before choosing the *Checkout and Import Projects...* option. This reduces the amount of files to checkout and download to your local file system.

Note

The existence of a `.project` file in a package identifies the package as being associated with a project.

The SAP HANA studio checks out the content of the selected package and displays any projects it finds in the *Projects* screen area.

5. Select the projects to import.
If multiple projects are available for import, select the projects you want to import.
6. Choose *Finish* to import the selected projects.
You can add the imported project to your *Working Sets*.

Note

A working set is a concept similar to favorites in a Web browser, which contain the objects you work on most frequently.

4.4 Maintaining Repository Packages

All content delivered as part of the application you develop for SAP HANA is stored in packages in the SAP HANA repository. The packages are arranged in a hierarchy that you define to help make the process of maintaining the packages transparent and logical.

Context

To perform the high-level tasks that typically occur during the process of maintaining repository packages, you need to be familiar with the concepts of packages and package hierarchies. Packages enable you to group together the artifacts you create and maintain for your applications. You must also be aware of the privileges the application developers require to access (and perform operations on) the packages.

Note

You can also create and delete packages in the *Project Explorer*, for example, by creating or deleting folders in shared projects and committing and activating these changes. However, to maintain advanced package

properties (for example, privileges, component, the package maintainer, and so on) you must use the [Modeling](#) perspective in the SAP HANA studio.

As part of the process of maintaining your application packages, you typically perform the following tasks:

Procedure

1. Define the package hierarchy

The package hierarchy is essential for ease of maintenance as well as the configuration of access to packages and the privileges that are required to perform actions on the packages.

2. Define package privileges

You can set package authorizations for a specific user or for a role. Authorizations that are assigned to a repository package are implicitly assigned to all sub-packages, too.

3. Create a package

Packages are necessary to group logically distinct artifacts together in one object location that is easy to transport.

Related Information

[Creating a Package \[page 79\]](#)

[Defining the Package Hierarchy \[page 71\]](#)

[Defining Package Privileges \[page 76\]](#)

4.4.1 Define the Repository Package Hierarchy

Packages belonging to an application-development delivery unit (DU) should be organized in a clear hierarchical structure under a single [root package](#) representing the vendor, for example, **com.acme**.

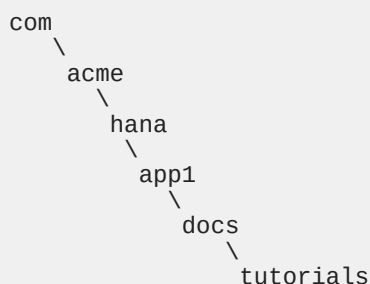
Context

The package hierarchy for a new project might include sub-packages, for example, to isolate the data model from the business logic. Although there are no package interfaces to enforce visibility of objects across packages, this separation of logical layers of development is still a recommended best practice.

❗ Note

You can only assign one project per package; this is important to remember if you have a mixture of design-time objects that need to be used in multiple projects, for example: server-side JavaScript (XSJS), SAPUI5, and a general project (for procedures).

The following simple example shows a package structure containing tutorials for the use of a new application:



- Package hierarchy
Each vendor uses a dedicated namespace, for example, `com.acme`.

Note

Do not use the namespace `sap` to build your application hierarchy. The namespace `sap` is reserved for use by SAP; packages created in the `sap` namespace are overwritten by system updates.

- Package type
Some packages contain content; other packages contain only other (sub)packages. Packages can also contain both objects and (sub)packages.
- Package naming conventions
There are recommendations and restrictions regarding package names.

To set up a package hierarchy in the SAP HANA repository, perform the following steps:

Procedure

1. Create a new root package.
Open the [SAP HANA Development](#) perspective, choose the [Systems](#) view, and perform the following steps:
 - a. Select the SAP HANA system where you want to create a new package and expand the [Content](#) node to display the namespace hierarchy for package content.
 - b. Choose  [New > Package](#) .
2. Maintain the package details.
In the [Create Package](#) dialog, perform the following steps:
 - a. Enter the name of the package (mandatory).
Guidelines and conventions apply to package names.
 - b. Enter a package description (optional).
 - c. Specify the delivery unit that the package is assigned to.
You can add additional packages to a delivery unit at a later point in time, too.
 - d. Specify a language for the package content.
 - e. Assign responsibility of the package to a specific user (optional).
By default, the responsible user for a new package is the database user connected to the SAP HANA repository in the current SAP HANA studio session.
 - f. Maintain translation details.

If you plan to have the content translated, you need to maintain the translation details; this is covered in another topic.

g. Choose **OK** to save the changes and create the new package.

3. Create a new subpackage.

In the **Systems** view of the **SAP HANA Development** perspective, perform the following steps:

a. Right-click the package to which you want to add a new subpackage.

b. In the pop-up menu, choose **► New > Package... ►**

4. Maintain the subpackage details.

In the **Create Package** dialog, perform the following steps:

a. Enter the name of the subpackage (mandatory).

Guidelines and conventions apply to package names.

b. Enter a description for the new subpackage (optional).

c. Specify the delivery unit that the subpackage is assigned to.

You can add additional packages to a delivery unit at a later point in time, too.

d. Specify a language for the subpackage content.

e. Assign responsibility of the subpackage to a specific user (optional).

By default, the responsible user for a new package is the database user connected to the SAP HANA repository in the current SAP HANA studio session.

f. Maintain translation details.

If you plan to have the content translated, you need to maintain the translation details; this is covered in another topic.

g. Choose **OK** to save the changes and create the new subpackage.

Related Information

[SAP HANA Delivery Unit Naming Conventions \[page 61\]](#)

4.4.1.1 Repository Package Hierarchy

A package hierarchy can include sub-packages, for example, to isolate the data model from the business logic.

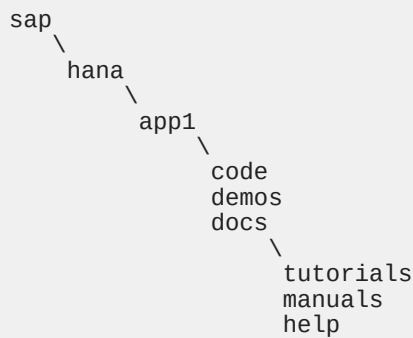
You can create a package hierarchy, for example, by establishing a parent-child type relationship between packages. The assignment of packages to delivery units is independent of the package hierarchy; packages in a parent-child relationship can belong to different delivery units. SAP recommends that you assign to one specific delivery unit all packages that are part of a particular project or project area.

The package hierarchy for a new project typically includes sub-packages, for example, to isolate the data model from the business logic. Although there are no package interfaces to enforce visibility of objects across packages, this separation of logical layers of development is still a recommended best practice.

📘 Note

You can only assign one project per package; this is important to remember if you have a mixture of design-time objects that need to be used in multiple projects, for example: server-side JavaScript (XSJS), SAPUI5, and a general project (for procedures).

The following simple example shows a package structure containing tutorials for the use of a new application:



All content delivered by SAP should be in a sub-package of "sap". Partners and customers should choose their own root package to reflect their own name (for example, the domain name associated with the company) and must not create packages or objects under the "sap" root structural package. This rule ensures that customer- or partner-created content will not be overwritten by an SAP update or patch.

📘 Note

SAP reserves the right to deliver without notification changes in packages and models below the "sap" root structural package.

There are no system mechanisms for enforcing the package hierarchy. The "sap" root structural package is not automatically protected. However, by default you cannot change the content of packages that did not originate in the system. In addition, an authorization concept exists, which enables you to control who can change what inside packages.

4.4.1.2 SAP HANA Repository Packages and Namespaces

In SAP HANA, a *package* typically consists of a collection of repository objects, which can be transported between systems. Multiple packages can be combined in a delivery unit (DU).

An SAP HANA package specifies a namespace in which the repository objects exist. Every repository object is assigned to a package, and each package must be assigned to a specific delivery unit. In the repository, each object is uniquely identified by a combination of the following information:

- Package name
- Object name
- Object type

📘 Note

Multiple objects of the same type can have the same object name if they belong to different packages.

Before you start the package development process, consider the following important points:

- **Package hierarchy**
Each vendor uses a dedicated namespace, and the package hierarchy you create enables you to store the various elements of an application in a logical order that is easy to navigate.
- **Package type**
Packages can be structural or non-structural; some packages contain content; other packages contain only other (sub)packages.
- **Package naming conventions**
There are recommendations and restrictions regarding package names, for example, the name's maximum length and which characters must not be used.

Package Naming Conventions

The following rules apply to package names:

- **Permitted characters**
Lower/upper case letters (aA-zZ), digits (0-9), hyphens (-), underscores (_), and dots (.) are permitted in package names. Dots in a package name define a logical hierarchy. For example, "a.b.c" specifies a package "a" that contains sub-package "b", which in turn contains sub-package "c".
- **Forbidden characters**
A package name must not start with either a dot (.) or a hyphen (-) and cannot contain two or more consecutive dots (..).
- **Package name length**
The name of the complete package namespace hierarchy (for example, "aa.bb.cc.zz" including dots) must not be more than 190 characters long. In addition, on object activation, the maximum permitted length of a generated **catalog** name (which includes the package path, the separating dots, and the object base name) is restricted to 127 characters.
 - hdbtable hdbview, hdbsequence, hdbstructure, hdbprocedure objects

```
sap.test.hana.db::myObject
```

- CDS objects

```
sap.test.hana.db::myContext.myEntity
```

4.4.2 Assign Repository Package Privileges

In the SAP HANA repository, you can set package authorizations for a specific user or for a role.

Prerequisites

The following prerequisites are assumed for assigning package privileges:

- Administrator access to the SAP HANA repository
- Permission to modify user privileges (for example, to grant privileges to other SAP HANA users)

Context

Authorizations that are assigned to a repository package are implicitly assigned to all sub-packages, too. You can also specify if the assigned user authorizations can be passed on to other users. To set user (or role) authorizations for repository packages, perform the following steps:

Procedure

1. Open the [Systems](#) view in the SAP HANA studio's [SAP HANA Development](#) perspective.
2. In the [Systems](#) view, expand the [Security > Roles/Users](#) node for the system hosting the repository that contains the packages you want to grant access to.
You can also define roles via source files; roles defined in this way can be assigned to a delivery unit and transported to other systems.
3. Double click the user (or role) to whom you want to assign authorizations.
4. Open the [Package Privileges](#) tab page.
5. Choose [\[+\]](#) to add one or more packages. Press and hold the [Ctrl](#) key to select multiple packages.
6. In the [Select Repository Package](#) dialog, use all or part of the package name to locate the repository package that you want to authorize access to.
7. Select one or more repository packages that you want to authorize access to; the selected packages appear in the [Package Privileges](#) tab.
8. Select the packages to which you want authorize access and, in the [Privileges for](#) tab, check the required privileges, for example:
 - REPO.READ
Read access to the selected package and design-time objects (both native and imported)
 - REPO.EDIT_NATIVE_OBJECTS
Authorization to modify design-time objects in packages originating in the system the user is working in

Related Information

[Package Privilege Options \[page 77\]](#)

[Package Privileges \[page 730\]](#)

4.4.2.1 Package Privilege Options

Package privileges authorize actions on individual packages in the SAP HANA repository. In the context of repository package authorizations, there is a distinction between native packages and imported packages.

Note

To be able perform operations in all packages in the SAP HANA repository, a user must have privileges on the root package `.REPO_PACKAGE_ROOT`.

Privileges for Native Repository Packages

A native repository package is created in the current SAP HANA system and expected to be edited in the current system. To perform application-development tasks on **native** packages in the SAP HANA repository, developers typically need the privileges listed in the following table:

Package Privilege	Description
REPO.READ	Read access to the selected package and design-time objects (both native and imported)
REPO.EDIT_NATIVE_OBJECTS	Authorization to modify design-time objects in packages originating in the system the user is working in
REPO.ACTIVATE_NATIVE_OBJECTS	Authorization to activate/reactivate design-time objects in packages originating in the system the user is working in
REPO.MAINTAIN_NATIVE_PACKAGES	Authorization to update or delete native packages, or create sub-packages of packages originating in the system in which the user is working

Privileges for Imported Repository Packages

An imported repository package is created in a remote SAP HANA system and imported into the current system. To perform application-development tasks on **imported** packages in the SAP HANA repository, developers need the privileges listed in the following table:

Note

It is not recommended to work on imported packages. Imported packages should only be modified in exceptional cases, for example, to carry out emergency repairs.

Package Privilege	Description
REPO.READ	Read access to the selected package and design-time objects (both native and imported)
REPO.EDIT_IMPORTED_OBJECTS	Authorization to modify design-time objects in packages originating in a system other than the one in which the user is currently working
REPO.ACTIVATE_IMPORTED_OBJECTS	Authorization to activate (or reactivate) design-time objects in packages originating in a system other than the one in which the user is currently working
REPO.MAINTAIN_IMPORTED_PACKAGES	Authorization to update or delete packages, or create sub-packages of packages, which originated in a system other than the one in which the user is currently working

Related Information

[Package Privileges \[page 730\]](#)

4.4.3 Create a Repository Package

In SAP HANA, a package contains a selection of repository objects. You assemble a collection of packages into a delivery unit, which you can use to transport the repository objects between SAP HANA systems.

Context

You can use repository packages to manage the various elements of your application development project in the SAP HANA repository. To create a package, perform the following steps:

Procedure

1. In the SAP HANA studio, start the *SAP HANA Development* perspective.
2. In the *Systems* view, select the SAP HANA system where you want to create a new package and expand the *Content* node to display the namespace hierarchy for package content.
3. Right-click the package where you want to add a new package and choose  in the context-sensitive popup menu.
SAP HANA studio displays the *New Package* dialog.
4. Maintain the package details.
In the *New Package* dialog, enter information in the following fields:
 - a. Enter a name for the new package.
The package *Name* is mandatory. Add the new name to the end of the full package path, for example, `acme.com.package1`.
 - b. Fill in the other optional information as required:
Use the *Delivery Unit* drop-down list to assign the new package to a delivery unit.
Choose *Translation* if you intend to have the package content localized. You must maintain the translation details.
5. Create the new package.
In the *New Package* dialog, click *OK* to create a new package in the specified location.
6. Activate the new package.
In the *Systems* view, right-click the new package and choose *Activate* from the context-sensitive popup menu.

4.4.3.1 Repository Package Types

SAP HANA enables the use of various types of package, which are intended for use in particular scenarios.

SAP HANA Application Services provide or allow the following package **types**:

- **Structural**
Package only contains sub-packages; it cannot contain repository objects.
- **Non-Structural**
Package contains both repository objects and subpackages.

The following packages are delivered by default with the repository:

- **sap**
Transportable package reserved for content delivered by SAP. Partners and customers must not use the `sap` package; they must create and use their own root package to avoid conflicts with software delivered by SAP, for example when SAP updates or overwrites the `sap` package structure during an update or patch process.
- **system-local**
Non-transportable, structural packages (and subpackages). Content in this package (and any subpackages) is considered system local and cannot be transported. This is similar to the concept of the `$tmp` development class in SAP NetWeaver ABAP.
- **system-local.generated**
Non-transportable, structural packages for generated content, that is; content not created by manual user interaction
- **system-local.private**
Non-transportable, structural package reserved for objects that belong to individual users, for example, `system-local.private.<user_name>`. To avoid compatibility issues with future functionality, do not use the `system-local.private` package or any of its sub-packages.

4.4.4 Delete a Repository Package

In SAP HANA development, repository packages are used to manage various elements of your application development project. Sometimes you need to delete a package that contains repository objects from other developers.

Prerequisites

To perform this task, your user must be assigned the `REPO.WORK_IN_FOREIGN_WORKSPACE` system privilege.

Context

You use repository packages to manage the various elements of your application development project in the SAP HANA repository. To delete a package, perform the following steps:

Procedure

1. In the SAP HANA studio, start the *SAP HANA Development* perspective.
2. Open the *Repositories* view and locate the package that you want to delete.
3. Delete the package.
 1. Click the alternate mouse button on the package that you want to delete and choose *Delete*.
 2. When prompted, choose *OK*.
A message box appears indicating that you are deleting a package with active and inactive objects.
 3. Choose *OK* to delete the package.
Choose *Cancel* to stop the deletion of the package and objects.

4.5 Creating the Application Descriptors

The application descriptors describe the framework in which an SAP HANA XS application runs. The framework defined by the application descriptors includes the root point in the package hierarchy where content is to be served to client requests, and who has access to the content.

Prerequisites

- You must be familiar with the concept of the application descriptor file (*.xsapp*), the application-access file (*.xsaccess*), and if required, the application-privileges file (*.xsprivileges*).

Context

When you develop and deploy applications in the context of SAP HANA Extended Application Services (SAP HANA XS), you must define the application descriptors. Maintaining the application descriptors involves the following tasks:

Procedure

1. Create an application-descriptor file.
The package that contains the application descriptor file becomes the root path of the resources exposed to client requests by the application you develop.
2. Create an application-access file.
The application-access file enables you to specify who or what is authorized to access the content exposed by a SAP HANA XS application package and what content they are allowed to see. You can use keywords in the application-access file to set authentication rules, define package-privilege levels (for example, EXECUTE or ADMIN, specify the connection security level (for example, SSL/HTTPS), and allow (or prevent) the creation of entity tags (Etags). You can also define rewrite rules for URLs exposed by an application, for example, to hide internal details of URL paths from external users, clients, and search engines.
3. Create an application-privileges file. (Optional)
The application-privileges file enables you to define the authorization privileges required for access to an SAP HANA XS application, for example, to start the application (EXECUTE) or to perform administrative actions on an application (ADMIN). The privileges defined here are activated for a particular application in the application-access file. These privileges can be checked by an application at runtime. Privileges defined apply to the package where the privileges file is located as well as any packages further down the package hierarchy unless an additional privileges file is present, for example, in a subpackage.

Related Information

[Create an application descriptor \[page 83\]](#)

[Create an application-access file \[page 85\]](#)

[Create an application-privileges file \[page 100\]](#)

4.5.1 Create an Application Descriptor File

Each application that you want to develop and deploy on SAP HANA Extended Application Services (SAP HANA XS) must have an application-descriptor file. The application descriptor is the core file that you use to describe an application's framework within SAP HANA XS.

Prerequisites

- A repository workspace with a shared project
- A root package for your application, for example, `MyAppPackage`

Note

The namespace `sap` is restricted. Place the new package in your own namespace, for example, `com.acme`, which you can create alongside the `sap` namespace.

Context

The application descriptor is the core file that you use to indicate an application's availability within SAP HANA XS. The application descriptor marks the point in the package hierarchy at which an application's content is available to clients. The application-descriptor file has no contents and no name; it only has the file extension `.xsapp`. The package that contains the application-descriptor file becomes the root path of the resources exposed by the application you develop.

Note

For backward compatibility, content is allowed in the `.xsapp` file but ignored.

To create an application descriptor for your new application, perform the following steps.

Procedure

1. In the SAP HANA studio, open the *SAP HANA Development* perspective.
2. In the *Project Explorer* view, right-click the folder where you want to create the new (`.xsapp`) file.
3. In the context-sensitive popup menu, choose **New** > *Other...*.
4. In the *Select a Wizard* dialog, choose **SAP HANA** > *Application Development* > *XS Application Descriptor File*.
5. Enter or select the parent folder. Note that the default file name for the XS application descriptor is `.xsapp` and cannot be changed.
6. Select a template to use. Templates contain sample source code to help you get started.

7. Choose *Finish*.

If you are using the SAP HANA Studio to create artifacts in the SAP HANA Repository, the file creation wizard adds the required file extension `.xsapp` automatically.

→ Tip

Files with names that begin with the period (`.`), for example, `.xsapp`, are sometimes not visible in the *Project Explorer*. To enable the display of all files in the *Project Explorer* view, use the  *Customize View*  *Available Customization*  option and clear all check boxes.

8. Save and activate your changes and additions.
 - a. In the *SAP HANA Development* perspective, open the *Project Explorer* view and right-click the new (`.xsapp`) package.
 - b. In the context-sensitive popup menu, choose  *Team*  *Activate* .

4.5.1.1 The SAP HANA XS Application Descriptor

Each application that you want to develop and deploy on SAP HANA Extended Application Services (SAP HANA XS) must have an application descriptor file. The application descriptor is the core file that you use to describe an application's framework within SAP HANA XS.

The package that contains the application descriptor file becomes the root path of the resources exposed to client requests by the application you develop.

📌 Note

The application-descriptor file has no name and no content; it only has the file extension “`xsapp`”, for example, `.xsapp`. For backward compatibility, content is allowed in the `.xsapp` file but ignored.

The application root is determined by the package containing the `.xsapp` file. For example, if the package `sap.test` contains the file `.xsapp`, the application will be available under the URL `http://<host>:<port>/sap.test/`. Application content is available to requests from users.

⚠ Caution

Make sure that the folder containing the `.xsapp` application descriptor file also contains an `.xsaccess` file, which controls access to the application.

The contents of the package where the `.xsapp` file resides (and any subfolders) are exposed to user requests and, as a result, potentially reachable by attackers. You can protect this content with the appropriate authentication settings in the corresponding application-access (`.xsaccess`) file, which resides in the same package. Bear in mind that by exposing Web content, you run the risk of leaking information; the leaked information can be used in the following ways:

- Directly
Data files such as `.csv` files used for the initial database load can contain confidential information.
- Indirectly
File descriptors can give details about the internal coding of the application, and files that contain the names of developers are useful; they can be used by an attacker in combination with social-engineering techniques.

To help protect your application from security-related issues, place the application descriptor (`.xsapp`) as deep as possible in the package hierarchy. In addition, include only the index page in this package; all other application data should be placed in sub-folders that are protected with individual application-access files.

→ Tip

Keep the application package hierarchy clean. Do not place in the same package as the `.xsapp` file (or sub-package) any unnecessary content, for example, files which are not required for the application to work.

Related Information

[The Application-Access File \[page 87\]](#)

4.5.2 Enable Access to SAP HANA XS Application Packages

The application-access file enables you to specify who or what is authorized to access the content exposed by the application package and what content they are allowed to see.

Prerequisites

- A repository workspace with a shared project
- A root package for your application, for example, `MyAppPackage`

ⓘ Note

The namespace `sap` is restricted. Place the new package in your own namespace, for example, `com.acme`, which you can create alongside the `sap` namespace.

- An application descriptor file (`.xsapp`) for the selected application

Context

The application-access file is a JSON-compliant file with the file suffix `.xsaccess`. You can use a set of keywords in the application-access file `.xsaccess` to specify if authentication is required to enable access to package content, which data is exposed, and if rewrite rules are in place to hide target and source URLs, for example, from users and search engines. You can also specify what, if any, level of authorization is required for the package and whether SSL is mandatory for client connections.

ⓘ Note

The application-access file does not have a name before the dot (`.`); it only has the suffix `.xsaccess`.

To create the application access file, perform the following steps:

Procedure

1. Create a file called `.xsaccess` and place it in the root package of the application to which you want to enable access.

A basic `.xsaccess` file must, at the very least, contain a set of curly brackets, for example, `{}`. Note that the `.xsaccess` file uses keyword-value pairs to set access rules; if a mandatory keyword-value pair is not set, then the default value is assumed.

- a. In the SAP HANA studio, open the *SAP HANA Development* perspective.
- b. In the *Project Explorer* view, right-click the folder where you want to create the new (`.xsaccess`) file.
- c. In the context-sensitive popup menu, choose **New > Other...**
- d. In the *Select a Wizard* dialog, choose **SAP HANA > Application Development > XS Application Access File**

- e. **→ Tip**

If you are using the SAP HANA Studio to create artifacts in the SAP HANA Repository, the file creation wizard adds the required file extension `.xsaccess` automatically and enables direct editing of the file.

Enter or select the parent folder where the `.xsaccess` file is to be located.

Note

The default name for the core application-access file is `.xsaccess` and cannot be changed.

- f. Select a template to use. Templates contain sample source code to help you.
 - g. Choose *Finish*.
2. Enable application access to data.

You use the *expose* keyword to enable or disable access to content at a package or subpackage level.

```
{
  "exposed" : true,
  "prevent_xsrf" : true
}
```

Note

It is highly recommended to always use the `prevent_xsrf` keyword to help protect your application against attacks that use cross-site request forgery vector.

3. Define the application authentication method.

To ensure that form-based logon works when you enable it using the *SAP HANA XS Administration Tool*, the *authentication* keyword is required in the `.xsaccess` file, too, and must be set to the value `"form"`, as illustrated in the following example.

```
{
  "authentication" : { "method" : "Form"}
}
```

Note

Use the *SAP HANA XS Administration Tool* to configure applications to use additional authentication methods, for example, basic, logon tickets, or Single Sign On (SSO) providers such as SAML2 and X509. You must also enable the *Form-based authentication* checkbox, if you want your application (or applications) to use form-based logon as the authentication method. Any other keywords in the authentication section of the `.xsaccess` file are ignored.

4. Specify the application privileges if required. (*Optional*)

Use the authorization keyword in the `.xsaccess` file to specify which authorization level is required by a user for access to a particular application package. The *authorization* keyword requires a corresponding entry in the `.xsprivileges` file, for example, *execute* for basic privileges or *admin* for administrative privileges on the specified package.

```
{
  "authorization":
    [
      "sap.xse.test::Execute",
      "sap.xse.test::Admin"
    ]
}
```

5. Save the `.xsaccess` file in the package with which you want to associate the rules you have defined.

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.

6. Activate the `.xsaccess` file to the repository.

In the *Project Explorer* view, right click the object you want to activate and choose **Team > Activate** in the context-sensitive popup menu.

Related Information

[Create an Application Descriptor File \[page 83\]](#)

[Application-Access File Keyword Options \[page 89\]](#)

[The Application-Privileges File](#)

4.5.2.1 The Application-Access File

SAP HANA XS enables you to define access to each individual application package that you want to develop and deploy.

The application-access file enables you to specify who or what is authorized to access the content exposed by a SAP HANA XS application package and what content they are allowed to see. For example, you use the application-access file to specify if authentication is to be used to check access to package content and if rewrite rules are in place that hide or expose target and source URLs.

The application-access file does not have a name; it only has the file extension `.xsaccess`. The content of the `.xsaccess` file is formatted according to JSON rules, and the settings specified in an `.xsaccess` file apply not only to the package the `.xsaccess` file belongs to but also any subpackages lower in the package hierarchy. Multiple `.xsaccess` files are allowed, but only at different levels in the package hierarchy. You cannot place two `.xsaccess` files in the same package.

Note

The settings specified in an `.xsaccess` file in a **subpackage** take precedence over any settings specified in a `.xsaccess` file higher up the package hierarchy; the subpackage settings are also inherited by any packages further down the package hierarchy. Any settings **not** modified by the `.xsaccess` in the subpackage remain unchanged, that is: as defined in the parent package or, where applicable, the default settings.

Using multiple `.xsaccess` files enables you to specify different application-access rules for individual subpackages in the package hierarchy. Following the inheritance rule, any applications below the application package containing the modified access settings inherit the new, modified settings.

The following example shows the composition and structure of the SAP HANA XS application access (`.xsaccess`) file, which comprises a list of key-value pairs that specify how the application service responds to client requests. For example, in this file, `"exposed" : true` indicates that data is available to client requests; `"force_ssl" : true` specifies that standard HTTP requests are not allowed by the Web browser.

Note

Some elements can also be specified in the application's runtime configuration, for example, using the [SAP HANA XS Administration Tool](#). For example, you can configure applications to refuse insecure HTTP connections, allow the use of e-tags, or enable additional authentication methods such as Single Sign On (SSO) providers SAML2 and X509.

Example: The Application-Access (`.xsaccess`) File

```
{
  "exposed" : true,                // Expose data via http
  "authentication" :
    {
      "method": "Form"
    },
  "authorization":                // Privileges for application access
    [
      "sap.xse.test::Execute",
      "sap.xse.test::Admin"
    ],
  "rewrite_rules" :              // URL rewriting rules
    [ {
      "source": "/entries/(\\d+)/(\\d+)/(\\d+)/",
      "target": "/logic/entries.xsjs?year=$1&month=$2&day=$3"
    } ],
  "mime_mapping" :              // Map file-suffix to MIME type
    [ {
      "extension": "jpg", "mimetype": "image/jpeg"
    } ],
  "force_ssl" : true,            // Accept only HTTPS requests
  "enable_etags" : true,        // Allow generation of etags
}
```

```

"prevent_xsrp" : true,           // Prevent cross-site request forgery
"anonymous_connection" : "sap.hana.sqlcon::AnonConn", //xsqlcc object
"default_connection" : "sap.hana.sqlcon::sqlcc", //xsqlcc object
"cors" :                        // Permit cross-origin browser requests
{
  "enabled" : false
},
"default_file" : "homepage.html", // Override default access setting
"cache_control" : "no-cache, no-store", // Manage static Web-content cache
"headers":                // Enable X-Frame-Options HTTP header field
{
  "enabled": true,
  "customHeaders":
  [ {
    "name": "X-Frame-Options", "value": "SAMEORIGIN"
  } ]
}
}

```

Related Information

[Application-Access File Keyword Options \[page 89\]](#)

[Set up Application Security \[page 105\]](#)

4.5.2.2 Application-Access File Keyword Options

The application-access (.xsaccess) file enables you to specify whether or not to expose package content, which authentication method is used to grant access, and what content is visible.

Example: The Application Access (.xsaccess) File

Note

This example of the .xsaccess file is **not** a working model; it is used to illustrate the syntax for all possible options.

```

{
  "exposed" : false,
  "authentication" :
  {
    "method": "Form"
  },
  "authorization":
  [
    "sap.xse.test::Execute",
    "sap.xse.test::Admin"
  ],
  "anonymous_connection" : "sap.hana.sqlcon::AnonConn",
  "default_connection" : "sap.hana.sqlcon::sqlcc",
  "cache_control" : "no-store",

```

```

"cors" :
{
  "enabled" : false
},
"default_file" : "index_1.html",
"enable_etags" : false,
"force_ssl" : true,
"mime_mapping" :
[
  {
    "extension": "jpg", "mimetype": "image/jpeg"
  }
],
"prevent_xsrp" : true,
"rewrite_rules" :
[
  {
    "source" : "...",
    "target" : "..."
  }
]
"headers":
{
  "enabled": true,
  "customHeaders": [ {"name": "X-Frame-Options", "value": "<VALUE>"} ]
}

```

exposed

```

{
  "exposed" : false,
}

```

The `exposed` keyword enables you to define if content in a package (and its subpackages) is to be made available by HTTP to client requests. Values are Boolean `true` or `false`. If no value is set for `exposed`, the default setting (`false`) applies.

→ Tip

Only expose content that is absolutely necessary to enable the application to run.

Consider whether it is necessary to expose data via HTTP/S. Not exposing data via HTTP enables you to keep your files accessible to other programs but prevent direct access to the data via URL. Since the application's `index.html` page must normally remain reachable, consider storing the `index.html` file separately with a dedicated `.xsaccess` file that enables access (`"exposed": true`). You can keep all other content hidden, for example, in separate package to which access is denied (`"exposed": false`).

Packages without a dedicated `.xsaccess` file inherit the application-access settings defined in the parent folder. If an `.xsaccess` file exists but the `exposed` keyword is not defined, the default setting `false` applies.

anonymous_connection

```
{  
  "anonymous_connection" : "sap.hana.sqlcon::AnonConn",  
}
```

The `anonymous_connection` keyword enables you to define the name of the `.xssqlcc` file that will be used for SQL access when no user credentials are provided. SAP HANA XS enables you to define the configuration for individual SQL connections. Each connection configuration has a unique name, for example, [Registration](#), [AnonConn](#), or [AdminConn](#), which is generated from the name of the corresponding connection-configuration file (`Registration.xssqlcc`, `AnonConn.xssqlcc`, or `AdminConn.xssqlcc`) on activation in the repository. If no value is set, the default setting is "null".

→ Tip

It is not recommended to enable anonymous access.

If it is necessary to provide anonymous access to an application, design your application in such a way that all files requiring anonymous access are placed together in the same package, which you can then protect with the permissions defined in a dedicated `.xsaccess` file. Remember that the behavior of the anonymous connection depends on the details specified in the corresponding SQL configuration file (`.xssqlcc`).

default_connection

```
{  
  "default_connection" : "sap.hana.sqlcon::sqlcc",  
}
```

If the `default_connection` is set in the `.xsaccess` file, the specified SQL connection configuration (for example, defined in `sqlcc`) is used for **all** SQL executions in this package, whether or not the requesting user is authenticated in SAP HANA or not. The difference between the `default_connection` and the `anonymous_connection` is that the anonymous SQL connection configuration is only used if the requesting user is not authenticated. Like any other property of the `xsaccess` file, the `default_connection` is inherited down the package hierarchy, for example, from package to subpackage. The `default_connection` can also be overwritten, for example, by locating an `xsaccess` file with a different `default_connection` in one or more subpackages.

→ Tip

If the requesting user is authenticated, the user name will be available in the connection as the `APPLICATIONUSER` session variable.

The credentials to use for an SQL execution are determined according to the following order of priority:

1. The SQL connection configuration (SQLCC) specified in `$.db.getConnection(sqlcc)`; this applies only in XS JavaScript (not OData, for example)
2. The value specified in `default_connection` (if set)
3. An authenticated user
4. The value specified in `anonymous_connection` (if set)

The `default_connection` is intended for use with anonymous parts of the application that require the **same** privileges for all users. If the anonymous part of an application is designed to behave according to the privileges granted to authenticated users, the `anonymous_connection` should be used. This is particularly important if analytic privileges are involved, for example, to restrict the amount of returned rows (not overall access to the table). In most cases, the `default_connection` should be used.

authentication

```
{
  "authentication" :
  {
    "method": "Form"
  },
}
```

The `authentication` keyword is required in the `.xsaccess` file and must be set to the value `"form"`, for example `"method" : "Form"`, to ensure that form-based logon works when you enable it using the [SAP HANA XS Administration Tool](#).

Note

Use the [SAP HANA XS Administration Tool](#) to configure applications to use additional authentication methods, for example, basic, logon tickets, or Single Sign On (SSO) providers such as SAML2 and X509. You must also enable the [Form-based authentication](#) checkbox, if you want your application (or applications) to use form-based logon as the authentication method. Any other keywords in the authentication section of the `.xsaccess` file are ignored.

- Form-based authentication
Redirect the logon request to a form to fill in, for example, a Web page.
To ensure that, during the authentication process, the password is transmitted in encrypted form, it is strongly recommended to enable SSL/HTTPS for all application connections to the XS engine, for example, using the `force_ssl` keyword. If you set the `force_ssl` option, you must ensure that the SAP Web Dispatcher is configured to accept and manage HTTPS requests.
Form-based authentication requires the `libxsauthenticator` library, which must not only be available but also be specified in the list of trusted applications in the [xsengine](#) application container. The application list is displayed in the SAP HANA studio's [Administration Console](#) perspective in the following location:
► [Administration](#) ► [Configuration tab](#) ► [xsengine.ini](#) ► [application_container](#) ► [application_list](#) ►. If it is not displayed, ask the SAP HANA administrator to add it.

Note

If you need to troubleshoot problems with form-based logon, you can configure the generation of useful trace information in the [XSENGINE](#) section of the database trace component using the following entry: `xsa:sap.hana.xs.formlogin`.

authorization

```
{
  "authorization":
    [
      "sap.xse.test::Execute",
      "sap.xse.test::Admin"
    ],
}
```

The `authorization` keyword in the `.xsaccess` file enables you to specify which authorization level is required for access to a particular application package, for example, `execute` or `admin` on the package `sap.xse.test`.

Note

The authorization levels you can choose from are defined in the `.xsprivileges` file for the package, for example, `"execute"` for basic privileges, or `"admin"` for administrative privileges on the specified package. If you do not define any authorization requirements, any user can launch the application.

If you use the `authorization` keyword in the `.xsaccess` file, for example, to require `"execute"` privileges for a specific application package, you must create a `.xsprivileges` file for the same application package (or a parent package higher up the hierarchy, in which you define the `"execute"` privilege level declared in the `.xsaccess` file).

Authorization settings are inherited down the package hierarchy from a package to a subpackage. However, you can specify different authorization levels for different subpackages; this new setting is then inherited by any subpackages further down the hierarchy. To disable authorization for a subpackage (for example, to prevent inheritance of authorizations from the parent package), you can create a (sub)package-specific `.xsaccess` file with the `authorization` keyword explicitly set to `null`, as illustrated in the following example.

```
{
  "authorization": null
}
```

Bear in mind that the `"authorization":null` setting applies not only to the package in which the `.xsaccess` with the `null` setting is located but also to any subpackages further down the package hierarchy. You can re-enable authorization in subpackage levels by creating new a `.xsaccess` file.

cache_control

```
{
  "cache_control": "no-store",
}
```

The `cache_control` keyword enables you to override the cache-control header for Web content served by the SAP HANA XS Web server. So-called cache-control *directives* (for example, `public`, `private`, `no-store`) enable you to control the behavior of the Web browser and proxy caches, for example, whether or not to store a page, how to store it, or where. For more information about the values you can use to set `cache_control`, see

the HTTP standard for cache-control directives. If no value for `cache_control` is set in the `.xsaccess` file, the default setting is "null".

→ Tip

For security reason, it is recommended to set the `cache_control` keyword to "no-cache, no-store". However, if nothing is cached or stored, there is an obvious impact on application performance.

If application performance allows, the `no-cache, no-store` setting is advisable for the following reasons:

- From a client perspective:
If an application is handling sensitive data, it is bad practice to cache the data in the local browser since this could lead to unintended disclosure of information.
- From a server perspective:
Allowing an application to cache data can open up the application to attack. For example, if attackers build a malicious page and host it on a proxy server between your server and the requesting client, it would be possible to steal data from the client or prevent access to the application altogether. Since the risk of such an attack is small, you might want to consider allowing caching, as long as it does not adversely affect performance.

cors

```
{
  "cors" :
  {
    "enabled" : false
  },
}
```

The `cors` keyword enables you to provide support for cross-origin requests, for example, by allowing the modification of the request header. Cross-origin resource sharing (CORS) permits Web pages from other domains to make HTTP requests to your application domain, where normally such requests would automatically be refused by the Web browser's security policy.

If CORS support is disabled (`"enabled" : false`), the following settings apply on the Web server:

- The server does not respond to any CORS preflight requests
- The server does not add CORS response headers to any CORS requests
- The server refuses to execute the resource specified in the request

To enable support for CORS, set the `cors` keyword to `{"enabled": true}`, which results in the following default `cors` configuration:

```
{"cors":{"enabled":true,"allowMethods":
["GET","POST","HEAD","OPTIONS"],"allowOrigin":["*"], "maxAge":"3600"}}
```

The following table describes the options that are supported with the `cors` keyword:

```
{"cors":{"enabled":true, "allowMethods":<ALLOWED_METHODS>,
"allowOrigin":<ALLOWED_ORIGIN>,
"maxAge":<MAX_AGE>, "allowHeaders":<ALLOWED_HEADERS>,
"exposeHeaders":<EXPOSED_HEADERS>}}
```

Default Settings for CORS Options

CORS Option	Description
ALLOWED_METHODS	A single permitted method or a comma-separated list of methods that are allowed by the server, for example, "GET", "POST". If <code>allowMethods</code> is defined but no method is specified, the default "GET", "POST", "HEAD", "OPTIONS" (all) applies. Note that matching is case-sensitive.
ALLOWED_ORIGIN	A single host name or a comma-separated list of host names that are allowed by the server, for example: <code>www.sap.com</code> or <code>*.sap.com</code> . If <code>allowOrigin</code> is defined but no host is specified, the default "*" (all) applies. Note that matching is case-sensitive.
ALLOW_HEADERS	A single header or a comma-separated list of request headers that are allowed by the server. If <code>allowHeaders</code> is defined but no header is specified as allowed, no default value is supplied.
MAX_AGE	A single value specifying how long a preflight request should be cached for. If <code>maxAge</code> is defined but no value is specified, the default time of "3600" (seconds) applies.
EXPOSE_HEADERS	A single header or a comma-separated list of response headers that are allowed to be exposed. If <code>exposeHeaders</code> is defined but no response header is specified for exposure, no default value is supplied.

Alternatively, you can isolate the part of the application where CORS must be allowed, for example, in a specific subpackage. By adding a dedicated `.xsaccess` file to this CORS-related subpackage, you can set the `cors` option in the dedicated `.xsaccess` file to true.

default_file

```
{
  "default_file" : "new_index.html",
}
```

The [default_file](#) keyword enables you to override the default setting for application access (`index.html`) when the package is accessed without providing a file in the URI. If you use the [default_file](#) but do not specify a value, the default setting "`index.html`" is assumed.

→ Tip

It is good practice to specify a default file name manually. Changing the default from `index.html` to something else can help make your application less vulnerable to automated hacker tools.

rewrite_rules

```
{
  "rewrite_rules" :
  [{
    "source": "...",
    "target": "...",
  }],
}
```



```
}
```

The `rewrite_rules` keyword enables you hide the details of internal URL paths from external users, clients, and search engines. Any rules specified affect the local application where the `.xsaccess` file resides (and any subpackage, assuming the subpackages do not have their own `.xsaccess` files); it is not possible to define global rewrite rules. URL rewrite rules are specified as a source-target pair where the source is written in the JavaScript regex syntax and the target is a simple string where references to found groups can be inserted using `$groupnumber`.

→ Tip

It is not recommended to rely on rewrite rules to make an application secure.

In the following example, the rule illustrated hides the `filename` parameter and, as a result, makes it harder to guess that the parameter provided after `/go/` will be used as a filename value. Note that it is still necessary to validate the received input

```
{
  "rewrite_rules" :
  [{
    "source": "/go/(\\d+)/",
    "target": "/logic/users.xsjs?filename=$1"
  }],
}
```

mime_mapping

```
{
  "mime_mapping" :
  [
    {
      "extension": ".jpg", "mimetype": "image/jpeg"
    }
  ],
}
```

The `mime_mapping` keyword enables you to define how to map certain file suffixes to required MIME types. For example, you can map files with the `.jpg` file extension to the MIME type `image/jpeg`.

This list you define with the `mime_mapping` keyword supersedes any default mapping defined by the server; the Web browser uses the information to decide how to process the related file types.

⚠ Caution

Make sure you do not instruct the browser to execute files that are not meant to be executed, for example, by mapping `.jpg` image files with the MIME type `application/javascript`.

The default MIME mappings remain valid for any values you do not define with the `mime_mapping` keyword. Consider restricting any explicit mappings to file types where the default behavior does not work as expected or where no default value exists, for example, for file types specific to your application.

force_ssl

```
{
  "force_ssl" : false,
}
```

The `force_ssl` keyword enables you to refuse Web browser requests that do not use secure HTTP (SSL/HTTPS) for client connections. If no value is set for `force_ssl`, the default setting (`false`) applies and non-secured connections (HTTP) are allowed.

→ Tip

To ensure that, during the authentication process, passwords are transmitted in encrypted form, it is strongly recommended to enable SSL/HTTPS for all application connections to the XS engine. If you set the [force_ssl](#) option, you must ensure that the SAP Web Dispatcher is configured to accept and manage HTTPS requests. For more information, see the SAP HANA XS section of the *SAP HANA Administration Guide*.

Enabling the `force_ssl` option ensures that your application is reachable only by means of an HTTPS connection. If your application must support standard HTTP (without SSL), make sure that no sensitive data is being sent either **to** or **from** the application. Disabling the `force_ssl` option allows attackers to read whatever is sent over the network. Although it is possible to use message-based encryption for sensitive data while allowing HTTP, it is much better to work with HTTPS.

⚠ Caution

If a runtime configuration exists for your application, the `force_ssl` setting in the runtime configuration supersedes the `force_ssl` in the `.xsaccess`.

enable_etags

```
{
  "enable_etags" : true,
}
```

You can allow or prevent the generation of entity tags (etags) for static Web content using the `enable_etags` keyword. If no value is set, the default setting (`true`) applies, in which case etags are generated. Etags are used to improve caching performance, for example, so that the same data is not resent from the server if no change has occurred since the last time a request for the same data was made.

If etags are enabled, the browser sends with each HTTP request the etag retrieved from its cached page. If the etag from the cached page matches the etag from the server, the server answers with the status code 304 (not modified) and does not send the full requested page. Although enabling etags has the positive side-effect of helping to prevent cache poisoning attacks, there is no direct security risk associated with disabling etags from the developer's perspective.

prevent_xsrp

```
{
  "prevent_xsrp" : true,
}
```

You can use the `prevent_xsrp` keyword in the `.xsaccess` file to protect applications from cross-site request-forgery (XSRF) attacks. XSRF attacks attempt to trick a user into clicking a specific hyperlink, which shows a (usually well-known) Web site and performs some actions on the user's behalf, for example, in a hidden iframe. If the targeted end user is logged in and browsing using an administrator account, the XSRF attack can compromise the entire Web application. There is no good reason why you would explicitly set this keyword to false.

Note

It is recommended to enable the `prevent_xsrp` feature for **all** applications that are not read-only.

The `prevent_xsrp` keyword prevents the XSRF attacks by ensuring that checks are performed to establish that a valid security token is available for a given Browser session. The existence of a valid security token determines if an application responds to the client's request to display content; if no valid security token is available, a **403 Forbidden** message is displayed. A security token is considered to be valid if it matches the token that SAP HANA XS generates in the back end for the corresponding session.

Note

The default setting is false, which means there is no automatic prevention of XSRF attacks. If no value is assigned to the `prevent_xsrp` keyword, the default setting (false) applies.

Setting the `prevent_xsrp` keyword to `true` ensures XSRF protection only on the server side. On the client side, to include the XSRF token in the HTTP headers, you must first fetch the token as part of a GET request, as illustrated in the following example:

```
xmlHttp.setRequestHeader("X-CSRF-Token", "Fetch");
```

You can use the fetched XSRF token in subsequent POST requests, as illustrated in the following code example:

```
xmlHttp.setRequestHeader("X-CSRF-Token", xsrp_token);
```

headers

```
{
  "headers":
  {
    "enabled": true,
    "customHeaders": [ {"name": "X-Frame-Options", "value": "<VALUE>" } ]
  }
}
```

Enable support for the X-Frame-Options HTTP header field, which allows the server to instruct the client browser whether or not to display transmitted content in frames that are part of other Web pages. You can

also enable this setting in the application's corresponding runtime configuration file, for example, using the [XS Administration Tool](#).

⚠ Caution

Runtime settings override any settings specified in the design-time configuration.

<VALUE> can be one of the following:

- DENY
- SAMEORIGIN
- ALLOW-FROM **<URL>**

You can only specify one URL with the ALLOW-FROM option, for example: "value": "ALLOW-FROM http://www.site.com".

📌 Note

To allow an application to use custom headers, you must enable the headers section.

Related Information

[Server-Side JavaScript Security Considerations \[page 533\]](#)

[The SQL Connection Configuration File \[page 584\]](#)

4.5.2.3 Application-Access URL Rewrite Rules

Rewriting URLs enables you to hide internal URL path details from external users, clients, and search engines. You define URL rewrite rules in the application-access file (`.xsaccess`) for each application or for an application hierarchy (an application package and its subpackages).

The rewrite rules you define in the `.xsaccess` file apply only to the local application to which the `.xsaccess` file belongs; it is not possible to define global rules to rewrite URLs. Rules are specified as a source-target pair where the source is written in the JavaScript `regex` syntax, and the target is a simple string where references to found groups can be inserted using `$groupnumber`.

The following examples show how to use a simple set of rewrite rules to hide internal URLs from requesting clients and users.

The first example illustrates the package structure that exists in the repository for a given application; the structure includes the base package `apptest`, the subpackages `subpackage1` and `subpackage2`, and several other subpackages:

```
sap---apptest
|---logic
|   |---users.xsjs
|   |---posts.xsjs
|---posts
|   |---2011...
|---subpackage1
|   |---image.jpg
```

```

|---subpackage2
|   |---subsubpackage
|   |   |---secret.txt
|   |---script.xsjs
|---subpackage3
|   |---internal.file
|---users
|   |---123...
|---.xsapp
|---.xsaccess
|---index.html

```

The application-access file for the package `apptest` (and its subpackages) includes the following rules for rewriting URLs used in client requests:

```

{
  "rewrite_rules": [
    {
      "source": "/users/(\\d+)/",
      "target": "/logic/users.xsjs?id=$1"
    },
    {
      "source": "/posts/(\\d+)/(\\d+)/(\\d+)/",
      "target": "/logic/posts.xsjs?year=$1&month=$2&day=$3"
    }
  ]
}

```

Assuming we have the package structure and URL rewrite rules illustrated in the previous examples, the following valid URLs would be exposed; bold URLs require authentication:

```

/sap/apptest/
/sap/apptest/index.html
/sap/apptest/logic/users.xsjs
/sap/apptest/logic/posts.xsjs

```

The rewriting of the following URLs would be allowed:

```

/sap/apptest/users/123/ ==> /sap/appTest/logic/users.xsjs?id=123
/sap/apptest/posts/2011/10/12/ ==> /sap/appTest/logic/posts.xsjs?
year=2011&month=10&day=12

```

4.5.3 Create an SAP HANA XS Application Privileges File

The application-privileges (`.xsprivileges`) file enables you to define the authorization levels required for access to an application, for example, to start the application or perform administrative actions on an application. You can assign the application privileges to the individual users who require them.

Prerequisites

- A repository workspace with a shared project
- A root package for your application, for example, `MyAppPackage`

Note

The namespace `sap` is restricted. Place the new package in your own namespace, for example, `com.acme`, which you can create alongside the `sap` namespace.

- An application descriptor file (`.xsapp`) for the selected application
- An application access file (`.xsaccess`) for the selected application

Context

The `.xsprivileges` file must reside in the same application package that you want to define the access privileges for.

Note

If you use the `.xsprivileges` file to define application-specific privileges, you must also add a corresponding entry to the same application's `.xsaccess` file, for example, using the [authorization](#) keyword.

Procedure

1. Create the application-privileges (`.xsprivileges`) file and place it in the application package whose access privileges you want to define.

The application-privileges file does not have a name; it only has the file extension `.xsprivileges`. The contents of the `.xsprivileges` file must be formatted according to JavaScript Object Notation (JSON) rules.

Note

Multiple `.xsprivileges` files are allowed, but only at different levels in the package hierarchy; you cannot place two `.xsprivileges` files in the same application package. The privileges defined in a `.xsprivileges` file are bound to the package to which the file belongs and can only be applied to this package and its subpackages.

- a. In the SAP HANA studio and open the [SAP HANA Development](#) perspective.
- b. In the [Project Explorer](#) view, right-click the folder where you want to create the new (`.xsprivileges`) file.
- c. In the context-sensitive popup menu, choose **New > Other...**
- d. In the [Select a Wizard](#) dialog, choose **SAP HANA > Application Development > XS Application Privileges File**
- e. Enter or select the parent folder, where the application-privileges file is to be located.
- f. Enter a name for the application-privileges file.

→ Tip

If you are using the SAP HANA Studio to create artifacts in the SAP HANA Repository, the file creation wizard adds the required file extension `.xsprivileges` automatically and enables direct editing of the file.

- g. Select a template to use. Templates contain sample source code to help you.
 - h. Choose *Finish*.
 - i. Activate the new (`.xsprivileges`) file.
2. Define the required application privileges.

In the `.xsprivileges` file, you define a privilege for an application package by specifying an entry name with an optional description. This entry name is then automatically prefixed with the package name in which the `.xsprivileges` file is located to form a unique privilege name. For example, `com.acme.myapp::Execute` would enable *execute* privileges on the package `com.acme.myapp`. The privilege name is unique to the package to which it belongs and, as a result, can be used in multiple `.xsprivileges` files in different packages.

ⓘ Note

The `.xsprivileges` file lists the authorization levels defined for an application package. A corresponding entry is required in the same application's access file `.xsaccess` file to define which authorization level is assigned to which application package.

```
{
  "privileges" :
  [
    { "name" : "Execute", "description" : "Basic execution
privilege" },
    { "name" : "Admin", "description" : "Administration
privilege" }
  ]
}
```

3. Specify which privileges are required for access to the application or application package.

If you use the `.xsprivileges` file to define application-specific privileges, you must also add a corresponding entry to the same application's `.xsaccess` file, for example, using the *authorization* keyword.

ⓘ Note

The `.xsprivileges` file lists the authorization levels that are available for access to an application package; the `.xsaccess` file defines which authorization level is assigned to which application package.

- a. Locate and open the XS application access file (`.xsaccess`) for the application for which you want to define application privileges.
- b. Specify the privileges required for access to the application or application package.
Use the *authorization* keyword in the `.xsaccess` file to specify which authorization level is required by a user for access to a particular application package.

Note

If you enable the [authorization](#) keyword in the `.xsaccess` file, you must add a corresponding entry to the `.xsprivileges` file, too.

```
{
  "exposed" : true,
  "authentication" :
    [
      { "method" : "Form" }
    ],
  "authorization":
    [
      "com.acme.myApp::Execute",
      "com.acme.myApp::Admin"
    ]
}
```

4. Save and activate your changes and additions.

The activation of the application privileges creates the corresponding objects, which you can use to assign the specified privileges to an author.

5. Assign the application privilege to the users who require it.

After activation of the `.xsprivileges` object, the only user who by default has the application privileges specified in the `.xsprivileges` file is the `_SYS_REPO` user. To grant the specified privilege to (or revoke them from) other users, use the `GRANT_APPLICATION_PRIVILEGE` or `REVOKE_APPLICATION_PRIVILEGE` procedure in the `_SYS_REPO` schema.

To grant the execute application privilege to a user, run the following command in the SAP HANA studio's [SQL Console](#):

```
call
  "_SYS_REPO"."GRANT_APPLICATION_PRIVILEGE"('com.acme.myApp::Execute', '<UserNa
me>')
```

To revoke the execute application privilege to a user, run the following command in the SAP HANA studio's [SQL Console](#):

```
call
  "_SYS_REPO"."REVOKE_APPLICATION_PRIVILEGE"('com.acme.myApp::Execute', '<UserN
ame>')
```

Related Information

[Create an Application Descriptor File \[page 83\]](#)

[Enable Access to SAP HANA XS Application Packages \[page 85\]](#)

4.5.3.1 The Application-Privileges File

In SAP HANA Extended Application Services (SAP HANA XS), the application-privileges (`.xsprivileges`) file can be used to create or define the authorization privileges required for access to an SAP HANA XS application,

for example, to start the application or to perform administrative actions on an application. These privileges can be checked by an application at runtime.

The application-privileges file has only the file extension `.xsprivileges`; it does not have a name and is formatted according to JSON rules. Multiple `.xsprivileges` files are allowed, but only at different levels in the package hierarchy; you cannot place two `.xsprivileges` files in the same application package. The package privileges defined in a `.xsprivileges` file are bound to the package to which the `.xsprivileges` file belongs and can only be used in this package and its subpackages.

Inside the `.xsprivileges` file, a privilege is defined by specifying an entry name with an optional description. This entry name is then automatically prefixed with the package name to form the unique privilege name, for example, `sap.hana::Execute`.

As an application privilege is created during activation of an `.xsprivileges` file, the only user who has the privilege by default is the `_SYS_REPO` user. To grant or revoke the privilege to (or from) other users you can use the `GRANT_APPLICATION_PRIVILEGE` or `REVOKE_APPLICATION_PRIVILEGE` procedure in the `_SYS_REPO` schema.

Note

The `.xsprivileges` file lists the authorization levels that are available for access to an application package; the `.xsaccess` file defines which authorization level is assigned to which application package.

In the following above, if the application-privileges file is located in the application package `sap.hana.xse`, then the following privileges are created:

- `sap.hana.xse::Execute`
- `sap.hana.xse::Admin`

The privileges defined apply to the package where the `.xsprivileges` file is located as well as any packages further down the package hierarchy unless an additional `.xsprivileges` file is present, for example, in a subpackage. The privileges do not apply to packages that are not in the specified package path, for example, `sap.hana.app1`.

Example: The SAP HANA XS Application-Privileges File

The following example shows the composition and structure of a basic SAP HANA XS application-privileges file.

```
{
  "privileges" :
  [
    { "name" : "Execute", "description" : "Basic execution
privilege" },
    { "name" : "Admin", "description" : "Administration privilege" }
  ]
}
```

If the `.xsprivileges` file shown in the example above is located in the package `sap.hana.xse`, you can assign the `Execute` privilege for the package to a particular user by calling the `GRANT_APPLICATION_PRIVILEGE` procedure, as illustrated in the following code:

```
call "_SYS_REPO"."GRANT_APPLICATION_PRIVILEGE"('sap.hana.xse::Execute',
'<user>')
```

4.6 Maintaining Application Security

As part of the application-development process, you must decide how to provide access to the applications you develop. Application access includes security-related matters such as authentication methods and communication protocols

In addition to the features and functions you can enable with keywords in the `.xsaccess` file, SAP HANA Extended Application Services (SAP HANA XS) provides a dedicated SAP HANA XS administration tool that is designed to help you configure and maintain the authentication mechanism used to control access to the applications you develop. The [SAP HANA XS Administration Tool](#) enables you to configure the following runtime elements for an application:

- **Security**
Choose the security level you want to set to provide access to the application. For example, you can expose the application with/without requiring authentication (public/private) and force the application to accept only requests that use SSL/HTTPS.
- **Authentication**
Select an authentication type to use when checking user credentials before authorizing access to an application, for example: form-based authentication (with user name and password), SAML (SSO with Security Assertion Markup Language), SAP logon tickets...

Related Information

[Set up Application Security \[page 105\]](#)

[Application Security \[page 107\]](#)

[Application Authentication \[page 110\]](#)

4.6.1 Set up Application Security

To restrict access to the applications you develop, you must configure the application to work with particular authentication methods and communication protocols.

Prerequisites

To perform the steps in this task, you must ensure the following prerequisites are met:

- You have access to an SAP HANA system
- You have the privileges granted in the following SAP HANA XS user roles:
 - [sap.hana.xs.admin.roles::RuntimeConfAdministrator](#)

Context

You must specify whether or not to expose application content, which authentication method is used to grant access to the exposed content, and what content is visible.

Procedure

1. Start the *SAP HANA XS Administration Tool*.

The tool is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/admin/`.

Note

In the default configuration, the URL redirects the request to a logon screen, which requires the credentials of an authenticated SAP HANA database user to complete the logon process. To ensure access to all necessary features, the user who logs on should have the SAP HANA XS role *sap.hana.xs.admin.roles::RuntimeConfAdministrator*.

2. Select the security options your applications use.

You can setup the following application-related security options:

Note

Security settings are automatically inherited by applications further down the application hierarchy. However, you can override the inherited security settings at any application level by modifying the settings for a particular application. Applications below the application with the modified security settings inherit the new, modified settings.

- a. Use the *Public (no authentication required)* option to specify if applications require user authentication to start.
 - Disabled
This is the default setting. In **disabled** mode, *Form-based authentication* and *Basic authentication* options are enabled automatically in the *Authentication* screen area.
 - Enabled
If you **enable** the *Public* option, no authentication is required to start an application; the *Authentication* screen area is hidden, and you cannot select any authentication-method options.
- b. Use the *Force SSL* option to specify if client requests must use secure HTTP (HTTPS).
 - Disabled
This is the default setting. With *Force SSL* disabled, the application returns a response to all requests (both HTTP and HTTPS).
 - Enabled
If you **enable** the *Force SSL* option, requests from browsers using standard HTTP are refused.

Note

Enabling the *Force SSL* option only ensures that the selected application refuses any request that does not use HTTPS; it does not set up the Secure Sockets Layer (SSL) protocol for you.

The SAP HANA administrator must configure the SAP Web Dispatcher to accept (and forward) HTTPS requests in addition.

Related Information

[SAP HANA XS Application Security \[page 107\]](#)

[Set up Application Authentication \[page 108\]](#)

[SAP HANA XS Application Authentication \[page 110\]](#)

[The Application-Access File \[page 87\]](#)

4.6.1.1 SAP HANA XS Application Security

You can set some basic security options to increase the security of the applications you develop for SAP HANA.

SAP HANA Extended Application Services (SAP HANA XS) provides a dedicated tool, the [SAP HANA XS Administration Tool](#), that is designed to help you configure and maintain some of the basic aspects of runtime security relating to the applications you develop. For example, you can use the [SAP HANA XS Administration Tool](#) to specify if the applications you develop are publicly available for anyone to start, or if the applications can only be started by an authenticated user.

You can use the [SAP HANA XS Administration Tool](#) to set the following security-related options for the application you develop for SAP HANA XS:

- [Public \(no authentication required\)](#)
Use the [Public](#) option to specify if applications require user authentication to start. By default, the [Public](#) option in the application [Security](#) screen area is disabled and the [Form-based authentication](#) and [Basic authentication](#) options are enabled automatically in the [Authentication](#) screen area. However, you can disable both form-based and basic authentication and enable other, additional authentication methods (for example, SAP logon tickets or X509 authentication).

Note

If you **enable** the [Public](#) option in the application [Security](#) screen area, no authentication is required to start an application; the [Authentication](#) screen area is hidden, and you cannot select any authentication-method options.

- [Force SSL](#)
The [force SSL](#) option enables you to refuse Web browser requests that do not use secure HTTP (SSL/HTTPS) for client connections. If no value is set for `force_ssl`, the default setting (false) applies and non-secured connections (HTTP) are allowed.

Related Information

[SAP HANA XS Application Authentication \[page 110\]](#)

4.6.2 Set up Application Authentication

To restrict access to the applications you develop, you must configure the application to work with particular authentication methods and communication protocols.

Prerequisites

To perform the steps in this task, you must ensure the following prerequisites are met:

- You have access to an SAP HANA system
- You have the privileges granted in the following SAP HANA XS user roles:
 - `sap.hana.xs.admin.roles::RuntimeConfAdministrator`

Context

Before you define which authentication methods an application uses to grant access to the application content, you must use the application security tools to define whether or not to expose application content and, if so, which content to expose. SAP HANA XS enables you to define multiple authentication methods to verify the credentials of users who request access to the exposed content; multiple authentication methods are considered according to a specific order of priority. For example, if the first authentication method fails, SAP HANA tries to authenticate the user with the next authentication method specified. To configure the authentication method an application uses to verify user credentials, perform the following steps:

Procedure

1. Start the *SAP HANA XS Administration Tool*.

The tool is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/admin/`.

Note

In the default configuration, the URL redirects the request to a logon screen, which requires the credentials of an authenticated SAP HANA database user to complete the logon process. To ensure access to all necessary features, the user who logs on should have the SAP HANA XS role `sap.hana.xs.admin.roles::RuntimeConfAdministrator`.

2. Select the security options your applications use.

If you have already set the application security level, you can safely skip this step. You can setup the following application-related security options:

Note

Security settings are automatically inherited by applications further down the application hierarchy. However, you can override the inherited security settings at any application level by modifying the settings for a particular application. Applications below the application with the modified security settings inherit the new, modified settings.

- a. Use the *Public (no authentication required)* option to specify if applications require user authentication to start.
 - Disabled
This is the default setting. In **disabled** mode, *Form-based authentication* and *Basic authentication* options are enabled automatically in the *Authentication* screen area.
 - Enabled
If you **enable** the *Public* option, no authentication is required to start an application; the *Authentication* screen area is hidden, and you cannot select any authentication-method options.
- b. Use the *Force SSL* option to specify if client requests must use secure HTTP (HTTPS).
 - Disabled
This is the default setting. With *Force SSL* disabled, the application returns a response to all requests (both HTTP and HTTPS).
 - Enabled
If you **enable** the *Force SSL* option, requests from browsers using standard HTTP are refused.

Note

Enabling the *Force SSL* option only ensures that the selected application refuses any request that does not use HTTPS; it does not set up the Secure Sockets Layer (SSL) protocol for you. The SAP HANA administrator must configure the SAP Web Dispatcher to accept (and forward) HTTPS requests in addition.

3. Select the authentication method your applications must use.

Authentication settings are automatically inherited by applications further down the application hierarchy. However, you can override the inherited authentication settings at any application level by modifying the settings for a particular application. Applications below the application with the modified authentication settings inherit the new, modified settings.

Note

Enabling an application-security option (for example, *SAML2* or *X509*) only ensures that the selected application uses the enabled authentication method when required; it does not perform any setup operation for the authentication method itself. The SAP HANA administrator must maintain the selected authentication infrastructure (SAML2, X509, or SAP logon tickets) in an additional step.

You can choose any selection of the following application-related authentication methods; if you enable multiple authentication methods for your application, a priority applies depending on whether the application logon is interactive or non-interactive:

- a. Enable the *SAML2* option.
The SAP HANA administrator must already have configured the authentication infrastructure, for example, to enable the creation of SAML2 assertions to permit SSO in Web browsers.
- b. Enable the *X509 Authentication* option

The SAP HANA administrator must already have configured the appropriate authentication infrastructure, for example, to enable users to be authenticated by client certificates signed by a trusted Certification Authority (CA).

- c. Enable the *SAP logon ticket* option

The SAP HANA administrator must already have configured the appropriate authentication infrastructure, for example, to enable users to be authenticated by a logon ticket that is issued when the same user logs on to an SAP system that is configured to create logon tickets (for example, the SAP Web Application Server or Portal).

- d. Enable the *Form-based authentication* option

If the *Public* security option is **disabled**, the *Form-based authentication* option is enabled by default.

- e. Enable the *Basic authentication* option

If the *Public* security option is **disabled**, the *Basic authentication* option is enabled by default.

Related Information

[Set up Application Authentication \[page 105\]](#)

[SAP HANA XS Application Security \[page 107\]](#)

[SAP HANA XS Application Authentication \[page 110\]](#)

[The Application-Access File \[page 87\]](#)

4.6.2.1 SAP HANA XS Application Authentication

The authentication method determines whether or not authentication is required to access an application, and if required, which authentication methods must be used.

SAP HANA Extended Application Services (SAP HANA XS) provides a dedicated tool, the *SAP HANA XS Administration Tool*, that is designed to help you configure and maintain the authentication mechanism used to control runtime access to the applications you develop. The authentication method you select for access to your application depends on which authentication methods are supported by SAP HANA and whether or not your system administrator has configured the authentication method in the system backend.

You can use the *SAP HANA XS Administration Tool* to configure applications running in SAP HANA XS to use the following authentication mechanisms:

- SAML2
Choose this option if you have configured SAML2 assertions to enable SSO in Web browsers. SAML2 is version 2 of the Security Assertion Markup Language (SAML), which enables Web-based authentication including single sign-on across domains.

Note

The user who connects to the database using an external authentication provider must also have a database user known to the database. SAP HANA maps the external identity to the identity of the internal database user.

- SPNego

Choose this option if you want to SAP HANA XS applications to use Simple and Protected GSSAPI Negotiation Mechanism (SPNego) for authentication by means of Kerberos for Web-based (HTTP) access.

- X509 Authentication

X.509 client certificates For secure HTTP (HTTPS) access to SAP HANA XS applications, users can be authenticated by client certificates signed by a trusted Certification Authority (CA), which can be stored in the SAP HANA XS trust store.

- SAP logon ticket

For HTTPS access to SAP HANA XS applications, a user can be authenticated by a logon ticket that is issued when the same user logs on to an SAP system that is configured to create logon tickets (for example, the SAP Web Application Server or Portal).

To configure the trust relationship between the issuer of the SAP logon ticket and SAP HANA, you must specify the path to the SAP logon ticket trust store, which contains the trust chain for the ticket issuer. You can use the [SapLogonTicketTrustStore](#) keyword in the `xsengine.ini` file. Default values are: `$SECUDIR/saplogon.pse` or `$HOME/.ssl/saplogon.pem`.

Note

SAP HANA XS does not issue SAP logon tickets; it only accepts them. Since the tickets usually reside in a cookie, the issuer and SAP HANA XS need to be in the same domain to make sure that your browser sends the SAP logon ticket cookie with each call to SAP HANA XS.

- Form-based authentication

This option is used if interactive logon is desired. With form-based authentication, the logon request is redirected to a form to fill in, for example, displayed in Web page. The [Form-based authentication](#) option is enabled by default if the [Public](#) option is disabled in the application [Security](#) screen area.

Note

You must also enable the [Form-based authentication](#) in the `.xsaccess` file, if you want your application (or applications) to use form-based logon as the authentication method. Note that any other keywords in the authentication section of the `.xsaccess` file are ignored.

Form-based authentication requires the `libxsauthenticator` library, which must not only be available but also be specified in the list of trusted applications in the [xsengine](#) application container. The application list is displayed in the SAP HANA studio's [Administration Console](#) perspective in the following location:

► [Administration](#) ► [Configuration tab](#) ► [xsengine.ini](#) ► [application_container](#) ► [application_list](#) ►. If it is not displayed, ask the SAP HANA administrator to add it.

→ Tip

If you need to troubleshoot problems with form-based authentication, you can configure the generation of useful trace information in the [XSENGINE](#) section of the database trace component using the following entry: [xsa:sap.hana.xs.formlogon](#).

- Basic authentication

Logon with a recognized database user name and password. This option is used if non-interactive logon is desired. The [Basic authentication](#) option is enabled by default if the [Public](#) option is disabled in the application [Security](#) screen area.

The [authentication](#) configuration enables you to define the authentication methods to use for Browser requests either at the application level or for single packages in an application.

Note

The authentication mechanism set at the root of the application/package hierarchy is inherited by applications further down the application hierarchy.

By default, the *Public* option in the application *Security* screen area is disabled and the *Form-based authentication* and *Basic authentication* options are enabled automatically in the *Authentication* screen area. However, you can disable both form-based and basic authentication and enable other, additional authentication methods (for example, SAP logon tickets or X509 authentication). If multiple authentication methods are enabled, SAP HANA XS enforces the following order of priority:

- For non-interactive logon:
 1. X509 authentication
 2. SPNego
 3. SAP logon ticket
 4. Basic authentication
- For interactive logon:
 1. SAML
 2. Form-based authentication

If you **enable** the *Public* option in the application *Security* screen area, no authentication is required to start an application; the *Authentication* screen area is hidden, and you cannot select any authentication-method options.

Related Information

[The Application-Access File \[page 87\]](#)

[Application-Access File Keyword Options \[page 89\]](#)

4.7 Maintaining HTTP Destinations

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. The definition can be referenced by an application.

Context

If you want to configure an SAP HANA XS application to access data on a specific server that offers a specific service, for example, a service that is only available outside your network, it is recommended to configure the HTTP connection parameters in an HTTP destination file that you store locally as a design-time artifact. You can use an HTTP destination to call an external resource directly from a server-side JavaScript application. You can also use an HTTP destination when configuring a transport route, for example, to automate the process

of exporting a delivery unit from one system and importing it into another. To create an HTTP destination configuration for an SAP HANA XS application, you must perform the following high-level steps.

Procedure

1. Create a package for the SAP HANA XS application that will use the HTTP destination you define.
2. Define the details of the HTTP destination.

You define the details of an HTTP destination in a configuration file and using a specific syntax. The configuration file containing the details of the HTTP destination must have the file extension `.xshttpdest` and be located in the same package as the application that uses it or one of the application's subpackages.

3. Define any extensions to the HTTP destination configuration.

You can extend a configured HTTP destination, for example, by providing additional details concerning proxy servers and logon details. The details concerning the extensions to the HTTP destination must be specified in a separate configuration file. Like the original HTTP destination that the extension modifies, the configuration-file extension must have the file extension `.xshttpdest` and be located in the same package as the HTTP destination configuration file it extends and the application that uses it.

4. Check the HTTP destination configuration using the [SAP HANA XS Administration Tool](#).

The [SAP HANA XS Administration Tool](#) is available on the SAP HANA XS Web server at the following URL:
`http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/admin/cockpit`.

Note

Access to details of HTTP destinations in the [SAP HANA XS Administration Tool](#) requires the credentials of an authenticated database user and one of the following SAP HANA roles:

- [HTTPDestViewer](#)
- [HTTPDestAdministrator](#)

Related Information

[Create an HTTP Destination Configuration \[page 114\]](#)

[Extend an HTTP Destination Configuration \[page 126\]](#)

[HTTP Destination Configuration Syntax \[page 118\]](#)

4.7.1 Tutorial: Create an HTTP Destination

Create an HTTP destination defining connection details for services running on specific hosts. The definition can be referenced by an application.

Prerequisites

Since the artifacts required to create a simple HTTP destination are stored in the repository, it is assumed that you have already performed the following tasks:

- Create a development workspace in the SAP HANA repository
- Create a project in the workspace
- Share the new project
- Assigned your user the following SAP HANA roles:
 - [HTTPDestAdministrator](#)
 - [RuntimeConfAdministrator](#)

Context

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. The definition can be referenced by an application. You can also provide more (or modified) connection details in additional files called “extensions”; values specified in extensions overwrite values specified in the original HTTP destination configuration.

Note

HTTP destinations configurations are defined in a text file; you can use the editing tools provided with SAP HANA studio or your favorite text editor.

Procedure

1. Create a package for the SAP HANA XS application that will use the HTTP destination you define in this tutorial.
For example, create a package called `testApp`. Make sure you can write to the schema where you create the new application.
 - a. Start the SAP HANA studio and open the [SAP HANA Development](#) perspective.
 - b. In the [Systems](#) view, right-click the node in the package hierarchy where you want to create the new package and, in the pop-up menu that displays, choose [Packages...](#)
 - c. In the [New Package](#) dialog that displays, enter the details of the new package (`testApp`) that you want to add and click [OK](#).
2. Define the details of the HTTP destination.

You define the details of an HTTP destination in a configuration file that requires a specific syntax. The configuration file containing the details of the HTTP destination must have the file extension `.xshttpdest`. 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 enables direct editing of the file.

⚠ Caution

You must place the HTTP destination configuration and the XSJS application that uses it in the same application package. An application cannot reference an HTTP destination configuration that is located in another application package.

- a. Create a plain-text file called `yahoo.xshttpdest` and open it in a text editor.
- b. Enter the following code in the new file `yahoo.xshttpdest`.

```
host = "download.finance.yahoo.com";
port = 80;
description = "my stock-price checker";
useSSL = false;
pathPrefix = "/d/quotes.csv?f=a";
authType = none;
useProxy = false;
proxyHost = "";
proxyPort = 0;
timeout = 0;
```

- c. Save and activate the file.

📌 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.

3. View the activated HTTP destination.
You can use the *SAP HANA XS Administration Tool* to check the contents of an HTTP destination configuration.

📌 Note

To make changes to the HTTP Destination configuration, you must use a text editor, save the changes and reactivate the file.

- a. Start the *SAP HANA XS Administration Tool*.

The *SAP HANA XS Administration Tool* is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/admin/cockpit`.

→ Tip

Access to details of HTTP destinations in the *SAP HANA XS Administration Tool* requires the credentials of an authenticated database user and the permissions granted by the following SAP HANA roles:

- *RuntimeConfAdministrator*
- *HTTPDestAdministrator*

- b. In the *XS Artifact Administration* tab, expand the nodes in the *Application Objects* tree to locate the application `testApp`.
- c. Choose `yahoo.xshttpdest` to display details of the HTTP destination.

If you are using the Web-based *XS Administration Tool*, you can only make limited changes to the displayed HTTP destination configuration, as follows:

- *Save*
Commit to the repository any modifications made to the HTTP destination configuration in the current session.
- *Edit*
Display details of the corresponding **extension** to the selected HTTP destination configuration. If no extension exists, the *Edit* option is not available.
- *Extend*
Enables you to create an extension to the selected XS HTTP destination and associate the extension with another (new or existing) package.

Note

This option is only available if the selected HTTP destination is provided as part of a delivery unit, for example, as a destination template.

Related Information

[Tutorial: Extend an HTTP Destination \[page 126\]](#)

[The HTTP Destination Configuration \[page 116\]](#)

[HTTP Destination Configuration Syntax \[page 118\]](#)

4.7.1.1 The HTTP Destination Configuration

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. The definition can be referenced by an application.

You use the HTTP destination file to define not only the details of the host you want an application to reach by means of HTTP but also any further details that are necessary to establish the connection, for example, any proxy settings. If necessary, the proxy settings can also be defined in a separate, so-called "extension file". Both the configuration file you use to define an HTTP destination and the file used to specify any extensions to the HTTP destination are text files that must have the suffix `.xshttpdest`, for example, `myHTTPdestination.xshttpdest` or `myHTTPdestExtension.xshttpdest`.

Note

For security reasons, the HTTP destination configuration and the XSJS application that uses it must be in the same application package or one of the application's subpackages. An application cannot reference an HTTP destination configuration that is located in a different application package structure.

You configure an HTTP destination in a text file that contains the details of the connection to the HTTP destination, using a mandatory syntax comprising a list of *keyword=value* pairs, for example, `host = "download.finance.yahoo.com";`. After creating and saving the HTTP destination, you must activate it in the SAP HANA repository.

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.

The following configuration file for the HTTP destination `yahoo.xshttpdest` illustrates how to define an HTTP destination that can be used to access a financial service running on an external host.

```
host = "download.finance.yahoo.com";
port = 80;
description = "my stock-price checker";
useSSL = false;
pathPrefix = "/d/quotes.csv?f=a";
authType = none;
proxyType = none;
proxyHost = "";
proxyPort = 0;
timeout = 0;
```

After activating the configuration in the SAP HANA repository, you can view the details of the new HTTP destination using the [SAP HANA XS Administration Tool](#).

Note

Access to details of HTTP destinations in the [SAP HANA XS Administration Tool](#) requires the credentials of an authenticated database user and one of the following SAP HANA roles:

- [HTTPDestViewer](#)
- [HTTPDestAdministrator](#)

If you are using the Web-based [XS Administration Tool](#), you can only make limited changes to the displayed HTTP destination configuration, as follows:

- [Save](#):
Commit to the repository any modifications made to the HTTP destination configuration in the current session.
- [Edit](#):
Display details of the corresponding **extension** to the selected HTTP destination configuration. If no extension exists, the [Edit](#) option is not available.
- [Extend](#):
Enables you to create an extension to the selected XS HTTP destination and associate the extension with another (new or existing) package.

Note

This option is only available if the selected HTTP destination is provided as part of a delivery unit, for example, as a destination template.

Related Information

[HTTP Destination Configuration Syntax \[page 118\]](#)

[Tutorial: Create an HTTP Destination \[page 114\]](#)

4.7.1.2 HTTP Destination Configuration Syntax

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. Syntax rules apply to the contents of the HTTP destination configuration are checked when you activate the configuration in the repository.

Example: The .xshttpdest Configuration File

The following example shows all possible keyword combinations in the SAP HANA XS application-access (.xshttpdest) file.

Note

In the form shown below, the .xshttpdest file is **not** a working model; it is used to illustrate the syntax for all possible options.

```
host = "download.finance.yahoo.com";
port = 80;
//All of the following keywords are optional
description = "";
useSSL = false;
sslAuth = client;
sslHostCheck = true;
pathPrefix = "/d/quotes.csv?f=a";
authType = none;
samlProvider = "";
samlACS = "header";
samlAttributes = "";
samlNameId = ["email"];
proxyType = none;
proxyHost = ""; //in-line comments are allowed
proxyPort = 0;
timeout = 0;
remoteSID = "Q7E";
remoteClient = "007";
oAuthAppConfigPackage = "sap.hana.test";
oAuthAppConfig = "abapTest";
```

When you are defining the HTTP destination, bear in mind the following important syntax rules:

- A semi-colon (;) is required at the end of each line in the HTTP destination configuration, including the last line in the file.
- String values must be wrapped in quotes (""), for example:
host = "download.finance.yahoo.com";

ⓘ Note

The *host* and *port* keywords are mandatory; all other keywords are optional.

host

```
host = "download.finance.yahoo.com";
```

The *host* keyword is mandatory: it enables you to specify the hostname of the HTTP destination providing the service or data you want your SAP HANA XS application to access.

port

```
port = 80;
```

The *port* keyword is mandatory; it enables you to specify the port number to use for connections to the HTTP destination hosting the service or data you want your SAP HANA XS application to access.

description

```
description = "my short description of the HTTP connection";
```

The optional keyword *description* enables you to provide a short description of the HTTP destination you want to configure. If you do not want to provide a description, include the *description* but leave the entry between the quotes empty, for example, "".

useSSL

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

The optional keyword *useSSL* is of type Boolean and enables you to specify if the outbound connections between SAP HANA XS and the HTTP destination is secured with the Secure Sockets Layer (SSL) protocol (HTTPS).

ⓘ Note

Setting this option does not configure SSL; if you want to use SSL to secure connections to the configured destination, you must ensure that SAP HANA is already set up to enable secure outbound connections using SSL.

If `useSSL = true`, you can set the authentication type with the keyword `sslAuth`. You can also use the `sslHostCheck` to enable a check which ensures that the certificate used for authentication is valid (matches the host).

sslAuth

```
sslAuth = [client | anonymous];
```

If `useSSL = true`, you can use the keyword `sslAuth` to set the authentication type. The following values are permitted:

- **client**
(Default setting). You must create a TRUST store entry in the [SAP HANA XS Admin Tool's Trust manager](#) (or use an existing one that is known to the HTTP destination configuration) and maintain the trust relationship with the SSL server, for example, by adding a certificate to the trust store that is used for the authentication process.
- **anonymous**
A built-in key is used for SSL encryption; no TRUST store is needed.. No authentication via SSL is possible.

sslHostCheck

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

If `useSSL = true`, you can use the keyword `sslHostCheck` to enable a check which ensures that the certificate used for authentication is valid (matches the host). The following values are permitted:

- **true**
(Default setting). The SSL certificate subject must match the host name. For example, if SSL server certificate `CN=server1.acme.com`, then the host parameter must be `server1.acme.com`. If there is no match, SSL terminates.
- **false**
No host check is performed. Note that if the SSL server certificate is `CN=server1.acme.com`, and you use `"localhost"` as a connection parameter (because this certificate is installed on its own server), then this works with `sslHostCheck` deactivated (`sslHostCheck=false`).

pathPrefix

```
pathPrefix = "";
```

The optional keyword [*pathPrefix*](#) enables you to specify a text element to add to the start of the URL used for connections to the service specified in the HTTP destination configuration. For example, `pathPrefix = "/d/quotes.csv?f=a"` inserts the specified path into the URL called by the connection.

authType

```
authType = [none | basic | AssertionTicket | SamlAssertion |  
SamlAssertionPropagation];
```

The optional keyword `authType` enables you to specify the authentication method that must be used for connection requests for the service located at the HTTP destination specified in the configuration, for example, "basic", which requires users to provide a user name and password as authentication credentials. Permitted values for the `authType` are "none", "basic", and "AssertionTicket". If no authentication type is specified, the default setting "none" applies.

The `AssertionTicket` option is for use with XSJS applications that want to enable access to HTTP services running on remote SAP servers using single sign-on (SSO) with SAP assertion tickets. If the `AssertionTicket` option is enabled, a user with administration privileges in SAP HANA must use the parameter `saplogontickettruststore` to specify the location of the trust store containing the assertion tickets.

→ Tip

The `saplogontickettruststore` parameter can be set in ► [\[indexserver | xsengine\].ini](#) ► [authentication](#) ► [saplogontickettruststore](#) ►.

If `authType = AssertionTicket` is set you also need to set values for the keywords `remoteSID` and `remoteclient`.

For `authType = SamlAssertion`, you must also set the subproperties `samlProvider`, `samlACS`, `samlAttributes`, and `samlNameId`.

samlProvider

```
samlProvider = "";
```

If you set `authType = SamlAssertion`, you must also set the subproperty `samlProvider`, which enables you to specify the entityID of the remote SAML party.

samlACS

```
samlACS = "header";
```

If you set `authType = SamlAssertion`, you must also set the subproperty `samlACS`, which enables you to specify the way in which SAML assertions or responses are sent. The following values are supported:

- "" (empty string)
A SAML response (including the SAML assertion) is sent to the HTTP destination end point as a POST parameter.
- `/saml/acs/sso`

If you provide a URL path, the SAML response (including the SAML Assertion) is sent to the specified endpoint in an additional Web connection to establish the authentication context (session). When the outbound communication is being established, there are two connections: first to the specified end point (for example, /saml/asc/sso) and then to the destination service end point.

- **header**
The SAML response (including the SAML assertion) is sent in the HTTP header authorization with the following syntax: `Authorization: SAML2.0 <base-64-saml-response>`.
- **parameter:assertion**
The SAML Assertion is sent as a POST parameter. This flavor is needed for JAM integrations.

samlAttributes

```
samlAttributes = "name1=<property>&name2=<property>";
```

If you set `authType = SamlAssertion`, you must also set the subproperty `samlAttributes`, which enables you to specify additional attributes for the SAML Assertion.

samlNameId

```
samlNameId = ["email", "unspecified"];
```

If you set `authType = SamlAssertion`, you must also set the subproperty `samlNameId`, which enables you to define a list of name-ID mappings. The following values are supported:

- `email`
- `unspecified`

For example, if you have an e-mail maintained in SAP HANA User Self Services (USS), the SAML assertion contains your e-mail address; if you do **not** have a e-mail address maintained in SAP HANA USS, the mapping is “unspecified”.

proxyType

```
proxytype = none;
```

The optional keyword `proxyType` enables you to specify if a proxy server must be used to resolve the host name specified in the HTTP destination configuration file, and if so, which **type** of proxy. The following values are allowed:

- `none`
- `http`
- `socks`

⚠ Caution

`proxyType` replaces and extends the functionality previously provided with the keyword `useProxy`. For backward compatibility, the `useProxy` is still allowed but should not be used any more.

To define the proxy host and the port to connect on, use the keywords `proxyHost` and `proxyPort` respectively.

If you want to include the proxy-related information in a separate configuration (a so-called **extension** to the original HTTP destination configuration), you must set `proxyType = none` in the original HTTP destination configuration. In the HTTP destination extension that references and modifies the original HTTP destination, you can change the proxy setting to `proxyType = http`. You must then provide the corresponding host name of the proxy server and a port number to use for connections.

proxyHost

```
proxyHost = "";
```

If you use the keyword `useProxy = true` to specify that a proxy server must be used to resolve the target host name specified in the HTTP destination configuration, you must use the [proxyHost](#) and [proxyPort](#) keywords to specify the fully qualified name of the host providing the proxy service (and the port number to use for connections). The name of the proxy host must be wrapped in quotes, as illustrated in the following example,

```
proxyHost = "myproxy.hostname.com"
```

proxyPort

```
proxyPort = 8080;
```

If you use the keyword `useProxy = true` to indicate that a proxy server must be used to resolve the host name specified in the HTTP destination configuration, you must also use the [proxyPort](#) keyword (in combination with [proxyHost](#) =) to specify the port on which the proxy server accepts connections.

timeout

```
timeout = -1;
```

The optional keyword [timeout](#) enables you to specify for how long (in milliseconds) an application tries to connect to the remote host specified in the HTTP destination configuration, for example, `timeout = 5000`; (5 seconds). By default, the timeout interval is set to -1, which means that there is no limit to the time required to connect to the server specified in the HTTP destination configuration. In the default setting, the application keeps trying to connect to the destination server either until the server responds, however long this takes, or

the underlying request-session timeout (300 seconds) is reached. The default setting (-1) is intended to help in situations where the destination server is slow to respond, for example, due to high load.

remoteSID

```
remoteSID = "Q7E";
```

The optional keyword `remoteSID` enables you to specify the SID of a remote ABAP system. You use this keyword in combination with the `remoteClient` keyword, for example, to enable an application to log on to an ABAP system that is configured to provide SAP assertion tickets. If the XSJS application service requires access to remote services, you can create an HTTP destination that defines the logon details required by the remote ABAP system and specifies SSO with SAP assertion tickets as the logon authentication method.

Note

In the *XS Administration Tool*, the value specified in an HTTP destination configuration file with the `remoteSID` keyword is displayed in the *SAP SID* field in the *AUTHENTICATION* section of the application's runtime configuration. The *SAP SID* option is only available if you select *SAP Assertion Ticket* as the authentication type in the application's runtime configuration.

remoteClient

```
remoteClient = "007";
```

The optional keyword `remoteClient` enables you to specify the client number to use when logging on to a remote ABAP system. You use this keyword in combination with the `remoteSID` keyword, for example, to enable an application to logon to an ABAP system that is configured to provide SAP assertion tickets. If the XSJS application service requires access to remote services, you can create an HTTP destination that defines the logon details required by the remote ABAP system and specifies SSO with SAP assertion tickets as the logon authentication method.

Note

In the *XS Administration Tool*, the value specified in an HTTP destination configuration file with the `remoteClient` keyword is displayed in the *SAP Client* field in the *AUTHENTICATION* section of the application's runtime configuration. The *SAP Client* option is only available if you select *SAP Assertion Ticket* as the authentication type in the application's runtime configuration.

oAuthAppConfigPackage

```
oAuthAppConfigPackage = "sap.hana.test";
```

Use the optional keyword `oAuthAppConfigPackage` enables you to specify the location of the package that contains the OAuth application configuration to be used by an HTTP destination configuration.

oAuthAppConfig

```
oAuthAppConfig = "abapTest";
```

Use the optional keyword `oAuthAppConfig` enables you to specify the name of the OAuth application configuration to be used by an HTTP destination configuration. The OAuth application configuration is a file describing the application-specific OAuth parameters that are used to enable access to a resource running on a remote HTTP destination. The OAuth application configuration is defined in a design-time artifact with the mandatory file suffix `.xsoauthappconfig`; the configuration file must be specified using the JSON format.

modifies

```
modifies pkg.path.testApp:yahoo.xshttpdest;
```

The keyword *modifies* can only be used in an HTTP **extension** file and enables you to reference an existing HTTP destination (or extension) whose settings you want to further extend or modify. The settings in an HTTP destination **extension** overwrite any identical settings in the original HTTP destination configuration. The HTTP destination configuration referenced by the *modifies* keyword must already exist.

Note

The HTTP destination **extension** does not have to be tied to a particular XSJS application; it can be located in any application package or subpackage. For this reason, you must include the full package path to the HTTP destination extension when using the *modifies* keyword.

Related Information

[The HTTP Destination Configuration \[page 116\]](#)

[The HTTP Destination Extension \[page 129\]](#)

4.7.2 Tutorial: Extend an HTTP Destination

Extend an HTTP destination defining connection details for services running on specific hosts, for example, by providing additional details. The definition and the extension details can be referenced by an application.

Prerequisites

Since the artifacts required to create an HTTP destination extension are stored in the repository, it is assumed that you have already performed the following tasks:

- Create a development workspace in the SAP HANA repository
- Create a project in the workspace
- Share the new project
- Assigned your user the following SAP HANA roles:
 - [HTTPDestAdministrator](#)
 - [RuntimeConfAdministrator](#)

📘 Note

This tutorial shows you how to modify an HTTP destination by providing details of a proxy server that must be used to resolve host names specified in the connection details; you must supply the name of a working proxy server that is available in your environment.

Context

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. The definition can be referenced by an application. You can also provide more (or modified) connection details in additional files called “extensions”; values specified in extensions overwrite values specified in the original HTTP destination configuration.

📘 Note

HTTP destinations configurations and any extensions are defined in a plain-text file; you can use the editing tools provided with SAP HANA studio or your favorite text editor to add entries to the configuration file.

Procedure

1. Create a package for the SAP HANA XS application that will use the HTTP destination (and extension) you define in this tutorial.
For example, create a package called `testApp`. Make sure you can write to the schema where you create the new application.

- a. Start the SAP HANA studio and open the *SAP HANA Development* perspective.
 - b. In the *Systems* view, right-click the node in the package hierarchy where you want to create the new package and, in the pop-up menu that displays, choose *Packages...*
 - c. In the *New Package* dialog that displays, enter the details of the new package (*testApp*) that you want to add and click *OK*.
2. Define the details of the new HTTP destination.

You define the details of an HTTP destination in a configuration file that requires a specific syntax. The configuration file containing the details of the HTTP destination must have the file extension `.xshttpdest`.

⚠ Caution

You must place the HTTP destination configuration in the application package that uses it. An application cannot reference an HTTP destination configuration that is located in another application package.

- a. Create a plain-text file called `yahoo.xshttpdest` and open it in a text editor.
- b. Enter the following code in the new file `yahoo.xshttpdest`.

```
host = "download.finance.yahoo.com";
port = 80;
description = "my stock-price checker";
useSSL = false;
pathPrefix = "/d/quotes.csv?f=a";
authType = none;
proxyType = none;
proxyHost = "";
proxyPort = 0;
timeout = 0;
```

- c. Save and activate the file.

📌 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.

3. View the activated HTTP destination.
- You can use the *SAP HANA XS Administration Tool* to check the contents of an HTTP destination configuration.

📌 Note

To make changes to the HTTP Destination configuration, you must use a text editor, save the changes and reactivate the file.

- a. Open a Web browser.
- b. Start the *SAP HANA XS Administration Tool*.

The *SAP HANA XS Administration Tool* tool is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/admin/`.

Note

Access to details of HTTP destinations in the *SAP HANA XS Administration Tool* requires the credentials of an authenticated database user and the permissions granted by the following SAP HANA roles:

- *RuntimeConfAdministrator*
- *HTTPDestAdministrator*

- In the *XS Artifact Administration* tab, expand the nodes in the *Application Objects* tree to locate the application `testApp`.
- Choose `yahoo.xshttpdest` to display details of the HTTP destination.

- Define the details of the **extension** to the HTTP destination you created in the previous steps.

Like the HTTP destination itself, you define an extension to an HTTP destination in a configuration file that requires a specific syntax. The configuration file containing the details of the HTTP destination must have the file suffix `.xshttpdest`.

Caution

You must place the HTTP destination configuration (and any extensions to the configuration) in the application package that uses them. An application cannot reference an HTTP destination configuration (or an extension) that is located in another application package.

- Create a plain-text file called `yahooProxy.xshttpdest` and open it in a text editor.
- Enter the following code in the new file `yahooProxy.xshttpdest`.

```
modifies testApp:yahoo.xshttpdest;  
proxyType = http;  
proxyHost = "proxy.mycompany.com";  
proxyPort = 8080;
```

Note

Replace the value in `proxyHost` with the name of the host providing the proxy service.

- Save and activate the file.
- View and check the details of the activated HTTP destination **extension** `yahooProxy.xshttpdest`. You can use the *SAP HANA XS Administration Tool* to check the contents of an HTTP destination configuration or an extension to the configuration.

Note

To make changes to the HTTP Destination configuration (or any extension), you must use a text editor, save the changes and reactivate the file.

- Open a Web browser.
- Start the *SAP HANA XS Administration Tool*.

The *SAP HANA XS Administration Tool* tool is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/admin/cockpit`.

Note

In the default configuration, the URL redirects the request to a logon screen, which requires the credentials of an authenticated SAP HANA database user to complete the logon process.

- c. In the *XS Artifact Administration* tab, expand the nodes in the *Application Objects* tree to locate the application `testApp`.
- d. Choose `yahooProxy.xshttpdest` to display details of the HTTP destination extension.

Related Information

[Tutorial: Create an HTTP Destination \[page 114\]](#)

[The HTTP Destination Configuration \[page 116\]](#)

[HTTP Destination Configuration Syntax \[page 118\]](#)

4.7.2.1 The HTTP Destination Extension

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. An extension to an HTTP destination provides additional information or modifies values set in the original configuration.

You can use one or more **extension** to an HTTP destination configuration; the extensions include additions to the original settings or modifications to the values set in the original configuration. For example, you could include basic configuration settings in an HTTP destination and provide details of any required proxy settings in a separate, so-called “extension”.

You define an extension to an HTTP destination configuration in a text file that contains the details of the modifications you want to apply to the connection details for the original HTTP destination. The HTTP destination extension uses a mandatory syntax comprising a list of *keyword=value* pairs, for example, `host = "download.finance.myhoo.com"`; . The same syntax rules apply for the basic HTTP destination configuration and any extensions. Both files must also have the file suffix `.xshttpdest`, for example, `myHTTPdestination.xshttpdest` or `myHTTPextension.xshttpdest`. After creating and saving the HTTP destination extension, you must activate it in the SAP HANA repository.

Note

The HTTP destination **extension** does not have to be tied to a particular XSJS application; it can be located in any application package or subpackage. For this reason, you must include the full package path to the HTTP destination extension.

The following configuration file for the HTTP destination `yahooProxy.xshttpdest` illustrates how to modify the proxy settings specified in the HTTP destination `yahoo.xshttpdest`, located in the application package `pkg.path.testApp`.

```
modifies pkg.path.testApp:yahoo.xshttpdest;  
proxyType = http;  
proxyHost = "proxy.host.name.com";  
proxyPort = 8080;
```

Note

For backward compatibility, the keyword `userProxy` still works; however, it has been replaced with the keyword `proxyType`, which takes the values: `[none | http | socks]`.

After activation, you can view the details of the new HTTP destination extension using the [SAP HANA XS Administration](#) tool.

Note

Access to details of HTTP destinations in the [SAP HANA XS Administration Tool](#) requires the credentials of an authenticated database user and one of the following SAP HANA roles:

- [HTTPDestViewer](#)
- [HTTPDestAdministrator](#)

4.7.3 Tutorial: Create an OAuth Configuration Package

Create the files required to enable a service that uses OAuth to authorize access to a resource running on a remote HTTP destination.

Prerequisites

Since the artifacts required to create an XS OAuth configuration package are stored in the SAP HANA repository, it is assumed that you have the following:

- A development workspace in the SAP HANA repository
- A shared project in the workspace
- Access to SAP HANA development tools, for example:
 - SAP HANA studio
 - SAP HANA Web-based Workbench
- An HTTP destination configuration (`.xshttpdest`)
- Your SAP HANA database user has the permissions granted by the following roles:
 - [RuntimeConfAdministrator](#)
 - [HTTPDestAdministrator](#)
 - [oAuthAdmin](#)

Context

An OAuth configuration package is a collection of configuration files that define the details of how an application uses OAuth to enable login to a resource running on a remote HTTP destination.

An HTTP destination defines connection details for services running on specific hosts whose details you want to define and distribute. Additional syntax rules apply to the contents of the HTTP destination configuration are checked when you activate the configuration in the repository.

An OAuth configuration requires the following **dependent** configuration files:

- OAuth application configuration (`<filename>.xsoauthappconfig`)
Describes the configuration of the OAuth application parameters including the name and package location of the associated client configuration and any mandatory or optional scopes.
- OAuth client configuration (`<filename>.xsoauthclientconfig`)
Describes the configuration of the OAuth client including: the client ID, the client authentication type, and the name and package location of the associated client **flavor**.
- OAuth client flavor configuration (`<filename>.xsoauthclientflavor`)
Describes the OAuth client flavor setup used by the XS OAuth client configuration, including: the protocol steps and the parameters to be set. Note that, normally, you do not need to change the OAuth client flavor configuration.

→ Tip

You connect the OAuth configuration to the HTTP destination configuration in the HTTP destination's runtime configuration. Access to the runtime configuration tools requires the permissions included in an administrator role.

Procedure

1. Create an OAuth application configuration.

You need to create the base configuration for your OAuth application in a design-time file with the mandatory file-extension `.xsoauthappconfig`. The application configuration is stored in the SAP HANA repository and must be activated to create the corresponding catalog objects.

- a. Create the design-time file that contains your OAuth application configuration, for example, `oauthDriveApp.xsoauthappconfig`
- b. Define the details of the new OAuth application configuration, as follows:

```
{
  "clientConfig"      :
  "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac",
  "mandatoryScopes"   : ["OAUTH2_TEST_SCOPE1", "OAUTH2_TEST_SCOPE2"],
  "description"       : "ABAP Testapplication for OAuth"
}
```

Note

In this example, the OAuth client configuration is located in the package `sap.hana.xs.oauth.lib.providerconfig.providermodel`; you can change the path to suit your own requirements.

2. Create an OAuth client configuration (optional).

You create the client configuration for your OAuth application in a design-time file with the mandatory file-extension `.xsoauthclientconfig`. You can either use an existing client configuration from the package `sap.hana.xs.oauth.lib.providerconfig.providermodel` or create your own client configuration. The application configuration is stored in the SAP HANA repository and must be activated to create the corresponding catalog objects.

- a. Create the design-time file that contains your OAuth client configuration, for example, `ABAPv1.xsoauthclientconfig`
- b. Define the details of the new OAuth client configuration, as follows:

```
{
  "clientFlavor"
  :
  "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac",
  "clientId"
  : "<OAuth ClientId registered at ABAP>",
  "clientAuthType"
  : "basic",
  "authorizationEndpointURL"
  : "/sap/bc/sec/oauth2/authorize",
  "tokenEndpointURL"
  : "/sap/bc/sec/oauth2/token",
  "revocationEndpointURL"
  : "/sap/bc/sec/oauth2/revoke",
  "redirectURL"
  : "<External_XS_HOST>:<PORT>/sap/hana/xs/
  oauth/lib/runtime/tokenRequest.xsjs",
  "flow"
  : "authCode",
  "scopeReq"
  : "maxScopes",
  "description"
  : "OAuth Client for SAP Application Server
  ABAP - Authorization Code Flow"
}
```

3. Create the OAuth client flavor (optional).

The OAuth client flavor file is a design-time artifact that provides details of the OAuth protocol for a client application which uses the services provided by a corresponding OAuth application. The OAuth client flavor steps are defined in a design-time artifact with the mandatory file suffix `.xsoauthclientflavor`; the configuration file must be specified using the JSON format.

→ Tip

You do not have to create the OAuth client flavor from scratch; SAP HANA provides some example OAuth client flavors which you can use. The example OAuth client flavors are located in the following package: `sap.hana.xs.oauth.lib.providerconfig.providermodel`.

The following example shows the required format and syntax for the contents of the `.xsoauthclientflavor` artifact.

📌 Note

The example below is not complete; it is intended for illustration purposes only.

```
{ "parameters":[
  { "flavorStep":"1Aut", "paramLocation":"uri", "paramName":"client_id",
    "paramValue":"client_id", "valueType":"eval",
    "paramMandatory":"true" },
  { "flavorStep":"2Gra", "paramLocation":"head", "paramName":"Authorization",
    "paramValue":"Basic Authentication", "valueType":"sec",
    "paramMandatory":"true" },
  { "flavorStep":"3Prc", "paramLocation":"head", "paramName":"Bearer",
    "paramValue":"access_token", "valueType":"sec",
    "paramMandatory":"true" },
  { "flavorStep":"4Ref", "paramLocation":"head", "paramName":"Authorization",
    "paramValue":"Basic Authentication", "valueType":"sec",
    "paramMandatory":"true" },
  { "flavorStep":"5Rev", "paramLocation":"para", "paramName":"token",
    "paramValue":"access_token", "valueType":"sec",
    "paramMandatory":"true" },
  ] }
```

4. Activate all the XS OAuth configuration files.

Activating the configuration files creates the corresponding catalog objects.

5. Add the OAuth configuration to the runtime configuration of the HTTP destination configuration that requires it.

The *SAP HANA XS Administration Tool* is available on the SAP HANA XS Web server at the following URL:
`http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/admin/cockpit`.

Note

Access to details of HTTP destinations in the *SAP HANA XS Administration Tool* requires the credentials of an authenticated database user and one of the following SAP HANA roles:

- *RuntimeConfAdministrator*
- *HTTPDestAdministrator*
- *oAuthAdmin*

- a. Start the *XS Artifact Administration* tool.
- b. In the *Application Objects* list, locate and choose the HTTP destination configuration that you want to modify.
- c. Choose the *OAuth Details* tab.
- d. Choose **Edit** > *Browse OAuth App Configs*.
- e. Select an OAuth application configuration from the list displayed.
The name of the application configuration you choose and the absolute path to the package where it is located are displayed in the appropriate fields, for example.
 - *OAuth App Config Package*: `sap.hana.test`
 - *OAuth App Config Name*: `abapTest`

Note

The values displayed here must also be present in the HTTP destination configuration to which the OAuth configuration applies.

For example, the HTTP destination corresponding to the OAuth configuration you are setting up in this task must also contain entries that describe the name and package location of the OAuth application configuration to use.

```
oAuthAppConfigPackage = "sap.hana.test";  
oAuthAppConfig = "abapTest";
```

- f. Navigate to the OAuth client configuration and set the client secret.
- g. Choose *Save* to update the run-time configuration for the HTTP destination.

Related Information

[Tutorial: Create an HTTP Destination \[page 114\]](#)

[OAuth Application Configuration Syntax \[page 134\]](#)

[OAuth Client Configuration Syntax \[page 135\]](#)

[OAuth Client Flavor Syntax \[page 140\]](#)

4.7.3.1 OAuth Application Configuration Syntax

The format and syntax required in a design-time artifact describing an OAuth application configuration.

The OAuth application configuration is a file describing the application-specific OAuth parameters that are used to enable access to a resource running on a remote HTTP destination. The OAuth application configuration is defined in a design-time artifact with the mandatory file suffix `.xsoauthappconfig`; the configuration file must be specified using the JSON format.

Note

The following code example is not a working example; it is provided for illustration purposes, only.

```
{
  "clientConfig": "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac",
  "description": "ABAP test application for OAuth",
  "mandatoryScopes": ["OAUTH2_TEST_SCOPE1", "OAUTH2_TEST_SCOPE2"],
  "optionalScopes": ["OAUTH2_TEST_SCOPE3", "OAUTH2_TEST_SCOPE4"],
  "modifies": "sap.hana.test:abapTest"
}
```

An OAuth configuration requires the following **dependent** configuration files:

- OAuth application configuration (`.xsoauthappconfig`)
- OAuth client configuration (`.xsoauthclientconfig`)
- OAuth client flavor configuration (`.xsoauthclientflavor`)

clientConfig

Use the `clientConfig` keyword to specify the fully qualified name of the associated `xsoauthclientconfig` artifact, using the format `<path.to.package>:<XSOAuthClientConfigObjectName>`.

```
"clientConfig": "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac",
```

Note

It is mandatory to specify the name and location of the package containing the associated OAuth client configuration.

description

Use the `description` keyword to provide an optional short description of the contents of the OAuth application configuration.

```
"description": "ABAP test application for OAuth",
```

mandatoryScopes

Use the `mandatoryScopes` keyword to specify one or more (in an array) of strings describing the mandatory permissions requested by the client.

```
"mandatoryScopes": ["OAUTH2_TEST_SCOPE1", "OAUTH2_TEST_SCOPE2"],
```

optionalScopes

Use the `optionalScopes` keyword to specify one or more (in an array) of strings describing the optional permissions to be used by the client.

```
"optionalScopes": ["OAUTH2_TEST_SCOPE3", "OAUTH2_TEST_SCOPE4"],
```

modifies

Use the `modifies` keyword to indicate that the current XS OAuth application configuration (for example, `abapTest2.xsoauthappconfig` is based on (and extends) another SAP HANA XS OAuth application configuration (for example, `abapTest.xsoauthappconfig`). You must specify the fully qualified name of the associated SAP HANA XS OAuth application configuration artifact (`xsoauthappconfig`), using the format `<path.to.package>:<ObjectName>`.

```
"modifies": "sap.hana.test:abapTest.xsoauthappconfig",
```

Related Information

[OAuth Client Configuration Syntax \[page 135\]](#)

[OAuth Client Flavor Syntax \[page 140\]](#)

[Tutorial: Create an OAuth Configuration Package \[page 130\]](#)

4.7.3.2 OAuth Client Configuration Syntax

The format and syntax required in a design-time artifact describing the OAuth client configuration.

The OAuth client configuration is a file describing details of the client parameters for an application which uses the services provided by a corresponding OAuth application that enables access to a resource running on a remote HTTP destination. The OAuth client configuration is defined in a design-time artifact with the mandatory file suffix `.xsoauthclientconfig`; the configuration file must be specified using the JSON format. The following code example shows the contents of a typical OAuth client configuration.

Note

The following code example is not a working example; it is provided for illustration purposes, only.

```
{
  "clientFlavor": "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac",
  "clientId": "<The OAuth ClientId you registered at ABAP>",
  "clientAuthType": "basic",
  "authorizationEndpointURL": "/sap/bc/sec/oauth2/authorize",
  "tokenEndpointURL": "/sap/bc/sec/oauth2/token",
  "revocationEndpointURL": "/sap/bc/sec/oauth2/revoke",
  "flow": "authCode",
  "description": "OAuth Client for ABAP server",
  "samlIssuer": "",
  "redirectURL": "<HOST>:<PORT>/sap/hana/xs/oauth/lib/runtime/tokenRequest.xsjs",
  "scopeReq": "maxScopes",
  "shared": "true",
  "modifies": "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac"
}
```

In this example, the OAuth client configuration is located in the package `com.acme.oauth.lib`; change the path specified in `clientFlavor` to suit your own requirements. You will also have to change the value specified for `clientId` and `redirectURL`.

→ Tip

SAP HANA provides some example OAuth client configurations which you can use; you can find them in the following package: `sap.hana.xs.oauth.lib.providerconfig.providermodel`

clientFlavor

Use the `clientFlavor` keyword to specify the fully qualified name of the associated XS OAuth client flavor configuration artifact, for example, `ABAPv1.xsoauthclientfavor`; you must use the format `<path.to.package>:<ObjectName>` (no file extension is required).

```
"clientFlavor": "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac",
```

Note

It is mandatory to specify the name and location of the package containing the associated OAuth client **flavor** configuration.

clientId

Use the `clientId` keyword to define a string that specifies the customer's ID, which is used to identify the client with the server. The `clientId` must be changed to suit your requirements. Typically, the `clientId` is obtained by registering with a specific service provider.

```
"clientId" : "<The OAuth ClientId you registered at ABAP>",
```

Note

It is mandatory to define the `clientId`.

clientAuthType

Use the `clientAuthType` keyword to define a number that specifies the client authentication type, for example, `"cert"` or `"basic"`.

```
"clientAuthType" : "basic",
```

Note

It is mandatory to define the `clientAuthType`.

The following values are permitted:

- `basic` (user and password)
- `cert` (authentication by client certificate)

authorizationEndpointURL

Use the `authorizationEndpointURL` keyword to specify a string that defines the authorization endpoint. The authorization endpoint is the endpoint on the authorization server where the resource owner logs on and grants authorization to the client application.

```
"authorizationEndpointURL" : "/sap/bc/sec/oauth2/authorize",
```

Note

It is mandatory to define the `authorizationEndpointURL`.

tokenEndpointURL

Use the `tokenEndpointURL` keyword to specify a string that defines the token endpoint. The token endpoint is the endpoint on the authorization server where the client application exchanges the authorization code, the client ID, and the client secret for an access token.

```
"tokenEndpointURL" : "/sap/bc/sec/oauth2/token",
```

Note

It is mandatory to define the `tokenEndpointURL`.

revocationEndpointURL

Use the `revocationEndpointURL` keyword to specify a string that defines the token endpoint. The token endpoint is the endpoint on the authorization server where the client application exchanges the authorization code, the client ID, and the client secret for an access token.

```
"revocationEndpointURL" : "/sap/bc/sec/oauth2/revoke",
```

Note

It is mandatory to define a value for the `revocationEndpointURL`.

flow

Use the `flow` keyword to specify a number that defines the authorization flow used during the authentication exchange, for example, `saml2Bearer` or `authCode`.

```
"flow" : "saml2Bearer",
```

Note

It is mandatory to define a value for `flow`.

The following values are permitted:

- `saml2Bearer`
- `authCode`

description

Use the optional `description` keyword to provide a short description of the OAuth client configuration.

```
"description": "OAuth Client for SAP App Server ABAP - Authorization Code Flow"
```

samlIssuer

Use the optional `samlIssuer` keyword to specify a string that defines the [SAML issuer ID](#). The SAML issuer ID describes the issuer of the SAML token. The SAML bearer extension enables the validation of SAML tokens as part of granting the OAuth access token.

Note

You set this parameter only if the parameter `flow` is set to `saml2Bearer`, for example,
`"flow" : "saml2Bearer".`

```
"samlIssuer" : "" ,
```

redirectURL

Use the `redirectURL` keyword to specify a string that defines the *redirection endpoint*. The redirection endpoint is the endpoint in the client application where the resource owner is redirected to, after having granted authorization at the authorization endpoint. The `redirectURL` must be changed to suit your requirements.

```
"redirectURL" : "<HOST>:<PORT>/sap/hana/xs/oauth/lib/runtime/tokenRequest.xsjs",
```

Note

If `"flow" : "authCode"`, it is mandatory to define a value for the `redirectURL`.

scopeReq

Use the `scopeReq` keyword to specify whether the maximum available scope from all applications using this client configuration is **always** requested or the scope set is specified iteratively.

```
"scopeReq" : "maxScopes",
```

The following values are permitted:

- `maxScopes`
- `iterativeScopes`

Note

Currently only `maxScopes` is implemented.

shared

Use the `shared` keyword to specify a number that defines whether the if the XS OAuth client configuration can be shared between applications.

```
"shared" : "false",
```

The following values are permitted:

- `true` (shared)
- `false` (**not** shared)

Note

Currently only `true` is implemented.

modifies

Use the `modifies` keyword to indicate that the current XS OAuth client configuration, for example, `abap_ac1.xsoauthclientconfig`, is based on (and extends) another SAP HANA XS OAuth client configuration (for example, `abap_ac.xsoauthclientconfig`). You must specify the fully qualified name of the associated OAuth client configuration artifact (`<fileName>.xsoauthclientconfig`), using the format `<path.to.package>:<ArtifactName>.xsoauthclientconfig`.

```
"modifies": "sap.hana.xs.oauth.lib.providerconfig.providermodel:abap_ac.xsoauthclientconfig",
```

Related Information

[OAuth Client Flavor Syntax \[page 140\]](#)

[OAuth Application Configuration Syntax \[page 134\]](#)

[Tutorial: Create an OAuth Configuration Package \[page 130\]](#)

4.7.3.3 OAuth Client Flavor Syntax

The format and syntax required in a design-time artifact that describes the OAuth client flavors.

The OAuth client flavor file provides details of the OAuth protocol for a client application that uses the services provided by a corresponding OAuth application. The OAuth client flavor steps are defined in a design-time artifact with the mandatory file suffix `.xsoauthclientflavor`; the configuration file must be specified using the JSON format.

Note

The following example of an OAuth client flavor configuration is incomplete; it is intended for illustration purposes only.

```
{ "parameters": [
  { "flavorStep": "1Aut", "paramLocation": "uri", "paramName": "client_id",
    "paramValue": "client_id", "valueType": "eval",
    "paramMandatory": "true" },
```

```

{ "flavorStep": "1Aut", "paramLocation": "uri", "paramName": "redirect_uri",
  "paramValue": "redirect_uri", "valueType": "eval",
  "paramMandatory": "true" },
{ "flavorStep": "1Aut", "paramLocation": "uri", "paramName": "scope",
  "paramValue": "scope", "valueType": "eval",
  "paramMandatory": "true" },
{ "flavorStep": "1Aut", "paramLocation": "uri", "paramName": "response_type",
  "paramValue": "code", "valueType": "litr",
  "paramMandatory": "true" },
{ "flavorStep": "1Aut", "paramLocation": "uri", "paramName": "state",
  "paramValue": "state", "valueType": "eval",
  "paramMandatory": "true" },
{ "flavorStep": "2Gra", "paramLocation": "head", "paramName": "Authorization",
  "paramValue": "Basic Authentication", "valueType": "sec",
  "paramMandatory": "true" },
{ "flavorStep": "2Gra", "paramLocation": "head", "paramName": "Content-Type",
  "paramValue": "application/x-www-form-urlencoded", "valueType": "litr",
  "paramMandatory": "true" },
{ "flavorStep": "2Gra", "paramLocation": "para", "paramName": "code",
  "paramValue": "code", "valueType": "eval",
  "paramMandatory": "true" },
{ "flavorStep": "2Gra", "paramLocation": "para", "paramName": "grant_type",
  "paramValue": "authorization_code", "valueType": "litr",
  "paramMandatory": "true" },
{ "flavorStep": "2Gra", "paramLocation": "para", "paramName": "client_id",
  "paramValue": "client_id", "valueType": "eval",
  "paramMandatory": "true" },
{ "flavorStep": "2Gra", "paramLocation": "para", "paramName": "redirect_uri",
  "paramValue": "redirect_uri", "valueType": "eval",
  "paramMandatory": "true" },
{ "flavorStep": "3Prc", "paramLocation": "head", "paramName": "Bearer",
  "paramValue": "access_token", "valueType": "sec",
  "paramMandatory": "true" },
{ "flavorStep": "4Ref", "paramLocation": "head", "paramName": "Authorization",
  "paramValue": "Basic Authentication", "valueType": "sec",
  "paramMandatory": "true" },
{ "flavorStep": "4Ref", "paramLocation": "head", "paramName": "Content-Type",
  "paramValue": "application/x-www-form-urlencoded", "valueType": "litr",
  "paramMandatory": "true" },
{ "flavorStep": "4Ref", "paramLocation": "para", "paramName": "grant_type",
  "paramValue": "refresh_token", "valueType": "litr",
  "paramMandatory": "true" },
{ "flavorStep": "4Ref", "paramLocation": "para", "paramName": "refresh_token",
  "paramValue": "refresh_token", "valueType": "sec",
  "paramMandatory": "true" },
{ "flavorStep": "5Rev", "paramLocation": "para", "paramName": "token",
  "paramValue": "access_token", "valueType": "sec",
  "paramMandatory": "true" },
] }

```

It is not necessary to create your own OAuth client flavor from scratch; SAP HANA provides some OAuth client flavors for a selection of OAuth server scenarios, which you can use without modification.

→ Tip

The example OAuth client flavors are located in the package
`sap.hana.xs.oauth.lib.providerconfig.providermodel`.

However, you **do** need to modify the OAuth client flavor artifact for the following scenarios:

- Modifications are required (or have already been made) to the API of an available OAuth server.
- A connection is required to a new OAuth server not covered by the scenarios included in the SAP HANA configuration templates.

parameters

Use the `parameters` keyword to define a list of parameter-values pairs, for example, `"paramLocation":"uri"` that support the specification defined in the OAuth client configuration file `<filename>.oauthclientconfig`.

flavorStep

Use the `flavorStep` keyword to specify a step in the procedure used by the client flavor, as illustrated in the following example

```
"flavorStep":"saml",
```

The following values are permitted:

- `IAut`
- `2Gra`
- `3Prc`
- `4Ref`
- `5Rev`
- `saml`

paramLocation

Use the `paramLocation` keyword to specify the location of the parameter defined, as shown in the following example:

```
"paramLocation":"uri",
```

The following values are permitted:

- `uri`
Universal resource indicator
- `head`
In the request header
- `para`
In the request body

paramName

Use the `paramName` keyword to specify the name of the parameter defined in “`paramLocation`”, as shown in the following example:

```
"paramName": "token",
```

The parameter name depends on the local setup of your client configuration.

paramValue

Use the `paramValue` keyword to specify a value for the parameter name specified in “`paramName`”.

```
"paramValue": "access_token",
```

The parameter name depends on the local setup of your client configuration.

valueType

Use the `valueType` keyword to specify the type of value expected by the parameter defined in “`paramValue`”.

```
"valueType": "sec",
```

The following values are permitted:

- `litr`
Literal value
- `eval`
The value is evaluated by the OAuth client runtime
- `sec`
The value is evaluated by the OAuth client runtime in a secure way

paramMandatory

Use the `paramMandatory` keyword to specify if a parameter is required or not.

```
"paramMandatory": "true",
```

The following values are permitted:

- `true`
Required
- `false`
Not Required

Related Information

[OAuth Client Configuration Syntax \[page 135\]](#)

[OAuth Application Configuration Syntax \[page 134\]](#)

[Tutorial: Create an OAuth Configuration Package \[page 130\]](#)

4.8 Maintaining Application Artifacts

The design-time building blocks of an SAP HANA applications are called development objects (or artifacts), and many have a mandatory file extension, for example, `.hdbtable` (design-time table definition), `.hdbview` (design-time SQL-view definition), or `.hdbrole` (design-time role definition).

Some of the development objects you encounter when creating an application, such as projects and packages, are designed to help you structure your application. Other objects such as schemas, table definitions, or analytical and attribute views, help you organize your data. Design-time definitions of procedures and server-side JavaScript code are the core objects of an SAP HANA application; these, too, have mandatory file extensions, for example, `.hdbprocedure` or `.xsjs`. Other types of development objects help you control the access to runtime objects.

When you activate an application artifact, the file extension (for example, `.hdbdd`, `.xsjs`, or `hdbprocedure`, ...) is used to determine which runtime plug-in to call during the activation process. The plug-in reads the repository artifact selected for activation (for example, a table definition, a complete CDS document, or server-side JavaScript code), interprets the object description in the file, and creates the appropriate runtime object in the designated catalog schema.

The file extensions associated with application artifacts are used in other contexts, too. For example, in SAP HANA studio, a context-sensitive menu is displayed when you click an artifact with the alternate mouse button; the options displayed in the menu is determined, amongst other things, according to the file extension.

Related Information

[Design-Time Application Artifacts \[page 144\]](#)

[Studio-Based SAP HANA Development Tools \[page 147\]](#)

4.8.1 Design-Time Application Artifacts

The design-time building blocks of your SAP HANA applications have a mandatory file extension, for example, `.hdbtable` (design-time table definition) or `.hdbview` (design-time SQL-view definition).

In SAP HANA, application artifacts have a mandatory file extension, which is used to determine the Repository tools required to parse the contents of the design-time artifact on activation. The following tables list the most commonly used building blocks of an SAP HANA application; the information provided shows any mandatory

file extension and, if appropriate, indicates where to find more information concerning the context in which the object can be used.

Design-time Application Building Blocks

File Extension	Object	Description
<code>.aflpmmml</code>	Procedure	A file used by the application function modeler to store details of a procedure defined using application functions in the Predictive Analysis Library * (PAL) or Business Function Library * (BFL). Using the AFM also generates a <code>.diagram</code> and a <code>.aflmodel</code> file.
<code>.analyticview</code>	Analytic view	A file containing a design-time definition of an analytic view; the view can be referenced in an OData service definition.
<code>.attributeview</code>	Attribute view	A file containing a design-time definition of an attribute view; the view can be referenced in an OData service definition.
<code>.calculationview</code>	Calculation view	A file containing a design-time definition of an calculation view; the view can be referenced in an OData service definition.
<code>.hdbdd</code>	CDS document	A file containing a design-time definition of a CDS-compliant data-persistence object (for example, an entity or a data type) using the Data Definition Language (DDL).
<code>.hdbprocedure</code>	Procedure	Replaces <code>.procedure</code> . A design-time definition of a database function for performing complex and data-intensive business logic that cannot be performed with standard SQL.
<code>.hdbrole</code>	Role	A file containing a design-time definition of an SAP HANA user role.
<code>.hdbscalarfunction</code>	Scalar user-defined function	A file containing a design-time definition of a scalar user-defined function (UDF), which is a custom function that can be called in the SELECT and WHERE clauses of an SQL statement.
<code>.hdbschema</code>	Schema	A design-time definition of a database schema, which organizes database objects into groups.
<code>.hdbsequence</code>	Sequence	A design-time definition of a database sequence, which is set of unique numbers, for example, for use as primary keys for a specific table.
<code>.hdbstructure</code>	Table type	A design-time definition of a database table type using the <code>.hdbtable</code> syntax. Used for defining reusable table types, for example, for parameters in procedures.
<code>.hdbsynonym</code>	Database synonym	A design-time definition of a database synonym using the <code>.hdbsynonym</code> syntax.
<code>.hdbtable</code>	Table	A design-time definition of a database table using the <code>.hdbtable</code> syntax.
<code>.hdbtablefunction</code>	Table user-defined function	A file containing a design-time definition of a table user-defined function (UDF), which is a custom function that can be called in the FROM-clause of an SQL statement.
<code>.hdbtextbundle</code>	Resource Bundle	A file for defining translatable UI texts for an application. Used in SAP UI5 applications.
<code>.hdbti</code>	Table Import definition	A table-import configuration that specifies which <code>.csv</code> file is imported into which table in the SAP HANA system.

File Extension	Object	Description
.hdbview	SQL View	A design-time definition of a database view, which is a virtual table based on an SQL query.
.procedure	Procedure	A design-time definition of a database function for performing complex and data-intensive business logic that cannot be performed with standard SQL.
.proceduretemplate	Procedure template	A design-time artifact containing a base script with predefined placeholders for objects such as tables, views and columns.
.project	Project	An Eclipse project for developing your application or part of an application. The .project file is a design-time artifact that is stored in the SAP HANA repository.
.searchruleset	Search Rule Set *	A file that defines a set of rules for use with fuzzy searches. The rules help decide what is a valid match in a search.
.xsaccess	Application Access File	An application-specific configuration file that defines permissions for a native SAP HANA application, for example, to manage access to the application and running objects in the package.
.xsapp	Application Descriptor	An application-specific file in a repository package that defines the root folder of a native SAP HANA application. All files in that package (and any subpackages) are available to be called via URL.
.xsappsite	Application Site	A file that defines an application site
.xshttpdest	HTTP destination configuration	A file that defines details for connections to a remote destination by HTTP (or HTTPS)
.xsjob	Scheduled XS job	A JSON-compliant file used to define recurring tasks that run in the background (independent of any HTTP request/response process); a scheduled job can either execute a JavaScript function or call a SQLScript procedure.
.xsjs	Server-Side JavaScript Code	A file containing JavaScript code that can run in SAP HANA Extended Application Services and be accessed via URL
.xsjslib	Server-Side JavaScript Library	A file containing JavaScript code that can run in SAP HANA Extended Application Services but cannot be accessed via URL. The code can be imported into an .xsjs code file.
.xsoauthappconfig	OAuth application configuration file	A file describing high-level details of an application that enables logon to a service running on a remote HTTP destination using OAuth
.xsoauthclientconfig	OAuth client configuration file	A file containing detailed information about a client application that uses OAuth as the authentication mechanism for logon to a remote HTTP destination
.xsoauthclientflavor	OAuth client flavor file	The corresponding OAuth flavors file for the OAuth client configuration
.xsodata	OData Descriptor	A design-time object that defines an OData service that exposes SAP HANA data from a specified end point.
.xsprivileges	Application Privilege	A file that defines a privilege that can be assigned to an SAP HANA Extended Application Services application, for example, the right to start or administer the application.

File Extension	Object	Description
.xssecurestore	Application secure store	The design-time file that creates an application-specific secure store; the store is used by the application to store data safely and securely in name-value form.
.xsqlcc	SQL Connection Configuration	A file that enables execution of SQL statements from inside server-side JavaScript code with credentials that are different to those of the requesting user
.xswidget	Widget	A file that defines a standalone SAP HANA application for the purpose of integration into an application site

⚠ Caution

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

Additional Application Building Blocks

Object	Description	File Extension
Package	A container in the repository for development objects.	Packages are represented by folders.
Attribute, Analytic and Calculation View	A view created with modeling tools and designed to model a business use case.	Created with the Systems view.
Decision Table	A table used to model business rules, for example, to manage data validation and quality.	
Analytic Privilege	A set of rules that allows users to seeing a subset of data in a table or view.	

4.8.2 Studio-Based SAP HANA Development Tools

The [SAP HANA Development](#) perspective in SAP HANA studio provides context-sensitive access to a variety of useful developer tools.

In SAP HANA studio's [SAP HANA Development](#) perspective, the view you are using determines what tools are available and the action that can be performed on the displayed objects. For example, in the [Project Explorer](#) view, the application developer can use the alternate mouse button to display a context-sensitive menu that provides access to Repository activation features, debugging configuration tools, and so on.

Project Explorer View

The following table lists a selection of the most frequently used tools and features that are available in the context-sensitive menu for artifacts in the [Project Explorer](#) view of the [SAP HANA Development](#) perspective.

SAP HANA XS Development Options

Menu Group	Menu Option	Description
Team	Commit	Copy the most recent version of the design-time artifact from the local file system to the Repository. Note that every local saved change is immediately committed to the user's corresponding inactive workspace in the SAP HANA Repository.
	Activate	Use the corresponding design-time definition in the Repository to generate a catalog object for the currently selected inactive artifact.
	Activate All...	Generate a catalog object based on the corresponding design-time definition in the Repository for all currently inactive artifacts in a particular workspace; you can choose to include/exclude individual artifacts from the displayed list. Inactive artifacts are local copies of Repository artifacts saved in your workspace.
	Check	Simulate an activate operation (including a syntax check)
	Regenerate	Force generation of a run-time catalog object without starting the corresponding design-time activation process
	Remove from Client	Undo a check-out operation without the risk of deleting content in the SAP HANA Repository
	Show in	Display details of the selected repository artifact in the Repositories , Synchronize , or History view.
	Synchronize	Synchronize changes made to local file version with the version of the file in the repository
Debug as...	Name/ID	Debug the code in the selected design-time artifact using an existing debug configuration.
	Debug configuration...	Debug the code in the selected design-time artifact using an new debug configuration that you define now, for example: XS JavaScript, SAP HANA stored procedure...
Run as...	HTML/XS Service/...	Test the selected Repository artifact in a Web browser directly from the Project Explorer view using the services provided by the currently connected SAP HANA server; the artifact's file extension is used to determine how to display the content.
	Run configuration...	Run the selected Repository artifact in a Web browser using a new runtime configuration, for example, for SAP HANA XS JavaScript artifacts, on a specific SAP HANA instance, and with defined user logon credentials.
Refresh	Refresh	Triggers a recursive checkout of Repository content, synchronizes differences between the Repository workspace and the local file system by fetching changes from the server

The following table lists **additional** tools and features that are available in the context-sensitive menu for artifacts in the [Repositories](#) view of the [SAP HANA Development](#) perspective.

Additional SAP HANA XS Development Options

Tool	Description
Inactive testing	Test repository objects that have not yet been activated, for example: XSJS, XSOData, XSJSlib, ...  Note The SAP HANA server must be running in <code>developer_mode</code>, and you must set a client-side cookie named <code>sapXsDevWorkspace</code> to the name of your Repository workspace.
Compare with active version	Display the differences between two versions of the same repository artifact or two different artifacts. You can select and compare multiple artifacts (CTRL and click the alternate mouse button). You can also compare an individual repository artifact with the version of the artifact that is currently active in the repository or a version from the artifact's revision-history list .
Get Where-Used List	Look for any references to the currently selected artifact and display the results in the Search view. The search includes both inactive artifacts (in your Repository workspace) and activated artifacts in the Repository. The Get Where-Used List option is available in both the Project Explorer and the Repositories view.
Share Project	Connect the local (client) project folders with SAP HANA repository and synchronizes the contents between client and server. This option is only available with an unshared project artifact.
Unshare Project	Cancel any synchronization between the local file system and the SAP HANA repository; the Unshare action does not delete any files, unless you specifically enable the delete option. The Unshare option is only available with an already shared project artifact.
Move	Moves selected SAP HANA artifacts or an entire package within or across projects in the same Repository workspace. All SAP HANA artifacts referencing the moved artifacts are updated too. You must manually activate all the moved and referencing artifacts. You can move the following SAP HANA artifacts: • Attribute View • Analytical View • Calculation View • Analytic Privilege
Paste Special	Clones one or more packages and all their artifacts and copies them to a target package. While copying, this feature detects if the target contains any other artifacts from a previous Paste Special operation. If any other cloned artifacts exist, you can update references to the existing cloned artifacts. You must manually activate the cloned artifacts. You can paste the following artifacts: • Attribute View • Analytical View • Calculation View • Analytic Privilege

Repositories View

The following table lists the most frequently used tools and features that are available in the context-sensitive menu for artifacts in the [Repositories](#) view of the [SAP HANA Development](#) perspective.

Note

The items displayed in the [Team](#) popup menu are context-sensitive; the options available in the menu depend on the type of repository object selected.

Tool/Feature	Description
Add package	This option is only available when you select another package.
Activate	Generate a catalog object based on the corresponding design-time definition in the Repository for the selected artifact
Activate All...	Generate a catalog object based on the corresponding design-time definition in the Repository for all currently inactive artifacts; you can choose to include/exclude individual artifacts from the displayed list. Inactive artifacts are local copies of Repository artifacts saved in your workspace.
Check	Simulate an activate operation (including a syntax check)
Check out	Copy package content from the Repository to the local workspace folder. Synchronize the repository with the local workspace (refresh)
Create Repository Workspace	Start the repository workspace wizard.
Delivery Unit management	(Package only): Start the lifecycle-management tools and display details of the corresponding delivery unit (DU) if available.
Edit package	(Package only): Display and edit details of the selected package, for example: the delivery unit the package is assigned to, the package type, and the person responsible for the package's creation and maintenance.
Get Where-Used List	Display any references to the currently selected artifact in the Search view. The search includes both inactive artifacts (in your Repository workspace) and activated artifacts in the SAP HANA Repository. The Get Where-Used List option is available in both the Project Explorer and the Repositories view.
Open	Open the selected file in the appropriate editor.
Product management	(Package only): Start the lifecycle-management tools and display details of the corresponding product, if available.
Remove from client	Remove the selected file(s) from the local file system; the repository version remains untouched.
Refresh	Synchronize the contents of the selected repository package with the local workspace (F5)
Reset to	Replace the selected file with the base version or the currently active version
Caution When you choose base version, you restore the original version of the object you are currently editing. When you choose active version, the version that you are currently editing becomes the new active version.	
Show in history view	Display the complete list of revisions available for the selected item; the details displayed include the version number, the date created, and the file owner. Right-click an entry in the history list to display further menu options, for example, to compare two versions of the file.

Tool/Feature	Description
Moves	Moves selected SAP HANA artifacts or an entire package within or across projects in the same Repository workspace. All SAP HANA artifacts referencing the moved artifacts are updated too. You must manually activate all the moved and referencing artifacts. You can move the following SAP HANA artifacts: • Attribute View • Analytical View • Calculation View • Analytic Privilege
Paste Special	Clones one or more packages and all their artifacts and copies them to a target package. While copying, this feature detects if the target contains any other artifacts from a previous Paste Special operation. If any other cloned artifacts exist, you can update references to the existing cloned artifacts. You must manually activate the cloned artifacts. You can paste the following artifacts: • Attribute View • Analytical View • Calculation View • Analytic Privilege

5 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 \[page 152\]](#)
- [SAP HANA XS Advanced Model \[page 153\]](#)

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.

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	<code>.hdbschema *</code>	<code>.hdbschema</code>
Synonym	<code>.hdbsynonym*</code>	<code>.hdbsynonym</code>
Table	<code>.hdbdd</code>	<code>.hdbtable</code>
Table Type	<code>.hdbdd</code>	<code>.hdbstructure</code>
View	<code>.hdbdd</code>	<code>.hdbview</code>
Association	<code>.hdbdd</code>	-
Sequence	<code>.hdbsequence*</code>	<code>.hdbsequence</code>
Structured Types	<code>.hdbdd</code>	-
Data import	<code>.hdbti</code>	<code>.hdbti</code>

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.

Note

In general, CDS works in XS advanced (HDI) in the same way that it does in the SAP HANA XS classic Repository. For XS advanced, however, there are some incompatible changes and additions, for example,

in the definition and use of name spaces, the use of annotations, the definition of entities (tables) and structure types. For more information, see *CDS Documents in XS Advanced* in the list of *Related Links* below.

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

1. Define the data model.

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

→ Tip

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

2. Set up the SAP HANA HDI deployment infrastructure.

This includes the following components:

- The HDI configuration
Map the design-time database artifact type (determined by the file extension, for example, `.hdbprocedure`, or `.hdbcds` in XS advanced) to the corresponding HDI build plug-in in the HDI configuration file (`.hdiconfig`).
- Run-time name space configuration (**optional**)
Define rules that determine how the run-time name space of the deployed database object is formed. For example, you can specify a base prefix for the run-time name space and, if desired, specify if the name of the folder containing the design-time artifact is reflected in the run-time name space that the deployed object uses.
Alternatively, you can specify the use of freestyle names, for example, names that do not adhere to any name-space rules.

3. Deploy the data model.

Use the design-time representations of your database artifacts to generate the corresponding active objects in the database catalog.

4. Consume the data model.

Reference the deployed database objects from your application, for example, using OData services bound to UI elements.

Related Information

[Creating Data Persistence Artifacts with CDS in XS Classic \[page 155\]](#)

[Creating Data Persistence Artifacts with CDS in XS Advanced](#)

[CDS Documents in XS Advanced](#)

5.1 Creating Data Persistence Artifacts with CDS in XS Classic

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

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

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

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

Note

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

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

Related Information

[Create a CDS Document \[page 159\]](#)

[Create an Entity in CDS \[page 183\]](#)

[Create a User-defined Structured Type in CDS \[page 206\]](#)

[Create an Association in CDS \[page 220\]](#)

[Create a View in CDS \[page 235\]](#)

[CDS Annotations \[page 172\]](#)

5.1.1 CDS Editors

The provides editing tools specially designed to help you create and modify CDS documents.

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

- [CDS Text Editor \[page 156\]](#)
View and edit DDL source code in a CDS document as text with the syntax elements highlighted for easier visual scanning.
Right-click a CDS document: 
- [CDS Graphical Editor \[page 157\]](#)
View a graphical representation of the contents of a CDS source file, with the option to edit the source code as text with the syntax elements highlighted for easier visual scanning.
Right-click a CDS document: 

CDS Text Editor

includes a dedicated text-based editor that you can use to define data-persistence objects using the CDS syntax. SAP HANA recognizes the `.hdbcds` file extension required for CDS object definitions and calls the appropriate repository plug-in.

The CDS editor provides the following features:

- Syntax highlights
The CDS DDL editor supports syntax highlighting, for example, for keywords and any assigned values. To customize the colors and fonts used in the CDS text editor, choose and select a theme and font size.
- Keyword completion
The editor displays a list of DDL suggestions that could be used to complete the keyword you start to enter. To change the settings, choose .
- Code validity
The CDS text editor provides syntax validation, which checks for parser errors as you type. Semantic errors are only shown when you build the XS advanced application module to which the CDS artifacts belong; the errors are shown in the console tab.
- Comments
Text that appears after a double forward slash (//) or between a forward slash and an asterisk (/ * . . * /) is interpreted as a comment and highlighted in the CDS editor (for example, `//this is a comment`).

CDS Graphical Editor

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

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

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

→ Tip

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

5.1.1.1 CDS Text Editor

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

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

The CDS editor provides the following features:

- Syntax highlights
The CDS DDL editor supports syntax highlighting, for example, for keywords and any assigned values (`@Schema: 'MySchema'`). You can customize the colors and fonts used in the Eclipse Preferences ([Window](#) > [Preferences](#) > [General](#) > [Appearance](#) > [Colors and Fonts](#) > [CDS DDL](#)).
- **Note**
The CDS DDL editor automatically inserts the mandatory keyword `namespace` into any new DDL source file that you create using the [New DDL Source File](#) dialog. The following values are assumed:
 - `namespace = <repository package name>`
- Keyword completion
The editor displays a list of DDL suggestions that could be used to complete the keyword you start to enter. You can insert any of the suggestions using the `[SPACE] + [TAB]` keys.
- Code validity
You can check the validity of the syntax in your DDL source file before activating the changes in the SAP HANA repository. Right-click the file containing the syntax to check and use the [Team](#) > [Check](#) option in the context menu.

Note

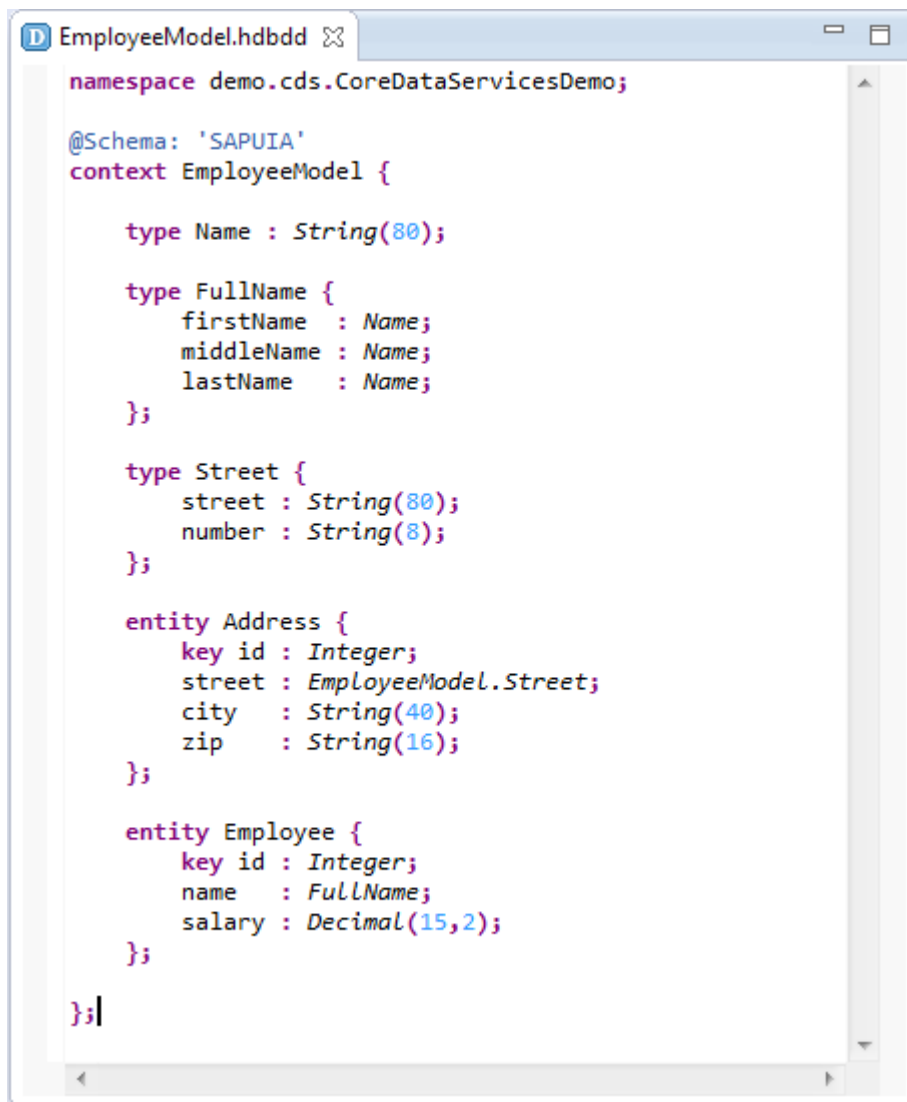
Activating a file automatically commits the file first.

- Comments

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

→ Tip

The [Project Explorer](#) view associates the .hdbdd file extension with the DDL icon. You can use this icon to determine which files contain CDS-compliant DDL code.



```
namespace demo.cds.CoreDataServicesDemo;

@Schema: 'SAPUIA'
context EmployeeModel {

    type Name : String(80);

    type FullName {
        firstName : Name;
        middleName : Name;
        lastName : Name;
    };

    type Street {
        street : String(80);
        number : String(8);
    };

    entity Address {
        key id : Integer;
        street : EmployeeModel.Street;
        city : String(40);
        zip : String(16);
    };

    entity Employee {
        key id : Integer;
        name : FullName;
        salary : Decimal(15,2);
    };

};
```

5.1.2 Create a CDS Document

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

Prerequisites

To complete this task successfully, note the following prerequisites:

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

Context

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

Procedure

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

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

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

→ Tip

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

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

- c. Choose *Finish* to save the changes and commit the new CDS document to the repository.

The file-creation wizard creates a basic CDS document with the following elements:

- **Namespace**
The name of the repository package in which you created the new CDS document, for example, `acme.com.hana.cds.data`
- **Top-level element**
The name of the top-level element in a CDS document must match the name of the CDS document itself; this is the name you enter when using the file-creation wizard to create the new CDS document, for example, `MyModel`, `MyContext`, or `MyEntity`. In this example, the top-level element is a context.

```
namespace acme.com.hana.cds.data;  
context MyModel {  
  
};
```

5. Define the details of the CDS artifacts.

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

Note

The following code examples are provided for illustration purposes only.

- a. Add a schema name.

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

Note

If the schema you specify does not exist, you cannot activate the new CDS document.

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

- b. Add structured types, if required.

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

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

```
};
```

- c. Add a new context, if required.

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

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

- d. Add new entities.

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

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

6. Save the CDS document.

Note

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

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

Note

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

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

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

Note

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

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

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

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

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

Note

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

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

Catalog Object	Location
Entities	► <SID> ► <i>Catalog</i> ► <MYSCHEMA> ► <i>Tables</i> ►

Catalog Object	Location
Types	► <SID> ► <i>Catalog</i> ► <MYSCHEMA> ► <i>Procedures</i> ► <i>Table Types</i> ►

- c. Open a data preview for the new object.
Right-click the new object and choose *Open Data Preview* in the pop-up menu.

Related Information

[CDS Namespaces \[page 168\]](#)

[CDS Naming Conventions \[page 167\]](#)

[CDS Contexts \[page 169\]](#)

[CDS Annotations \[page 172\]](#)

[CDS Comment Types \[page 181\]](#)

5.1.2.1 CDS Documents

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

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

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

If you use the file-creation wizard to create a new CDS document, the name space is inserted automatically; the inserted name space reflects the repository location you select to create the new CDS document.
- A schema definition
The schema you specify is used to store the catalog objects that are defined in the CDS document, for example: entities, structured types, and views. The objects are generated in the catalog when the CDS document is activated in the SAP HANA repository.
 - CDS artifact definitions
The objects that make up your persistence model, for example: contexts, entities, structured types, and views

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

Note

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

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

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

Catalog Location for CDS-generated Artifacts

CDS Artifact	Catalog Location
Entity	▶ <SID> ▶ Catalog ▶ <MYSCHEMA> ▶ Tables ▶
View	▶ <SID> ▶ Catalog ▶ <MYSCHEMA> ▶ Views ▶
Structured type	▶ <SID> ▶ Catalog ▶ <MYSCHEMA> ▶ Procedures ▶ Table Types ▶

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

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

Note

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

Sample Code

```
namespace acme.com.hana.cds.data;
using acme.com.hana.cds.data::MyModelB.MyContextB1 as ic;
@Schema: 'SAP_HANA_CDS'
context MyModel {
```

```

type BusinessKey : String(10);
type SString : String(40);
type <[...]>
context MasterData {
  @Catalog.tableType : #COLUMN
  Entity Addresses {
    key AddressId: BusinessKey;
    City: SString;
    PostalCode: BusinessKey;
    <[...]>
  };
  @Catalog.tableType : #COLUMN
  Entity BusinessPartner {
    key PartnerId: BusinessKey;
    PartnerRole: String(3);
    <[...]>
  };
};
context Sales {
  @Catalog.tableType : #COLUMN
  Entity Header {
    key SalesOrderId: BusinessKey;
    <[...]>
  };
  @Catalog.tableType : #COLUMN
  @MyAnnotation : 'foo'
  Entity Item {
    key SalesOrderId: BusinessKey;
    key SalesOrderItem: BusinessKey;
    <[...]>
  };
};
context Purchases {
  <[...]>
};
};

```

Related Information

[Create a CDS Document \[page 159\]](#)

[CDS Namespaces \[page 168\]](#)

[CDS Annotations \[page 172\]](#)

[External Artifacts in CDS \[page 165\]](#)

5.1.2.2 External Artifacts in CDS

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

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

→ Tip

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

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

```
//Filename = Pack1/Distributed/ContextB.hdbdd
namespace Pack1.Distributed;
using Pack1.Distributed::ContextA.T1;
using Pack1.Distributed::ContextA.ContextAI as ic;
using Pack1.Distributed::ContextA.ContextAI.T3 as ict3;
using Pack1.Distributed::ContextA.ContextAI.T3.a as a; // error, is not an
artifact
context ContextB {
  type T10 {
    a : T1; // Integer
    b : ic.T2; // String(20)
    c : ic.T3; // structured
    d : type of ic.T3.b; // String(88)
    e : ict3; // structured
    x : Pack1.Distributed::ContextA.T1; // error, direct reference not allowed
  };
  context ContextBI {
    type T1 : String(7); // hides the T1 coming from the first using declaration
    type T2 : T1; // String(7)
  };
};
```

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

ⓘ Note

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

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

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

Note

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

5.1.2.3 CDS Naming Conventions

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

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

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

Caution

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

Related Information

[Create a CDS Document \[page 159\]](#)

[CDS Documents \[page 163\]](#)

[CDS Namespaces \[page 168\]](#)

5.1.2.4 CDS Namespaces

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

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

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

Note

You can also use quotation marks ("") to wrap the names of CDS artifacts (entities, views) and elements (columns...).

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

Note

A CDS document cannot contain more than one namespace declaration.

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

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

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

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

- pack1.pack2;

`pack1` is spelled incorrectly; the namespace element requires a capital P to match the corresponding location in the Repository, for example, `Pack1`.

- `"Pack1.pack2"`;
You cannot quote the entire namespace path; only individual elements of the namespace path can be quoted, for example, `"Pack1".pack2`; or `Pack1."pack2"`;
- `Pack1.pack2.MyDataModel`;
The namespace declaration must not include the names of elements specified in the CDS document itself, for example, `MyDataModel`.
- `Jack.Jill`;
The package path `Jack.Jill`; does not exist in the Repository.

Related Information

[Create a CDS Document \[page 159\]](#)

[CDS Documents \[page 163\]](#)

5.1.2.5 CDS Contexts

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

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

Note

If you are using a CDS document to define a CDS context, the name of the CDS document must match the name of the context defined in the CDS document, for example, with the “context” keyword.

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

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

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

```
@Catalog.index : [ { name: 'MYINDEX2', unique: true, order: #DESC,
elementNames: ['AuthorNationality'] } ]
entity Author {
    key AuthorName      : String(100);
    AuthorNationality   : String(20);
    AuthorBirthday       : String(100);
    AuthorAddress        : String(100);
};
```

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

Note

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

Nested Contexts

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

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

Note

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

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

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

```

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

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

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

```

Name Resolution Rules

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

Note

An identifier may be used before its definition without the need for forward declarations.

```

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

```

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

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

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

Related Information

[CDS User-Defined Data Types \[page 208\]](#)

[Create a CDS Document \[page 159\]](#)

5.1.2.6 CDS Annotations

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

Example

```
namespace mycompany.myapp1;
@Schema : 'MYSHEMA'
context Books {
  @Catalog.tableType: #COLUMN
  @Catalog.index: [ { name : 'MYINDEX1', unique : true, order : #DESC,
elementNames : ['ISBN'] } ]
  entity BOOK {
    key Author : String(100);
    key BookTitle : String(100);
    ISBN : Integer not null;
    Publisher : String(100);
  };
  @Catalog.tableType : #COLUMN
  @nokey
  entity MyKeylessEntity
  {
```

```

    element1 : Integer;
    element2 : UTCTimestamp;
    @SearchIndex.text: { enabled: true }
    element3 : String(7);
};
@GenerateTableType : false
Type MyType1 {
    field1 : Integer;
    field2 : Integer;
    field3 : Integer;
};
};

```

Overview

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

- [@Catalog](#)
- [@nokey](#)
- [@Schema](#)
- [@GenerateTableType](#)
- [@SearchIndex](#)
- [@WithStructuredPrivilegeCheck](#)

@Catalog

The [@Catalog](#) annotation supports the following parameters, each of which is described in detail in a dedicated section below:

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

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

```

namespace com.acme.myapp1;
@Catalog.tableType : #COLUMN
@Schema: 'MYSCHEMA'
@Catalog.index:[ { name:'Index1', unique:true, order:#ASC, elementNames:['fint',
'futcshrt' ] } ]
entity MyEntity1 {
    key fint:Integer;
    fstr      :String(5000);
    fstr15    :String(51);
    fbin      :Binary(4000);
    fbin15    :Binary(51);
    fint32    :Integer64;
    fdec53    :Decimal(5,3);
}

```

```
fdecf    :DecimalFloat;
fbinf    :BinaryFloat;
futcshrt:UTCDateTime not null;
flstr    :LargeString;
flbin    :LargeBinary;
};
```

You can define the following values for the [@Catalog.index](#) annotation:

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

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

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

```
};
```

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

- **#COLUMN**
Create a column-based table. If the majority of table access is through a large number of tuples, with only a few selected attributes, use COLUMN-based storage for your table type.
- **#ROW**
Create a row-based table. If the majority of table access involves selecting a few records, with all attributes selected, use ROW-based storage for your table type.
- **#GLOBAL_TEMPORARY**
Set the scope of the created table. Data in a **global** temporary table is session-specific; only the owner session of the global temporary table is allowed to insert/read/truncate the data. A global temporary table exists for the duration of the session, and data from the global temporary table is automatically dropped when the session is terminated. A global temporary table can be dropped only when the table does not have any records in it.

Note

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

@nokey

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

Note

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

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


@Schema

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

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

Note

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

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

@GenerateTableType

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

Note

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

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

@SearchIndex

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

```
entity MyEntity100
{
```

```

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

```

→ Tip

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

@WithStructuredPrivilegeCheck

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

```

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

```

Related Information

[Create a CDS Document \[page 159\]](#)

[User-Defined CDS Annotations \[page 177\]](#)

[CDS Structured Type Definition \[page 211\]](#)

5.1.2.6.1 User-Defined CDS Annotations

In CDS, you can define your own custom annotations.

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

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

```

@Catalog.tableType : #ROW
@MyAnnotation : 'foo'

```

```
entity MyEntity {
    key Author : String(100);
    key BookTitle : String(100);
    ISBN : Integer not null;
    Publisher : String(100);
}
```

CDS supports the following types of user-defined annotations:

- Scalar annotations
- Structured annotations
- Annotation arrays

Scalar Annotations

The following example shows how to define a scalar annotation.

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

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

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

Structured Annotations

The following example shows how to define a structured annotation.

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

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

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

Array Annotations

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

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

5.1.2.6.2 CDS Annotation Usage Examples

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

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

User-defined CDS Annotations

CDS Annotation Type	Description
Scalar annotations [page 179]	For use with simple integer or string annotations and enumeration or Boolean types
Structured annotations [page 180]	For use where you need to create a simple annotation structure or nest an annotation in an anonymous annotation structure
Annotation arrays [page 181]	For use where you need to assign the same annotation several times to the same object.

Scalar Annotations

The following examples show how to use a scalar annotation:

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

📌 Note

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

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

```
@MyAnnotation_3 : #red
```

```

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

```

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

```

@MyAnnotation_4 : true
type MyType7 : Integer;
@MyAnnotation_4 // same as explicitly assigning the value “true”
type MyType8 : Integer;
@MyAnnotation_4 : false
type MyType9 : Integer;

```

Structured Annotations

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

```

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

```

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

Note

It is not necessary to assign a value for each element.

```

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

```

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

```

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

```

Array-like Annotations

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

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

Related Information

[CDS Annotations \[page 172\]](#)

[CDS Documents \[page 163\]](#)

[Create a CDS Document \[page 159\]](#)

5.1.2.7 CDS Comment Types

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

Example: Comment Formats in CDS Object Definitions

```
namespace com.acme.myapp1;

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

Overview

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

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

In-line Comments

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

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

End-of-Line Comment

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

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

Complete-Line Comment

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

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

Multi-Line Comments

The multi-line comment enables you to insert comment text that extends over multiple lines of a CDS document. To indicate the start of the multi-line comment, insert a forward-slash (/) followed by an asterisk (*)

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

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

Related Information

[Create a CDS Document \[page 159\]](#)

5.1.3 Create an Entity in CDS

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

Prerequisites

To complete this task successfully, note the following prerequisites:

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

Context

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

CDS entity creates the corresponding table in the specified schema. To create a CDS entity-definition file in the repository, perform the following steps:

Procedure

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

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

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

→ Tip

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

- c. Choose *Finish* to save the changes and commit the new entity-definition file in the repository.
5. Define the structure of the CDS entity.

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

ⓘ Note

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

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

6. Save the CDS entity-definition file.

ⓘ Note

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

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

Note

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

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

Note

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

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

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

9. Check that the new entity has been successfully created.
 CDS entities are created in the *Tables* folder in the catalog.
 - a. In the *SAP HANA Development* perspective, open the *Systems* view.
 - b. Navigate to the catalog location where you created the new entity.
▶ <SID> > Catalog > <MYSCHEMA> > Tables ▶
 - c. Open a data preview for the new entity `MyEntity`.
 Right-click the new entity `<package.path>::MyEntity` and choose *Open Data Preview* in the pop-up menu.

→ Tip

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

Related Information

[CDS Entities \[page 186\]](#)

[Entity Element Modifiers \[page 188\]](#)

[CDS Entity Syntax Options \[page 193\]](#)

5.1.3.1 CDS Entities

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

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

Note

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

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

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

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

Note

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

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

► <SID> ► Catalog ► <MYSCHEMA> ► Tables ►

Entity Element Definition

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

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

Note

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

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

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

Spatial Data

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

Related Information

[CDS Primitive Data Types \[page 216\]](#)

[Entity Element Modifiers \[page 188\]](#)

[CDS Entity Syntax Options \[page 193\]](#)

5.1.3.2 Entity Element Modifiers

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

Example

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

key

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

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

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

Note

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

null

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

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

⚠ Caution

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

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

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

default

```
default <literal_value>
```

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

📌 Note

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

```
entity WithDefaults  
{  
  key id : Integer;  
  field1 : Integer          default -42;  
  field2 : Integer64        default 9223372036854775807;  
  field3 : Decimal(5, 3)    default 12.345;
```

```

field4 : BinaryFloat    default 123.456e-1;
field5 : LocalDate      default date'2013-04-29';
field6 : LocalTime      default time'17:04:03';
field7 : UTCDateTime    default timestamp'2013-05-01 01:02:03';
field8 : UTCTimestamp   default timestamp'2013-05-01 01:02:03';
field9 : Binary(32)     default x'0102030405060708090a0b0c0d0e0[...]' ;
field10 : String(10)    default 'foo';
};

```

generated always as <expression>

```

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

```

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

⚠ Restriction

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

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

generated [always | by default] as identity

```

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

```

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

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

⚠ Restriction

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

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

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

⚠ Restriction

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

Column Migration Behavior

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

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

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

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

Related Information

[Create an Entity in CDS \[page 183\]](#)

[Create an Entity in CDS](#)

[CDS Entity Syntax Options \[page 193\]](#)

[SAP HANA SQL and System Views Reference \(CREATE TABLE\)](#)

5.1.3.3 CDS Entity Syntax Options

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

Example

Note

This example is not a working example; it is intended for illustration purposes only.

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

```

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

```

Note

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

Overview

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

- [Key definition \[page 194\]](#)
- [Index definition \[page 195\]](#)
- [Table type specification \[page 196\]](#)
- [Calculated Fields \[page 197\]](#)
- [Technical Configuration \[page 197\]](#)
- [Spatial data * \[page 199\]](#)
- [Series Data * \[page 200\]](#)

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

Note

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

Key Definition

```

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

```

```
};
```

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

Note

To define an entity without a key, use the `@nokey` annotation.

Index Definition

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

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

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

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

Table-Type Definition

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

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

Table-Type Syntax Options

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

Note

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

Calculated Fields

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

Sample Code

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

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

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

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

Note

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

technical configuration

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

- [Storage type](#)

- [Indexes](#)
- [Full text indexes](#)

📌 Note

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

Storage type

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

📄 Sample Code

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

⚠️ Restriction

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

Indexes

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

📄 Sample Code

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

⚠ Restriction

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

Full text indexes

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

Sample Code

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

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

⚠ Restriction

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

Spatial Types *

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

Sample Code

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

Series Data *

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

Note

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

Sample Code

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

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

⚠ Restriction

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

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

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

⚠ Caution

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

Related Information

[Create an Entity in CDS \[page 183\]](#)

[Create an Entity in CDS](#)

[CDS Annotations \[page 172\]](#)

[CDS Primitive Data Types \[page 216\]](#)

5.1.4 Migrate an Entity from `hdbtable` to CDS (`hdbdd`)

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

Prerequisites

To complete this task successfully, note the following prerequisites:

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

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

Context

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

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

Note

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

Procedure

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

Note

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

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

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

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

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

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

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

→ Tip

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

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

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

Related Information

[Migration Guidelines: hdbtable to CDS Entity \[page 203\]](#)

[SAP HANA to CDS Data-Type Mapping \[page 204\]](#)

5.1.4.1 Migration Guidelines: hdbtable to CDS Entity

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

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

Note

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

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

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

Related Information

[SAP HANA to CDS Data-Type Mapping \[page 204\]](#)

[SAP HANA SQL Data Type Conversion](#)

5.1.4.2 SAP HANA to CDS Data-Type Mapping

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

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

Mapping SAP HANA and CDS Types

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

Related Information

[Migrate an Entity from hdbtable to CDS \(hdbdd\) \[page 201\]](#)

[CDS Entity Syntax Options \[page 193\]](#)

[SAP HANA SQL Data Type Conversion](#)

5.1.5 Create a User-Defined Structured Type in CDS

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

Prerequisites

To complete this task successfully, note the following prerequisites:

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

Context

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

Procedure

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

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

- a. Right-click the folder where you want to save the definition file for the user-defined structured type and choose [► New ► Other... ► Database Development ► DDL Source File ►](#) in the context-sensitive popup menu.
- b. Enter the name of the user-defined structured type in the [File Name](#) box, for example, `MyStructuredType`.

→ Tip

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

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

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

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

Note

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

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

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

Note

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

7. Activate the changes in the repository.
 - a. Locate and right-click the new CDS definition file in the *Project Explorer* view.
 - b. In the context-sensitive pop-up menu, choose **Team > Activate**.If you cannot activate the new CDS artifact, check that the specified schema already exists and that there are no illegal characters in the name space, for example, the hyphen (-).

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

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

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

Note

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

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

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

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

Related Information

[CDS User-Defined Data Types \[page 208\]](#)

[CDS Structured Type Definition \[page 211\]](#)

[CDS Structured Types \[page 214\]](#)

5.1.5.1 CDS User-Defined Data Types

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

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

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

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

Note

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

```

Restrictions

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

⚠ Caution

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

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

```

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

```

```
};
```

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

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

Note

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

Related Information

[Create a User-Defined Structured Type in CDS](#)

[Create a User-Defined Structured Type in CDS \[page 206\]](#)

[CDS Structured Type Definition \[page 211\]](#)

[CDS Primitive Data Types \[page 216\]](#)

5.1.5.2 CDS Structured Type Definition

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

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

Note

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

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

► <SID> ► *Catalog* ► <MYSCHEMA> ► *Procedures* ► *Table Types* ►

Structured User-Defined Types

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

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

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

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

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

Note

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

Nested Structured Types

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

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

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

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

Generated Table Types

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

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

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

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

Note

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

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

→ Tip

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

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

Example

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

```

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

```

Related Information

[Create a User-Defined Structured Type in CDS \[page 206\]](#)

[Create a User-Defined Structured Type in CDS](#)

[CDS User-Defined Data Types \[page 208\]](#)

[CDS Structured Types \[page 214\]](#)

[CDS Primitive Data Types \[page 216\]](#)

5.1.5.3 CDS Structured Types

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

Example

```

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

```

```

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

```

type

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

```

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

```

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

→ Tip

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

```

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

```


type of

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

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

Related Information

[Create a User-Defined Structured Type in CDS \[page 206\]](#)

[Create a User-Defined Structured Type in CDS](#)

[CDS Primitive Data Types \[page 216\]](#)

[CDS User-Defined Data Types \[page 208\]](#)

[CDS Structured Type Definition \[page 211\]](#)

5.1.5.4 CDS Primitive Data Types

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

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

- [DDL data types \[page 216\]](#)
- [Native SAP HANA data types \[page 218\]](#)

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

Supported SAP HANA DDL Primitive Types

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

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

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

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

Note

* In CDS, the name of SAP HANA data types are prefixed with the word "hana", for example, hana.ALPHANUM, or hana.SMALLINT, or hana.TINYINT.

Supported Native SAP HANA Data Types

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

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

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

Note

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

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

Related Information

[Create a User-Defined Structures Type in CDS \[page 206\]](#)

[Create a CDS User-Defined Structure in XS Advanced](#)

5.1.6 Create an Association in CDS

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

Prerequisites

To complete this task successfully, note the following prerequisites:

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

Context

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

Procedure

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

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

- a. Right-click the folder where you want to save the entity-definition file and choose ► [New](#) ► [Other...](#)
► [Database Development](#) ► [DDL Source File](#) ► in the context-sensitive popup menu.
- b. Enter the name of the CDS document in the [File Name](#) box, for example, **MyMode11**.

→ Tip

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

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

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

5. Define the underlying CDS entities and structured types.

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

Note

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

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

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

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

Note

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

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

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

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

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

```

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

```

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

Note

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

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

Note

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

Related Information

[CDS Associations \[page 222\]](#)

[CDS Association Syntax Options \[page 228\]](#)

5.1.6.1 CDS Associations

Associations define relationships between entities.

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

Note

CDS supports both managed and unmanaged associations.

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

```

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

```

```

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

```

Cardinality in Associations

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

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

→ Tip

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

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

```

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

```



```

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

```

Target Entities in Associations

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

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

```

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

```

Filter Conditions and Prefix Notation

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

Sample Code

```

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

```

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

Sample Code

```

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

```

```

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

```

→ Tip

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

Foreign Keys in Associations

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

```

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

```

```
};  
};
```

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

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

Sample Code

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

Restrictions

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

⚠ Caution

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

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

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

Example: Complex (One-to-Many) Association

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

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

```

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

```

Related Information

[Create an Association in CDS \[page 220\]](#)

[Create a CDS Association in XS Advanced](#)

5.1.6.2 CDS Association Syntax Options

Associations define relationships between entities.

Example: Managed Associations

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

Example: Unmanaged Associations

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

Overview

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

Note

CDS supports both managed and unmanaged associations.

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

- **<Cardinality>** [\[page 229\]](#)
The relationship between the source and target in the association, for example, one-to-one, one-to-many, many-to-one
- **<targetEntity>** [\[page 231\]](#)
The target entity for the association
- **<forwardLink>** [\[page 231\]](#)
The foreign keys to use in a managed association, for example, element names in the target entity
- **<unmanagedJoin>** [\[page 233\]](#)
Unmanaged associations only; the **ON** condition specifies the elements of the source and target elements and entities to use in the association

Association Cardinality

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

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

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

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

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

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

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

```

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

```

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

Association Cardinality Syntax Examples

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

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

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

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

Sample Code

```

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

```

```

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

```

Association Target

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

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

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

```

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

```

Association Keys

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

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

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

```

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

```

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

```

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

```



```

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

```

Association Syntax Options

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

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

Sample Code

```

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

```

Unmanaged Associations

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

Note

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

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

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

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

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

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

Constants in Associations

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

Sample Code

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

Related Information

[Create a CDS Association in XS Advanced](#)

[CDS Associations \[page 222\]](#)

5.1.7 Create a View in CDS

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

Prerequisites

To complete this task successfully, note the following prerequisites:

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

Context

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

Note

The following code examples are provided for illustration purposes only.

Procedure

1. Start the SAP HANA studio.
2. Open the *SAP HANA Development* perspective.
3. Open the *Project Explorer* view.
4. Create the CDS-definition file which will contain the view you define in the following steps.
Browse to the folder in your project workspace where you want to create the new CDS-definition file and perform the following steps:
 - a. Right-click the folder where you want to save the view-definition file and choose ► *New* ► *Other...* ► *Database Development* ► *DDL Source File* in the context-sensitive pop-up menu.
 - b. Enter the name of the view-definition file in the *File Name* box, for example, `MyMode12`.

→ Tip

File extensions are important. If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file-creation wizard adds the required file extension automatically (for example, `MyModel12.hdbdd`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

- c. Choose *Finish* to save the changes and commit the new CDS definition file in the repository.
5. Define the underlying CDS entities and structured types.

If the new entity-definition file is not automatically displayed by the file-creation wizard, in the *Project Explorer* view double-click the entity-definition file you created in the previous step, for example, `MyModel12.hdbdd`, and add the code for the entity definitions and structured types to the file.

```
namespace com.acme.myapp1;
@Schema : 'MYSHEMA'
context MyModel12 {
  type StreetAddress {
    name  : String(80);
    number : Integer;
  };
  type CountryAddress {
    name : String(80);
    code : String(3);
  };
  @Catalog.tableType : #COLUMN
  entity Address {
    key id : Integer;
    street : StreetAddress;
    zipCode : Integer;
    city : String(80);
    country : CountryAddress;
    type : String(10); // home, office
  };
};
```

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

In the same entity-definition file you edited in the previous step, for example, `MyModel12.hdbdd`, add the code for the view `AddressView` below the entity `Address` in the CDS document.

ⓘ Note

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

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

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

ⓘ Note

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

8. Activate the changes in the repository.

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

📘 Note

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

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

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

📘 Note

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

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

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

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

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

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

► `<SID>` ► *Catalog* ► `<MYSCHEMA>` ► *Views* ►

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

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

Related Information

[CDS Views \[page 238\]](#)

[CDS View Syntax Options \[page 240\]](#)

[Spatial Types and Functions \[page 255\]](#)

5.1.7.1 CDS Views

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

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

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

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

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

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

► <SID> ► Catalog ► <MYSCHEMA> ► Views ►

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

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

→ Tip

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

Type Definition

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

Sample Code

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

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

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

Note

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

Related Information

[Create a CDS View in XS Advanced](#)
[CDS View Syntax Options \[page 240\]](#)
[CDS Associations \[page 222\]](#)

5.1.7.2 CDS View Syntax Options

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

Example

Note

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

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

```

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

```

Expressions and Functions

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

```

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

```

Note

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

Aggregates

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

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

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

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

Note

Expressions are not allowed in the `GROUP BY` clause.

Constants in Views

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

Sample Code

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

```

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

```

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

Sample Code

```

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

```

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

Constant Declaration and Result

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

Constant Expression	Result	Comments
: c	Error	Error: there is no alias for "c".

SELECT

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

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

Subqueries

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

⚠ Restriction

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

↔ Code Syntax

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

📌 Note

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

WHERE

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

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

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

FROM

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

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

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

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

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

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

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

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

GROUP BY

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

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

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

HAVING

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

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

ORDER BY

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

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

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

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

LIMIT/OFFSET

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

```
VIEW MyViewV AS SELECT FROM E  
{ a, b, c }
```

```
order by a limit 10 offset 30;
```

CASE

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

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

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

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

UNION

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

Note

View `MyView5` has elements `a`, `b`, and `c`.

```
entity E1 {
  key a : Integer;
  b : String(20);
  c : LocalDate;
};
entity E2 {
  key x : String(20);
  y : LocalDate;
  z : Integer;
};
VIEW MyView5 AS
```



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

JOIN

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

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

The following example shows a simple join.

Sample Code

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

TOP

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

Sample Code

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

⚠ Restriction

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

SELECT DISTINCT

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

Sample Code

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

With Parameters

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

Restriction

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

Sample Code

Parameters in a CDS View

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

Note

Keywords are case insensitive.

Parameters in View Queries

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

→ Tip

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

↗ Sample Code

Using Parameters in a View Query

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

Invoking a View with Parameters

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

⚠ Restriction

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

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

↗ Sample Code

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

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

```

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

```

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

Sample Code

Query a Parameterized CDS View (with Association)

```

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

```

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

Sample Code

Follow an Association in the FROM Clause

```

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

```

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

Sample Code

Select from a Parameterized View with JOIN Expression

```

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

```

Annotations in Parameter Definitions

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

→ Tip

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

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

Sample Code

Annotation Assignments to Parameter Definitions in CDS Views

```
annotation remark: String(100);

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

Associations, Filters, and Prefixes

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

Restriction

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

Sample Code

Associations as View Elements

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

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

Filter Conditions

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

Sample Code

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

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

Sample Code

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

→ Tip

Filter conditions and prefixes can be nested.

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

Sample Code

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

Prefix Notation

The prefix notation can also be used without filters. The following example shows how to get a list of all customers with details of their sales orders. In this example, all uses of the association **orders** are combined so that there is only one **JOIN** to the table **SalesOrder**. Similarly, both uses of the association **items** are combined, and there is only one **JOIN** to the table **Item**.

Sample Code

```
view C3 as select from Customer {
```

```

name,
orders.id          as orderId,
orders.date        as orderDate,
orders.items.descr as itemDescr,
orders.items.price as itemPrice
};

```

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

Sample Code

```

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

```

Type Definition

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

⚠ Restriction

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

Sample Code

```

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

```

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

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

Note

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

Spatial Functions

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

Sample Code

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

Caution

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

Related Information

[Create a View in CDS \[page 235\]](#)

[Create a View in CDS](#)

[Spatial Types and Functions \[page 255\]](#)

5.1.7.3 Spatial Types and Functions

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

Spatial data is data that describes the position, shape, and orientation of objects in a defined space; the data is represented as two-dimensional geometries in the form of points, line strings, and polygons. The following

examples shows how to use the spatial function `ST_Distance` in a CDS view. The underlying spatial data used in the view is defined in a CDS entity using the type `ST_POINT`.

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

Sample Code

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

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

⚠ Caution

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

Related Information

[Create a View in CDS \[page 235\]](#)

[Create a View in CDS](#)

[CDS View Syntax Options \[page 240\]](#)

[CDS Entity Syntax Options \[page 193\]](#)

[CDS Primitive Data Types \[page 216\]](#)

5.1.8 Modifications to CDS Artifacts

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

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

- [Removing an artifact from a CDS document \[page 257\]](#)
- [Changing the definition of an artifact in a CDS document \[page 257\]](#)
- [Modifying a catalog object generated by CDS \[page 259\]](#)
- [Transporting a DU that contains modified CDS documents \[page 260\]](#)

Removing an Artifact from a CDS Document

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

Note

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

Changing the Definition of an Artifact in a CDS Document

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

- Views
Views in the SAP HANA catalog are dropped and recreated according to the new design-time specification for the artifact in the CDS document.
- Element types
Changing the type of an element according to the implicit conversion rules described in the SAP HANA SQL documentation (*SAP HANA SQL Data Type Conversion*). Note: For some type conversions the activation will succeed only if the data in the corresponding DB table is valid for the target type (for example the conversion of String to Integer will succeed only if the corresponding DB table column contains only numbers that match the Integer type)
- Element modifier: `NULL/NOT NULL`
Adding, removing or changing element modifiers “`NULL`” and “`not null`” to make an element nullable or not nullable respectively can lead to problems when activating the resulting artifact; the activation will

succeed only if the data in the database table corresponding to the CDS entity matches the new modifier. For example, you cannot make an element not nullable, if in the corresponding column in the database table some null values exist for which there is no default value defined.

- **Element modifier: Default Value**
If the default value modifier is removed, this has no effect on the existing data in the corresponding database table, and no default value will be used for any subsequently inserted record. If the default value is modified or newly added, the change will be applicable to all subsequent inserts in the corresponding database table. In addition, if the element is not nullable (irrespective of whether it was defined previously as such or within the same activation), the existing null values in the corresponding table will be replaced with the new default value.
- **Element modifier: Primary Key**
You can add an element to (or remove it from) the primary key by adding or removing the “key” modifier.

Note

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

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

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

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

Note

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

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

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

```

{
  entity Entity2
  {
    key x : Integer;
      y : String(10);
      z : LocalDate;
  };
};

```

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

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

```

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

```

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

Modifying a Catalog Object Generated from CDS

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

Transporting a DU that Contains Modified CDS Documents

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

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

Related Information

[SAP HANA to CDS Data-Type Mapping \[page 204\]](#)

[SAP HANA SQL Data Type Conversion](#)

5.1.9 Tutorial: Get Started with CDS

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

Prerequisites

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

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

Context

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

→ Tip

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

Procedure

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

Browse to the folder in your project workspace where you want to create the new CDS document (for example, in a project you have already created and shared) and perform the following tasks:

- a. Right-click the folder where you want to create the CDS document and choose ► [New](#) ► [DDL Source File](#) in the context-sensitive popup menu.

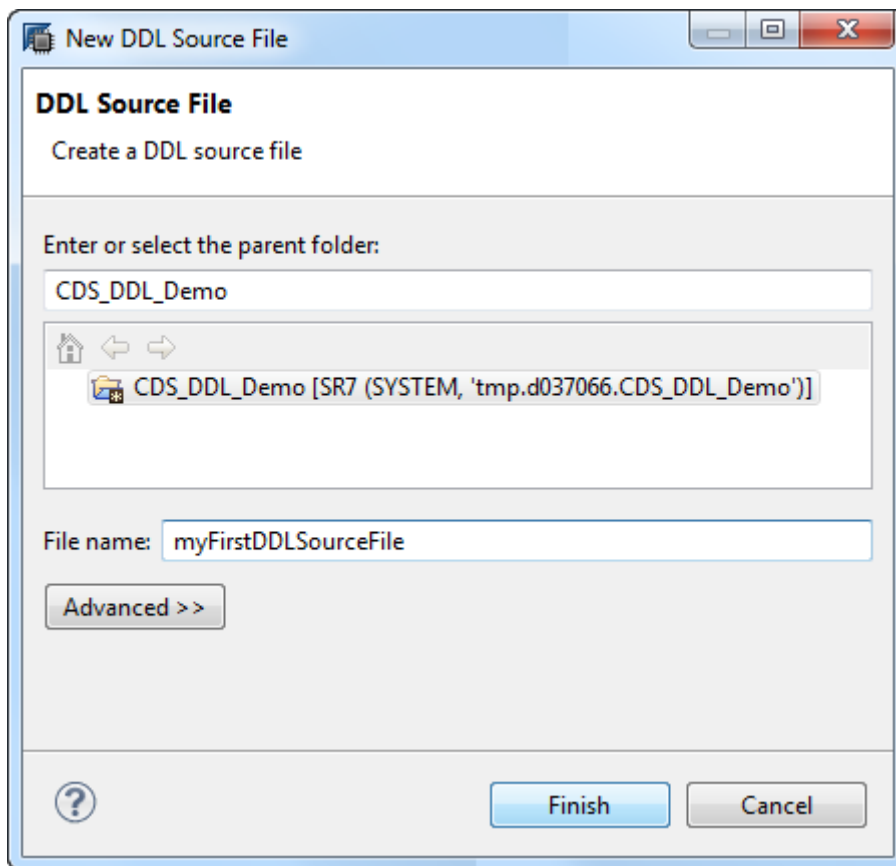
📘 Note

This menu option is only available from shared projects; projects that are linked to the SAP HANA repository.

- b. Enter the name of the entity in the [File Name](#) box, for example, `MyFirstCDSSourceFile`.

📘 Note

The file extension `.hdbdd` is added automatically to the new DDL file name. The repository uses the file extension to make assumptions about the contents of repository artifacts, for example, that `.hdbdd` files contain DDL statements.



- c. Choose *Finish* to save the new empty CDS document.

Note

If you are using a CDS document to define a single CDS-compliant entity, the name of the CDS document must match the name of the entity defined in the CDS document, for example, with the *entity* keyword. In the example in this tutorial, you would save the entity definition "BOOK" in the CDS document `BOOK.hdbdd`.

5. Define the table entity.

To edit the CDS document, in the *Project Explorer* view double-click the file you created in the previous step, for example, `BOOK.hdbdd`, and add the entity-definition code:

Note

The CDS DDL editor automatically inserts the mandatory keywords *namespace* and *context* into any new DDL source file that you create using the *New DDL Source File* dialog. The following values are assumed:

- *namespace* = `<Current Project Name>`
- *context* = `<New DDL File Name>`

The name space declared in a CDS document must match the repository package in which the object the document defines is located.

In this example, the CDS document `BOOK.hdbdd` that defines the CDS entity “BOOK” must reside in the package `mycompany.myapp1`.

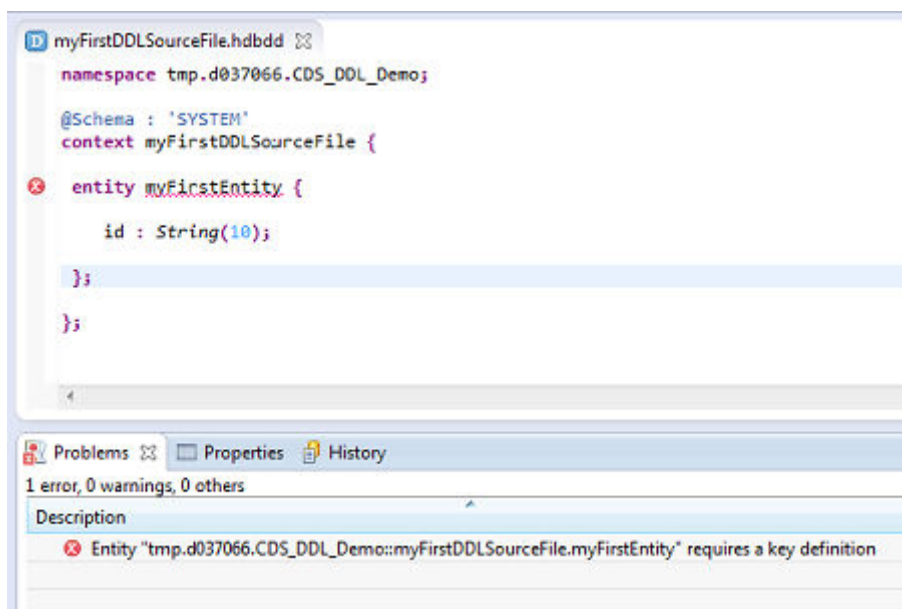
```
namespace mycompany.myapp1;
@Schema : 'MYSCHEMA'
@Catalog.tableType: #COLUMN
@Catalog.index: [ { name : 'MYINDEX1', unique : true, order : #DESC,
elementNames : ['ISBN'] } ]
entity BOOK {
    key Author      : String(100);
    key BookTitle   : String(100);
    ISBN           : Integer not null;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

6. Save the CDS document `BOOK.hdbdd`.

Note

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

7. Activate the new CDS document in the repository.
 - a. In the *Project Explorer* view, locate the newly created artifact `BOOK.hdbdd`.
 - b. Right-click `BOOK.hdbdd` and choose **Team > Activate** in the context-sensitive popup menu. The CDS/DDl editor checks the syntax of the source file code, highlights the lines where an error occurs, and provides details of the error in the *Problems* view.



The activation creates the following table in the schema `MYSCHEMA`, both of which are visible using the SAP HANA studio:

```
"MYSCHEMA"."mycompany.myapp1::BOOK"
```


The following public synonym is also created, which can be referenced using the standard SQL query notation:

```
"mycompany.myapp1::BOOK"
```

8. Add an entry to the BOOK entity using SQL.

```
INSERT INTO "mycompany.myapp1::BOOK" VALUES ( 'Shakespeare', 'Hamlet',  
'1234567', 'Books Incorporated' );
```

9. Save and activate the modifications to the entity.
10. Check the new entry by running a simply SQL query.

```
SELECT COUNT(*) FROM "mycompany.myapp1::BOOK" WHERE Author = 'Shakespeare'
```

Related Information

[Create a Schema \[page 278\]](#)

5.1.10 Import Data with CDS Table-Import

The table-import function is a data-provisioning tool that enables you to import data from comma-separated values (CSV) files into SAP HANA tables.

Prerequisites

Before you start this task, make sure that the following prerequisites are met:

- An SAP HANA database instance is available.
- The SAP HANA database client is installed and configured.
- You have a database user account set up with the roles containing sufficient privileges to perform actions in the repository, for example, add packages, add objects, and so on.
- The SAP HANA studio is installed and connected to the SAP HANA repository.
- You have a development environment including a repository workspace, a package structure for your application, and a shared project to enable you to synchronize changes to the project files in the local file system with the repository.

Note

The names used in the following task are for illustration purposes only; where necessary, replace the names of schema, tables, files, and so on shown in the following examples with your own names.

Context

In this tutorial, you import data from a CSV file into a table generated from a design-time definition that uses the `.hdbdd` syntax, which complies with the Core Data Services (CDS) specifications.

→ Tip

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

Procedure

1. Create a root package for your table-import application.
In SAP HANA studio, open the *SAP HANA Development* perspective and perform the following steps:
 - a. In the package hierarchy displayed in the *Systems* view, right-click the package where you want to create the new package for your table-import configuration and choose .
 - 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`

📘 Note

Naming conventions exist for package names, for example, a package name must not start with either a dot (.) or a hyphen (-) and cannot contain two or more consecutive dots (..). In addition, the name must not exceed 190 characters.

- a. Choose *OK* to create the new package.
2. Create a set of table-import files.

For the purposes of this tutorial, the following files must all be created in the same package, for example, a package called **TiTest**. However, the table-import feature also allows you to use files distributed in different packages

→ Tip

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

- The table-import configuration file, for example, `TiConfiguration.hdbti`
Specifies the source file containing the data values to import and the target table in SAP HANA into which the data must be inserted
- A CSV file, for example, `myTiData.csv`
Contains the data to be imported into the SAP HANA table during the table-import operation; values in the `.csv` file can be separated either by a comma (,) or a semi-colon (;).
- A target table.
The target table can be either a runtime table in the catalog or a table definition, for example, a table defined using the `.hdbtable` syntax (`TiTable.hdbtable`) or the CDS-compliant `.hdbdd` syntax (`TiTable.hdbdd`).

Note

In this tutorial, the target table for the table-import operation is `TiTable.hdbdd`, a design-time table defined using the CDS-compliant `.hdbdd` syntax.

- The schema named `AMT`
Specifies the name of the schema in which the target import table resides

When all the necessary files are available, you can import data from a source file, such as a CSV file, into the desired target table.

3. If it does not already exist, create a schema named `AMT` in the catalog; the `AMT` schema is where the target table for the table-import operation resides.
4. Create or open the table-definition file for the target import table (`inhabitants.hdbdd`) and enter the following lines of text; this example uses the `.hdbdd` syntax.

Note

In the CDS-compliant `.hdbdd` syntax, the `namespace` keyword denotes the path to the package containing the table-definition file.

```
namespace mycompany.tests.TiTest;

@Schema : 'AMT'
@Catalog.tableType : #COLUMN
entity inhabitants {
    key ID : Integer;
    surname : String(30);
    name : String(30);
    city : String(30);
};
```

5. Open the CSV file containing the data to import, for example, `inhabitants.csv` in a text editor and enter the values shown in the following example.

```
0,Annan,Kwesi,Accra
1,Essuman,Wiredu,Tema
2,Tetteh,Kwame,Kumasi
3,Nterful,Akye,Tarkwa
4,Acheampong,Kojo,Tamale
5,Assamoah,Adjoa,Takoradi
6,Mensah,Afua,Cape Coast
```

Note

You can import data from multiple `.csv` files in a single, table-import operation. However, each `.csv` file must be specified in a separate code block (`{table= ...}`) in the table-import configuration file.

6. Create or open the table-import configuration file (`inhabitants.hdbti`) and enter the following lines of text.

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

7. Deploy the table import.
 - a. Select the package that you created in the first step, for example, `mycompany.tests.TiTest`.
 - b. Click the alternate mouse button and choose [Commit](#).
 - c. Click the alternate mouse button and choose [Activate](#).

This activates all the repository objects. The data specified in the CSV file `inhabitants.csv` is imported into the SAP HANA table `inhabitants` using the data-import configuration defined in the `inhabitants.hdbti` table-import configuration file.

8. Check the contents of the runtime table `inhabitants` in the catalog.

To ensure that the import operation completed as expected, use the SAP HANA studio to view the contents of the runtime table `inhabitants` in the catalog. You need to confirm that the correct data was imported into the correct columns.

- a. In the [SAP HANA Development](#) perspective, open the [Systems](#) view.
- b. Navigate to the catalog location where the `inhabitants` object resides, for example:

► <SID> ► [Catalog](#) ► [AMT](#) ► [Tables](#) ►

- c. Open a data preview for the updated object.

Right-click the updated object and choose [Open Data Preview](#) in the context-sensitive menu.

5.1.10.1 Data Provisioning Using Table Import

You can import data from comma-separated values (CSV) into the SAP HANA tables using the SAP HANA Extended Application Services (SAP HANA XS) table-import feature.

In SAP HANA XS, you create a table-import scenario by setting up an table-import configuration file and one or more comma-separated value (CSV) files containing the content you want to import into the specified SAP HANA table. The import-configuration file links the import operation to one or more target tables. The table definition (for example, in the form of a `.hdbdd` or `.hdbtable` file) can either be created separately or be included in the table-import scenario itself.

To use the SAP HANA XS table-import feature to import data into an SAP HANA table, you need to understand the following table-import concepts:

- Table-import configuration
You define the table-import model in a configuration file that specifies the data fields to import and the target tables for each data field.

Note

The table-import file must have the `.hdbti` extension, for example, `myTableImport.hdbti`.

CSV Data File Constraints

The following constraints apply to the CSV file used as a source for the table-import feature in SAP HANA XS:

- The number of table columns must match the number of CSV columns.
- There must not be any incompatibilities between the data types of the table columns and the data types of the CSV columns.

- Overlapping data in data files is not supported.
- The target table of the import must not be modified (or appended to) outside of the data-import operation. If the table is used for storage of application data, this data may be lost during any operation to re-import or update the data.

Related Information

[Table-Import Configuration \[page 268\]](#)

[Table-Import Configuration-File Syntax \[page 270\]](#)

5.1.10.2 Table-Import Configuration

You can define the elements of a table-import operation in a design-time file; the configuration includes information about source data and the target table in SAP HANA.

SAP HANA Extended Application Services (SAP HANA XS) enables you to perform data-provisioning operations that you define in a design-time configuration file. The configuration file is transportable, which means you can transfer the data-provisioning between SAP HANA systems quickly and easily.

The table-import configuration enables you to specify how data from a comma-separated-value (.csv) file is imported into a target table in SAP HANA. The configuration specifies the source file containing the data values to import and the target table in SAP HANA into which the data must be inserted. As further options, you can specify which field delimiter to use when interpreting data in the source .csv file and if keys must be used to determine which columns in the target table to insert the imported data into.

ⓘ 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";`

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
```

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.

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 into tables together in a single package and set the appropriate (restrictive) application-access permissions for that package, for example, with a dedicated `.xsaccess` file.

Related Information

[Table-Import Configuration-File Syntax \[page 270\]](#)

5.1.10.3 Table-Import Configuration-File Syntax

The design-time configuration file used to define a table-import operation requires the use of a specific syntax. The syntax comprises a series of `keyword=value` pairs.

If you use the table-import configuration syntax to define the details of the table-import operation, you can use the keywords illustrated in the following code example. The resulting design-time file must have the `.hdbti` 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_DS0", "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.

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";
```

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";
```

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, `hdtable = "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`.

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 = ";";
```

Note

By default, the table-import operation assumes that data values in the `.csv` source file are separated by a comma (,). If no delimiter field is specified in the `.hdbti` table-import configuration file, the default setting is assumed.

delimEnclosing

Use the `delimEnclosing` keyword in the table-import configuration to specify a single character that indicates both the start and end of a set of characters to be interpreted as a single value in the `.csv` file, for example "This is all one, single value". This feature enables you to include in data values in a `.CSV` file even the character defined as the field delimiter (in `delimField`), for example, a comma (,) or a semi-colon (;).

Tip

If the value used to separate the data fields in your `.csv` file (for example, the comma (,)) is also used inside the data values themselves ("This, is, a, value"), you **must** declare and use a delimiter enclosing character and use it to enclose all data values to be imported.

The following example shows how to use the `delimEnclosing` keyword to specify the quote (") as the delimiting character that indicates both the start and the end of a value in the `.csv` file. Everything enclosed between the `delimEnclosing` characters (in this example, "") is interpreted by the import process as one, single value.

```
delimEnclosing="\\";
```

Note

Since the `hdbti` syntax requires us to use the quotes (") to specify the delimiting character, and the delimiting character in this example is, itself, also a quote ("), we need to use the backslash character (\) to escape the second quote (").

In the following example of values in a `.csv` file, we assume that `delimEnclosing="\\"`, and `delimField=","`. This means that imported values in the `.csv` file are enclosed in the quote character ("value") and multiple values are separated by the comma ("value1", "value 2"). Any commas **inside** the quotes are interpreted as a comma and not as a field delimiter.

```
"Value 1, has a comma","Value 2 has, two, commas","Value3"
```

You can use other characters as the enclosing delimiter, too, for example, the hash (#). In the following example, we assume that `delimEnclosing="#"` and `delimField=";"`. Any semi-colons included **inside** the hash characters are interpreted as a semi-colon and not as a field delimiter.

```
#Value 1; has a semi-colon#;#Value 2 has; two; semi-colons#;#Value3#
```

distinguishEmptyFromNull

Use the `distinguishEmptyFromNull` keyword in combination with `delimEnclosing` to ensure that the table-import process correctly interprets any **empty** value in the .CSV file, which is enclosed with the value defined in the `delimEnclosing` keyword, for example, as an empty space. This ensures that an empty space is imported “as is” into the target table. If the empty space is incorrectly interpreted, it is imported as **NULL**.

```
distinguishEmptyFromNull = true;
```

Note

The default setting for `distinguishEmptyFromNull` is **false**.

If `distinguishEmptyFromNull=false` is used in combination with `delimEnclosing`, then an empty value in the .CSV (with or without quotes "") is interpreted as **NULL**.

```
"Value1",, "", Value2
```

The table-import process would add the values shown in the example .csv above into the target table as follows:

Value1		NULL		NULL		Value2

keys

Use the `keys` keyword in the table-import configuration to specify the key range to be considered when importing the data from the .csv source file into the target table.

```
keys = [ "GROUP_TYPE" : "BW_CUBE", "GROUP_TYPE" : "BW_DSO", "GROUP_TYPE" :  
"BW_PSA" ];
```

In the example above, all the lines in the .csv source file where the `GROUP_TYPE` column value matches one of the given values (`BW_CUBE`, `BW_DSO`, or `BW_PSA`) are imported into the target table specified in the table-import configuration.

```
;;;BW_CUBE;;40000000;3;40000000;slave  
;;;BW_DSO;;40000000;3;40000000;slave  
;;;BW_PSA;;2000000000;1;2000000000;slave
```

In the following example, the `GROUP_TYPE` column is specified as `empty("")`.

```
keys = [ "GROUP_TYPE" : "" ];
```

All the lines in the .csv source file where the GROUP_TYPE column is empty are imported into the target table specified in the table-import configuration.

```
;;;;;40000000;2;40000000;all
```

5.1.10.4 Table-Import Configuration Error Messages

During the course of the activation of the table-import configuration and the table-import operation itself, SAP HANA checks for errors and displays the following information in a brief message.

Table-Import Error Messages

Message Number	Message Text	Message Reason
40200	Invalid combination of table declarations found, you may only use [cdstable hdbtable table]	The table keyword is specified in a table-import configuration that references a table defined using the .hdbtable (or .hdbdd) syntax. The hdbtable keyword is specified in a table-import configuration that references a table defined using another table-definition syntax, for example, the .hdbdd syntax. The cdstable keyword is specified in a table-import configuration that references a table defined using another table-definition syntax, for example, the .hdbtable syntax.
40201	If you import into a catalog table, please provide schema	You specified a target table with the table keyword but did not specify a schema with the schema keyword.
40202	Schema could not be resolved	The schema specified with the schema keyword does not exist or could not be found (wrong name). The public synonym for an .hdbtable or .hdbdd (CDS) table definition cannot be resolved to a catalog table.

Message Number	Message Text	Message Reason
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 (<code>.hdbti</code>), for example, an inconsistent database status.
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 (<code>.hdbti</code>) contains one or more syntax errors.
40211	Table import constraint checks failed	The same key is specified in multiple table-import configurations (<code>.hdbti</code> 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 <code>.CSV</code> source file) or An (unexpected) error occurred on the SQL level.
40213	CSV table column count mismatch	Either the number of columns in the <code>.CSV</code> record is higher than the number of columns in the table, or The number of columns in the <code>.CSV</code> record is higher than the number of columns in its header.

Message Number	Message Text	Message Reason
40214	Column type mismatch	The <code>.CSV</code> 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 <code>.CSV</code> 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.

5.2 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.

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 <code>.hdbschema</code> , for example, <code>MYSCHEMA.hdbschema</code> .
Create a synonym	Define a design-time synonym and maintain the synonym definition in the repository; the transportable synonym has the file extension <code>.hdbsynonym</code> , for example, <code>MySynonym.hdbsynonym</code> .

Task	Description
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.

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

[Create a Schema \[page 278\]](#)

[Create a Table \[page 282\]](#)

[Create an SQL View \[page 302\]](#)

[Create a Synonym \[page 308\]](#)

5.2.1 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.

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, `MYSHEMA.hdbschema`, and add the schema-definition code to the file:

Note

The following code example is provided for illustration purposes only.

```
schema_name="MYSHEMA" ;
```

6. Save the schema file.

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

[Schema \[page 280\]](#)

5.2.1.1 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**.

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.

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.

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](#)

5.2.2 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 `MYSHEMA.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:

ⓘ 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.

ⓘ 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

[Tables \[page 284\]](#)

[Table Configuration Syntax \[page 285\]](#)

[Create a Schema \[page 278\]](#)

5.2.2.1 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.

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.

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 can contain up to 255 characters.

Related Information

[Table Configuration Syntax \[page 285\]](#)

[Create a Table \[page 282\]](#)

5.2.2.2 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.

Note

Keywords are case-sensitive, for example, *tableName* 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 `.hdtable` 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 \[page 289\]](#).

Table Index Type

To define the index type for the specified table using the `.hdtable` 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.

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 `.hdtable` 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

[Tables \[page 284\]](#)

[Create a Table \[page 282\]](#)

5.2.3 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  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:

ⓘ 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.

ⓘ 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* ► *MYSHEMA* ► *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* ► *MYSHEMA* ► *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

[Reusable Table Structures \[page 293\]](#)

[Create a Table \[page 282\]](#)

5.2.3.1 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 can also use table-type definitions in table user-defined functions (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;
};
```

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.

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

[Create a Table Structure \[page 291\]](#)

[Table Configuration Syntax \[page 285\]](#)

[Create a Reusable Table Structure](#)

5.2.4 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, `MYSHEMA.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.

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.

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

[Sequences \[page 296\]](#)

[Sequence Configuration Syntax \[page 298\]](#)

5.2.4.1 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`.

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.

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

[Create a Sequence \[page 294\]](#)

[Sequence Configuration Syntax \[page 298\]](#)

5.2.4.2 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.

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.

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;
```

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;
```

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;
```

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;
```

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

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_VIEW_NAME1>"];
```

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

[Create a Sequence \[page 294\]](#)

[Sequences \[page 296\]](#)

5.2.5 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, `MYSHEMA.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.:

📌 Note

The following code example is provided for illustration purposes only.

```
schema="MYSHEMA";
query="SELECT T1.\"Column2\" FROM
\"MYSHEMA\".\"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.

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

[SQL Views \[page 304\]](#)

[SQL View Configuration Syntax \[page 306\]](#)

5.2.5.1 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.

ⓘ 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

[Create an SQL View \[page 302\]](#)

5.2.5.2 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.

❗ 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
```

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>"];
```

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

[Create an SQL View \[page 302\]](#)

5.2.6 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, `MySynonym1.hdbsynonym`, and add the synonym-definition code to the new file, as illustrated in the following example.

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

[Synonyms \[page 310\]](#)

[Synonym Configuration Syntax \[page 311\]](#)

[Schema \[page 280\]](#)

5.2.6.1 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 design-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

[Schema \[page 280\]](#)

[Create a Synonym \[page 308\]](#)

5.2.6.2 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.

ⓘ 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 312]": "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 \[page 310\]](#)

[Create a Synonym \[page 308\]](#)

5.2.7 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 .
 - 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`

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.

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**).

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,Wiredue,Tema
2,Tetteh,Kwame,Kumasi
3,Nterful,Akye,Tarkwa
4,Acheampong,Kojo,Tamale
5,Assamoah,Adjoa,Takoradi
6,Mensah,Afua,Cape Coast
```

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:

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.

- Select the package that you created in the first step, for example, `mycompany.tests.TiTest`.
- Click the alternate mouse button and choose [Commit](#).
- 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.

- In the [SAP HANA Development](#) perspective, open the [Systems](#) view.
- Navigate to the catalog location where the `inhabitants` object resides, for example:

► <SID> ► [Catalog](#) ► [AMT](#) ► [Tables](#) ►

- Open a data preview for the updated object.

Right-click the updated object and choose [Open Data Preview](#) in the context-sensitive menu.

5.2.7.1 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.

Note

The table-import file must have the `.hdbti` extension, for example, `myTableImport.hdbti`.

CSV Data File Constraints

The following constraints apply to the CSV file used as a source for the table-import feature in SAP HANA XS:

- The number of table columns must match the number of CSV columns.
- There must not be any incompatibilities between the data types of the table columns and the data types of the CSV columns.
- Overlapping data in data files is not supported.
- The target table of the import must not be modified (or appended to) outside of the data-import operation. If the table is used for storage of application data, this data may be lost during any operation to re-import or update the data.

Related Information

[Table-Import Configuration \[page 268\]](#)

[Table-Import Configuration-File Syntax \[page 270\]](#)

5.2.7.2 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.

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";`

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

```

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.

Note

The configuration-file syntax does not support wildcards in the key definition; the full value of a selectable column value has to be specified.

Security Considerations

In SAP HANA XS, design-time artifacts such as tables (`.hdbtable` or `.hdbdd`) and table-import configurations (`.hdbti`) are not normally exposed to clients via HTTP. However, design-time artifacts containing comma-separated values (`.csv`) could be considered as potential artifacts to expose to users through HTTP. For this reason, it is essential to protect these exposed `.csv` artifacts by setting the appropriate application privileges; the application privileges prevents data leakage, for example, by denying access to data by users, who are not normally allowed to see all the records in such tables.

→ Tip

Place all the `.csv` files used to import content to into tables together in a single package and set the appropriate (restrictive) application-access permissions for that package, for example, with a dedicated `.xsaccess` file.

Related Information

[Table-Import Configuration-File Syntax \[page 270\]](#)

5.2.7.3 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_DS0", "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.

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";
```

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";
```

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`.

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 = ";";
```

Note

By default, the table-import operation assumes that data values in the `.csv` source file are separated by a comma (,). If no delimiter field is specified in the `.hdbti` table-import configuration file, the default setting is assumed.

delimEnclosing

Use the `delimEnclosing` keyword in the table-import configuration to specify a single character that indicates both the start and end of a set of characters to be interpreted as a single value in the `.csv` file, for example "This is all one, single value". This feature enables you to include in data values in a `.CSV` file even the character defined as the field delimiter (in `delimField`), for example, a comma (,) or a semi-colon (;).

→ Tip

If the value used to separate the data fields in your `.csv` file (for example, the comma (,)) is also used inside the data values themselves ("This, is, a, value"), you **must** declare and use a delimiter enclosing character and use it to enclose all data values to be imported.

The following example shows how to use the `delimEnclosing` keyword to specify the quote (") as the delimiting character that indicates both the start and the end of a value in the `.csv` file. Everything enclosed between the `delimEnclosing` characters (in this example, "") is interpreted by the import process as one, single value.

```
delimEnclosing="\"";
```

Note

Since the `hdbti` syntax requires us to use the quotes (") to specify the delimiting character, and the delimiting character in this example is, itself, also a quote ("), we need to use the backslash character (\) to escape the second quote (").

In the following example of values in a `.csv` file, we assume that `delimEnclosing="\\"",` and `delimField=","`. This means that imported values in the `.csv` file are enclosed in the quote character ("value") and multiple values are separated by the comma ("value1", "value 2"). Any commas **inside** the quotes are interpreted as a comma and not as a field delimiter.

```
"Value 1, has a comma","Value 2 has, two, commas","Value3"
```

You can use other characters as the enclosing delimiter, too, for example, the hash (#). In the following example, we assume that `delimEnclosing="#"` and `delimField=";"`. Any semi-colons included **inside** the hash characters are interpreted as a semi-colon and not as a field delimiter.

```
#Value 1; has a semi-colon#;#Value 2 has; two; semi-colons#;#Value3#
```

distinguishEmptyFromNull

Use the `distinguishEmptyFromNull` keyword in combination with `delimEnclosing` to ensure that the table-import process correctly interprets any **empty** value in the `.CSV` file, which is enclosed with the value defined in the `delimEnclosing` keyword, for example, as an empty space. This ensures that an empty space is imported "as is" into the target table. If the empty space is incorrectly interpreted, it is imported as `NULL`.

```
distinguishEmptyFromNull = true;
```

Note

The default setting for `distinguishEmptyFromNull` is `false`.

If `distinguishEmptyFromNull=false` is used in combination with `delimEnclosing`, then an empty value in the `.CSV` (with or without quotes "") is interpreted as `NULL`.

```
"Value1",, "", Value2
```

The table-import process would add the values shown in the example `.csv` above into the target table as follows:

Value1		NULL		NULL		Value2

keys

Use the [keys](#) keyword in the table-import configuration to specify the key range to be considered when importing the data from the .csv source file into the target table.

```
keys = [ "GROUP_TYPE" : "BW_CUBE", "GROUP_TYPE" : "BW_DSO", "GROUP_TYPE" :  
"BW_PSA"];
```

In the example above, all the lines in the .csv source file where the GROUP_TYPE column value matches one of the given values (BW_CUBE, BW_DSO, or BW_PSA) are imported into the target table specified in the table-import configuration.

```
;;;BW_CUBE;;400000000;3;400000000;slave  
;;;BW_DSO;;400000000;3;400000000;slave  
;;;BW_PSA;;20000000000;1;20000000000;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.

```
;;;;;400000000;2;400000000;all
```

5.2.7.4 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 <i>table</i> keyword is specified in a table-import configuration that references a table defined using the <i>.hdbtable</i> (or <i>.hdbdd</i>) syntax. The <i>hdbtable</i> keyword is specified in a table-import configuration that references a table defined using another table-definition syntax, for example, the <i>.hdbdd</i> syntax. The <i>cdstable</i> keyword is specified in a table-import configuration that references a table defined using another table-definition syntax, for example, the <i>.hdbtable</i> syntax.
40201	If you import into a catalog table, please provide schema	You specified a target table with the <i>table</i> keyword but did not specify a schema with the <i>schema</i> keyword.
40202	Schema could not be resolved	The schema specified with the <i>schema</i> keyword does not exist or could not be found (wrong name). The public synonym for an <i>.hdbtable</i> or <i>.hdbdd</i> (CDS) table definition cannot be resolved to a catalog table.
40203	Schema resolution error	The schema specified with the <i>schema</i> 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 (<i>.hdbti</i>), 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 key-word does not exist or could not be found (wrong name or wrong schema name).
40210	Table import syntax error	The table-import configuration file (<code>.hdbti</code>) contains one or more syntax errors.
40211	Table import constraint checks failed	The same key is specified in multiple table-import configurations (<code>.hdbti</code> 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 <code>.CSV</code> source file) or An (unexpected) error occurred on the SQL level.
40213	CSV table column count mismatch	Either the number of columns in the <code>.CSV</code> record is higher than the number of columns in the table, or The number of columns in the <code>.CSV</code> record is higher than the number of columns in its header.
40214	Column type mismatch	The <code>.CSV</code> 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 <code>.CSV</code> 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.

6 Setting Up the Analytic Model

Modeling refers to an activity of refining or slicing data in database tables by creating views to depict a business scenario. The views can be used for reporting and decision-making.

The modeling process involves the simulation of entities, such as CUSTOMER, PRODUCT, and SALES, and relationships between them. These related entities can be used in analytics applications such as SAP BusinessObjects Explorer and Microsoft Office. In SAP HANA, these views are known as information views.

Information views use various combinations of content data (that is, non-metadata) to model a business use case. Content data can be classified as follows:

- Attribute: Descriptive data, such as customer ID, city, and country.
- Measure: Quantifiable data, such as revenue, quantity sold and counters.

You can model entities in SAP HANA using the *Modeler* perspective, which includes graphical data modeling tools that allow you to create and edit data models (content models) and stored procedures. With these tools, you can also create analytic privileges that govern the access to the models, and decision tables to model related business rules in a tabular format for decision automation.

You can create the following types of information views:

- Attribute Views
- Analytic Views
- Calculation Views

6.1 Setting Up the Modeling Environment

6.1.1 Set Modeler Preferences

Launch the modeler preferences screen to view and manage the default settings that the system must use each time you logon to the *SAP HANA Modeler* perspective.

Procedure

1. Choose **Window** > **Preferences** > **SAP HANA** > **Modeler**.
2. Choose the type of preference you want to specify.
3. Choose **Apply** and **OK**.

Note

Choose **Restore Defaults** to restore your earlier preferences.

Related Information

[Modeler Preferences](#)
[Keyboard Shortcuts](#)

6.1.2 Set Keyboard Shortcuts

You can enable keyboard shortcuts for modeling actions such as, activate and validate.

The supported commands with the default keyboard shortcuts are as follows:

Command	Binding	When	Category
Activate	Ctrl+Shift+A	Navigator	Modeler Keys
Activate	Ctrl+Shift+A	In Windows	Modeler Keys
Add Table/Model	Ctrl+Shift+=	In Windows	Modeler Keys
Auto Arrange	Ctrl+L	In Windows	Modeler Keys
Data Preview	Ctrl+Shift+P	Navigator	Modeler Keys
Data Preview	Ctrl+Shift+P	In Windows	Modeler Keys
Display XML	Alt+D	In Windows	Modeler Keys
Find	Ctrl+F	Navigator	Modeler Navigator
Fit to Window	Ctrl+O	In Windows	Modeler Keys
Move Element in Output Pane (Direction: Down)	Ctrl+]	In Windows	Modeler Keys
Move Element in Output Pane (Direction: Up)	Ctrl+[	In Windows	Modeler Keys
Open	Ctrl+O	Navigator	Modeler Keys
Show View (View: History)	Alt+Shift+Q, R	In Windows	Views
Show View (View: Job Log)	Alt+Shift+Q, G	In Windows	Views
Show View (View: Where-Used List)	Alt+Shift+Q, U	In Windows	Views
Validate	Ctrl+Shift+V	In Windows	Modeler Keys
Validate	Ctrl+Shift+V	Navigator	Modeler Keys
Zoom (Type: In)	Ctrl+=	In Windows	Modeler Keys
Zoom (Type: Out)	Ctrl+-	In Windows	Modeler Keys
Zoom (Type: Reset)	Alt+Shift+O	In Windows	Modeler Keys

Note

By default all the modeler key board shortcuts are available in the default scheme. You cannot add new commands, but you can customize the commands as follows:

- Copy Command - to provide a different keyboard shortcut for an existing command.
- Unbind Command - to clear the key bindings with the command and provide a new keyboard shortcut for an existing command.
- Restore Command - to restore the default key bindings provided by the Modeler for an existing command.

6.2 Creating Views

6.2.1 Attributes and Measures

Attributes and measures form content data that you use for data modeling. The attributes represent the descriptive data, such as region and product. The measures represent quantifiable data, such as revenue and quantity sold.

Attributes

Attributes are the non-measurable analytical elements.

Attributes	Description	Example
Simple Attributes	Individual non-measurable analytical elements that are derived from the data sources.	For example, PRODUCT_ID and PRODUCT_NAME are attributes of product data source.
Calculated Attributes	Derived from one or more existing attributes or constants.	For example, deriving the full name of a customer (first name and last name), assigning a constant value to an attribute that can be used for arithmetic calculations.
Local Attributes	Local attributes that you use in an analytic view allow you to customize the behavior of an attribute for only that view.	For example, if an analytic view or a calculation view includes an attribute view as an underlying data source, then the analytic view inherits the behavior of the attributes from the attribute view.

Note

Local attributes convey the table fields available in the default node of analytic views.

Measures

Measures are measurable analytical elements. That are derived from analytic and calculation views.

Measures	Description	Example
Simple Measures	A simple measure is a measurable analytical element that is derived from the data foundation.	For example, PROFIT.
Calculated Measures	Calculated measures are defined based on a combination of data from other data sources, arithmetic operators, constants, and functions.	For example, you can use calculated measures to calculate the net profit from revenue and operational cost.
Restricted Measures	Restricted measures or restricted columns are used to filter attribute values based on the user-defined rules.	For example, you can choose to restrict the value for the REVENUE column only for REGION = APJ, and YEAR = 2012.
Counters	Counters add a new measure to the calculation view definition to count the distinct occurrences of an attribute.	For example, to count how many times product appears and use this value for reporting purposes.

Related Information

[Working With Attributes and Measures](#)

6.2.2 First Steps to View Creation

You create views to model various slices of the data stored in an SAP HANA database. In SAP HANA terminology they are known as Information Views.

Context

Information views use various combinations of content data (that is, non-metadata) to model a business use case.

Content data can be classified as follows:

Attribute - Represents the descriptive data like customer ID, city, country, and so on.

Measure - Represents the quantifiable data such as revenue, quantity sold, counters, and so on.

Information views are often used for analytical use cases such as operational data mart scenarios or multidimensional reporting on revenue, profitability, and so on. There are three types of information views: attribute view, analytic view, and calculation view. All three types of information views are non-materialized views. This creates agility through the rapid deployment of changes.

Before you start modeling your data as information views, you perform the following subtasks:

Procedure

1. Create a development workspace.
The workspace is the link between the SAP HANA repository and your local file system, where you work on project-related objects.
2. Create a project.
Create a new project for a particular application or package; you can use the project to collect in a convenient place all your application-related artifacts. For information views, create a General project.
3. Share a project.
Sharing a project enables you to ensure that changes you make to project-related files are visible to other team members and applications. Shared projects are available for import by other members of the application-development team.
4. Select a project and in the context menu, choose ► **New** ► **Other...** .
5. In the pop-up wizard, select **SAP HANA Modeler**.
 - a. Select the required view **Attribute View**, **Analytic View**, **Calculation View** or **Analytic Privilege** as required.
 - b. Choose **Next**.
 1. In the **New Information View** dialog, enter a name and description.

Note

If the project is shared, the **Package** field specifies the package that is associated with the project.

2. In case of an attribute view, select the required option in the **Subtype** as follows:

Scenario	Substeps
Create a view with table attributes.	In the Sub Type drop-down list, choose Standard .

Scenario	Substeps
Create a view with time characteristics.	1. In the <i>Sub Type</i> drop-down list, choose <i>Time</i>. 2. Select the required calendar type as follows: 1. If the calendar type is Fiscal, select a variant schema, and a fiscal variant. 2. If the calendar type is Gregorian, select the granularity for the data. 3. To use the system-generated time attribute view, select <i>Auto Create</i>.
	Note The system creates a time attribute view based on the default time tables, and defines the appropriate columns/attributes based on the granularity. It also creates the required filters. The tables used for time attribute creation with calendar type Gregorian are, M_TIME_DIMENSION, M_TIME_DIMENSION_YEAR, M_TIME_DIMENSION_MONTH, M_TIME_DIMENSION_WEEK and for calendar type Fiscal is M_FISCAL_CALENDAR. If you want to do a data preview for the created attribute view, you need to generate time data into the mentioned tables from the <i>Quick Launch</i> tab page.
Derive a view from an existing view – in this case, you cannot modify the derived view that acts as a reference to the base attribute view.	1. In the <i>Sub Type</i> drop-down, choose <i>Derived</i>. 2. Select the required attribute view.

Note

If the project is not shared, the auto-creation of time attribute view and creation of derived attribute view is not possible.

3. In case of a calculation view, perform the following:
 1. Select the required *Subtype* as described below:
 - Graphical - to use to the graphical modeling features for creation of calculation view
 - SQL Script - to write SQL statements for calculation view script
 2. If the subtype is *SQL Script*, set the *Parameter case sensitive* to true or false as you want the calculation view output parameter naming convention.
 3. If the subtype is *Graphical*, select *Enable Multidimensional Reporting* option if you want to make the view available for reporting purposes.

Note

If you do not enable multidimensional reporting, you can create a calculation view without any measure. In this case it works like a attribute view and is not available for reporting. Also, when this property is disabled, the input to the Semantics node is via projection view. If the property is enabled, the input to the Semantics node is via aggregation view. You can also change the value of this property in the *Properties* panel.

4. Choose *Finish*.

→ Tip

For more information about projects, repository workspaces, and sharing of projects, see [Using SAP HANA Projects \[page 62\]](#).

The view editor opens. Based on the view the *Scenario* panel of the editor consist of the following nodes:

- In case of an attribute view - two nodes, *Data Foundation* and *Semantics*. The *Data Foundation* node represents the tables used for defining the output structure of the view. The *Semantics* node represents the output structure of the view, that is, the dimension. In the *Details* panel you define the relationship between data sources and output elements.
- In case of an analytic view - three nodes
 - *Data Foundation* - represents the tables used for defining the fact table of the view.
 - *Logical Join* - represents the relationship between the selected table fields (fact table) and attribute views that is, used to create the star schema.
 - *Semantics* - represents the output structure of the view.
- In case of a graphical calculation view - *Semantics* node with a default *Aggregation* or *Projection* node, based on the selection of *Enable Multi Dimensional Reporting* checkbox.
- In case of a Script based calculation view - *Semantics* node with the default *SQL Script* node.

6.2.3 Create Attribute Views

You can create a view that is used to model descriptive attribute data by using attributes, that is data that does not contain measures. Attribute views are used to define joins between tables and to select a subset or select all the columns and rows of the table.

Prerequisites

You have imported SAP system tables T009 and T009B tables of type *Time* to create time attribute views.

Procedure

1. Launch SAP HANA studio.
2. In *SAP HANA System* view, expand the content node.
3. In the navigation pane, select a package where you want to create the new calculation view.
4. In the context menu of the package, select ► *New* ► *Attribute View* ►
5. Provide name and description.
6. In the *Subtype* dropdown list, select the type of the attribute view.
7. Choose *Finish*.
8. Add data sources.

- a. Select the data foundation node.
- b. In the context menu, choose [Add Objects](#).
- c. In [Find Data Sources](#) dialog box, enter the name of the data source and select it from the list.

Note

You cannot add column views to the [Data Foundation](#).

- d. Choose [OK](#).

Note

You can add the same table again in [Data Foundation](#) using table aliases in the editor.

9. Define output columns.
 - a. Select the data foundation node.
 - a. In the [Details](#) pane, select the columns that you want to add to the output of the data foundation node.
 - b. In the context menu, choose [Add To Output](#).

Note

If you want to add all columns from the data source to the output, in the context menu of the data source, choose [Add All To Output](#).

10. Hide attributes in reporting tools.

If you want to hide the attributes from the client tools or reporting tools when you execute the attribute view, then

- a. Select the [Semantics](#) node.
- b. Choose the [Columns](#) tab.
- c. Select an attribute.
- d. Select the [Hidden](#) checkbox.

11. Define key attributes.

Define at least one attribute as a key attribute. If there are more than one key attribute, all the key attributes must point to the same table, also referred to as the central table, in the data foundation.

- a. Select the [Semantics](#) node.
- b. Choose the [Columns](#) tab.
- c. Select an attribute.
- d. Select the [Key](#) checkbox.
- e. In the [Attributes](#) tab page of the [Column](#) pane, select the required attribute and select the [Type](#) as [Key Attribute](#).

Note

For auto generated time attribute views, the attributes, and key attributes are automatically assigned.

12. Activate the attribute view.

- If you are in the [SAP HANA Modeler](#) perspective:
 - [Save and Activate](#) - to activate the current view and redeploy the affected objects if an active version of the affected object exists. Otherwise only current view gets activated.

- [Save and Activate All](#) - to activate the current view along with the required and affected objects.

Note

You can also activate the current view by selecting the view in the [SAP HANA Systems](#) view and choosing [Activate](#) in the context menu. The activation triggers validation check for both the client side and the server side rules. If the object does not meet any validation check, the object activation fails.

- If you are in the [SAP HANA Development](#) perspective:
 1. In the [Project Explorer](#) view, select the required object.
 2. In the context menu, select [Team](#) > [Activate](#).

Note

The activation triggers the validation check only for the server side rules. Hence, if there are any errors on the client side, they are skipped and the object activation goes through if no error found at the server side.

13. Assign Changes

- a. In the [Select Change](#) dialog box, either create a new ID or select an existing change ID that you want to use to assign your changes.
- b. Choose [Finish](#).

For more information on assigning changes, see chapter **SAP HANA Change Recording** of the *SAP HANA Developer Guide*.

14. Choose [Finish](#).

Results

Restriction

The behavior of attribute views with the new editor is as follows:

- Consider that you have added an object to the editor and the object was modified after it was added. In such cases, close and open the editor. The helps reflect the latest changes of the modified object in the editor. For more information, see SAP Note [1783668](#).

Next Steps

After creating an attribute view, you can perform certain additional tasks to obtain the desired output. The following table lists the additional tasks that you can perform to enrich the attribute view.

Requirement	Task to Perform
If you want to filter the output of data foundation node.	Filter Output of Data Foundation Node.
Working With Attributes	
Requirement	Task to perform
If you want to create new output columns and calculate its values at runtime using an expression.	Create Calculated Columns
If you want to assign semantic types to provide more meaning to attributes in the attribute views.	Assign Semantics
If you want to create level hierarchies to organize data in reporting tools.	Create Level Hierarchies
If you want to create parent-child hierarchies to organize data in reporting tools.	Create Parent-Child Hierarchies
Working With Attribute View Properties	
Requirement	Task to perform
If you want to filter the view data either using a fixed client value or using a session client set for the user.	Filter Data for Specific Clients
If you want to execute time travel queries on attribute views.	Enable Information Views for Time Travel Queries
If you want to invalidate or remove data from the cache after specific time intervals.	Invalidate Cached Content
If you want to maintain object label texts in different languages.	Maintain Modeler Objects in Multiple Languages
If you do not recommend using an attribute view.	Deprecate Information Views

Related Information

[Attribute View Types](#)

[Create Calculated Columns \[page 348\]](#)

[Assign Semantics](#)

[Create Level Hierarchies \[page 366\]](#)

[Create Parent-Child Hierarchies \[page 368\]](#)

[Deprecate Information Views](#)

[Filter Data for Specific Clients](#)

[Enable Information Views for Time Travel Queries](#)

[Invalidate Cached Content](#)

[Maintain Modeler Object Labels in Multiple Languages](#)

[Quick Reference: Information View Properties](#)

[Preview Information View Output](#)

6.2.3.1 Attribute Views

Attribute views are used to model an entity based on the relationships between attribute data contained in multiple source tables.

For example, customer ID is the attribute data that describes measures (that is, who purchased a product). However, customer ID has much more depth to it when joined with other attribute data that further describes the customer (customer address, customer relationship, customer status, customer hierarchy, and so on).

You create an attribute view to locate the attribute data and to define the relationships between the various tables to model how customer attribute data, for example, will be used to address business needs.

You can model the following elements within an attribute view:

- Columns
- Calculated Columns

Note

In the Semantics node, you can classify the columns as attributes and build calculated columns of attribute type.

- Hierarchies

Note

For more information about the attributes and hierarchies mentioned above, see sections Attributes and Measures, and Hierarchies.

You can choose to further fine-tune the behavior of the attributes of an attribute view by setting the properties as follows:

- Filters to restrict values that are selected when using the attribute view.
- Attributes can be defined as *Hidden* so that they can be used in processes but are not visible to end users.
- Attributes can be marked as key attribute which will be used to identify a central table.
- The *Drill Down Enabled* property can be used to indicate if an attribute is available for further drill down when consumed.

Attribute views can later be joined to tables that contain measures within the definition of an analytic view or a calculation view to create virtual star schema on the SAP HANA data.

6.2.3.2 Generate Time Data

Generate time data into default time-related tables present in the `_SYS_BI` schema and use these tables in information views to add a time dimension.

Context

For modeling business scenarios that require time dimension, you generate time data in default time related tables available in the `_SYS_BI` schema. You can select the calendar type and granularity and generate the time data for a specific time span.

Procedure

1. Launch SAP HANA studio.
2. In the *Quick View* pane, choose *Generate Time Data*.
3. Select a system where you want to perform this operation.
4. Choose *Next*.
5. In the *Calendar Type* dropdown list, select a calendar type.
6. In the *From Year* and *To Year* textboxes, enter the time range for which you want to generate time data into time-related tables.
7. If you have selected the *Gregorian* calendar type, in the *Granularity* dropdown list select the required granularity.

ⓘ Note

For the granularity level *Week*, specify the first day of the week.

8. If you have selected the *Fiscal* calendar type,
 - a. In *Variant Schema* dropdown list, select a variant schema that contains tables having variant data.

ⓘ Note

Tables T009 and T009B contain variant data.

- b. Select the required variant.

The variant specifies the number of periods along with the start and end dates.

9. Choose *Finish*.

ⓘ Note

For the *Gregorian* calendar type, modeler generates time dimension data into `M_TIME_DIMENSION_YEAR`, `M_TIME_DIMENSION_MONTH`, `M_TIME_DIMENSION_WEEK`, `M_TIME_DIMENSION` tables and for the *Fiscal* calendar type, the modeler populates the generated time dimension data into the `M_FISCAL_CALENDAR` table. These tables are present in `_SYS_BI` schema.

Related Information

[Supported Calendar Types to Generate Time Data](#)
[Time Range to Generate Time Data](#)

6.2.4 Native HANA Models

Creating native HANA models can be one way to improve performance compared to development options outside of the database, or in some cases also compared to pure SQL development.

Native HANA models can be developed in the new XS Advanced (XSA) development environment using SAP Web IDE for SAP HANA. These models supersede older artifacts like Analytic and Attribute Views; these views should now be replaced by graphical Calculation Views which can be used to model complex OLAP business logic. Native HANA modeling provides various options to tune performance by, for example, helping to achieve complete unfolding of the query by the calculation engine or modeling join cardinalities between two tables (that is, the number of matching entries (1...n) between the tables) and optimizing join columns.

For more information about modeling graphical calculation views refer to the *SAP HANA Modeling Guide for SAP Web IDE for SAP HANA*.

A number of blogs are available about the details of modeling:

- <https://blogs.sap.com/2017/09/01/overview-of-migration-of-sap-hana-graphical-view-models-into-the-new-xsa-development-environment/> Overview: Migration of Models into the XSA Development Environment
- <https://blogs.sap.com/2017/10/27/join-cardinality-setting-in-calculation-views/> Join cardinality setting in Calculation Views
- <https://blogs.sap.com/2018/08/10/optimize-join-columns-flag/> Optimize Join Columns Flag

The following SAP Notes provide further background information:

- <https://launchpad.support.sap.com/#/notes/2441054> 2441054 - High query compilation times and absence of plan cache entries for queries against calculation views.
- <https://launchpad.support.sap.com/#/notes/2465027> 2465027 - Deprecation of SAP HANA extended application services, classic model and SAP HANA Repository.

Related Information

[SAP HANA Modeling Guide for SAP Web IDE for SAP HANA](#)

6.2.4.1 Analytic Views

Analytic views are used to model data that includes measures.

For example, an operational data mart representing sales order history would include measures for quantity, price, and so on.

The data foundation of an analytic view can contain multiple tables. However, measures that are selected for inclusion in an analytic view must originate from only one of these tables (for business requirements that include measure sourced from multiple source tables, see calculation view).

Analytic views can be simply a combination of tables that contain both attribute data and measure data. For example, a report requiring the following:

```
<Customer_ID Order_Number Product_ID Quantity_Ordered Quantity_Shipped>
```

Optionally, attribute views can also be included in the analytic view definition. In this way, you can achieve additional depth of attribute data. The analytic view inherits the definitions of any attribute views that are included in the definition. For example:

```
<Customer_ID/Customer_Name Order_Number Product_ID/Product_Name/Product_Hierarchy  
Quantity_Ordered Quantity_Shipped>
```

You can model the following elements within an analytic view:

- Columns
- Calculated Columns
- Restricted Columns

→ Remember

In the Semantics node, you can classify columns and calculated columns as type attributes and measures. The attributes you define in an analytic view are Local to that view. However, attributes coming from attribute views in an analytic view are Shared attributes. For more information about the attributes and measures mentioned above, see section Attributes and Measures.

- Variables
- Input parameters

📘 Note

For more information about the variables and input parameters mentioned above, see sections Assigning Variables and Creating Input Parameters.

You can choose to further fine-tune the behavior of the attributes and measures of an analytic view by setting the properties as follows:

- Filters to restrict values that are selected when using the analytic view.
- Attributes can be defined as *Hidden* so that they are able to be used in processes but are not viewable to end users.
- The *Drill Down Enabled* property can be used to indicate if an attribute is available for further drill down when consumed.
- Aggregation type on measures
- *Currency* and *Unit of Measure* parameters (you can set the *Measure Type* property of a measure, and also in *Calculated Column* creation dialog, associate a measure with currency and unit of measure)

→ Tip

If there is a name conflict that is, more than one element having the same name among the local and shared attributes, calculated columns, restricted columns, and measures of an analytic view, the activation of the view does not go through. You can resolve such conflict using the aliases. Aliases must also have unique names. You can assign an alias to the required element in the [Column](#) view of the [Semantics](#) node by editing its name inline. Hereinafter, the element is referred by its alias.

If two or more shared columns have a name conflict, during save the aliases for the conflicting name columns are proposed. You can choose to overwrite the proposed names.

In case of old models, if you find any error while opening the object due to aliasing that was caused due to swapping of column names with the alias names, use the [Quick Fix](#). To use the [Quick Fix](#), select the error message that is, the problem in the [Problems](#) view, and choose [Quick Fix](#) in the context menu. This resolves the swapping issue by assigning right names to the column and alias.

You can choose to hide the attributes and measures that are not required for client consumption by assigning value **true** to the property [Hidden](#) in the [Properties](#) pane, or selecting the [Hidden](#) checkbox in the [Column](#) view. The attributes or measures marked as hidden are not available for input parameters, variables, consumers or higher level views that are build on top of the analytic view. For old models (before SPS06), if the hidden attribute is already used, you can either unhide the element or remove the references.

For an analytic view, you can set the property [MultiDimensional Reporting](#) to true or false. If the [MultiDimensional Reporting](#) property of the analytic view is set to false, the view will not be available for multidimensional reporting purposes. If the value is set to true, an additional column [Aggregation](#) is available to specify the aggregation type for measures.

You can enable relational optimization for your analytic view such as, Optimize stacked SQL for example, convert

```
SELECT a, SUM(X) FROM ( SELECT * FROM AV) GROUP BY A
```

to

```
SELECT A, SUM(X) FROM AV GROUP BY A
```

by setting the property [Allow Relational Optimization](#).

Setting this property would be effective only for analytic views having complex calculations such that deployment of analytic view generates catalog calculation view on top of the generated catalog OLAP view.

⚠ Caution

In this case, if this flag is set counters and SELECT COUNT may deliver wrong results

6.2.4.2 Create Temporal Joins

Temporal joins allow you to join the master data with the transaction data (fact table) based on the temporal column values from the transaction data and the time validity from the master data.

Procedure

1. Open the analytic view or calculation view with star join node in the view editor.
2. Select the [Star Join](#) node.

The star join node must contain the master data as a data source. The input to the star join node (the data foundation node) provides the central fact table.

3. Create a join

Create a join by selecting a column from one data source (master table), holding the mouse button down and dragging to a column in the other data source (fact table).

4. Select the join.
5. In the context menu, choose [Edit](#).
6. Define join properties.

In the [Properties](#) section, define the join properties.

Note

For temporal joins in analytic views, you can use [Inner](#) or [Referential](#) join types only and for temporal joins in calculation views, you can use [Inner](#) join type only.

7. Define temporal column and temporal conditions

In the [Temporal Properties](#) section, provide values to create temporal join.

- a. In the [Temporal Column](#) dropdown list, select a time column in the analytic view.
 - b. In the [From Column](#) and [To Column](#) dropdown list specify the start and end time values from the attribute view to fetch the records.
 - c. In the [Temporal Condition](#) dropdown list, select a condition.
8. Choose [OK](#).

Related Information

[Temporal Joins](#)

[Temporal Conditions](#)

[Example: Temporal Joins](#)

6.2.5 Create Graphical Calculation Views

Create graphical calculation views using a graphical editor to depict a complex business scenario. You can also create graphical calculation views to include layers of calculation logic and with measures from multiple data sources.

Context

Graphical calculation views can bring together normalized data that are generally dispersed. You can combine multiple transaction tables and analytic views, while creating a graphical calculation view.

❗ Note

If you want to execute calculation views in SQL engine, see SAP NOTE [1857202](#) .

Procedure

1. Launch SAP HANA studio.
2. In *SAP HANA System* view, expand the content node.
3. In the navigation pane, select a package where you want to create the new calculation view.
4. In the context menu of the package, select **► New ► Calculation View ►**.
5. Provide name and description.
6. Select calculation view type.
In the *Type* dropdown list, select *Graphical*.
7. Select a *Data Category* type.
8. Choose *Finish*.

Modeler launches a new graphical calculation view editor with the semantics node and default aggregation or projection node depending on the data category of the calculation view.

9. Continue modeling the graphical calculation view by dragging and dropping the necessary view nodes from the tool palette.
10. Add data sources.

If you want to add data sources to your view node, then

- a. Select a view node.
- b. In the context menu, choose *Add Objects*.
- c. In the *Find* dialog box, enter the name of the data source and select it from the list.

You can add one or more data sources depending on the selected view node.

- d. Choose *OK*.
11. Define output columns.
 - a. Select a view node.

- b. In the *Details* pane, select the columns that you want to add to the output of the node.
- c. In the context menu, choose *Add To Output*.
- d. If you want to add all columns from the data source to the output, in the context menu of the data source, choose *Add All To Output*.

Note

Using keep flag column property. The keep flag property helps retrieve columns from the view node to the result set even if you do not request it in your query. In other words, if you want to include those columns into the SQL group by clause even if you do not select them in the query, then:

1. Select the view node.
2. In the *Output* pane, select an output column.
3. In the *Properties* pane, set the value of *Keep Flag* property to *True*.

12. Define attributes and measures.

If you are creating a calculation view with data category as cube, then to successfully activate the information view, specify at least one column as a measure.

- a. Select the *Semantics* node.
- b. Choose the *Columns* tab.
- c. In the *Local* section, select an output column.
- d. In the *Type* dropdown list, select *Measure* or *Attribute*.

If the value is set to *Cube*, an additional *Aggregation* column is available to specify the aggregation type for measures.

Note

If the default node of the calculation view is aggregation, you can always aggregate the measures even if no aggregation function is specified in the SQL.

1. Select the default aggregation node.
2. In the *Properties* tab, set the value of the property *Always Aggregate Results* to *True*

- e. If you want to hide the measure of attribute in the reporting tool, select the *Hidden* checkbox.
- f. If you want to force the query to retrieve selected attribute columns from the database even when not requested in the query, set the *Keep Flag* property to *True* for those attributes.

This means that you are including those columns into the group by clause even if you do not select them in the query. To set the *Keep Flag* property of attributes to *True*, select an attribute in the *Output* pane, and in the *Properties* pane set the *Keep Flag* property to *True*.

Note

If you are using any attribute view as a data source to model the calculation view, the *Shared* section displays attributes from the attribute views that are used in the calculation view.

13. Activate the calculation view.

- If you are in the *SAP HANA Modeler* perspective,
 - *Save and Activate* - to activate the current view and redeploy the affected objects if an active version of the affected object exists. Otherwise, only the current view is activated.
 - *Save and Activate All* - to activate the current view along with the required and affected objects.

Note

You can also activate the current view by selecting the view in the *SAP HANA Systems* view and choosing *Activate* in the context menu. The activation triggers validation check for both the client side and the server side rules. If the object does not meet any validation check, the object activation fails.

- If you are in the *SAP HANA Development* perspective,
 1. In the *Project Explorer* view, select the required object.
 2. In the context menu, select ► *Team* ► *Activate* ►.

Note

The activation only triggers the validation check for the server side rules. If there are any errors on the client side, they are skipped, and the object activation goes through if no error is found on the server side.

Note

1. For an active calculation view, you can preview output data of an intermediate node. This helps to debug each level of a complex calculation scenario (having join, union, aggregation, projection, and output nodes). Choose the *Data Preview* option from the context menu of a node.
When you preview the data of an intermediate now, SAP HANA studio activates the intermediate calculation model with the current user instead of the user `_SYS_REPO`. The data you preview for a node is for the active version of the calculation view. If no active version for the object exists then activate the object first.

14. Assign Changes

- a. In the *Select Change* dialog box, either create a new ID or select an existing change ID that you want to use to assign your changes.
- b. Choose *Finish*.

For more information on assigning changes, see chapter **SAP HANA Change Recording** of the *SAP HANA Developer Guide*.

15. Choose *Finish*.

Next Steps

After creating a graphical calculation view, you can perform certain additional tasks to obtain the desired output. The following table lists the additional tasks that you can perform to enrich the calculation view.

Working With View Nodes

Requirement	Task to Perform
If you want to query data from two data sources and combine records from both the data sources based on a join condition or to obtain language-specific data.	Create Joins
If you want to query data from database tables that contains spatial data.	Create Spatial Joins

Requirement	Task to Perform
If you want to validate joins and identify whether you have maintained the referential integrity.	Validate Joins
If you want to combine the results of two more data sources.	Create Unions
If you want to partition the data for a set of partition columns, and perform an order by SQL operation on the partitioned data.	Create Rank Nodes
If you want to filter the output of projection or aggregation view nodes.	Filter Output of Aggregation or Projection View Nodes.

Working With Attributes and Measures

Requirement	Task to perform
If you want to count the number of distinct values for a set of attribute columns.	Create Counters
If you want to create new output columns and calculate its values at runtime using an expression.	Create Calculated Columns
If you want to restrict measure values based on attribute restrictions.	Create Restricted Columns
If you want to assign semantic types to provide more meaning to attributes and measures in calculation views.	Assign Semantics
If you want to parameterize calculation views and execute them based on the values users provide at query runtime.	Create Input Parameters
If you want to, for example, filter the results based on the values that users provide to attributes at runtime.	Assign Variables
If you want to associate measures with currency codes and perform currency conversions.	Associate Measures with Currency
If you want to associate measures with unit of measures and perform unit conversions.	Associate Measures with Unit of Measure
If you want to create level hierarchies to organize data in reporting tools.	Create Level Hierarchies
If you want to create parent-child hierarchies to organize data in reporting tools.	Create Parent-Child Hierarchies
If you want to group related measures together in a folder.	Group Related Measures.

Working With Calculation View Properties

Requirement	Task to perform
If you want to filter the view data either using a fixed client value or using a session client set for the user.	Filter Data for Specific Clients
If you want to execute time travel queries on calculation views.	Enable Information Views for Time Travel Queries
If you want to invalidate or remove data from the cache after specific time intervals.	Invalidate Cached Content
If you want to maintain object label texts in different languages.	Maintain Modeler Objects in Multiple Languages
If you do not recommend using a calculation view.	Deprecate Information Views

Related Information

[Create Graphical Calculation Views With Star Joins](#)
[Use Table Functions as a Data Source](#)
[Use SAP HANA CDS Entities or Views as a Data Source](#)
[Supported View Nodes for Modeling Graphical Calculation Views](#)
[Modeling Graphical Calculation Views With Tenant Databases](#)
[Supported Data Categories for Information Views](#)
[Working With View Nodes](#)
[Preview Information View Output](#)
[Working With Attributes and Measures](#)
[Working With Information View Properties](#)
[Defining Data Access Privileges](#)
[Additional Functionality for Information Views \[page 365\]](#)

6.2.5.1 Calculation Views

A calculation view is used to define more advanced slices on the data in SAP HANA database. Calculation views can be simple and mirror the functionality found in both attribute views and analytic views. However, they are typically used when the business use case requires advanced logic that is not covered in the previous types of information views.

For example, calculation views can have layers of calculation logic, can include measures sourced from multiple source tables, can include advanced SQL logic, and so on. The data foundation of the calculation view can include any combination of tables, column views, attribute views and analytic views. You can create joins, unions, projections, and aggregation levels on the sources.

You can model the following elements within a calculation view:

- Attributes
- Measures
- Calculated measures
- Counters
- Hierarchies (created outside of the attribute view)

Note

For more information about the attributes, measures, counters, and hierarchies mentioned above, see sections [Attributes and Measures](#), and [Hierarchies](#).

- Variables
- Input parameters

Note

For more information about the variables and input parameters mentioned above, see sections [Assign Variables](#) and [Input Parameters](#).

Calculation views can include measures and be used for multi-dimensional reporting or can contain no measures and used for list-type of reporting. Calculation views can either be created using a graphical editor or using a SQL Console . These various options provide maximum flexibility for the most complex and comprehensive business requirements.

Related Information

[Input Parameters \[page 372\]](#)

[Assign Variables \[page 373\]](#)

6.2.5.2 Create Calculated Columns

Create new output columns and calculate its values at runtime based on the result of an expression. You can use other column values, functions, input parameters, or constants in the expression.

Context

For example, you can create a calculated column DISCOUNT using the expression `if("PRODUCT" = 'NOTEBOOK', "DISCOUNT" * 0.10, "DISCOUNT")`. In this sample expression, you use the function `if()`, the column `PRODUCT` and operator `*` to obtain values for the calculated column `DISCOUNT`.

Procedure

1. Open the required graphical calculation view in the view editor.
2. Select the view node in which you want to create the calculated column.
3. In the *Output* pane, choose the icon  dropdown.
4. Choose the *New Calculated Column* menu option.
5. In the *Calculated Column*, enter a name and description for the new calculated column.
6. In the *Data Type* dropdown list, select the data type of the calculated column.
7. Enter length and scale based on the data type you select.

Modeler ignores the length for `VARCHAR` data type. If you want to, for example, truncate length, you must use the relevant string functions in calculated column expression.

8. Select a column type.

You can create calculated attributes or calculated measures using attributes or measures respectively.

- a. In the *Column Type* dropdown list, select a value.

Note

If you want to create a calculated measure and enable client side aggregation for the calculated measure, select the [Enable client side aggregation](#) checkbox.

This allows you to propose the aggregation that client needs to perform on calculated measures.

9. If you want to hide the calculated column in reporting tools, select the [Hidden](#) checkbox.
10. Choose [OK](#).
11. Provide an expression.

You can create an expression using the SQL language or the column engine language.

- a. In the [Language](#) dropdown list, select the expression language.
- a. In the [Expression Editor](#), enter a valid expression.

Modeler computes this expression at runtime to obtain values of calculated columns.

For example, the expression in column engine language, `if("PRODUCT" = 'NOTEBOOK', "DISCOUNT" * 0.10, "DISCOUNT")` which is equivalent to, if attribute PRODUCT equals the string 'NOTEBOOK' then DISCOUNT equals to DISCOUNT multiplied by 0.10 should be returned. Else use the original value of the attribute DISCOUNT.

Note

You can also create an expression by dragging and dropping the expression elements, operators, and functions from the menus to the expression editor. For expression in SQL language, modeler supports only a limited list of SQL functions.

- b. Choose [Validate Syntax](#) to validate your expression.
12. Assign semantics to the calculated column.
 - a. Choose the [Semantics](#) tab.
 - b. In the [Semantic Type](#) dropdown list, select a semantic value.

Related Information

[Using Functions in Expressions \[page 377\]](#)

[Calculated Column Properties](#)

[Example: Calculated Measures](#)

[Example: Calculated Attributes](#)

6.2.5.3 Map Input Parameters

You can map the input parameters in the underlying data sources (attribute views, analytic views and calculation views) of the calculation view to the calculation view parameters. You can also map many data source parameters to one calculation view input parameter and perform a one-to-one mapping of the data source parameters to the calculation view parameters.

Context

Note

You can map attribute view input parameters to calculation view input parameters with the same name only. The calculation view input parameter provides runtime value selection to filter attribute data based on the filter defined in the attribute view. For example, you can define an attribute view GEO with filter set on Country column such that, the filter value is an input parameter having syntax \$\$IP\$\$\$. When you use this attribute view in a calculation view, you need to define a same name input parameter IP and map it with the attribute view parameter. When you perform data preview on the calculation view, the runtime help for the calculation view input parameter is shown. The value selected for calculation view parameter serves as input for the attribute view parameter to filter the data.

Procedure

1. To invoke the dialog from the default aggregation or projection node:
 - a. Select the default aggregation or projection node.
 - b. Right-click *Input Parameter* in the *Output* view.
 - c. In the context menu, choose *Manage Mappings*.
2. To invoke the dialog from the *Semantics* node:
 - a. Select the *Semantics* node.
 - b. In the *Variables/Input Parameters* view, choose .
 - c. Choose *Data sources* or *Views for value help*

Note

The system displays the option *Views for value help* only if your calculation view consists of external views as value help references in variables and input parameters. If you choose *Views for value help*, you can map the parameters/ variable of external views for value help with the parameters/ variables of a calculation view of any name.

3. In the *Map Input Parameters* dialog, map the data source input parameters (or parameters of external views for value help) with the calculation view parameters.

Note

You can choose the *Auto Map by Name* option to automatically create the input parameters corresponding to the source and perform a 1:1 mapping. You can also select a source input parameter and use the following context menu options:

- *Copy and Map 1:1* - to create the same input parameter for the calculation view as for the source, and create a 1:1 mapping between them.
- *Map By Name* - to map the source input parameter with the calculation view input parameter having the same name.
- *Remove Mapping* - to delete the mapping between the source and calculation view input parameter.

4. Select [Create Constant](#) to create a constant at the target calculation view.

📘 Note

You can change the constant name by double clicking it.

6.2.5.4 Create Unions

Use union nodes in graphical calculation views to combine the results of two or more data sources.

Context

A union node combines multiple data sources, which can have multiple columns. You can manage the output of a union node by mapping the source columns to the output columns or by creating a target output column with constant values.

For a source column that does not have a mapping with any of the output columns, you can create a target output column and map it to the unmapped source values. You can also create a target column with constant values.

Procedure

1. Open the required graphical calculation view in the view editor.
2. From the tools palette, drag and drop a union node to the editor.
3. Add data sources.
 - a. Select the union node.
 - b. In the context menu, choose [Add Objects](#).
 - c. In [Find Data Sources](#) dialog box, enter the name of the data source and select it from the list.
 - d. Choose [OK](#).
4. Define output columns.
 - a. In the [Details](#) pane, select the columns you want to add to the output of the union node.
 - b. In the context menu, choose [Add To Output](#).

📘 Note

If you want to add all columns from the data source to the output, in the context menu of the data source, choose [Add All To Output](#).

5. Assign constant value.

This helps to denote the underlying data of the source columns with constant values in the output.

If you want to assign a constant value to any of the target columns, then

- a. In the *Target* section, select an output column.
 - b. In the context menu, choose *Manage Mappings*.
 - c. In the *Manage Mappings* dialog box, set the *Source Column* value as blank.
 - d. In the *Constant Value* field, enter a constant value.
 - e. Choose *OK*.
6. Create a constant output column.

If you want to create a new output column and assign a constant value to it, then

- a. In the *Target* section, choose .
- b. In the *Create Target* dialog box, provide name and data type for the new output column.
- c. Choose *OK*.

Note

By default, the value of the constant output column is null.

Related Information

[Constant Column \[page 352\]](#)

[Example: Constant Columns](#)

[Empty Union Behavior](#)

[Prune Data in Union Nodes](#)

6.2.5.5 Constant Column

In a union view, a *Constant Column* is created if there are any target or output attributes for which there are no mappings to the source attributes. The default value for the constant column is NULL.

Note

The target attribute is mapped to all the sources.

For example, you have two tables with similar structures, Actual Sales and Planned Sales, corresponding to the sales of products. You want to see the combined data in a single view, but differentiate between the data from the two tables for comparison. To do so, you can create a union view between the two tables and have a constant column indicating constant values like A & P, as shown below:

Actual Sales

Sales	Product
5000	A1
2000	B1

Planned Sales

Sales	Product
3000	A1
6000	B1

The result of this query can be as follows:

Actual Planned Indicator	Sales	Product
A	5000	A1
P	3000	A1
A	2000	B1
P	6000	B1

6.2.5.6 Dynamic Joins

After creating a join between two data sources, you can define the join property as dynamic. Dynamic joins helps improve the join execution process by reducing the number of records processed by the join view node at runtime.

With dynamic joins, two or more fields from two data sources are joined using a join condition that changes dynamically based on the fields requested by the client. For example, Table1 and Table2 are joined on Field1 and Field2. But, if only one, Field1 or Field2 is requested by a client, the tables (Table1 and Table2) are joined based only on the requested field (Field1 or Field2).

⚠ Restriction

You can set the *Dynamic Join* property only if the two data sources are joined on multiple columns.

⚠ Caution

Use dynamic joins with caution. Dynamic join behavior is different from the classical join behavior. In the classical join, the join condition is static. This means that the join condition does not change irrespective of the client query. The difference in behavior can result in different query result sets.

Prerequisite

At least one of the fields involved in the join condition is part of the client query.

If you define a join as dynamic, the engine dynamically defines the join fields based on the fields requested by the client query. But, if the field is not part of the client query, it results in query runtime error.

Static Join Versus Dynamic Joins

- In static joins, the join condition isn't changed, irrespective of the client query.
- In a dynamic join, if the client query to the join doesn't request a join column, a query runtime error occurs. This behavior of dynamic join is different from the static joins.
- Dynamic join enforces aggregation before executing the join, but for static joins the aggregation happens after the join. This means that, for dynamic joins, if a join column is not requested by the client query, its value is first aggregated, and later the join condition is executed based on columns requested in the client query.

Related Information

[Example: Dynamic Joins](#)

6.2.5.7 Filter Output of Aggregation or Projection View Nodes

Apply filters on columns in the projection or the aggregation view nodes (except the default aggregation or projection node) to filter the output of these nodes.

Context

You apply filters, for example, to retrieve the sales of a product where (revenue $\geq$ 100 AND region = India) OR (revenue $\geq$ 50 AND region = Germany). You can also define filters using nested or complex expressions.

Filters on columns are equivalent to the HAVING clause of SQL. At runtime, the modeler executes the filters after performing all the operations that you have defined in the aggregation or projection. You can also use input parameters to provide values to filters at runtime.

Procedure

1. Applying filters on columns of calculation views.

If you want to define filters on columns of projection or aggregation view nodes in calculation views:

- a. Open the calculation view in the view editor.
- b. Select a projection or aggregation view node.
- c. In the *Details* pane, select a column.
- d. In the context menu, choose *Apply Filter*.

- e. In the [Apply Filter](#) dialog box, select an operator.
- f. In the [Value](#) field, select a fixed value or an input parameter from the value help.

Note

In the selected view node, if you are using other information views as data sources (and not tables), then you can use only input parameters to apply filters on columns.

- g. Choose [OK](#).
2. Choose [OK](#).
 3. If you want to apply filters on columns or at the node level using expressions.

You can create expression in SQL language or the column engine language to apply filters. For example, `match("ABC", '*abc*')` is an expression in the column engine language.

- a. Select the aggregation or projection node.
- b. In the [Output](#) pane, expand [Filters](#).
- c. In the context menu of [Expression](#), choose [Open](#).
- d. Enter the expression by selecting the required elements, operators, input parameters, calculated columns, and functions.
- e. Choose [OK](#).

Note

For expression in SQL language, modeler supports only a limited list of SQL functions.

Related Information

[Supported Operators for Filters](#)

6.2.6 Create Script-Based Calculation Views

Create script-based calculation views to depict complex calculation scenarios by writing SQL script statements. It is a viable alternative to depict complex business scenarios, which you cannot achieve by creating other information views (Attribute, Analytical, and Graphical Calculation views).

Context

For example, if you want to create information views that require certain SQL functions (i.e. window), or predictive functions (i.e. R-Lang), then you use script-based calculation views. Sufficient knowledge of SQL scripting including the behavior and optimization characteristics of the different data models is a prerequisite for creating script-based calculation views.

Procedure

1. Launch SAP HANA studio.
2. In *SAP HANA System* view, expand the content node.
3. In the navigation pane, select a package where you want to create the new calculation view.
4. In the context menu of the package, select ► *New* ► *Calculation View* .
5. Provide name and description.
6. Select calculation view type.
In the *Type* dropdown list, select *SQL Script*.
7. Set *Parameter Case Sensitive* to *True* or *False* based on how you require the naming convention for the output parameters of the calculation view.
8. Choose *Finish*.
9. Select default schema
 - a. Select the *Semantics* node.
 - b. Choose the *View Properties* tab.
 - c. In the *Default Schema* dropdown list, select the default schema.

Note

If you do not select a default schema while scripting, then provide fully qualified names of the objects used.

10. Choose *SQL Script* node in the *Semantics* node.

Note

The *IN* function does not work in SQL script to filter a dynamic list of values. Use *APPLY_FILTER* functions instead.

11. Define the output structure.
 - a. In the *Output* pane, choose *Create Target*.
 - b. Add the required output parameters and specify its length and type.
12. If you want to add multiple columns that are part of existing information views or catalog tables or table functions to the output structure of script-based calculation views, then:
 - a. In the *Output* pane, choose ► *New* ► *Add Columns From* .
 - b. Enter the name of the object that contains the columns you want to add to the output.
 - c. Select one or more objects from the dropdown list.
 - d. Choose *Next*.
 - e. In the *Source* pane, choose the columns that you want to add to the output.
 - f. If you want to add selective columns to the output, then select those columns and choose *Add*.
 - g. If you want to add all columns of an object to the output, then select the object and choose *Add*.

Note

For all duplicate column names in the *Target* pane, the modeler displays an error. You cannot add two columns with the same name to your output. If you want to retain both the columns, then change the name of columns in the *Target* pane before you add them to the output.

- h. If you want to override the existing output structure, select *Replace existing output columns in the Output*.
- i. Choose *Finish*.

Note

The defined order and data types of columns and parameters must match with the order and data types of the columns and parameters in the select query, which is assigned to the output function var_out.

13. Write the SQL Script statements to fill the output columns.

You can drag information views from the navigator pane to the SQL editor to obtain an equivalent SQL statement that represents the deployed schema name for the information view.

Note

For information on providing input parameters in script-based calculation views, see SAP Note [2035113](#).

14. Activate the script-based calculation view.

- If you are in the *SAP HANA Modeler* perspective:
 - *Save and Activate* - to activate the current view and redeploy the affected objects if an active version of the affected object exists. Otherwise, only the current view is activated.
 - *Save and Activate All* - to activate the current view along with the required and affected objects.

Note

You can also activate the current view by selecting the view in the *SAP HANA Systems* view and choosing *Activate* in the context menu. The activation triggers validation check for both the client side and the server side rules. If the object does not meet any validation check, the object activation fails.

- If you are in the *SAP HANA Development* perspective:
 1. In the *Project Explorer* view, select the required object.
 2. In the context menu, select **Team** > *Activate*.

Note

The activation only triggers the validation check for the server side rules. If there are any errors on the client side, they are skipped, and the object activation goes through if no error is found on the server side.

For more information about the details of the functions available on content assist (pressing Ctrl + Space in the SQL Console while writing procedures) in the *SAP HANA SQLScript Reference*.

15. Assign Changes

- a. In the *Select Change* dialog box, either create a new ID or select an existing change ID that you want to use to assign your changes.
- b. Choose *Finish*.

For more information on assigning changes, see *SAP HANA Change Recording* of the *SAP HANA Developer Guide*.

16. Choose *Finish*.

Next Steps

After creating a script-based calculation view, you can perform certain additional tasks to obtain the desired output. The following table lists the additional tasks that you can perform to enrich the calculation view.

Working With Attributes and Measures

Requirement	Task to perform
If you want to assign semantic types to provide more meaning to attributes and measures in calculation views.	Assign Semantics
If you want to parameterize calculation views and execute them based on the values users provide at query runtime.	Create Input Parameters
If you want to, for example, filter the results based on the values that users provide to attributes at runtime.	Assign Variables
If you want to associate measures with currency codes and perform currency conversions.	Associate Measures with Currency
If you want to associate measures with unit of measures and perform unit conversions.	Associate Measures with Unit of Measure
If you want to create level hierarchies to organize data in reporting tools.	Create Level Hierarchies
If you want to create parent-child hierarchies to organize data in reporting tools.	Create Parent-Child Hierarchies
If you want to group related measures together in a folder.	Group Related Measures.

Working With Calculation View Properties

Requirement	Task to perform
If you want to filter the view data either using a fixed client value or using a session client set for the user.	Filter Data for Specific Clients
If you want to execute time travel queries on script-based calculation views.	Enable Information Views for Time Travel Queries
If you want to invalidate or remove data from the cache after specific time intervals.	Invalidate Cached Content
If you want to maintain object label texts in different languages.	Maintain Modeler Objects in Multiple Languages
If you do not recommend using a script-based calculation view.	Deprecate Information Views

Related Information

[Use Script-based Calculation Views as Table Functions](#)

[Working With View Nodes](#)

[Preview Information View Output](#)

[Working With Attributes and Measures](#)

[Working With Information View Properties](#)

[Quick Reference: Information View Properties](#)

[Supported Data Categories for Information Views](#)

[SAP HANA SQLScript Reference](#)

6.2.7 Activating Objects

You activate objects available in your workspace to expose the objects for reporting and analysis.

Based on your requirements, you can do the following:

- [Activate](#) - Deploys the inactive objects.
- [Redeploy](#) - Deploys the active objects in one of the following scenarios:
 - If your runtime object gets corrupted or deleted, and you want to create it again.
 - In case of runtime problems during object activation, and the object status is still active.

The following activation modes are supported:

- [Activate and ignore the inconsistencies in affected objects](#) - To activate the selected objects even if it results in inconsistent affected objects. For example, if you choose to activate an object A that is used by B and C, and it causes inconsistencies in B and C but you can choose to go ahead with the activation of A. This is the default activation mode.
- [Stop activation in case of inconsistencies in affected objects](#) - To activate the selected objects only if there are no inconsistent affected objects.

Note

If even one of the selected objects fails (either during validation or during activation), the complete activation job fails and none of the selected objects is activated.

Depending on where you invoke the activation, redeployment or cascade activation, the behavior is as follows:

Context	Activate	Redeploy
Quick Launch tab page	A dialog box appears with a preselected list of all your inactive objects.	A dialog box appears with a list of active objects in your workspace.
Package context menu	A dialog box appears with a preselected list of all your inactive objects.	A dialog box appears with a list of active objects in your workspace.
Content context menu	A dialog box appears with a preselected list of all your inactive objects.	Not applicable
Editor	• If you select Save and Activate, current object is activated and the affected objects are redeployed if an active version for the affected objects exist. • If you select Save and Activate All, a dialog box appears with a preselected list of the selected object along with all the required and affected objects.	Not applicable
Object context menu	A dialog box appears with a preselected list of the selected object along with all the required objects.	A redeployment job is submitted for the selected object.

Note

- If an object is the only inactive object in the workspace, the activation dialog box is skipped and the activation job is submitted.
- If an object is inactive and you want to revert back to the active version, from the editor or object context menu, choose [Revert To Active](#).
- In the [Activate](#) dialog, you can select the [Bypass validation](#) checkbox in order to skip validation before activation to improve the activation time. For example, if you have imported a number of objects and want to activate them without spending time on validation.

Note

During delivery unit import, full server side activation is enabled, activation of objects after import is done. In this case all the imported objects are activated (moved to active table), even if there are errors in activated or affected objects. But the objects for which activation results in error are considered as broken or inconsistent objects which means that the current runtime representation of these objects is not in sync with the active design time version. The broken objects are shown in the [Navigator](#) view with an 'x' along side.

Note

The behavior of the activation job is as follows:

- The status (completed, completed with warnings, and completed with errors) of the activation job indicates whether the activation of the objects is successful or failed.
- In case of failure that is when the status is completed with errors, the process is rolled back. This means, even if there are individual objects successfully activated, since the activation job is rolled back, none of the objects are activated.
- When you open the job log, the summary list shows only those objects that are submitted for activation. It does not list all the affected objects. They are listed only in detail section.

Activation behavior in the view editor

The following table describes the availability and behavior of take over and activate options for an object from the view editor in the [SAP HANA Modeler](#) perspective.

Scenario	Object	in Team Provider		in SAP HANA Systems view	SAP HANA Systems view		Description
		User: U1,Workspace: WS1	User: U2,Workspace: WS2	User: U, Workspace: "" (default/other workspace)	Take Over	Activate	

Scenario	Object	in Team Provider		in SAP HANA Systems view	SAP HANA Systems view		Description
1	OBJ1	Inactive	Inactive	Inactive	Not Applicable	Allowed	If an object has multiple inactive versions, and the object version in Modeler is also inactive, for example, through delivery unit import or another workspace in Project Explorer, user can activate his own inactive object. After activation, the object is the scenario 2 as in the next row.
 Note If the logged-in user and the user to whom the object belongs are different, the activation is not allowed. For example, if the object is inactive in SYSTEM user's workspace and MB user opens the object, the object opens in read-only mode, and the activation is not allowed.							
2	OBJ1	Inactive	Inactive	Active	Not Allowed	Not Allowed	If an object has multiple inactive versions in the Project Explorer and the object version in Modeler is active, neither activation nor take over option is enabled.
3	OBJ1	Inactive	Active	Active	Allowed	Not Allowed	If an object has single inactive version in the Project Explorer, and the object version in Modeler is active, only take over option is enabled.
4	OBJ1	Inactive	Active	Inactive	Not Applicable	Allowed	If an object has inactive versions in the Project Explorer and Modeler, only activation option is enabled.
5	OBJ1	Active	Inactive	Active	Allowed	Not Allowed	If an object has multiple active versions such as, one in the Project Explorer and one in the Modeler, only take over option is enabled.
6	OBJ1	Active	Active	Inactive	Not Applicable	Allowed	If an object has single inactive version, and the object version in Modeler is inactive, only activation option is enabled.
7	OBJ1	Active	Inactive	Inactive	Not Allowed	Allowed	If an object has single active version, and the object version in Modeler is inactive, only activation option is enabled.

Scenario	Object	in Team	Provider	in SAP HANA Systems view	SAP HANA Systems view	Description
8	OBJ1	Active	Active	Active	Not Applicable (Redeploy)	If an object has multiple active versions, and the object version in Modeler is active, only take over activation (redeploy) option is enabled.

6.2.8 Description Mapping

Description mapping helps you to associate an attribute with another attribute, which describes it in detail. For example, when reporting via a Label Column, you can associate Region_ID with Region_Text.

For an attribute you can now maintain description mapping by selecting another attribute from the same model as *Label Column* in the *Semantics* node. The result is attribute description displaying as the label column in the data preview. The related columns appear side by side during data preview.

You can rename a label column attribute as <attribute>.description but not as <label column attribute.description>. For example, if product_text is the Label Column for product then, you can rename product_text to product.description but not as product_text.description.

Note

- On renaming a column as <attribute.description>, it is marked as *Hidden* and cannot be used in other places such as calculated columns, input parameters and so on.
- If you have created an object using the old editor (which supported the old style of description mapping) and try to open it using the new editor you will see a new column <attribute>.description (as an attribute) which is hidden and disabled. You can rename it maintain its properties and use it like other attributes.

6.2.9 Import BW Objects

You can import SAP Business Warehouse (SAP BW) models that are SAP HANA-optimized InfoCubes, Standard DataStore Objects, Query Snapshot InfoProviders, and InfoObjects of type Characteristics to the SAP HANA modeling environment.

Prerequisites

- You have implemented SAP Notes [1703061](#), [1759172](#), [1752384](#), [1733519](#), [1769374](#), [1790333](#), [1870119](#), [1994754](#), and [1994755](#).
- You have installed SAP HANA 1.0 SPS 05 Revision 50 or above.
- You have added BW schema in the SQL privileges for the Modeler user to import BW models.
- _SYS_REPO user has SELECT with GRANT privileges on the schema that contains the BW tables.

Context

Import SAP BW objects to expose it as SAP HANA models to the reporting tools.

Note

- You can only import those Standard DataStore objects that have *SID Generation* set to *During Activation*.
- For an InfoObject, you can import Characteristics having key figures as attributes.

Procedure

1. Open the *SAP HANA Modeler* perspective.
2. In the main menu, choose **File > Import**.
3. Expand the *SAP HANA Content* node.
4. Choose *SAP BW Models*, and choose *Next*.
5. Establish connection with your SAP BW system (underlying BW Application Server). In the *Provide Source System Details* page, enter the SAP BW system credentials and choose *Next*.

Note

To add new connection details, select *New Connection* option from the *Connection* dropdown list. The connection details are saved and are available as dropdown options on subsequent logons.

6. Optional Step: Provide SAProuter String

You can use SAProuter string to connect to the SAP BW System over the internet. You can obtain the SAProuter string information of your SAP BW system from your SAP Logon. In your SAP Logon screen, choose your **SAP BW system > Edit > Connection**.

7. Optional Step: Activate Secure Network Connections (SNC)

Select *Activate Secure Network Connections* and provide the *SNC Name* of your communication partner. You can use SNC to encrypt the data communication paths that exist between an SAP HANA Studio and your SAP BW system. You can obtain the SNC name of your SAP BW system from SAP Logon. In your SAP Logon screen, choose your **SAP BW system > Edit > Network**.

8. Select the target system (an SAP BW on SAP HANA) to, which you want to import the models, and choose *Next*.
9. Select the BW InfoProviders that you want to import and expose as SAP HANA information models.

Remember

In order to import the QuerySnapshot InfoProvider, make sure that the BW Query is unlocked in transaction RSDDDB, and an index is created via the same transaction before it can be used as InfoProviders.

10. Select the target package where you want to place the generated models, and analytic privileges.

Note

Your package selection is saved during the subsequent import. Hence, the next time you visit the same wizard you get to view the package that was selected previous time. You can though change the package where you want to import objects.

11. If you want to import the selected models along with the display attributes for IMO Cube and Standard DSO, select *Include display attributes*.
For InfoObjects all the attributes are added to the output and joined to their text tables if exists.
12. If you want to replace previously imported models in the target system with a new version, select *Overwrite existing objects*.
13. If you do not want to import the analysis authorizations associated with the selected InfoProviders, deselect *Generate InfoProvider based analytic privileges*.
14. If you want to import the role based analysis authorizations as analytic privileges, select *Generate Role based analytic privileges*, and choose *Next*.
If you have selected both the InfoProviders and InfoObjects, only authorizations set on InfoProviders can be imported after selecting the checkbox.
15. Select the roles to import the related analysis authorizations.
16. Choose *Finish*.

Note

While importing your SAP BW models, the SAP HANA system imports the column labels of these models in the language that you specify in its properties. However, in your SAP BW system, for any of the columns, if you do not maintain column labels in the language that you specify in your SAP HANA system properties, then those column labels appears as blank after import. If you want to check the default language for your SAP HANA system, then:

1. In the *Systems View*, select the SAP HANA system in which you are importing the models.
2. In the context menu, choose *Properties*.
3. In the *Additional Properties* tab, the dropdown list *Locale* specifies the language of objects, which you create in SAP HANA repository.

Results

The generated information models and analytic privileges are placed in the package selected above. In order to view the data of generated models, you need to assign the associated analytic privileges that are generated as part of the model import to the user. If these privileges are not assigned, user is not authorized to view the data.

Related Information

[Secure Network Communications \(SNC\)](#)

6.2.10 Group Related Measures

If your analytic view and calculation view has multiple measures and you want to organize them, for, example, segregate the planned measures with the actual measures, you can group the related measures in folders. These folders are called the display folders.

You can organize display folders in a hierarchical manner that is, by creating one display folder under the other.

To create display folders, select the [Display Folder](#) toolbar option in the [Column](#) panel of the [Semantics](#) node. In the [Display Folder](#) dialog create a new folder using the context menu option or using the toolbar option. Drag the required measures to the relevant folder. Note that one measure can be part of multiple display folders. Alternatively, you can associate a measure with a new or existing display folder by entering the value in the [Display Folder](#) property of the measure. If you enter a new value for this property a new display folder with the specified name is created.

Each measure is associated with the [Display Folder](#) property. The value for this property contains the fully qualified name of the display folder in which it appears. The fully qualified name of a display folder consists of the names of the display folders that represent the path to a given object. If the property contains the name of more than one display folder, indicating a hierarchy, each name is separated by a backslash character (\). If this property contains an empty string (""), the object is not associated with a display folder. The same measure can be part of multiple display folders. In such cases each folders should be separated by a semi colon (;). For example, if for the measure "Invoiced_amount" the value for Display Folder property is Reported\Amount, it means, Reported\Amount is a hierarchical display folder of "Invoiced_amount".

6.3 Additional Functionality for Information Views

After modeling information views or at design time you can perform certain additional functions, which helps improve the efficiency of modeling information views.

This section describes the different additional functions that SAP HANA modeler offers and how you can use these functions to efficiently model views.

Related Information

[Performance Analysis](#)

[Maintain Comments for Modeler Objects](#)

[Replacing Nodes and Data Sources](#)

[Renaming Information Views and Columns](#)

6.3.1 Create Level Hierarchies

In level hierarchies each level represents a position in the hierarchy. For example, a time dimension can have a hierarchy that represents data at the month, quarter, and year levels.

Context

Level hierarchies consist of one or more levels of aggregation. Attributes roll up to the next higher level in a many-to-one relationship and members at this higher level roll up into the next higher level, and so on, until they reach the highest level. A hierarchy typically comprises of several levels, and you can include a single level in more than one hierarchy. A level hierarchy is rigid in nature, and you can access the root and child node in a defined order only.

Procedure

1. Launch SAP HANA studio.
2. Open the required attribute view or graphical calculation view in the view editor.
3. Select the *Semantics* node.
4. Choose the *Hierarchies* tab.
5. Choose the icon  (Create).
6. Provide a name and description to the new hierarchy.
7. In *Hierarchy Type* dropdown list, select *Level Hierarchy*.
8. Define node style

The node style determines the node ID for the level hierarchy.

- a. In the *Node* section, choose *Add*.
 - b. In the *Node Style* dropdown list, select a value.
9. Create levels.
 - a. In the *Nodes* tab, choose *Add* to create a level.
 - b. In the *Element* dropdown list, select a column value for each level.
 - c. In *Level Type* dropdown list, select a required level type.

The level type specifies the semantics for the level attributes. For example, level type *TIMEMONTHS* indicates that the attributes are months such as, "January", February, and similarly level type *REGULAR* indicates that the level does not require any special formatting.

- d. In the *Order By* dropdown list, select a column value that modeler must use to order the hierarchy members.

Note

MDX client tools use attribute values to sort hierarchy members.

- a. In the *Sort Direction* dropdown list, select a value that modeler must use to sort and display the hierarchy members.

10. Define level hierarchy properties.

In the *Advanced* tab, you can define certain additional properties for your hierarchy.

- a. If you want to include the values of intermediate nodes of the hierarchy to the total value of the hierarchy's root node, in the *Aggregate All Nodes* dropdown list select True. If you set the *Aggregate All Nodes* value to False, modeler does not roll-up the values of intermediate nodes to the root node.

Note

The value of *Aggregate All Nodes* property is interpreted only by the SAP HANA MDX engine. In the BW OLAP engine, the modeler always counts the node values. Whether you want to select this property depends on the business requirement. If you are sure that there is no data posted on aggregate nodes, set the option to false. The engine then executes the hierarchy faster.

- b. In the *Default Member* textbox, enter a value for the default member.

This value helps modeler identify the default member of the hierarchy. If you do not provide any value, all members of hierarchy are default members.

- c. In the *Orphan Nodes* dropdown list, select a value.

This value helps modeler know how to handle orphan nodes in the hierarchy.

Note

If you select *Stepparent* option to handle orphan nodes, in the *Stepparent* text field, enter a value (node ID) for the step parent node. The step parent node must already exist in the hierarchy at the root level and you must enter the node ID according to the node style that you select for the hierarchy. For example if you select node style *Level Name*, the stepparent node ID can be [Level2].[B2]. The modeler assigns all orphan nodes under this node.

- d. In the *Root Node Visibility* dropdown list, select a value.

The value helps modeler know if it needs to add an additional root node to the hierarchy.

- e. If you want the level hierarchy to support multiple parents for its elements, select the *Multiple Parent* checkbox.

11. Create a Not Assigned Member, if required.

In attribute view or calculation views of type dimensions, you can create a new *Not Assigned Member* that captures all values in fact table, which do not have corresponding values in the master table. In level hierarchies, the not assigned member appears at each level of the hierarchy.

- a. Select the *Not Assigned Member* tab.
- b. If you want to capture values in the fact tables that do not have corresponding values in the master table, then in the *Not Assigned Members* dropdown list, select *Enable*.

By default, modeler does not provide a hierarchy member to capture such values. This means that, *Not Assigned Members* is disabled. You can either enable or choose *Auto Assign* to handle not assigned members.

Note

Selecting, *Auto Assign* to handle not assigned members impacts the performance of your calculation views. Select *Auto Assign* with caution.

- c. Provide a name and label to the hierarchy member.

This label value appears in reporting tools to capture not assigned members.

- d. If you want to drilldown this member in reporting tool, select the [Enable Drilldown](#) checkbox.
- e. If you want to use null convert values to process NULL values in the fact table, which do not have any corresponding records in the master table, select the [Null Value Processing](#) checkbox.

By default, modeler uses the string `_#_` as the null convert value. You can change this value in the [Name](#) field under the [Null Value Member Properties](#) section.

- f. Provide a label for the null value member.

This value appears in the reporting tools to capture null values.

Related Information

[Node Style](#)

[Level Hierarchy Properties](#)

[Root Node Visibility](#)

[Orphan Nodes](#)

[Query Shared Hierarchies](#)

6.3.2 Create Parent-Child Hierarchies

In parent-child hierarchies, you use a parent attribute that determines the relationship among the view attributes. Parent-child hierarchies have elements of the same type and do not contain named levels.

Context

Parent-child hierarchies are value-based hierarchies, and you create a parent-child hierarchy from a single parent attribute. You can also define multiple parent-child pairs to support the compound node IDs. For example, you can create a compound parent-child hierarchy that uniquely identifies cost centers with the following two parent-child pairs:

- CostCenter and ParentCostCenter and
- ControllingArea and ParentControllingArea,

A parent-child hierarchy is always based on two table columns and these columns define the hierarchical relationships amongst its elements. Others examples of parent-child hierarchies are bill of materials hierarchy (parent and child) or employee master (employee and manager) hierarchy.

Procedure

1. Launch SAP HANA studio.
2. Open the required attribute view or graphical calculation view in the view editor.
3. Select the *Semantics* node.
4. Choose the *Hierarchies* tab.
5. Choose the icon  (Create).
6. Provide a name and description to the new hierarchy.
7. In *Hierarchy Type* dropdown list, select *Parent-Child Hierarchy*.
8. Create parent-child elements
 - a. In the *Node* section, choose *Add*.
 - b. In the *Child column* dropdown list, select a column value as the child attribute.
 - c. In the *Parent column* dropdown list, select a column value as a parent attribute for the child column that you have selected.
 - d. If you want to place orphan nodes in the hierarchy under a step parent node, then in the *Stepparent column* dropdown list, enter a value (node ID) for the step parent node.
 - e. If you want to place the parent-child hierarchies under a root node, in the *Root Node* value help, select a value.
9. If you want to add additional attributes to execute the hierarchy, then
 - a. In *Additional Attributes* section, choose *Add*.
 - b. In the *Attributes* dropdown list, select an attribute value.
10. Define parent-child hierarchy properties.

In the *Advanced* tab, you can define certain additional properties for your hierarchy.

- a. If you want to include the values of intermediate nodes of the hierarchy to the total value of the hierarchy's root node, in the *Aggregate All Nodes* dropdown list select True. If you set the *Aggregate All Nodes* value to False, modeler does not roll-up the values of intermediate nodes to the root node.

Note

The value of *Aggregate All Nodes* property is interpreted only by the SAP HANA MDX engine. In the BW OLAP engine, the modeler always counts the node values. Whether you want to select this property depends on the business requirement. If you are sure that there is no data posted on aggregate nodes, set the option to false. The engine then executes the hierarchy faster.

- b. In the *Default Member* textbox, enter a value for the default member.

This value helps modeler identify the default member of the hierarchy. If you do not provide any value, all members of hierarchy are default members.

- c. In the *Orphan Nodes* dropdown list, select a value.

This value helps modeler know how to handle orphan nodes in the hierarchy.

Note

If you select *Stepparent* option to handle orphan nodes, then in the *Node* tab, enter a value (node ID) for the stepparent. The stepparent node must already exist in the hierarchy at the root level.

- d. In the *Root Node Visibility* dropdown list, select a value.

The value helps modeler know if it needs to add an additional root node to the hierarchy.

- e. Handling cycles in hierarchy

A parent-child hierarchy is said to contain cycles if the parent-child relationships in the hierarchy have a circular reference. You can use any of the following options to define the behavior of such hierarchies at load time.

Options	Description
Break up at load time	The nodes are traversed until a cycle is encountered. The cycles are broken-up at load time.
Traverse completely, then breakup	The nodes in the parent-child hierarchy are traversed once completely and then the cycles broken up.
Error	Displays error when a cycle is encountered.

- f. If you want the parent-child hierarchy to support multiple parents for its elements, select the *Multiple Parent* checkbox.

11. Order and sort hierarchy elements.

If you want to order and sort elements of a parent child hierarchy based on a column value,

- a. In the *Order By* section, choose *Add*.
- b. In the *Order By Column* dropdown list, select a column value that modeler must use to order the hierarchy members.
- c. In *Sort Direction* dropdown list, select a value that modeler must use to sort and display the hierarchy members.

Note

MDX client tools use attribute values to sort hierarchy members.

12. Enable hierarchy for time dependency

If elements in your hierarchy are changing elements (time dependent elements), you can enable the parent-child hierarchy as a time dependent hierarchy. In other words, if you are creating hierarchies that are relevant for specific time period, then enable time dependency for such hierarchies. This helps you display different versions on the hierarchy at runtime.

Note

Not all reporting tools support time dependent hierarchies. For example, time dependent hierarchies does not work with BI clients such as MDX or Design Studio.

- a. In the *Time Dependency* tab, select the *Enable Time Dependency* checkbox.
- b. In the *Valid From Column* dropdown list, select a column value.
- c. In the *Valid To Column* dropdown list, select a column value.

SAP HANA modeler uses *Valid From Column* and *Valid To Column* values as the validity time for the time dependent hierarchies.

13. If you want to use an input parameter to specify the validity of the time dependent hierarchy at runtime,
 - a. In the *Validity Period* section, select *Interval*.
 - b. In the *From Date Parameter* dropdown list, select an input parameter that you want to use to provide the valid from date at runtime.
 - c. In the *To Date Parameter* dropdown list, select an input parameter that you want to use to provide the valid to date at runtime.
14. If you want to use an input parameter to specify the key date at runtime,
 - a. In the *Validity Period* section, select *Key Date*.
 - b. In the *Key Date Parameter* dropdown list, select an input parameter value that you want to use to provide key date value at runtime.
15. Create a Not Assigned Member, if required.

In attribute views or calculation views of type dimensions, you can create a new *Not Assigned Member* that captures all values in fact table, which do not have corresponding values in the master table.

- a. Select the *Not Assigned Member* tab.
- b. If you want to capture values in the fact tables that do not have corresponding values in the master table, then in the *Not Assigned Members* dropdown list, select *Enable*.

By default, modeler does not provide a hierarchy member to capture such values. This means that, *Not Assigned Members* is disabled. You can either enable or choose *Auto Assign* to handle not assigned members.

Note

Selecting, *Auto Assign* to handle not assigned members impacts the performance of your calculation views. Select *Auto Assign* with caution.

- c. Provide a name and label to the hierarchy member.

This label value appears in reporting tools to capture not assigned members.
- d. If you want to drilldown this member in reporting tool, select the *Enable Drilldown* checkbox.
- e. If you want to use null convert values to process NULL values in the fact table, which do not have any corresponding records in the master table, select the *Null Value Processing* checkbox.

By default, modeler uses the string *_#_* as the null convert value. You can change this value in the *Name* field under the *Null Value Member Properties* section.

- f. Provide a label for the null value member.

This value appears in the reporting tools to capture null values.

Related Information

[Create Parent-Child Hierarchies \[page 368\]](#)

[Parent-Child Hierarchy Properties](#)

[Query Shared Hierarchies](#)

[Root Node Visibility](#)

[Orphan Nodes](#)

6.3.3 Input Parameters

Use input parameters to parameterize the view and to obtain the desired output when you run the view.

This means that the engine uses the parameter value that users provide at runtime, for example, to evaluate the expression defined for a calculated measure. The parameter value is passed to the engine through the **PLACEHOLDER** clause of the SQL statement. A parameter can only have a single value, for example, for currency conversion. However, when working with the **in()** function in filter expressions of calculation views, you can pass several values as an **IN List**. When defining the expression, quote the expression as described here:

For numerical type parameters

The filter expression of a calculation view CV1 is defined as follows:

```
in("attr", $$param$$)
```

Then pass several values as:

```
select ... from CV1( 'PLACEHOLDER' = ('$$var$$' = 'VAL1, VAL2, VAL3')
```

For string type parameters

The filter expression of a calculation view CV1 is defined as:

```
in("attr", $$param$$)
```

Then pass several values (with double quotes) as:

```
select ... from CV1( 'PLACEHOLDER' = ('$$var$$' = ''VAL1'', 'VAL2'', 'VAL3''))
```

The table here summarizes with some examples the input parameter expressions at design time and the query at runtime.

Input Parameter Data Type	Multiple Values	Expression	In Query
Integer	False	<code>in("ID", \$\$IP_1\$\$)</code>	<code>(placeholder."\$\$IP_1\$\$"=>1)</code>
Varchar	False	<code>in ("elem_2", '\$\$IP_1\$ \$')</code>	<code>(placeholder."\$\$IP_1\$ \$"=>'test')</code>
Varchar	False	<code>"elem_2" = '\$\$IP_1\$\$'</code>	<code>(placeholder."\$\$IP_1\$ \$"=>'test')</code>
Integer	True	<code>in("elem_3", \$\$IP_1\$\$)</code>	<code>(placeholder."\$\$IP_1\$ \$"=>'2,3')</code>
Varchar	True	<code>in("elem_2", \$\$IP_1\$\$)</code>	<code>(placeholder."\$\$IP_1\$ \$"=>'test')</code> Or <code>(placeholder."\$\$IP_1\$ \$"=>'test', 'test2')</code>

You use input parameters as placeholders, for example, during currency conversion, unit of measure conversion, or in calculated column expressions. When used in formulas, the calculation of the formula is based on the input that you provide at runtime during data preview.

The expected behavior of the input parameter when a value at runtime is not provided is as follows:

Default Value	Expected Behavior
Yes	Calculates the formula based on the default value
No	Results in error

The table implies that it is mandatory to provide a value for the input parameter at runtime, or assign a default value while creating the view, to avoid errors.

6.3.4 Assign Variables

You can assign variables to a filter at design time for obtaining data based on the values you provide for the variable. At runtime, you can provide different values to the variable to view the corresponding set of attribute data.

6.3.5 Using Currency and Unit of Measure Conversions

If measures in your calculation views or analytic views represent currency or unit values, associate them with currency codes or unit of measures. This helps you display the measure values along with currency codes or unit of measures at data preview or in reporting tools.

Associating measures with currency code or unit of measure is also necessary for currency conversion or unit conversions respectively.

Modeler performs currency conversions based on the source currency value, target currency value, exchange rate, and date of conversion. Similarly, it performs unit conversions based on the source unit and target unit.

Use input parameters in currency conversion and unit conversion to provide the target currency value, the exchange rate, the date of conversion or the target unit value at runtime.

Currency conversion or unit conversion are not supported for script-based graphical calculation views.

Related Information

[Associate Measures with Currency](#)

[Associate Measures with Unit of Measure](#)

[Create Input Parameters](#)

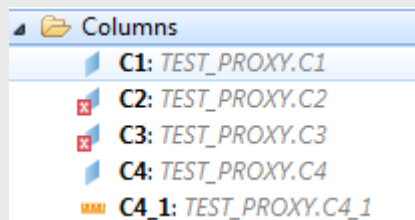
6.3.6 Manage Information Views with Missing Objects

If objects within an information view are missing, for example, if the objects or its references are deleted, then the information view is referred to as broken models. By using proxies, SAP HANA modeler helps you work with broken models and fix inconsistencies.

When you open broken models, the system displays red decorators for all missing objects, which are essential to activate the information view.

❁ Example

If you have defined an attribute view ATV1 on table T1 (C1, C2, C3) such that Attributes A1, A2, A3 is defined on columns C1, C2, C3 respectively. Now, if you remove column C2 and C3 from the table T1, then the attribute A3 becomes inconsistent. In such cases, the system injects proxies for C3, and when you open the attribute view in the editor, the system displays a red decorator for C2, C3, and an error marker for A3 to



indicate that it is inconsistent.

📌 Note

If the connection to SAP HANA system is not available, and if you try to open a view, then the system uses proxies for all required objects and opens the view in read-only mode. But, since the model is not broken, the red decorators and the error markers are not shown..

You can resolve inconsistencies in analytic views or attribute views or calculation views by performing one of the following:

- Deleting the missing objects, which the information view requires. This clears all references of missing object.
- Adjusting the mappings of inconsistent objects.
- Deleting inconsistent objects.

📌 Note

The system logs inconsistencies within information view in the *Problems* view of *SAP HANA Development* perspective.

6.4 Working with Views

6.4.1 Manage Editor Layout

You use this procedure to adjust the data foundation and logical view layout comprising user interface controls like, tables and attribute views in a more readable manner. This functionality is supported for attribute views and analytic views.

The options available are as follows:

Option	Purpose	Substeps
Auto Arrange	Use this option to arrange the user interface elements automatically.	In the editor tool bar, choose  (Auto Layout).
Show outline	Use this option to view an outline of the elements arranged so that, you do not have to navigate in the editor using horizontal and vertical scrollbars.	In the editor tool bar, choose  .
Highlight related tables in <i>Data Foundation</i>	Use this option if you want to view only those tables that are related to a table selected in the editor.	1. In the editor, right-click the selected table. 2. From the context menu, choose Highlight related tables.
Display	Use this option if you have a table with a large number of columns in the editor, and you want to view them in a way that meet your needs: for example, only the table name, or only joined columns, or the expanded form with all the columns.	1. In the editor, right-click the relevant table. 2. From the context menu, choose Display. 3. If you want to view only the table name, choose Collapsed. 4. If you want to view all the columns of the table, choose Expanded. 5. If you want to view only the joined columns of the table, choose Joins only.
Show Complete Name	Use this option to view the complete name of a truncated column.	1. In the Scenario pane, choose a view node. 2. In the Details pane, choose the required input. 3. In the context menu, choose Show Complete Name.
Show Description	Use this option to view the column description.	1. In the Scenario pane, choose a view node. 2. In the Details pane, choose the required input. 3. In the context menu, choose Show Description.

6.4.2 Validate Models

You can check if there are any errors in an information object and if the object is based on the rules that you specified as part of preferences.

For example, the "Check join: SQL" rule checks that the join is correctly formed.

Procedure

1. On the [Quick View](#) page, choose [Validate](#).
2. Select a system where you want to perform this operation.
3. From the [Available](#) list, select the required models that system must validate, and choose [Add](#).
4. Choose [Validate](#).

6.4.3 Maintain Search Attributes

You use this procedure to enable an attribute search for an attribute used in a view. Various properties related to attribute search are as follows:

- [Freestyle Search](#): Set to [True](#) if you want to enable the freestyle search for an attribute. You can exclude attributes from freestyle search by setting the property to [False](#).
- [Weights for Ranking](#): To influence the relevancy of items in the search results list, you can vary the weighting of the attribute. You can assign a higher or lower weighting (range 0.0 to 1.0). The higher the weighting of the attribute, the more influence it has in the calculation of the relevance of an item. Items with a higher relevance are located higher up the search results list. Default value: 0.5.

Note

To use this setting the property [Freestyle Search](#) must be set to [True](#).

- [Fuzzy Search](#): This parameter enables the fault-tolerant search. Default: [False](#).
- [Fuzziness Threshold](#): If you have to set the parameter Fuzzy Search to [True](#) you can fine-tune the threshold for the fault-tolerant search between **0** and **1**. Default: **0.8**

Note

We recommend using the default values for [Weights for Ranking](#) and [Fuzziness Threshold](#) to start with. Later on, you can fine-tune the search settings based on your experiences with the search. You can also fine-tune the search using feedback collected from your users.

6.4.4 Data Preview Editor

Use the data-preview editor to display and inspect raw data output or to view all attributes and measures in a graphical format.

The [Data Preview](#) editor includes the following tabs:

- Raw Data
- Distinct Values
- Analysis

Tab Title	Information Displayed	User Options
Raw Data	All attributes along with data in a table format.	• Filter data. For example, define filters on columns and filter the data based on company names. • Export data to different file formats to analyze them in other reporting tools.
Distinct values	All attributes along with data in a graphical format.	Basic data profiling
Analysis	All attributes and measures in a graphical format.	• Perform advance analysis using labels and value axis. For example, analyze sales based on country by adding Country to the labels axis and Sales to the value axis. • Use different charts to support analysis. You can view the data in the Chart, Table, Grid, and HTML formats and save the analysis as favorites. • Filter data. For example, define filters and filter the data based on company names.

📘 Note

If you refresh data in the [Analysis](#) tab, the data modeler clears the data in the [Raw Data](#) tab. Refresh the [Raw Data](#) tab to fetch the latest results.

6.4.5 Using Functions in Expressions

This section describes the functions, which you can use in expressions of column engine language. You create expressions, for example, while creating expressions for calculated attributes or calculated measures.

You can create expressions, for example in calculated columns, using the column engine (CS) language or the SQL language.

→ Recommendation

Use SQL expressions rather than column engine expressions, if possible. For hybrid scenarios, remember that the SAP Business Application Studio allows only SQL expressions to be created in calculation views that are newly created in the SAP HANA Cloud, SAP HANA database.

Note

Related SAP Notes. The SAP Note [2252224](#)  describes the differences between the CS and SQL string expression with respect to Unicode or multibyte encoding. The SAP Note [1857202](#)  describes the SQL execution of calculation views.

Related Information

[String Functions](#)

[Conversion Functions](#)

[Mathematical Functions](#)

[Date Functions](#)

[Miscellaneous Functions](#)

[Using Functions in Expressions \[page 377\]](#)

[Spatial Functions](#)

[Spatial Predicates](#)

6.4.6 Resolving Conflicts in Modeler Objects

You can resolve the conflicts between three different versions of a model by merging them with the help of 3-way merge feature. You can also compare two files for finding their differences with this feature. The common scenarios and the available options for the use of this feature are:

S.No.	Requirement	Option
1.	To compare two models in the Project Explorer to view their differences.	Compare With > Each Other
2.	To compare the inactive version of a model with the active version.	Compare With > Active Version

S.No.	Requirement	Option
3.	To resolve conflicts between the model versions encountered during activation in the following scenarios: - You modify a model in two SAP HANA studio instances and you commit and activate the model (one or several times) in the first instance. In the second instance when you try to activate the model you get an error message. - You modify a model in one of the SAP HANA studio instance, and commit and activate the model. If you modify the model in the other SAP HANA studio instance without updating it, you get an error while activating the model. - In a SAP HANA studio instance if you have an inactive model in the Project Explorer and an inactive version in the Modeler perspective. If you activate the model in the Modeler perspective, you get an error while activating the model from Project Explorer.	Team > Merge Tool Or Team > Resolve With

Compare ('Mini/ SAL...') *SQL Console 1 mini.MINI_D7_RENA... MINI_D7_RENAME.attr... Quick Launch VS3 - POC_ARCH.Cal... sap.hba.ecc.Busine...

Two-way compare of 'Mini/ SALES_ONLY_COPY.analyticview' with 'Mini/SALES_ONLY.analyticview'

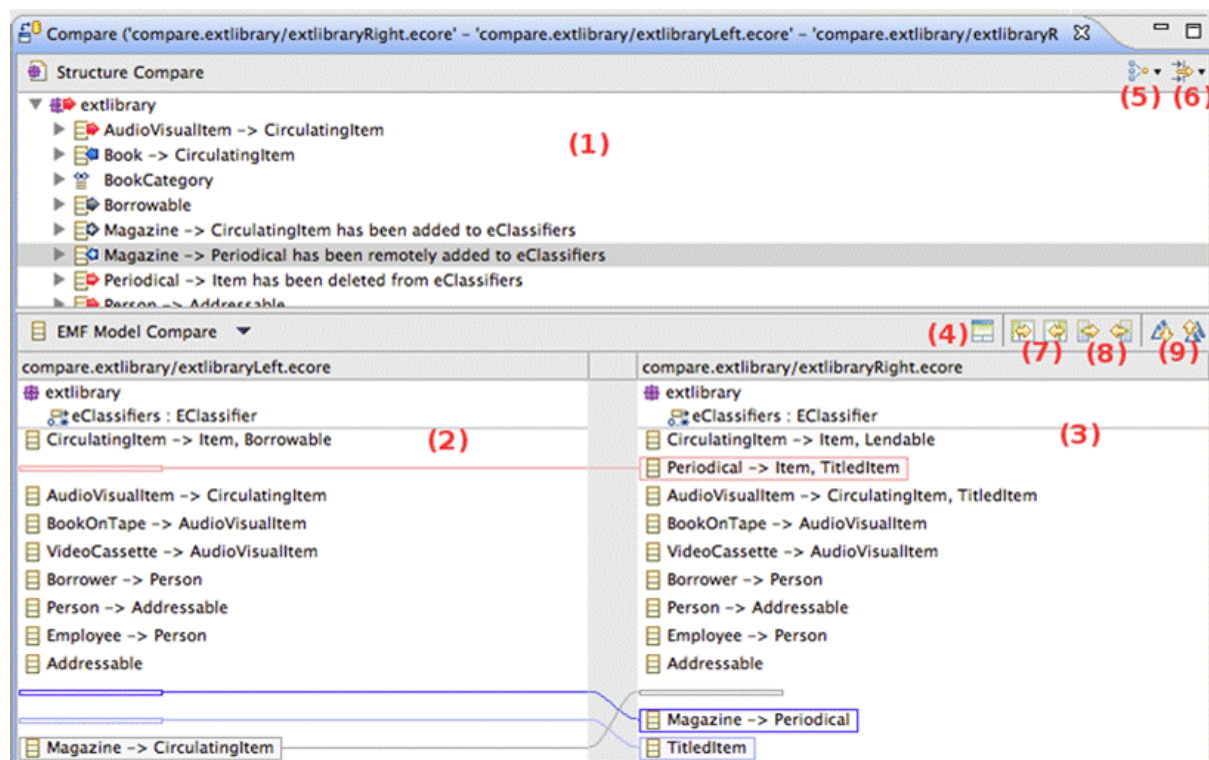
- 5 differences in resource %20SALES_ONLY_COPY.analyticview
 - 5 change(s) in model
 - 5 change(s) in View SALES_ONLY
 - Attribute name : Identifier in View SALES_ONLY_COPY has changed from 'SALES_ONLY' to 'SALES_ONLY_COPY'
 - 2 change(s) in Join Node Data Foundation
 - Column YEAR has been added
 - 1 change(s) in Input "MINI".SALES
 - 2 change(s) in Join Node Logical Join
 - Column YEAR has been added
 - 1 change(s) in Input Data Foundation
 - Mapping YEAR-> YEAR has been added

Visualization of Structural Differences

Mini/ SALES_ONLY_COPY.analyticview	Mini/SALES_ONLY.analyticview
platform/resource/Mini/%20SALES_ONLY_COPY.analyticview	platform/resource/Mini/SALES_ONLY.analyticview
View SALES_ONLY_COPY	View SALES_ONLY
Column CUSTOMER_ID	Column CUSTOMER_ID
Column YEAR	Column REVENUE
Column REVENUE	Simple Type
Join Node Data Foundation	Currency Conversion true
Column CUSTOMER_ID	Join Node Data Foundation
Column YEAR	Column CUSTOMER_ID
Simple Type	Column REVENUE
Column REVENUE	Input "MINI".SALES
Input "MINI".SALES	Join Node Logical Join
Join Node Logical Join	Column CUSTOMER_ID
Column CUSTOMER_ID	Column REVENUE
Column YEAR	Input Data Foundation
Simple Type	
Column REVENUE	
Simple Type	
Currency Conversion true	
Input Data Foundation	
Mapping CUSTOMER_ID-> CUSTOMER_ID	
Mapping YEAR-> YEAR	
Mapping REVENUE-> REVENUE	

Differences Denotation

The merge editor components are depicted below:



1. Overview of the differences detected between the given two (or three) models.
2. First version of the compared models.
3. Second version of the compared models.
4. This button will only be visible in the case of three-way comparisons (for example, comparing with a remote repository). It will make a third version of the compared model (the common ancestor of the two others) visible in the interface.
5. This button will allow you to group differences together in the structural view. For example, grouping all "Additions" or "Deletions" together.
6. This button will allow you to filter some differences out of the view according to a set predicate. For example, filtering out all "Additions" or "Moves".
7. Allows you to merge all non-conflicting differences (left to right, or right to left) at once.
8. Allows you to merge the single, currently selected difference in a given direction (left to right, or right to left).
9. Allows you to navigate through the detected differences.

6.5 Create Decision Tables

You use this procedure to create a decision table to model related business rules in a tabular format for decision automation. You can use decision tables to manage business rules, data validation, and data quality rules.

You use this procedure to create a decision table to model related business rules in a tabular format for decision automation. You can use decision tables to manage business rules, data validation, and data quality rules.

rules, without needing any knowledge of technical languages such as SQL Script or MDX. A data architect or a developer creates the decision table and activates it. The active version of the decision table can be used in applications.

Prerequisites

This task describes how to create a decision table. Before you start this task, note the following prerequisites:

- You must have access to an SAP HANA system.
- To activate and validate the decision table, the `_SYS_REPO` user requires the SELECT, EXECUTE, and UPDATE privileges on your schema.
- If you are using the SAP HANA Development perspective, you must ensure the following prerequisites are also met:
 - You must have already created a development workspace.
 - You must have checked out a package.
 - You must have created and shared a project so that the newly created files can be committed to (and synchronized with) the repository.

Note

For more information about projects, repository workspaces, and sharing of projects, see *Using SAP HANA Projects* in the *SAP HANA Developer Guide for SAP HANA Studio*.

Create a Decision Table

You can create a decision table by using one of the following options:

- If you are in the *SAP HANA Modeler* perspective, perform the following steps:
 1. In the *SAP HANA Modeler* perspective, expand **<System Name> > Content > <Package Name>**.
 2. In the context menu of the package, choose **New > Decision Table**.
 3. In the *New Decision Table* dialog box, enter a name and description for the decision table.
 4. To create a decision table from scratch or from an existing decision table, perform the following substeps:

Scenario	Substeps
Create a decision table from scratch	1. Choose <i>Create New</i>. 2. Choose <i>Finish</i>.
Create a decision table from an existing decision table	1. Choose <i>Copy From</i>. 2. Browse the required decision table. 3. Choose <i>Finish</i>.

- If you are in the *SAP HANA Development* perspective, perform the following steps:
 1. Go to the *Project Explorer* view in the *SAP HANA Development* perspective, and select the project.

2. In the context menu of the selected project, choose ► **New** ► **Other...** ►

ⓘ Note

You can also create a decision table from the *File* menu. Choose ► **New** ► **Other...** ►

3. In the popup wizard, open *SAP HANA* and expand ► **Database Development** ► **Modeler** ►.
 1. Select **Decision Table**.

ⓘ Note

You can also search for the decision table directly by using the search box in the wizard.

2. Choose **Next**.
 1. In the *New Decision Table* dialog, choose **Browse** to choose the project under which you want to create your decision table. Enter a name and description.

ⓘ Note

If the project is shared, the Package field specifies the package that is associated with the project.

2. Choose **Finish**.

The decision table editor opens. It consists of three panes: Scenario, Details, and Output.

- The *Scenario* pane of the editor consists of the *Decision Table* and *Data Foundation* nodes. Selecting any of these nodes shows the specific node information in the *Details* pane.
- The *Details* pane of the *Data Foundation* node displays the tables or information models used for defining the decision table. The *Details* pane of the *Decision Table* node displays the modeled rules in tabular format.
- The *Output* pane displays the vocabulary, conditions, and actions, and allows you to perform edit operations. Expand the vocabulary node to display the parameters, attributes, and calculated attributes sub-nodes. In the *Output* pane, you can also view properties of the selected objects within the editor.

Related Information

[SAP HANA Developer Guide for SAP HANA Studio \[page 8\]](#)

6.5.1 Changing the Layout of a Decision Table

You use this procedure to change the decision table layout by arranging the condition and action columns. By default, all the conditions appear as vertical columns in the decision table. You can choose to mark a condition as a horizontal condition, and view the corresponding values in a row. The evaluation order of the conditions is such that the horizontal condition is evaluated first, and then the vertical ones.

Note

You can only change the layout of a decision table if it has more than one condition. You can mark only one condition as a horizontal condition.

Procedure

Mark as Horizontal Condition

1. Select the *Decision Table* node.
2. In the context menu of the *Details* pane, choose *Change Layout*.
3. If you want to view a condition as a horizontal condition, in the *Change Decision Table Layout* dialog, select the *Table Has Horizontal Condition (HC)* checkbox.

Note

The first condition in the list of conditions is marked as horizontal by default.

4. Choose *OK*.
5. Save the changes.

Note

You can also set a condition as horizontal from the context menu of the condition in the *Output* pane. You can also arrange the conditions and actions in the desired sequence in the *Output* pane by using the navigation buttons in the toolbar.

Rearranging Conditions and Actions

1. Select the *Decision Table* node.
2. In the context menu of the *Details* pane, choose *Change Layout*.
3. In the *Conditions and Actions* section, choose the options on the right-hand side of the dialog box to arrange the conditions and actions in the desired sequence.

The following options are available for arranging the conditions in a sequence:

- *Move Condition to Top*
- *Move Condition Up*
- *Move Condition Down*
- *Move Condition to Bottom*

Note

You can also arrange the sequence by using the navigation buttons at the top of the *Output* pane.

6.5.2 Using Parameters in a Decision Table

You use this procedure to create a parameter that can be used to simulate a business scenario. You can use parameters as conditions and actions in the decision table at design time. Parameters used as conditions determine the set of physical table rows to be updated based on the parameter value that you provide at runtime during the procedure call. Parameters used as actions simulate the physical table without updating it.

The following parameter types are supported:

Type	Description
Static List	Use this if the value of a parameter comes from a user-defined list of values.
Empty	Use this if the value of a parameter could be any of the selected data types.

Example

Consider a sales order physical table with column headers as follows:

ID	Name	Supplier	Model	Price	Quantity

If you want to evaluate *Discount* based on the *Quantity* and *Order Amount*, you can create two parameters: *Order Amount* and *Discount*. Use *Quantity* and *Order Amount* as the condition, and *Discount* as the action. The sample decision table could look like this:

Quantity	Order Amount	Discount
>5	50000	10
>=10	100000	15

Procedure

1. Create a Parameter

1. In the *Output* panel, select the *Parameters* node.
2. From the context menu, choose *New* and do the following:
 1. Enter a name and description.
 2. Select the required data type from the dropdown list.
 3. Enter the length and scale as required.
 4. Choose the required *Type* from the dropdown list.

Note

If you have selected *Static List* for *Type*, choose *Add* in the *List of Values* section to add values. You can also provide an alias for the enumeration value.

5. Choose *OK*.

2. Use Parameter as Condition or Action

1. In the *Output* panel, select the *Parameters* node.
2. From the context menu of the parameter, choose *Add as Conditions/ Add as Actions*.

6.5.3 Using Calculated Attributes in Decision Tables

Context

You use this procedure to create calculated attributes that can be used as conditions in a decision table.

You can create a calculated attribute to perform a calculation using the existing attributes, parameters, and SQL functions.

Procedure

1. In the *Output* panel, select the *Calculated Attributes* node.
2. From the context menu, choose *New* and do the following:
 - a. Enter a name and description.
 - b. Select the required data type, length, and scale.
 - c. In the expression editor, enter the expression. For example, you can write a formula such as ("*NAME*" = "*FIRST_NAME*" + "*LAST_NAME*"). This expression is an example of the string concatenation function, which is used to derive the name of a person by using the first name and last name values from the table fields.

Note

You can also create the expression by dragging and dropping the expression elements from the options at the bottom of the editor. Only arithmetic operators and SQL functions are supported for expression creation.

3. Choose *OK*.
4. Add the required calculated attribute as a condition.

6.6 Generate Object Documentation

Use this procedure to capture the details of an information model or a package in a single document. This helps you view the necessary details from the document, instead of referring to multiple tables. The following table specifies the details that you can view from the document.

Type	Description
Attribute View	General object properties, attributes, calculated attributes (that is, calculated columns of type attribute), data foundation joins, cross references, and where-used
Analytic View	General object properties, private attributes, calculated attributes (that is, calculated columns of type attribute), attribute views, measures, calculated measures (that is, calculated columns of type measure), restricted measures (that is, restricted columns), variables, input parameters, data foundation joins, logical view joins, cross references, and where-used
Calculation View	General object properties, attributes, calculated attributes, measures, calculated measures, counters, variables, input parameters, calculation view SQL script, cross references, and where-used
Package	Sub-packages, general package properties, and list of content objects

Procedure

1. From the [Quick View](#) pane, choose [Auto Documentation](#).
2. Select a system where you want to perform this operation.
3. In the [Select Content Type](#) field, select one of the following options as required:

Option	Description
Model Details	To generate documentation for models such as attribute, analytic, and calculation views.
Model List	To generate documentation for packages.

4. Add the required objects to the [Target](#) list.
5. Browse the location where you want to save the file.
6. Choose [Finish](#).

7 Developing Procedures

SQL in SAP HANA includes extensions for creating procedures, which enables you to embed data-intensive application logic into the database, where it can be optimized for performance (since there are no large data transfers to the application and features such as parallel execution is possible). Procedures are used when other modeling objects, such as views, are not sufficient; procedures are also often used to support the database services of applications that need to write data into the database.

Reasons to use procedures instead of standard SQL, include:

- SQL is not designed for complex calculations, such as for financials.
- SQL does not provide for imperative logic.
- Complex SQL statements can be hard to understand and maintain.
- SQL queries return one result set. Procedures can return multiple result sets.
- Procedures can have local variables, eliminating the need to explicitly create temporary tables for intermediate results.

Procedures can be written in the following languages:

- SQLScript: The language that SAP HANA provides for writing procedures.
- R: An open-source programming language for statistical computing and graphics, which can be installed and integrated with SAP HANA.

There are additional libraries of procedures, called Business Function Library and Predictive Analysis Library, that can be called via SQL or from within another procedure.

HANA Database Explorer

HANA Database Explorer provides a comprehensive set of development tools that allow you to evaluate, revise, and optimize stored procedures. You can browse through the objects in the schema to locate the procedures, from there, a number of options are available from the context menu. Features include a code editor for running and testing procedures as well as debugging and SQLScript analysis tools. Refer to the documentation sections on Database Explorer in the *SAP HANA Administration Guide* for more details.

SQL Extensions for Procedures

SQL includes the following statements for enabling procedures:

- **CREATE TYPE**: Creates a table types, which are used to define parameters for a procedure that represent tabular results. For example:

```
CREATE TYPE tt_publishers AS TABLE (  
    publisher INTEGER,  
    name VARCHAR(50),  
    price DECIMAL,
```

```
cnt INTEGER);
```

- **CREATE PROCEDURE:** Creates a procedure. The **LANGUAGE** clause specifies the language you are using to code the procedure. For example:

```
CREATE PROCEDURE ProcWithResultView(IN id INT, OUT o1 CUSTOMER)
LANGUAGE SQLSCRIPT READS SQL DATA WITH RESULT VIEW ProcView AS
BEGIN
    o1 = SELECT * FROM CUSTOMER WHERE CUST_ID = :id;
END;
```

- **CALL:** Calls a procedure. For example:

```
CALL getOutput (1000, 'EUR', NULL, NULL);
```

Related Information

[Create and Edit Procedures \[page 391\]](#)

[Open the SAP HANA Database Explorer \(SAP HANA Cockpit\)](#)

7.1 SQLScript Security Considerations

You can develop secure procedures using SQLScript in SAP HANA by observing the following recommendations.

Using SQLScript, you can read and modify information in the database. In some cases, depending on the commands and parameters you choose, you can create a situation in which data leakage or data tampering can occur. To prevent this, SAP recommends using the following practices in all procedures.

- Mark each parameter using the keywords **IN** or **OUT**. Avoid using the **INOUT** keyword.
- Use the **INVOKER** keyword when you want the user to have the assigned privileges to start a procedure. The default keyword, **DEFINER**, allows only the owner of the procedure to start it.
- Mark read-only procedures using **READS SQL DATA** whenever it is possible. This ensures that the data and the structure of the database are not altered.

→ Tip

Another advantage to using **READS SQL DATA** is that it optimizes performance.

- Ensure that the types of parameters and variables are as specific as possible. Avoid using **VARCHAR**, for example. By reducing the length of variables you can reduce the risk of injection attacks.
- Perform validation on input parameters within the procedure.

Dynamic SQL

In SQLScript you can create dynamic SQL using one of the following commands: **EXEC** and **EXECUTE IMMEDIATE**. Although these commands allow the use of variables in SQLScript where they might not be

supported. In these situations you risk injection attacks unless you perform input validation within the procedure. In some cases injection attacks can occur by way of data from another database table.

To avoid potential vulnerability from injection attacks, consider using the following methods instead of dynamic SQL:

- Use static SQL statements. For example, use the static statement, `SELECT` instead of `EXECUTE IMMEDIATE` and passing the values in the `WHERE` clause.
- Use server-side JavaScript to write this procedure instead of using SQLScript.
- Perform validation on input parameters within the procedure using either SQLScript or server-side JavaScript.
- Use `APPLY_FILTER` if you need a dynamic `WHERE` condition
- Use the SQL Injection Prevention Function

Escape Code

You might need to use some SQL statements that are not supported in SQLScript, for example, the `GRANT` statement. In other cases you might want to use the Data Definition Language (DDL) in which some `<name>` elements, but not `<value>` elements, come from user input or another data source. The `CREATE TABLE` statement is an example of where this situation can occur. In these cases you can use dynamic SQL to create an escape from the procedure in the code.

To avoid potential vulnerability from injection attacks, consider using the following methods instead of escape code:

- Use server-side JavaScript to write this procedure instead of using SQLScript.
- Perform validation on input parameters within the procedure using either SQLScript or server-side JavaScript.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SQL Injection Prevention Functions](#)

7.2 Create and Edit Procedures

The SAP HANA SQLScript editor allows you to create, edit, and activate procedures.

Prerequisites

- You have created a development workspace. For more information, see *Create a Repository Workspace*.
- You have checked out a package.

Note

After checking out a package that contains active procedures, you can modify and debug the procedures.

- You have created and shared a project. For more information, see *Using SAP HANA Projects*.

Note

You can also share your project after you create your procedure.

- To enable semantic code completion, you must have the following user role permissions:
 - `sap.hana.xs.dt.base::restapi`
 - `sap.hana.xs.ide.roles::Developer`

Procedure

1. Open the *New Stored Procedure* wizard.
 - a. Go to the *Project Explorer* view in the *SAP HANA Development* perspective, right-click on the file name, choose **New** **Other** **SAP HANA** **Database Development** **Stored Procedure**. Click *Next*. The *New Stored Procedure* wizard appears.
 - b. Enter or select the parent folder, enter the file name, select *Text (.hdbprocedure)* for the file format, select the target schema, and click *Finish*. The 📄 icon shows that your procedure is created locally.

Note

The *XML (.procedure)* file format is compatible with the Modeler Procedure editor, but may not support new SQLScript features. You should also use this format if you want to create a procedure template instance.

The editor opens containing a default template for the procedure. The design-time procedure name is generated in a shared project containing the full path. In an unshared project, the full function name must be added manually. In the *Properties* view, you see the properties of your procedure, such as *Access Mode*, *Name*, and *Language*.

2. Confirm the project is shared.

If you have not yet shared your project, right-click the project name, choose **Team > Share Project**. The **Share Project** wizard appears. Click **Finish**. The  icon shows that your procedure is not committed and not activated.

3. Write a new procedure or make changes to an existing one.

Begin writing your code inside your new procedure and save it locally. The syntax is checked simultaneously and is highlighted. Auto-completion of the syntax appears as you type or by using the Semantic Code Completion feature.

Note

You can only write one stored procedure per file. The file name and the procedure name must be the same. Only SQLScript language is supported for **Text (.hdbprocedure)** procedures.

To enable Semantic Code Completion:

- a. Position the cursor where you want to insert an object.
- b. Press **CTRL** + **Space Bar**.

A suggested list of valid objects appear.

Note

Text based searches display the object names that begin with and contain the entered text. Searches are asynchronous, the suggested list is updated in parallel to the user's refined textual input.

- c. Use the arrow keys to scroll through the list, click **Enter** to select the object, or **Esc** to close the code completion window without selecting an object.

4. Confirm the procedure is Committed.

Confirm the procedure is synchronized to the repository as a design time object and the  icon shows that your procedure is committed. If not, click **Save**, right-click and select **Team > Commit**.

5. Activate the procedure.

When you have finished writing your procedure and you are ready to activate it, right-click the procedure, choose **Team > Activate**. Your procedure is created in the catalog as a runtime object and the  icon shows that your procedure is activated. This allows you and other users to call the procedure and debug it.

If an error is detected during activation, an error message appears in the **Problems** view.

→ Tip

You can also activate your procedure at the project and folder levels.

Related Information

[Maintain a Repository Workspace \[page 63\]](#)

[Using SAP HANA Projects \[page 62\]](#)

[SAP HANA Repository Packages and Namespaces \[page 74\]](#)

[The SAP HANA Development Perspective \[page 31\]](#)

7.2.1 Define and Use Table Types in Procedures

You can use a table type to define parameters for a procedure; the table type represents tabular results.

Prerequisites

- Access to the SAP HANA repository

Context

If you define a procedure that uses data provided by input and output parameters, you can use table types to store the parameterized data. These parameters have a type and are either based on a global table (for example, a catalog table), a global **table type**, or a local (inline) table type. This task shows you two ways to use the `.hdbprocedure` syntax to define a text-based design-time procedure artifact; the parameterized data for your procedure can be stored in either of the following ways:

- Global
In an externally defined (and globally available) table type, for example, using the Core Data Service (CDS) syntax
- Local:
In a table type that is defined inline, for example, in the procedure itself

Procedure

1. Create a procedure that uses data provided by a local (inline) table type.

To define a text-based design-time procedure, use the `.hdbprocedure` syntax. The procedure in this example stores data in a local table type defined **inline**, that is; in the procedure itself.

Note

If you plan to define a **global** table type (for example, using CDS) you can skip this step.

- a. Create a design-time artifact called `get_product_sale_price.hdbprocedure` and save it in the repository.
- b. Add the following code to the new repository artifact `get_product_sale_price.hdbprocedure`.

→ Tip

The table used to store the parameterized data is defined inline, in the procedure's OUT parameter.

```
PROCEDURE
SAP_HANA_EPM_NEXT."sap.hana.democontent.epmNext.procedures::get_product_sale_price" (
    IN im_productid NVARCHAR(10),
    OUT ex_product_sale_price table (
        "PRODUCTID" nvarchar(10),
        "CATEGORY" nvarchar(40),
        "PRICE" decimal(15,2),
        "SALEPRICE" decimal(15,2) ) )
LANGUAGE SQLSCRIPT
SQL SECURITY INVOKER
DEFAULT SCHEMA SAP_HANA_EPM_NEXT
READS SQL DATA AS
BEGIN
```

- c. Save and activate the new (hdb)procedure artifact.
2. Define a global table type using Core Data Services (CDS).

If you want to define a **global** table type to store data for your use by your procedure, you can use the CDS syntax to define the global table type, as illustrated in the following example:

Note

This is only required if you want to use a global table type. If you plan to define a table type inline, you can skip this step.

- a. Create a design-time artifact called `GlobalTypes.hdbdd` and save it in the repository.
- b. Add the following code to the new repository artifact `GlobalTypes.hdbdd`.

```
namespace sap.hana.democontent.epmNext.data;
@Schema: 'SAP_HANA_EPM_NEXT'
context GlobalTypes {
    type tt_product_sale_price {
        PRODUCTID: String(10);
        CATEGORY: String(40);
        PRICE: Decimal(15,2);
        SALEPRICE: Decimal(15,2);
    };
};
```

- c. Save and activate the new CDS table type.

This generates a table type called `GlobalTypes.tt_product_sale_price` in the package `sap.hana.democontent.epmNext.data`. You use this path to reference the table type in your procedure.

3. Create the procedure that uses data provided by a global table type.

To define a text-based design-time procedure, use the `.hdbprocedure` syntax. The procedure in this example stores data in a table with the structure defined in the CDS global data type `tt_product_sale_price`.

Note

This is only required if you want to use a global table type. If you plan to define a table type inline, you can skip this step.

- a. Create a design-time artifact called `get_product_sale_price.hdbprocedure` and save it in the repository.
- b. Add the following code to the new repository artifact `get_product_sale_price.hdbprocedure`.

→ Tip

The OUT parameter refers to the CDS type `tt_product_sale_price` defined in the CDS document `GlobalTypes.hdbdd`.

```
PROCEDURE
SAP_HANA_EPM_NEXT."sap.hana.democontent.epmNext.procedures::get_product_sale_price" (
    IN im_productid NVARCHAR(10),
    OUT ex_product_sale_price SAP_HANA_EPM_NEXT."
sap.hana.democontent.epmNext.data::GlobalTypes.tt_product_sale_price")
LANGUAGE SQLSCRIPT
SQL SECURITY INVOKER
DEFAULT SCHEMA SAP_HANA_EPM_NEXT
READS SQL DATA AS
BEGIN
```

- c. Save and activate the new (hdb)procedure artifact.

7.2.2 Tutorial: Create an SQLScript Procedure that Uses Imperative Logic

SQLScript procedures can make use of standard SQL statements to build a query that requests data and returns a specified result set.

Prerequisites

To complete this exercise successfully, bear in mind the following prerequisites:

- You have the user credentials required to log on to SAP HANA
- You have installed the SAP HANA studio
- You have a shared SAP HANA project available (preferably of type *XS Project*).
- The shared project contains a folder called **Procedures**.
- You have installed the SAP HANA Interactive Education (*SHINE*) **HCODEMOCONTENT** delivery unit (DU); this DU contains the tables and views that you want to consume with the procedure you build in this tutorial.
- You have generated data to populate the tables and views provided by the *SHINE* delivery unit and used in this tutorial. You can generate the data with tools included in the *SHINE* delivery unit.

📌 Note

You might have to adjust the paths in the code examples provided to suit the package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration tables and services) referenced in the tutorial.

Context

The stored procedure you create in this tutorial uses standard SQL statements (for example, SELECT statements) and some imperative logic constructs to determine the sale price of a product based on the product category.

Procedure

1. Open the SAP HANA studio.

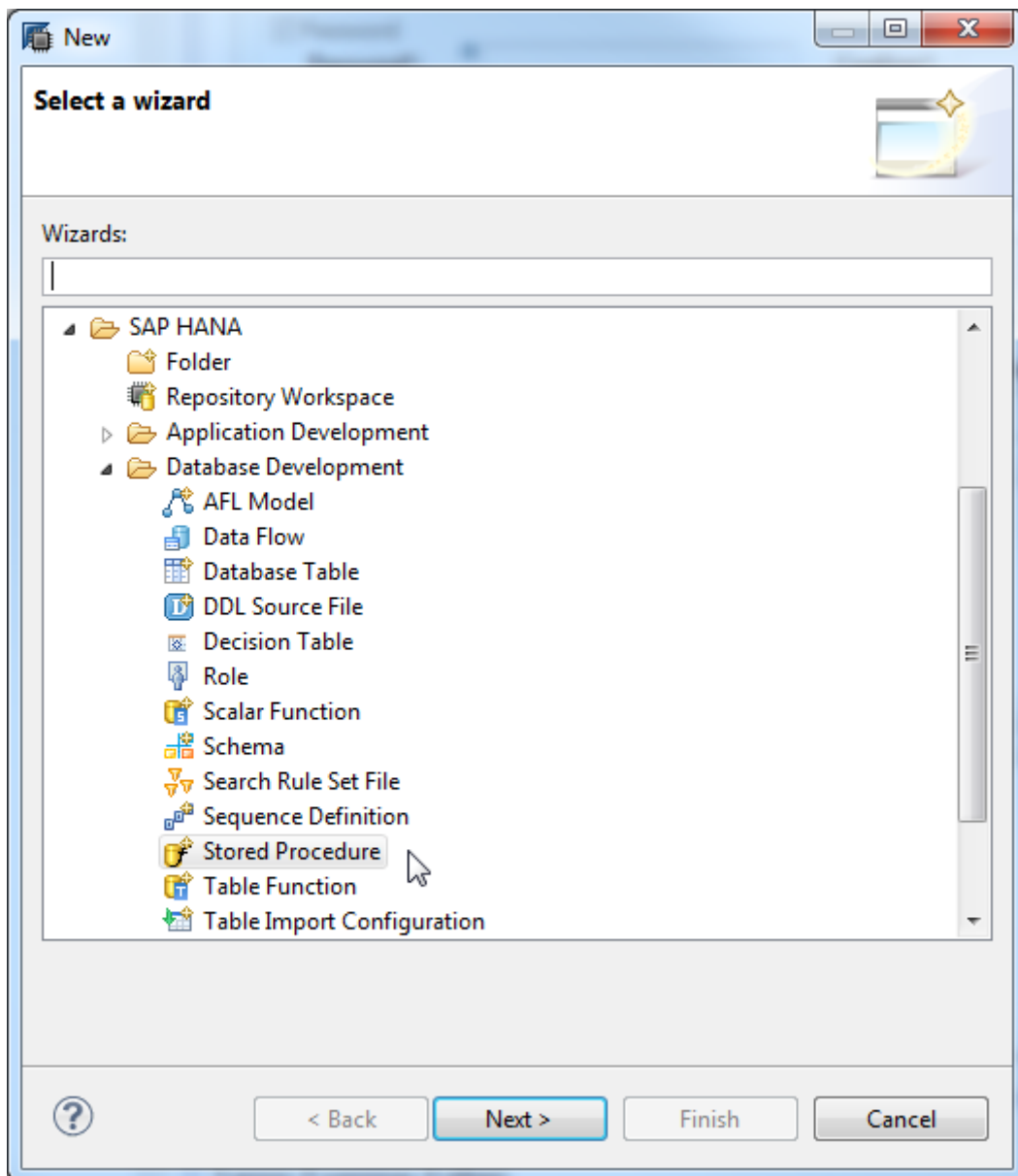
Switch to the *SAP HANA Development* perspective, open the *Project Explorer* view, and navigate to the shared project in which you want to create the new stored procedure.

2. Create the file that will contain the stored procedure.

If not already available, create a new folder (package) called `procedures` in the selected project.

- a. Start the *Create New Procedure* wizard.

In the *Project Explorer* view, right-click the `procedures` folder and choose ► *New* ► *Other...* from the context-sensitive pop-up menu. In the *Select a wizard* dialog, choose ► *SAP HANA* ► *Database Development* ► *Stored Procedure* ► .

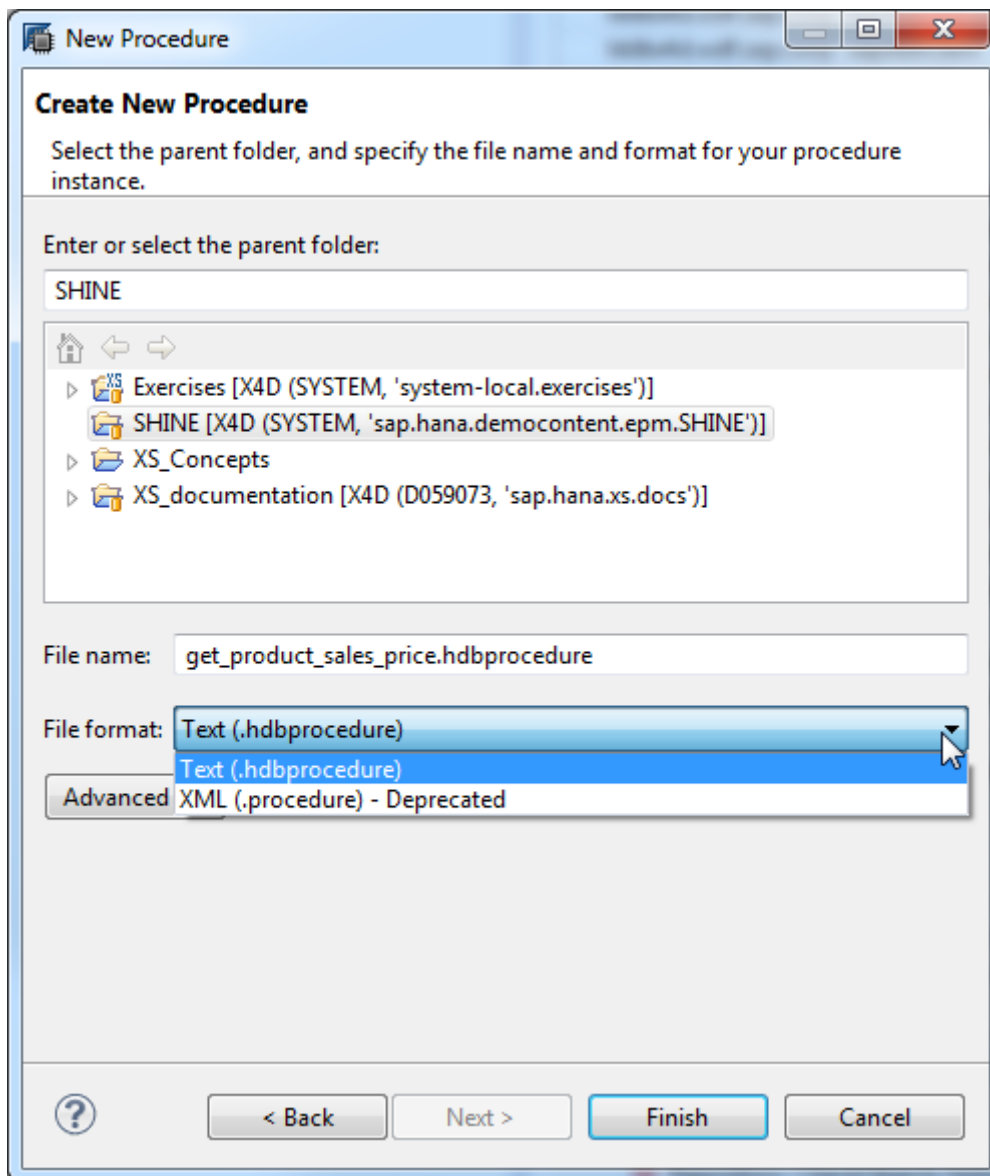


- b. Type the name of the new stored procedure.

Type **get_product_sales_price** in the *File name* box and choose *Text (.hdbprocedure)* in the *File format* drop-down menu.

→ Tip

The file-creation wizard adds the suffix (*.hdbprocedure*) automatically.



- c. Choose *Finish* to create the stored procedure and open it in the SAP HANA studio's embedded SQLScript Editor.
3. Define the new stored procedure.
This procedure uses standard SQL statements and some imperative logic constructs to determine the sale price of a product based on the product category.
- a. In the *SQLScript Editor*, define details of the stored procedure.
Use the following code to define the stored procedure.

```

PROCEDURE SAP_HANA_DEMO.get_product_sales_price (
    IN productid NVARCHAR(10),
    OUT product_sale_price
SAP_HANA_DEMO."sap.hana.democontent.epm.data::EPM.Procedures.tt_product_sale_price" )
    LANGUAGE SQLSCRIPT
    SQL SECURITY INVOKER
    READS SQL DATA AS
BEGIN
/*****

```

```

Write your procedure logic
*****/
declare lv_category nvarchar(40) := null;
declare lv_discount decimal(15,2) := 0;
lt_product = select PRODUCTID, CATEGORY, PRICE
              from "sap.hana.democontent.epm.data::EPM.MasterData.Products"
              where PRODUCTID = :productid;
select CATEGORY into lv_category from :lt_product;
if      :lv_category = 'Notebooks' then
    lv_discount := .20;
elseif :lv_category = 'Handhelds' then
    lv_discount := .25;
elseif :lv_category = 'Flat screens' then
    lv_discount := .30;
elseif :lv_category like '%printers%' then
    lv_discount := .30;
else
    lv_discount := 0.00; -- No discount
end if;
product_sale_price =
    select PRODUCTID, CATEGORY, PRICE,
           PRICE - cast((PRICE * :lv_discount) as decimal(15,2))
           as "SALEPRICE" from :lt_product;
END;

```

- b. Save the changes you have made to the new stored procedure.
- c. Activate the new stored procedure in the SAP HANA Repository.

In the *Project Explorer* view, right-click the new *get_product_sales_price* procedure and choose ► *Team* ► *Activate...* from the context-sensitive menu.

4. Test the new stored procedure using SAP HANA studio's embedded SQL console.

- a. Start the *SQL Console*.

In the *Project Explorer* view, right-click the *SAP HANA System Library* node and choose ► *SQL Console* from the context-sensitive menu.

- b. Call the new stored procedure.

Enter the following SQL statement (adjusting the path `sap.hana...` to the new procedure if necessary) and choose *Execute*.

```

call
SAP_HANA_DEMO."sap.hana.democontent.epm.Procedures::get_product_sales_price"
(
 productid => 'HT-1000',  product_sale_price => ? );

```

X4D (SYSTEM) sap.corp 00					
SQL SQL Result					
<pre>call SAP_HANA_DEMO."sap.hana.democontent.epm.Procedures::get_product_sales_price"(productid => 'HT-1000', product_sale_price => ?)</pre>					
	PRODUCTID	CATEGORY	PRICE	SALEPRICE	
1	HT-1000	Notebooks	956	764,8	

7.3 Create Scalar and Table User-Defined Functions

You can create, edit, and activate design-time scalar and table user-defined functions (UDF). These functions are added to a SELECT statement in the body of a stored procedure.

Procedure

1. Open a *New Scalar Function* or *New Table Function* wizard.
Go to the *Project Explorer* view in the *SAP HANA Development* perspective, right-click on the file name, choose ► *New* ► *Other* ► *SAP HANA* ► *Database Development* ► *Scalar Function* ► or *Table Function*. The *New Scalar Function* or *New Table Function* wizard appears.
2. Define the function parameters.
Enter or select the parent folder, enter the file name, and choose *Finish*.
The editor opens containing a default template for the function. In a shared project, the design-time function name is generated containing the full path. In an unshared project, the full function name must be added manually.
3. Commit and activate your function.

Related Information

[Create and Edit Procedures \[page 391\]](#)

7.3.1 Tutorial: Create a Scalar User-Defined Function

In SQL, a user-defined function (UDF) enables you to build complex logic into a single database object. A scalar UDF is a custom function that can be called in the SELECT and WHERE clauses of an SQL statement.

Prerequisites

To complete this exercise successfully, you must bear in mind the following prerequisites:

- You have the user credentials required to log on to SAP HANA
- You have installed the SAP HANA studio
- You have a shared SAP HANA project available (preferably of type *XS Project*)
- The shared project contains a folder called `functions`

- You have installed the SAP HANA Interactive Education ([SHINE](#)) HCODEMOCONTENT delivery unit (DU); this DU contains the demonstration content (tables and views) that you want to consume with the procedure you build in this tutorial.
- You have generated data to populate the tables and views provided by the [SHINE](#) delivery unit and used in this tutorial. You can generate the data with tools included in the [SHINE](#) delivery unit.

📘 Note

You might have to adjust the paths in the code examples provided to suit the/package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration tables and services) referenced in the tutorial.

Context

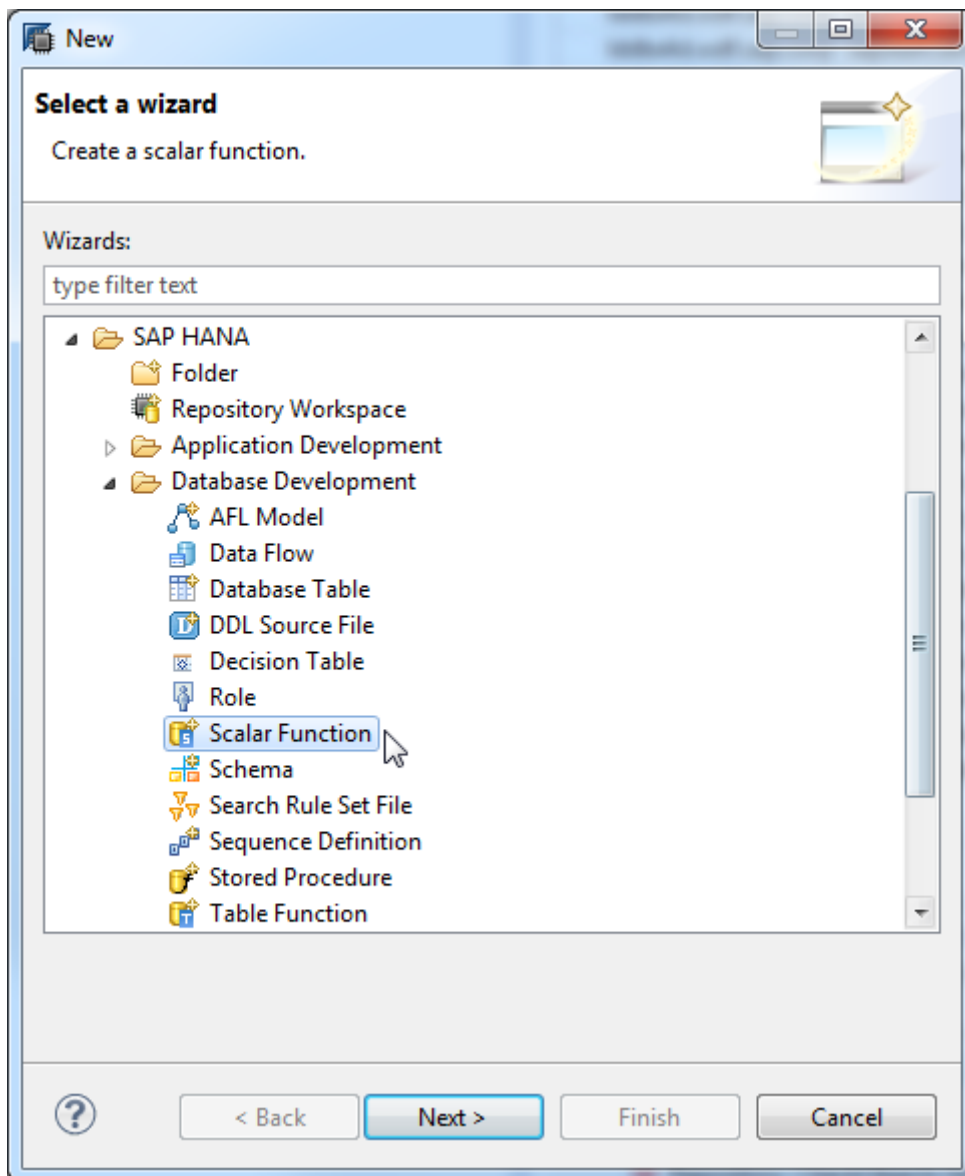
A scalar user-defined function has a list of input parameters and returns the scalar values specified in the RETURNS `<return parameter list>` option defined in the SQL function, for example, `decimal(15,2)`. The **scalar** UDF named `apply_discount` that you create in this tutorial performs the following actions:

- Applies a discount to the stored product price
- Calculates the sale price of a product including the suggested discount

To create the scalar UDF `apply_discount`, perform the following steps:

Procedure

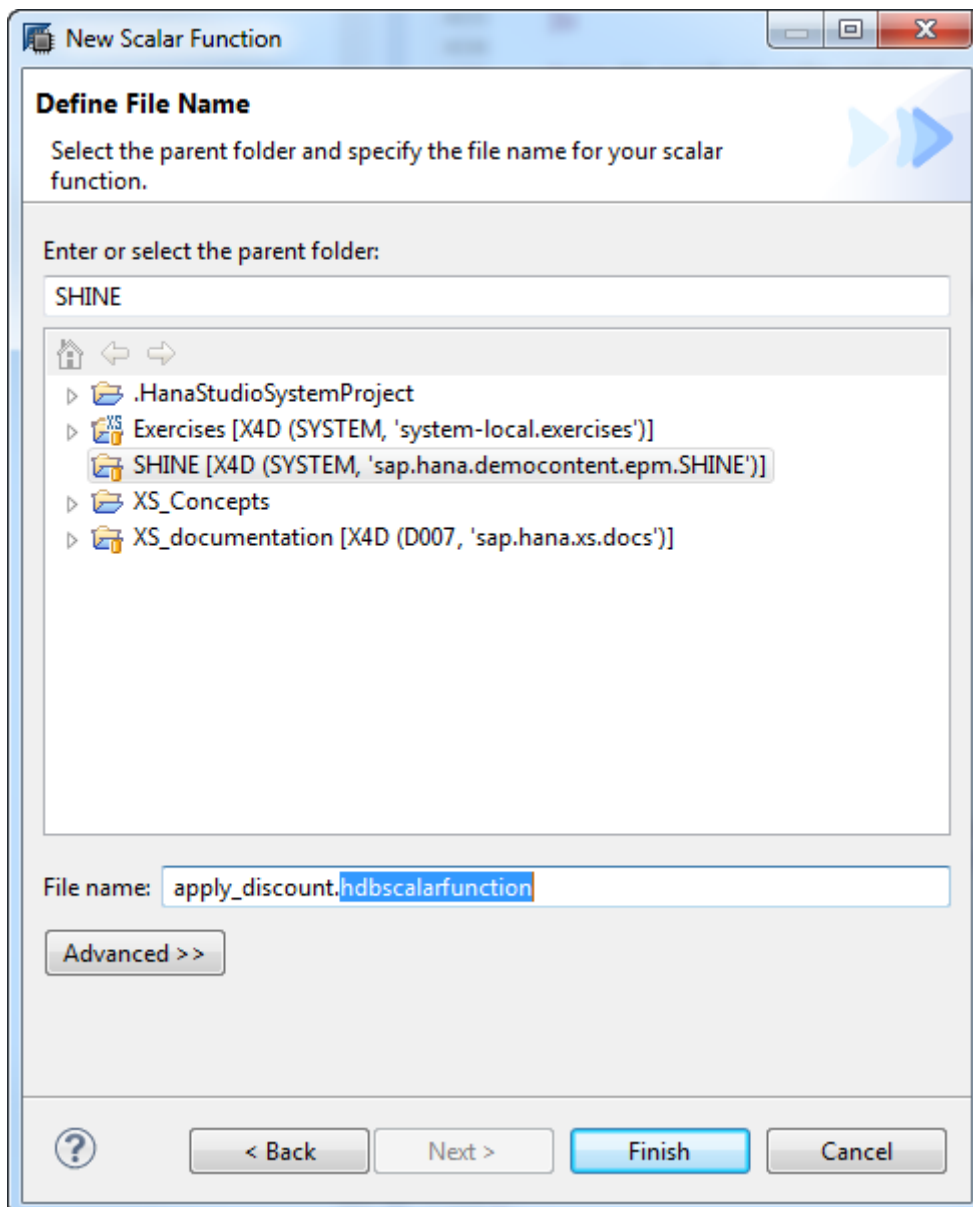
1. Open the SAP HANA studio.
Start the [SAP HANA Development](#) perspective, open the [Project Explorer](#) view, and navigate to the shared project in which you want to create the new scalar UDF.
2. Create the file that will contain the scalar UDF.
If not already available, create a new folder (package) called `functions` in the selected project.
 - a. Start the [Create New UDF](#) wizard.
In the [Project Explorer](#) view, choose ► [New](#) ► [Other...](#) ► [SAP HANA](#) ► [Database Development](#) ► [Scalar Function](#) ► and choose [Next](#).



- b. Type the name of the new scalar UDF.
Type **apply_discount** in the *File name* box.

→ Tip

The file-creation wizard adds the suffix (`.hdbscalarfunction`) automatically.



- c. Choose *Finish* to create the scalar UDF and open it in SAP HANA studio's embedded SQL editor.
3. Create the user-defined function.

The user-defined function (UDF) you create in this step applies a discount to the stored product price and calculates the sale price of a product including the suggested discount.

- a. In the *SQL Editor*, type the code that defines the new user-defined function.

You can use the following code example, but make sure the paths point to the correct locations in your environment, for example, the schema name, the package location for the new UDF, and the location of the demo tables referenced in the code.

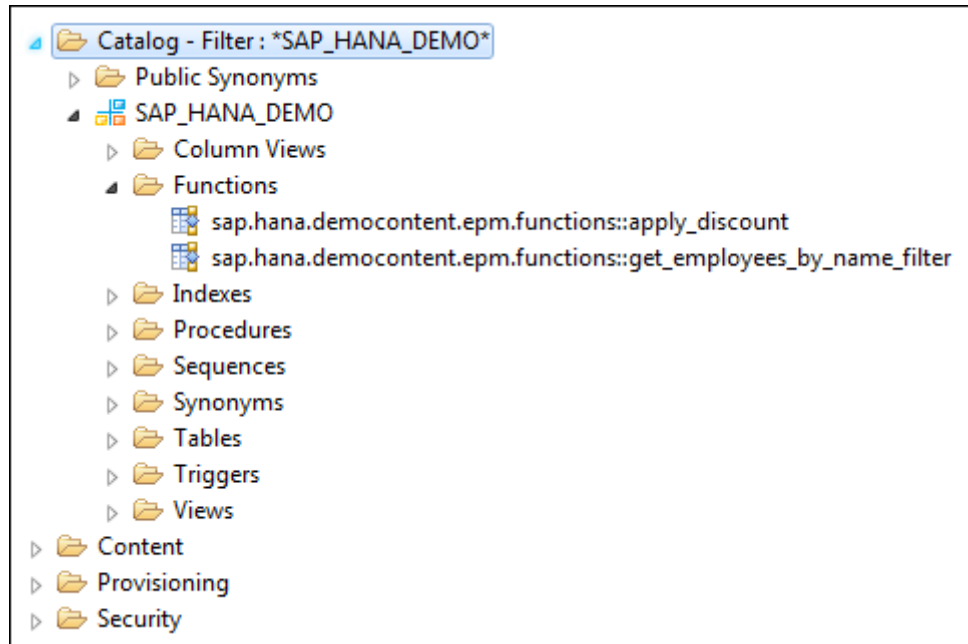
```
FUNCTION
"SAP_HANA_DEMO"."sap.hana.democontent.epm.functions::apply_discount"
(im_price decimal(15,2),
    im_discount decimal(15,2) )
RETURNS result decimal(15,2)
LANGUAGE SQLSCRIPT
SQL SECURITY INVOKER AS
```

```
BEGIN
result := :im_price - ( :im_price * :im_discount );
END;
```

- b. Save the changes you have made to the new scalar UDF.
- c. Activate the new scalar UDF in the SAP HANA Repository.

In the [Project Explorer](#) view, right-click the new `apply_discount.hdbscalarfunction` UDF artifact and choose **Team > Activate..** in the context-sensitive menu.

- d. Check the catalog to ensure the new UDF was successfully created in the correct location.



4. Use the new UDF in an SQL select statement.

You can use the following example statement, but make sure you modify the paths to point to the correct locations in your environment, for example, the schema name, the package location for the new UDF, and the location of the demo tables referenced in the code.

```
select PRODUCTID, CATEGORY, PRICE,
       "SAP_HANA_DEMO"."sap.hana.democontent.epm.functions::apply_discount"(PRICE,
0.33 )
  as "SalePrice" from
       "sap.hana.democontent.epm.data::EPM.MasterData.Products";
```

5. Check the results in the [Results](#) tab of the [SQL Console](#).

SQL Result

```
select PRODUCTID, CATEGORY, PRICE,
       "SAP_HANA_DEMO"."sap.hana.democontent.epm.functions::apply_discount"(PRICE, 0.33 )
       as "SalePrice" from
       "sap.hana.democontent.epm.data::EPM.MasterData.Products"
```

	PRODUCTID	CATEGORY	PRICE	SalePrice
1	HT-1000	Notebooks	956	640.52
2	HT-1001	Notebooks	1,249	836.83
3	HT-1002	Notebooks	1,570	1,051.9
4	HT-1003	Notebooks	1,650	1,105.5
5	HT-1007	Handhelds	499	334.33
6	HT-1010	Notebooks	1,999	1,339.33
7	HT-1011	Notebooks	2,299	1,540.33
8	HT-1020	Handhelds	129	86.43
9	HT-1021	Handhelds	149	99.83
10	HT-1022	Handhelds	205	137.35

7.3.2 Tutorial: Create a Table User-Defined Function

In SQL, a user-defined function (UDF) enables you to build complex logic into a single database object that you can call from a SELECT statement. You can use a table user-defined function (UDF) to create a parameterized, fixed view of the data in the underlying tables.

Prerequisites

To complete this exercise successfully, bear in mind the following prerequisites:

- You have the user credentials required to log on to SAP HANA
- You have installed the SAP HANA studio
- You have a shared SAP HANA project available (preferably of type *XS Project*)
- The shared project contains a folder called `functions`
- You have installed the *SHINE* (democontent) delivery unit (DU); this DU contains the tables and views that you want to consume with the procedure you build in this tutorial.
- You have generated data to populate the tables and views provided by the *SHINE* delivery unit and used in this tutorial. You can generate the data with tools included in the *SHINE* delivery unit.

📌 Note

You might have to adjust the paths in the code examples provided to suit the package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration tables and services) referenced in the tutorial.

Context

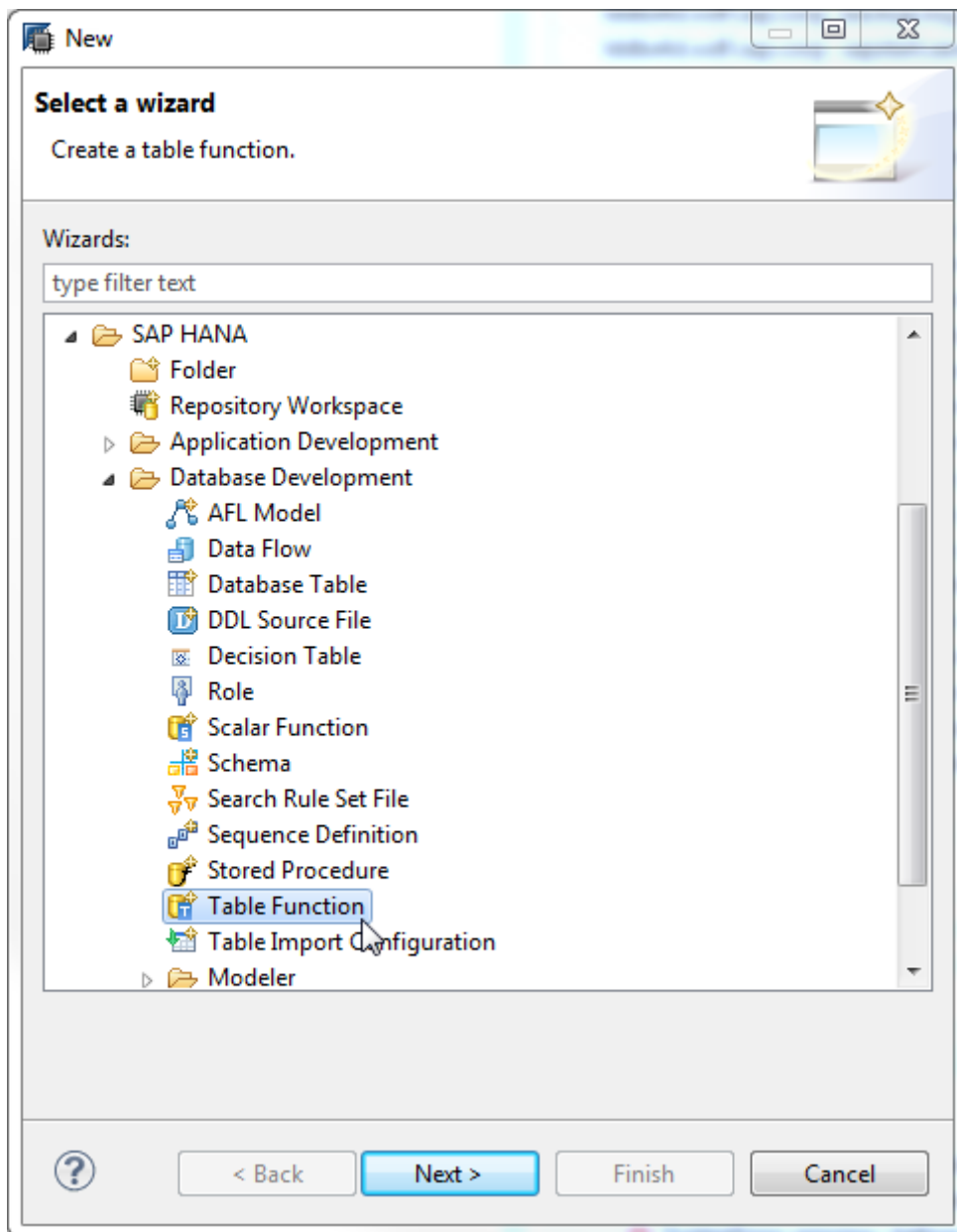
A table UDF has a list of input parameters and must return a table of the type specified in RETURNS *<return-type>*. The **table** UDF named `get_employees_by_name_filter` that you create in this tutorial performs the following actions:

- Executes a `SELECT (INNER JOIN)` statement against the employee and address tables
- Filters the results by performing a fuzzy search on the last name

To create a table user-defined function called `get_employees_by_name_filter`, perform the following steps:

Procedure

1. Open the SAP HANA studio.
Start the *SAP HANA Development* perspective, open the *Project Explorer* view, and navigate to the shared project in which you want to create the new table UDF.
2. Create the file that will contain the table UDF.
If not already available, create a new folder (package) called `functions` in the selected project.
 - a. Start the *Create New UDF* wizard.
In the *Project Explorer* view, choose ► *New* ► *Other...* ► *SAP HANA* ► *Database Development* ► *Table Function* ► and choose *Next*.

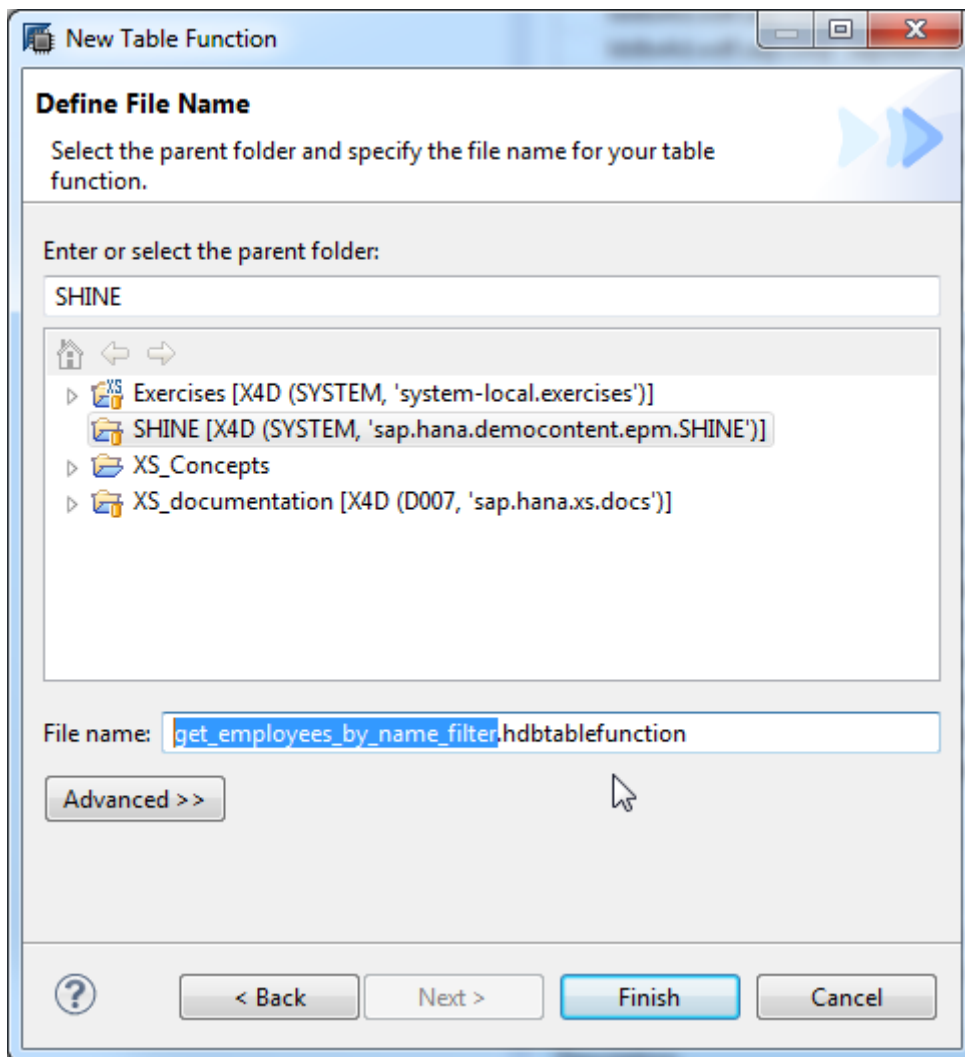


- b. Type the name of the new table UDF.

Type **get_employees_by_name_filter** in the *File name* box.

→ Tip

If the file-creation wizard does not automatically add the suffix (`.hdbtablefunction`), select the parent folder where you want to create the new function.



- c. Choose *Finish* to create the table UDF and open it in SAP HANA studio's embedded SQL editor.
3. Define details of the user-defined function.

The user-defined function you create in this step first executes a `SELECT (INNER JOIN)` statement against the employee and address tables and then filters the results by performing a fuzzy search on the last name.

- a. In the *SQL Editor*, type the code that defines the new user-defined function.

You can use the following code example, but make sure the paths point to the correct locations in your environment, for example, the schema name, the package location for the new UDF, and the location of the demo tables referenced in the code.

```
FUNCTION
"SAP_HANA_DEMO"."sap.hana.democontent.epm.functions::get_employees_by_name_
filter"
(lastNameFilter nvarchar(40))
RETURNS table ( EMPLOYEEID NVARCHAR(10),
                "Name.FIRST" NVARCHAR(40),
                "Name.LAST" NVARCHAR(40),
                EMAILADDRESS NVARCHAR(255),
                ADDRESSID NVARCHAR(10), CITY NVARCHAR(40),
                POSTALCODE NVARCHAR(10), STREET NVARCHAR(60))
LANGUAGE SQLSCRIPT
```

```

SQL SECURITY INVOKER AS

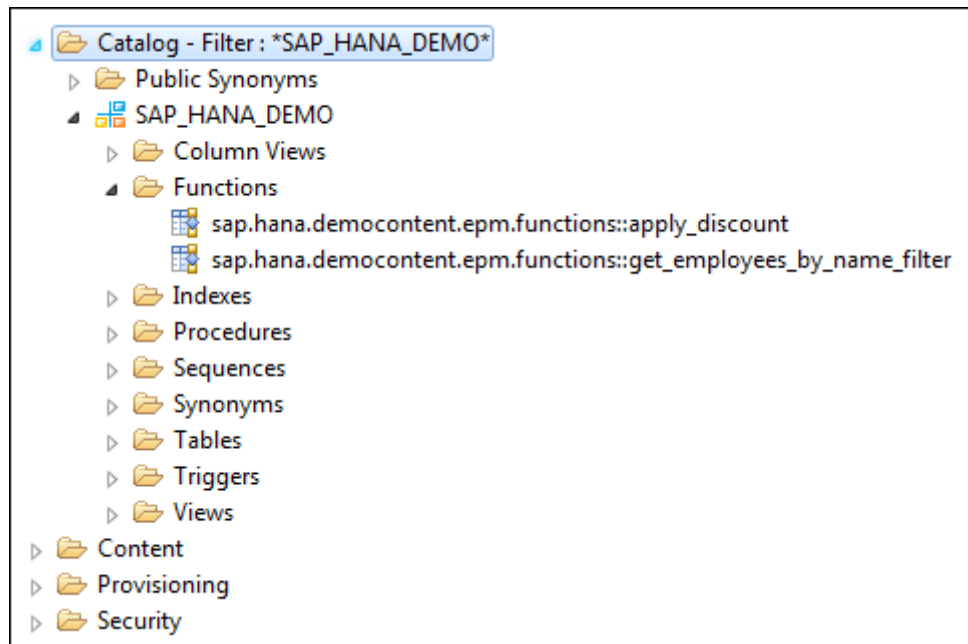
BEGIN
RETURN
  select a.EMPLOYEEID, a."Name.FIRST",
         a."Name.LAST", a.EMAILADDRESS,
         a.ADDRESSID, b.CITY, b.POSTALCODE, b.STREET
  from "sap.hana.democontent.epm.data::EPM.MasterData.Employees"
      as a
 inner join
      "sap.hana.democontent.epm.data::EPM.MasterData.Addresses"
      as b
  on a.ADDRESSID = b.ADDRESSID
     where contains("Name.LAST", :lastNameFilter, FUZZY(0.9));
END;

```

- b. Save the changes you have made to the new table UDF.
- c. Activate the new table UDF in the SAP HANA Repository.

In the *Project Explorer* view, right-click the new *get_employees_by_name_filter* UDF artifact and choose **Team > Activate..** in the context-sensitive menu.

- d. Check the catalog to ensure the new UDF was successfully created in the correct location.



4. Use the new UDF in an SQL select statement.

You can use the following example statement, but make sure you modify the paths to point to the correct locations in your environment, for example, the schema name, the package location for the new UDF, and the location of the demo tables referenced in the code.

```

select * from
"SAP_HANA_DEMO"."sap.hana.democontent.epm.functions::get_employees_by_name_fil
ter"('*ll*')

```

5. Check the results in the *Results* tab of the *SQL Console*.

SQL Result									
select * from "SAP_HANA_DEMO".sap.hana.democontent.epm.functions::get_employees_by_name_filter('*11*'))									
	EMPLOYEEID	Name.FIRST	Name.LAST	EMAILADDRESS	ADDRESSID	CITY	POSTALCODE	STREET	
1	0000000003	Franco	Fall	franco.fall@itelo.info	1000000004	San Francisco	94101	Risel Avenue	
2	0000000010	Michael	Miller	michael.miller@itelo.info	1000000011	London	EC1Y8SY	Castle Lane	
3	0000000018	Al	Wallace	al.wallace@itelo.info	1000000019	San Francisco	94101	Silver Aven...	
4	0000000027	Barbara	Ingalls	barbara.ingalls@itelo.in...	1000000028	San Francisco	94101	Fernwood ...	

7.4 Create Procedure Templates

A procedure template is an artifact containing a base script with predefined placeholders for objects such as tables, views and columns. The procedure template enables you to create procedures that contain the same script, but with different values.

Prerequisites

- You have created a development workspace. For more information, see *Create a Repository Workspace*.
- You have checked out a package. For more information, see *SAP HANA Repositories View*.

Note

After checking out a package that contains active procedures, you can modify and debug the procedures.

- You have created and shared a project. For more information, see *Using SAP HANA Projects*.

Note

You can also share your project after you create your procedure template.

Procedure

1. Open the *New File* wizard.

After you have created your workspace and your project, go to the *Project Explorer* view in the *SAP HANA Development* perspective, right-click on the file name, choose **New > File**. The *New File* wizard appears.

2. Enter or select the parent folder and enter the file name using the following naming convention `<filename>.proceduretemplate`.

3. Choose *Finish*.

The *Template Script* editor opens.

4. Define the template parameters.

Click the  icon from the toolbar in the *Template Parameters* table to add a parameter to the table. You can rename the parameter and give it a meaningful name. Add the parameters to the table and to the script where they are used as a placeholder for the following objects:

- Schema name
- Table name and table column name
- View name and view column name
- Procedure name

The parameters can only be used in the procedure body, between the BEGIN and END, and not as part of the procedure header. The parameters must follow the SQL identifier semantics. Each parameter should be wrapped using the less than (<) and greater than (>) symbols. For example:

```
SELECT <My_Column> FROM <My_Table>;
```

Caution

You cannot add a parameter as a placeholder for other objects or syntactic statements.

5. Commit and activate your procedure template.

Caution

To avoid errors during activation, you must make sure your procedure template is consistent. For example:

- A parameter that is a placeholder for a table must be in a valid position that is syntactically correct.
- A parameter name must be identical in the *Template Parameters* table and the *Template Script*.

Related Information

[Maintain a Repository Workspace \[page 63\]](#)

[The Repositories View \[page 32\]](#)

[Using SAP HANA Projects \[page 62\]](#)

[Create and Edit Procedures \[page 391\]](#)

7.4.1 Create Procedure Template Instances

A procedure template instance is an artifact that is created from a procedure template. It contains the same procedure script and uses specific values for the predefined placeholders in the script. Procedure template instances are coupled with the procedure template, which means any changes that are made to the template are also applied to the template instances. During activation, a template instance is generated as a procedure in the catalog.

Prerequisites

You have created a procedure template or checked out an existing one. For more information, see *Create Procedure Templates*.

Procedure

1. Open a *New Procedure* wizard.
 - a. Go to the *Project Explorer* view in the *SAP HANA Development* perspective, right-click the file name, choose  *New*  *Other* . The *New* wizard appears.

Note

The latest version of the procedure template must be checked out to your local workstation before you can select it.

- b. Expand the *Database Development* folder and select *Stored Procedure*. The *New Procedure* wizard appears.
2. Define the new procedure attributes.

Enter or select the parent folder, enter the file name, select *XML (.procedure) - Deprecated* for the file format. Choose *Advanced*, select the *Create from procedure template* checkbox, and choose *Browse*. Select the relevant template, choose *OK*, and choose *Finish*.
3. In the *Procedure Template Instance* editor, add a value in the *Value* column for each parameter, and choose *Save*.

Note

The value is the string that replaces the parameter in the template script.

4. Commit and activate your procedure template instance.

Note

During activation:

- The procedure is created in the catalog using the values specified in the instance with the active template in the repository.
- A reference between the instance and its template is created to link them together.

Related Information

[Create Procedure Templates \[page 410\]](#)

[Create and Edit Procedures \[page 391\]](#)

[Update Procedure Templates and Instances \[page 413\]](#)

[Delete Procedure Templates and Instances \[page 413\]](#)

7.4.2 Update Procedure Templates and Instances

The procedure template script and its parameters can be modified, which also modifies the template instances that refer to it. Only the template parameter values can be changed in the procedure template instances.

Procedure

1. To update a procedure template and its instances, double-click the relevant file in the *Project Explorer* view. The file appears in the *Template Script* editor.
2. You can change the list of template parameters or the template script. Choose *Save*.

Note

If you change the list of template parameters, you should also update the instances by choosing the *Refresh* button to update the list of parameters and enter the values. 

3. Commit and activate your procedure template and its instances.

Note

During activation, the corresponding instances are reactivated and the changes are applied accordingly.

Related Information

[Create and Edit Procedures \[page 391\]](#)

7.4.3 Delete Procedure Templates and Instances

A procedure template can be deleted if there are no instances linked to it. If there are instances linked to the procedure template, they must be deleted before you can delete the procedure template.

Procedure

1. To delete a procedure template or a procedure instance, right-click the relevant file in the *Project Explorer* view, choose *Delete*, and choose *OK*.
2. Commit and activate the package.

Note

If an error occurs during activation because there are instances linked to the procedure template that you are trying to delete, then right-click the project name and choose ► *Team* ► *Resolve* ►.

Related Information

[Create and Edit Procedures \[page 391\]](#)

7.5 Debugging Procedures

The SAP HANA SQLScript debugger allows you to debug and analyze procedures. In a debug session, your procedures are executed in serial mode, not in parallel (not optimized). The stored procedure call stack appears in the debug view allowing you to view the nested calls. This allows you to test the correctness of the procedure logic and is not intended for evaluating the performance.

The following debug session types are available:

- Design-Time - Enables you to debug a design-time procedure artifact (.procedure/.hdbprocedure)
- Catalog - Enables you to debug a runtime procedure object
- External - Enables you to debug procedures that are executed by an external session
- Unified - Enables you to debug targets of both XS JavaScript and SQLScript in the debug view

Related Information

[Debug Design-Time and Catalog Procedures \[page 415\]](#)

[Debug an External Session \[page 418\]](#)

[The Debug Perspective \[page 610\]](#)

7.5.1 Setup Debugger Privileges

Grant debugger privileges to your user.

Procedure

1. Go to the *Systems* view in the *SAP HANA Development* perspective and open ► *Security* ► *Users* ►.

Note

You can also grant authorization from the [SQL Console](#).

2. Double-click your user ID. Your system privileges information appears.
3. Choose the [Object Privileges](#) tab to grant debug privileges to a schema or to procedures. Choose the [Add](#) button, select the relevant schema or procedure, and choose [OK](#). Select the schema or procedure in the table and select [DEBUG](#).

Note

If you want to allow other users to debug your schema or procedures, select [Yes](#) under [Grantable to others](#).

4. Choose the [Deploy](#) button ([F8](#)).

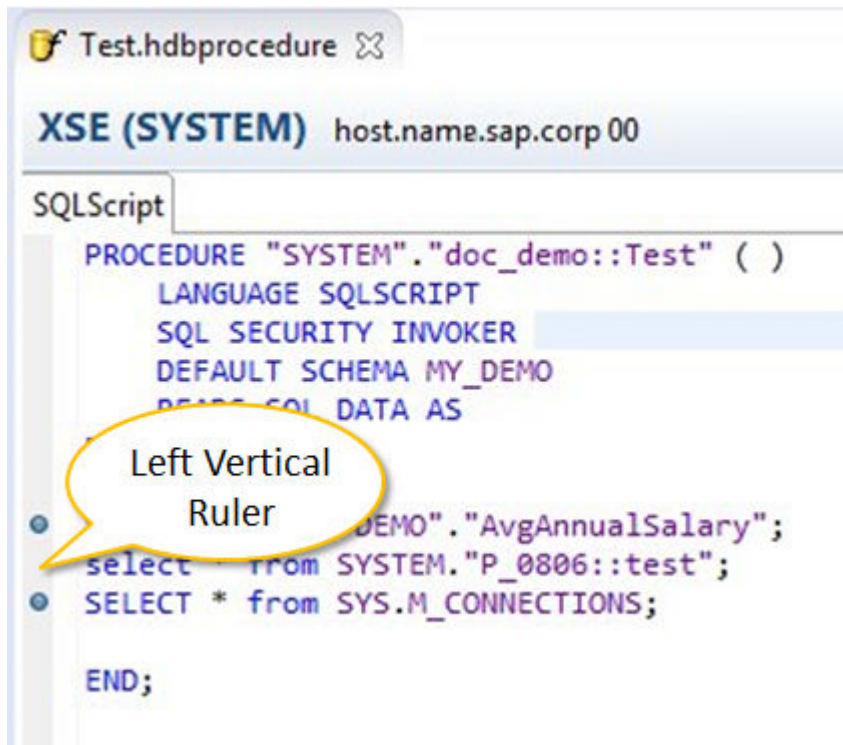
7.5.2 Debug Design-Time and Catalog Procedures

You can debug and analyze active SQLScript procedures that are in a local shared project (.hdbprocedure or .procedure).

Procedure

1. Open a procedure.
 - For catalog procedures go to the [Project Explorer](#) view in the [SAP HANA Development](#) perspective or to the [Systems](#) view, choose [Catalog](#), select the schema, and double-click the procedure to open it in the [SAP HANA Stored Procedure](#) viewer.
 - For design-time procedures open the [Debug](#) perspective in the SAP HANA studio and select the procedure you want to debug by choosing the relevant tab in the [Editor](#) view.
2. Add breakpoints.

To add breakpoints double-click the left vertical ruler to add breakpoints to your procedure. You can see a list of all of the breakpoints in the [Breakpoints](#) view.



From the *Breakpoints* view, you can:

- Deselect specific breakpoints or skip all of them.
- Delete a specific breakpoint or delete all of them.
- Double-click a breakpoint to see which line it belongs to in the *Editor* view.
- See the status of the breakpoint:
 - Pending
 - Valid
 - Invalid

3. Start a debug session.

To start a new debug session, you must first create a debug configuration. Choose and *Debug Configurations...*. The *Debug Configurations* wizard appears.

Note

You can also go to the *Project Explorer* view, right-click your procedure, choose *Debug As* *SAP HANA Stored Procedure*.

4. In the *General* tab, do the following:
 - a. Select the *Procedure to Debug* radio button and choose *Local projects* from the drop-down menu.
 - b. Choose *Browse...* and select a procedure from a schema in the relevant system to debug. Choose *OK*.
5. In the *Input Parameters* tab, a list of the parameters and types is displayed for your procedure. You must add values for each parameter in the *Value* column.

Note

For scalar types, insert a value. For table types, enter the name of a catalog table (schema . tablename) that contains the relevant input. For example, SYS . USERS.

Note

To debug a procedure that does not require you to define values for input parameters, double-click *SQLScript Procedure*, enter a name, choose *Apply*, and choose *Debug*.

6. If you want to control the way your procedures are compiled in debug mode, go to the *Advanced* tab, and select one of the following radio buttons:
 - *All procedures in the call stack* to compile all of the nested procedures that are referenced from the procedure stack in debug mode
 - *Procedures with breakpoints* to compile procedures with breakpoints in debug mode

Caution

Selecting *Procedures with breakpoints* will make the compilation and the procedure execution faster. However, it may prevent you from breaking in a procedure that was compiled in an optimized way.

7. Choose *Apply* and *Debug*.
8. To start your debug session, choose *Debug*.

The debug session begins and the status of the session appears in the *Debug* view. The debugger will stop at the first breakpoint and the session will be suspended until you resume it. After the server validates your breakpoints, the status and position of them may change. The position of the breakpoints is the next valid line where the debugger can stop. If your breakpoint is successfully set, the  valid status appears next to it in the *Breakpoints* view.

Caution

Selecting *Procedures with breakpoints* makes the compilation and the procedure execution faster. However, it may prevent you from breaking in a procedure that was compiled in an optimized way.

Note

You must set breakpoints in the lines you want to break at and resume the session again.

You can evaluate your local scalar and table variables in the *Variable* view. The view shows the values of the scalar variables and the number of rows in each table.

9. View the content of the listed tables in the *Variable* view.

Right-click the table name and choose *Open Data Preview*. The results will appear in the *Preview* view. This view will automatically close when you resume your debug session.

Results

The debug session is terminated when the procedure run has finished.

7.5.3 Debug an External Session

You can debug and analyze procedures that are executed by an external application.

Prerequisites

- You know the connection ID, the HANA user, or the HANA user and the application user that your external application uses to connect to the SAP HANA database and to call procedures.
- You have activated your stored procedures.
- You have granted debugger privileges to your user:

1. Go to the *Systems* view in the *SAP HANA Development* perspective and open ► *Security* ► *Users* ►.
2. Double-click your user ID. Your system privileges' information will appear.
3. Choose the *Object Privileges* tab to grant debug privileges to a schema or to procedures. Choose the  *Add* button, select the relevant schema or procedure, and choose *OK*. Select the schema or procedure in the table and select *DEBUG*.

Note

If you want to allow other users to debug your schema or procedures, select *Yes* under *Grantable to others*.

4. Choose the *Privileges on Users* tab to allow other users to debug procedures in your connection.
Choose the  *Add* button, select the relevant user, and select *ATTACH DEBUGGER*.
5. Choose the  *Deploy* button (*[F8]*).

Caution

Granting debugger privileges to your user enables you to connect to other user's sessions, and therefore debug procedures that you are not allowed to run and view data that you are not allowed to examine.

Procedure

1. Start a debug session.

To start a new debug session, you must first create a debug configuration. Choose  and *Debug Configurations...*. The *Debug Configurations* wizard appears.

2. In the *General* tab, select the *Debug an external session* radio button, and choose *SAP HANA System*.
 - a. Select the *Set filter attributes* radio button if you know the connection attributes that your external application uses to connect to the SAP HANA database. Enter *HANA User*, which is the SAP HANA database user, and optionally enter *Application User* if your external application sets this attribute for the connection.

Note

It is not mandatory for the connection to be established before you start the debug session.

- b. Select the *Select a connection after initiating the debugger* radio button if you know the connection ID that your external application uses to connect to the SAP HANA database. This option enables you to choose a specific connection after the debugger session has started.

If you want to save the debug configuration you created and debug your procedure later, choose *Apply* and *Close*. To start your debug session, choose *Debug* and trigger the call to the SAP HANA procedure from your external application.. The *Select Connection* wizard appears. Choose a connection ID and choose *OK*.

Note

It is mandatory for the connection to be established before you start the debug session.

Results

The debug session will begin and you will see the status of the session in the *Debug* view. The debugger will wait until your procedure is executed on the connection ID that your external application uses. Once your procedure is executed, the debugger will stop at the first breakpoint, and the session will be suspended until you resume it. You will also see the your procedure name in the third and fourth level of the *Debug* view.

After the server has validated your breakpoints, the status and position of them may change. The position of the breakpoints will be the next valid line where the debugger can stop. If your breakpoint is successfully set, the  valid status appears next to it in the *Breakpoints* view.

Caution

If more than one user tries to debug a procedure in the same connection that was either selected or identified by a user name, only the first user that chooses *Debug* will be able to stop at a breakpoint and debug the procedure.

Note

You must set breakpoints in the lines you want to break at and resume the session again.

You can evaluate your local scalar and table variables in the *Variable* view. The view shows the values of the scalar variables and the number of rows in each table.

7.6 Developing Procedures in the Modeler Editor

Context

To create procedures, use the SQLScript Editor, as described in [Create and Edit Procedures \[page 391\]](#).

If you need to create procedures with local table types, that is, table types created only for the procedure, perform the steps described in this section.

Procedure

1. On the [Quick Launch](#) tab page, choose [Procedure](#).
If the [Quick Launch](#) page is not open, go to ► [Help](#) ► [Quick Launch](#) ►.
2. Enter a name and description for the procedure.
3. For unqualified access in SQL, select the required schema from the [Default Schema](#) dropdown list.

Note

- If you do not select a default schema, while scripting you need to provide fully qualified names of the catalog objects that include the schema.
- If you specify a default schema, and write SQL such as `select * from myTable`, the specified default schema is used at runtime to refer to the table.

4. Select the package in which you want to save the procedure.
5. Select the required option from the [Run With](#) dropdown list to select which privileges are to be considered while executing the procedure.

Note

There are two types of rights, as follows:

Definer's right: If you want the system to use the rights of the definer while executing the procedure for any user.

Invoker's right: If you want the system to use the rights of the current user while executing the procedure.

6. Select the required access mode as follows:

Access Mode	Purpose
Read Only	Use this mode to create procedures for fetching table data.

Access Mode	Purpose
Read Write	Use this mode to create procedures for fetching and updating table data.

7. Select the language in which you are writing the procedure.

Note

You can choose to create procedures in Read Write mode and make use of L- Lang and R-lang languages only if you have done the repository configuration for the field `sqlscript_mode`. Two values for `sqlscript_mode` field exist, DEFAULT, and UNSECURE. By default DEFAULT is assigned which means Read Only mode with non-modifiable access mode and SQLScript as language. To change the configuration, go to administration console -> Configuration tab -> indexserver.ini -> repository -> sqlscript_mode, and assign the required value.

8. Choose *Finish*.
9. In the function editor pane, write a script for the function using the following data types:
 - Table or scalar data types for input parameters.
 - Table data types for output parameters.

Note

You can only write one function in the function body. However, you can refer to other functions.

10. Choose **File** > **Save**.
11. Activate the procedure using one of the following options in the toolbar:
 - **Save and Activate**: Activate the current procedure and redeploy the affected objects if an active version of the affected object exists. Otherwise only the current procedure gets activated.
 - **Save and Activate All**: Activate the current procedure along with the required and affected objects.

Note

You can also activate the current procedure by selecting the procedure in the *Navigator* view and choosing **Activate** in the context menu. For more information about activation, see [Activating Objects \[page 359\]](#).

7.7 Transforming Data Using SAP HANA Application Function Modeler

Overview of SAP HANA application function modeler.

A flowgraph is a development object. It is stored in a project and has extension `.hdbflowgraph`. By default, the activation of a flowgraph generates a procedure in the catalog.

ⓘ Note

If the optional additional cost SAP HANA smart data integration and SAP HANA smart data quality component is available, a flowgraph can be configured to generate a task plan run-time object instead of a procedure.

ⓘ Note

Columns that you do not map as inputs to a flowgraph will not be sent over the network to be processed by SAP HANA. Excluding columns as inputs can improve performance, for example if they contain large object types. You can also choose to enhance security by excluding columns that, for example, include sensitive data such as passwords.

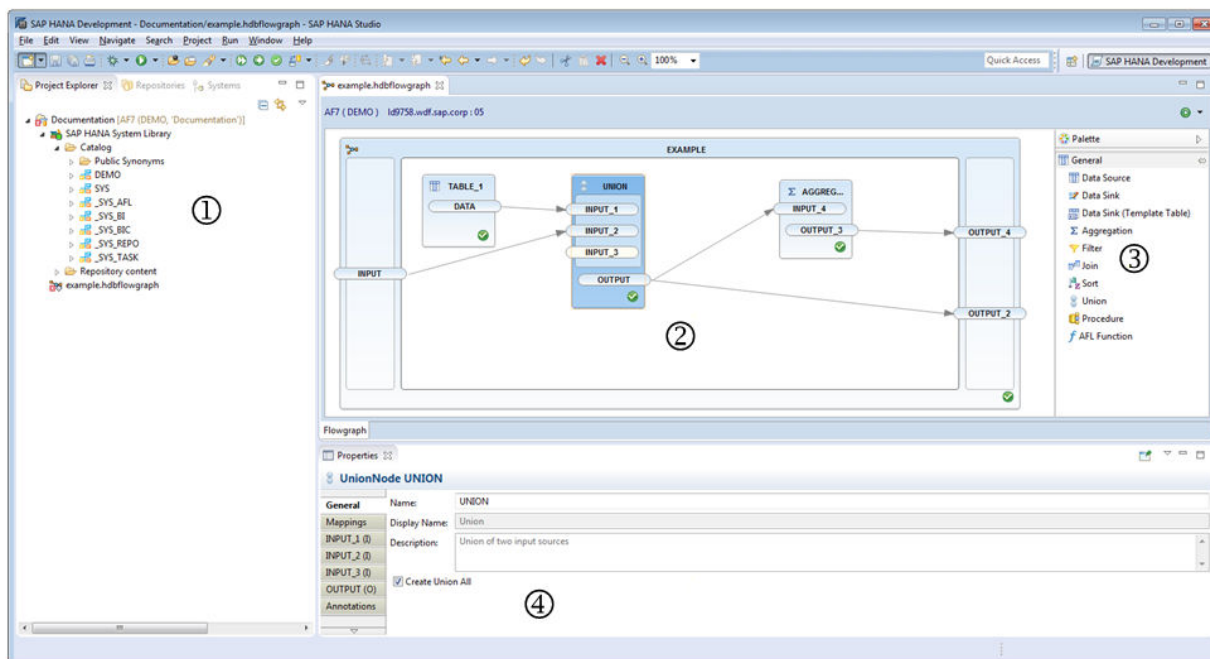
A flowgraph models a data flow that can contain:

- tables, views, and procedures from the catalog
- relational operators such as projection, filter, union, and join
- functions from Application Function Libraries (AFL) installed on your system
- attribute view and calculation view development objects

In addition, the application function modeler provides support for some optional, additional cost components of the SAP HANA Platform such as:

- the Business Function Library
- the Predictive Analysis Library
- R Scripts
- Data Provisioning operators
- the generation of task plans

The application function modeler is part of the *SAP HANA Development* perspective and utilizes the following components.



Components used by the SAP HANA application function modeler

	Area	Description
1	<i>Project Explorer</i> view	The <i>Project Explorer</i> is used as a source of objects that can be added to the <i>Editing Area</i> .
2	<i>Editing Area</i>	In the <i>Editing Area</i> , the flowgraph is modeled. Elements are added to the flowgraph by dragging objects from the <i>Project Explorer</i> or node templates from the <i>Node Palette</i> to the <i>Editing Area</i> . There, they can be selected and edited via the context button pad and the context menu. The <i>Editing Area</i> supports standard editing operations like copy, paste, and delete, as well as moving elements by drag and drop. The properties of selected flowgraph elements can be edited in the <i>Properties</i> view.
3	<i>Node Palette</i>	The <i>Node Palette</i> lists the node templates available to the application function modeler. These node templates can be added to the flowgraph by dragging them to the <i>Editing Area</i> . In case an optional, additional cost component of the SAP HANA Platform is detected by the application function modeler, an additional compartment with node templates for its functions is automatically added to the <i>Node Palette</i> .
4	<i>Properties</i> view	The <i>Properties</i> view shows the property details of the selected flowgraph element.

→ Tip

You can open the *SAP HANA Development* perspective by choosing ► *Window* ► *Open Perspective* ► *SAP HANA Development* ►, the *Properties* view by choosing ► *Window* ► *Show View* ► *Properties* ►, and the *Project Explorer* views by choosing ► *Window* ► *Show View* ► *Project Explorer* ►.

Related Information

[The SAP HANA Development Perspective \[page 31\]](#)

[The Project Explorer View \[page 33\]](#)

[SAP HANA Projects \[page 35\]](#)

[Attribute Views \[page 337\]](#)

[Calculation Views \[page 347\]](#)

7.7.1 Converting deprecated AFL Models (AFLPMML objects)

Convert a deprecated AFL Model development object that was created by a previous version of the SAP HANA application function modeler into a flowgraph.

Context

AFL Models are development objects with the extension `.aflpmml` that were created with a previous version of the SAP HANA application function modeler. They are deprecated in SAP HANA SPS09.

Compared to the complex data flows with various operators modeled by a flowgraph, an AFL Model object is restricted to model a single function from the Application Function Library together with the data sources and data sinks that are connected to this function.

An AFL Model can still be activated. However, since AFL Models are deprecated, it can no longer be directly edited with the application function modeler. Instead, the AFL Model first has to be converted to a flowgraph. Then this flowgraph can be edited with the application function modeler. For backward compatibility, the edited flowgraph can be re-converted to an AFL Model. This requires all changes to the flowgraph to be compatible with the restrictions of AFL Models.

❗ Note

From SAP HANA 2.0 SPS03, the deprecated `.aflpmml` file can no longer be activated. Before an upgrade to SPS03, any old `.aflpmml` objects must be either converted to or re-created in the format `.hdbflowgraph`.

Procedure

1. In the *Project Explorer* view right-click on the AFL Model that you want to convert to a flowgraph, and then choose *Convert to Flowgraph* in the context-sensitive menu.
The application function modeler creates a new flowgraph with the same prefix and the `.hdbflowgraph` extension. A dialog appears that lets you delete the AFL Model and its corresponding generated procedure. Afterward, you can edit the new flowgraph with the application function modeler.

Note

If you choose not to delete the converted application function modeler Model and try to activate a flowgraph, you get an error stating that there already exists an active catalog object with the same name (the new object tries to generate the same runtime object). You need to either delete or rename one of the two objects and activate the modification as well.

Note

A flowgraph cannot be activated on a SAP HANA SPS08 system.

2. (Optional) Convert a flowgraph to a AFL Model. In the *Project Explorer* view right-click on the flowgraph that you want to convert to an AFL Model, and then choose *Convert to AFLPMML* in the context-sensitive menu. The application function modeler creates a new AFL Model with the same prefix and the `.aflpmml` extension.

Note

AFL Model objects are deprecated. This conversion is available for backward compatibility. Most features of a flowgraph are not supported by the AFLPMML format.

7.7.2 Setting up the SAP HANA Application Function Modeler

Configure your system to use the SAP HANA Application Function Modeler.

Before modeling flowgraphs with the SAP HANA Application Function Modeler (AFM), make sure that the following system requirements are satisfied and that the following database access rights are granted to the respective database users.

System Requirements

The AFM has the following system requirements.

- You have installed the current version of SAP HANA.
- You have installed the Application Function Libraries (AFLs) that you want to use. For more information, see the section *Installing or Updating SAP HANA Components* in the *SAP HANA Server Installation and Update Guide*.
- You have enabled the Script Server in your SAP HANA instance. See SAP Note 1650957 for more information.

Privileges for the database user `_SYS_REPO`

The database user `_SYS_REPO` has to be granted the following object privileges:

- `SELECT` object privileges for objects that are used as data sources,
- `INSERT` object privileges for objects that are used as data sinks,
- `INSERT` and `DELETE` object privileges for objects that are used as data sinks with truncation.

Note

Granting access rights to the user `_SYS_REPO` may constitute a security risk. Make sure that you understand the privileges you grant to database users. Also see the *SAP HANA Security Guide*.

Privileges for the database user of the AFM

You have to be granted the `MODELING` role.

You have to be granted the `EXECUTE` privilege for the object `SYS.REPOSITORY_REST`.

You have to be granted the following package privileges:

- `repo.read` package privileges on your repository package
- `repo.activate_native_objects` package privileges on your repository package
- `repo.edit_native_objects` package privileges on your repository package
- `repo.maintain_native_packages` package privileges on your repository package

In addition, you have to be granted the following object privileges to the target schema of the flowgraph activation (default: `_SYS_BIC`):

- `CREATE ANY`
- `ALTER`
- `DROP`
- `EXECUTE`
- `SELECT`
- `INSERT`
- `UPDATE`

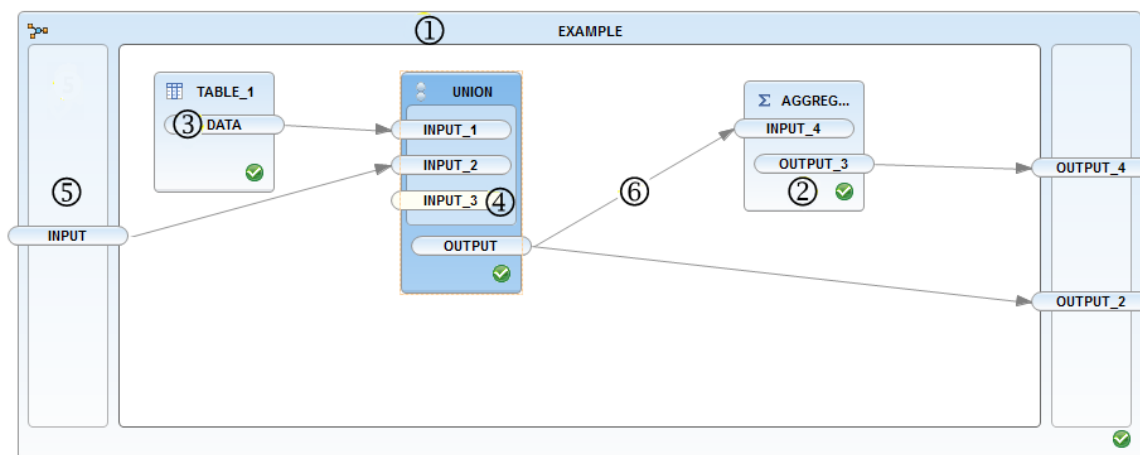
Note

Granting access rights to the user `_SYS_REPO` may constitute a security risk. Make sure that you understand the privileges you grant to database users. Also see the *SAP HANA Security Guide*.

7.7.3 Flowgraphs

This is an overview of all flowgraph elements.

A flowgraph consists of several flowgraph elements that are depicted in the [Editing Area](#). Every flowgraph element has a collection of properties that are displayed in the [Properties](#) view.



Flowgraph Elements

	Element	Description
1	Flowgraph container	The flowgraph container represents the operator defined by the flowgraph. Every flowgraph has exactly one flowgraph container. This flowgraph container has a name which has to differ from all other elements of the flowgraph. The flowgraph container can have several anchors. They represent the inputs and outputs of the operator defined by the flowgraph. The central free area of the flowgraph container is its canvas. All nodes of the flowgraph are contained in this canvas. The validation decorator in the right bottom corner of the flowgraph container indicates whether the flowgraph is configured correctly.
2	Node	Nodes are the functional elements in a flowgraph. There are several different types of nodes which represent data sources, data sinks, and operators. A node has a name which has to be unique in the flowgraph. Like the flowgraph container, a node can have several anchors. They represent the inputs and outputs of the node. The validation decorator in the right bottom corner of a node indicates whether the node is configured correctly.

	Element	Description
3	Anchor	An anchor represents an input or an output of the flowgraph container or of a node. Every anchor has a kind and a signature which define the input or output it represents. For input anchors, the supported kinds are Table, Column, and Scalar. For output anchors, the only supported kind is Table. Anchors of the Column kind are considered to be tables with a single column. Anchors of the Scalar kind are considered to be tables with a single column and a single row. This way, every anchor defines the table type of the input or output it represents.
4	Fixed content anchor	A fixed content anchor is an input anchor for which the fixed content flag is set in the properties. It is displayed in white color (in contrast to the light-blue colored standard anchors). A fixed content anchor cannot be the target of a connection. Instead, there is a table embedded in the flowgraph that is associated to the fixed content anchor. The table is displayed in the <i>Fixed Content</i> tab of the <i>Properties</i> view of the anchor.
5	Anchor region	The flowgraph container and some nodes (for example, the Join node and the Union node) can have a variable number of input and output anchors. This is represented by an anchor region for the corresponding set of anchors. Anchors can be added to or removed from the anchor region. They can also be reordered in the anchor region.
6	Connection	Connections represent the directed flow of data from a source to a target. The source and the target of a connection are anchors. The connection defines a table mapping between the table types defined by its source and target. The source of a connection is either an input anchor of the flowgraph container or an output anchor of a node. The target of a connection is either an output anchor of the flowgraph container or an input anchor of a node. An anchor can be the source of several connections. It can be the target of only one connection. A fixed content anchor cannot be the target of a connection.

Validation

There is a validation decorator in the right bottom corner of the flowgraph container and of each node. This decorator indicates if the complete flowgraph or the respective node is configured correctly. The details of a validation error are displayed by mouse-over on the validation decorator and in the *Problems* view.

→ Tip

You can open the Problems view by choosing ► *Window* ► *Show View* ► *Problems* ► in the main menu of the HANA Studio.

⚠ Caution

A flowgraph with validation errors will fail to activate.

Annotations

Annotations are nested key-value pairs that can be added to the flowgraph container and to nodes. The AFM uses annotations to store certain properties of the flowgraph such as custom palette information. An AFM user

can store arbitrary meta data in the annotations. When the flowgraph is activated, all annotations are exposed in a table with the name extension `.META` in the flowgraph target schema. This way, they can be consumed at runtime.

There are two main reasons for the user of the AFM to create annotations. The first reason is to add comments and documentation to the flowgraph. The second reason is to pass meta data about the flowgraph and its nodes to an application consuming the runtime procedure generated by the activation. In this case the application has to be specifically designed to process the meta data. Although this is a rather specific and uncommon use-case, it is a very versatile approach that utilizes flowgraphs to configure the analytic functionality of an application.

Related Information

[Using the Mapping Editor \[page 462\]](#)

[Using the Annotation Editor \[page 464\]](#)

[Customizing the Node Palette \[page 470\]](#)

7.7.4 Modeling a flowgraph

Model a flowgraph starting with its creation and concluding with the execution of the generated procedure.

Context

The SAP HANA Application Modeler (AFM) supports standard graphical editing operations like move, copy, paste, and delete on the elements of a flowgraph. Detailed properties of these elements are edited in the [Properties](#) view. After editing and saving a flowgraph, it can be activated by the AFM and the generated procedure can then be executed via the AFM.

If the flowgraph container has input anchors, the procedure has corresponding free inputs. It then cannot be executed directly. In this case, data sources have to be bound to the free inputs in order to execute the runtime object. The AFM provides a wizard for this.

Procedure

1. Create a new flowgraph or open an existing flowgraph in the [Project Explorer](#) view.
The flowgraph is opened in the [Editing Area](#) of the AFM.
2. Edit the flowgraph container.
3. Add and edit nodes.
4. Add and edit anchors.

5. Add and edit connections.
The validation decorators in the bottom right corners of the flowgraph container and the nodes indicate whether the flowgraph is valid.

Note

A flowgraph must be valid to be activated.

6. Save the flowgraph. Select **File > Save** in the HANA Studio main menu.
7. Activate the flowgraph. In the *Project Explorer* view, right-click the flowgraph object and choose **Team > Activate...** from the context-sensitive menu.
A new procedure is generated in the target schema which is specified in the properties of the flowgraph container.

Note

The generated procedure has inputs that correspond to the input anchors of the flowgraph container. To activate this procedure, these inputs have to be specified.

8. Select the black downward triangle next to the *Execute* button  in the top right corner of the AFM.
A context menu appears. It shows the options *Execute in SQL Editor* and *Open in SQL Editor* as well as the option *Execute and Explore* for every output of the flowgraph. In addition, the context menu shows the option *Edit Input Bindings*.
9. (Optional) If the flowgraph has input anchors, choose the option *Edit Input Bindings*.
A wizard appears that allows you to bind all inputs of the flowgraph to data sources in the catalog.
10. Choose one of the options *Execute in SQL Editor*, *Open in SQL Editor*, or *Execute and Explore* for one of the outputs of the flowgraph.
The behavior of the AFM depends on the execution mode.

Execution mode	Behavior
<i>Open in SQL Editor</i>	Opens a SQL console containing the SQL code to execute the runtime object.
<i>Execute in SQL Editor</i>	Opens a SQL console containing the SQL code to execute the runtime object and runs this SQL code.
<i>Execute and Explore</i>	Executes the runtime object and opens the <i>Data Explorer</i> view for the chosen output of the flowgraph.

11. Close the flowgraph. Select **File > Close** in the HANA Studio main menu.

7.7.4.1 Creating a flowgraph

Create a flowgraph development object with the [New Flowgraph Model Wizard](#).

Prerequisites

You have created and shared a project.

Procedure

1. In the [Project Explorer](#) view of the [SAP HANA Development](#) perspective, open an existing project.
2. If the project is not shared, right-click on the project and choose ► [Team](#) ► [Share](#) ► [Project](#) ► in the context-sensitive menu. In the [Share Project](#) wizard, choose SAP HANA Repository on the first page and an existing workspace on the second page.
3. In the [Project Explorer](#) view, right-click on the project and choose ► [New](#) ► [Other](#) ► in the context-sensitive menu.
The [New](#) wizard appears.
4. Choose ► [SAP HANA](#) ► [Database Development](#) ► [Flowgraph Model](#) ►, and then click [Next](#).
The [New Flowgraph Model](#) wizard appears.
5. In the text field [File Name](#) enter the base name of the new flowgraph.

Note

The system automatically adds the file extension `.hdbflowgraph`.

Note

The remaining steps explain advanced configuration options. In the standard use-case select [Finish](#) at this point to skip these steps.

6. (Optional) Choose the activation mode of the flowgraph in the [Usage](#) area.
This choice determines which runtime object is generated on activation of the flowgraph.

Note

The checkbox [Flowgraph for Activation as Task Plan](#) is only relevant when the flowgraph uses the additional cost SAP HANA smart data integration and SAP HANA smart data quality optional component.

7. (Optional) Select the [Advanced](#) button.
The [New Flowgraph Model](#) wizard expands and reveals the advanced options. This adds the option [Operator Palette Template](#) to the [Usage](#) area. It also shows the [Predefined Content](#) and [Node Palette](#) areas.
8. (Optional) Select the option [Operator Palette Template](#) in the [Usage](#) area.

If you choose this option the [New Flowgraph Model](#) wizard creates a node template flowgraph. A node template flowgraph has the file extension `.hdbf_lowgraph_template` and models a custom palette. It does not create any runtime object on activation.

9. (Optional) Select the checkbox [Use Flowgraph Template](#) in the [Predefined Content](#) area and specify a flowgraph in the text field below.

Instead of creating an empty flowgraph, the new flowgraph will be a copy of the specified flowgraph.

10. (Optional) Select a checkbox in the [Node Palette](#) area. If you select [Custom](#), specify a node palette flowgraph in the text field below.

This specifies the node palette of the new flowgraph.

- [AFM](#): the default AFM node palette,
- [Empty](#): a custom node palette that contains only the compartments specified in [Additional compartments for Node Palette](#),
- [Custom](#): a custom [Node Palette](#) based on the selected node template flowgraph.

Note

Node palette flowgraphs are nested structures that represent the hierarchy of a node palette. The nesting depth of the node palette flowgraph that represents a node palette is at least two and at most three.

11. (Optional) Select a list of node palette flowgraphs as additional node palette compartments of the new flowgraph.

The selected node palette flowgraphs are added to the end of the node palette of the new flowgraph.

Note

If a node palette flowgraph represents a node palette compartment, then its nesting depth is at most two.

12. Select [Finish](#).

The new flowgraph appears in the [Project Explorer](#) view and has the extension `.hdbf_lowgraph` (or `.hdbf_lowgraph_template` if you created a node palette flowgraph). The AFM is opened for editing the new flowgraph.

Related Information

[Node palette flowgraphs \[page 469\]](#)

7.7.4.2 Editing the flowgraph container

Edit the properties of the flowgraph in the [Properties](#) view of the flowgraph container.

Context

Tabs in the Properties view of the flowgraph container.

Tab name	Description	Optional
General	This tab contains the following entries: • Name: name of the flowgraph container, • Display Name; not used, • Description: not used, • Target Schema: schema in which the runtime object is generated during activation (default: <code>_SYS_BIC</code>), • Generator: the type of runtime object to be generated during activation. The option Task is only relevant if the flowgraph uses the additional cost SAP HANA smart data integration and SAP HANA smart data quality optional component, • Realtime Behavior: This option is only relevant when the flowgraph uses the additional cost SAP HANA smart data integration and SAP HANA smart data quality optional component and the chosen Generator option is Task	No
Variables	This tab is relevant only when the flowgraph uses the SAP HANA smart data integration and SAP HANA smart data quality optional component. For more information see the "Adding a Variable to the Container Node" topic in the <i>Modeling Guide for SAP HANA smart data integration and SAP HANA smart data quality</i> .	Yes
Mappings	The Mapping Editor in this tab is used to remove or re-order input and output anchors and their attributes.	Yes
INPUT (I) / OUTPUT (O)	These tabs correspond to the input and output anchors of the flowgraph container. They have the same names as the respective anchors and the same contents as the All tabs in the Properties views of the anchors.	Yes
Annotations	This tab contains the annotations of the flowgraph container.	No
All	This tab is a summary of all tabs in this view except for the input and output anchor tabs.	No

Procedure

1. Select the flowgraph container and open the [Properties](#) view.
2. In the [General](#) tab, specify the name of the flowgraph container, as well as the target schema, and the generator of the flowgraph.

Note

The name of the flowgraph container is initially auto-generated from the name of the flowgraph object in the *Project Explorer* view. This name has to be changed if it does not adhere to the naming rules for the flowgraph elements. Names of flowgraph elements may contain only upper-case letters, underscores, and digits and must be unique in the flowgraph.

Note

You need to be granted the CREATE ANY, ALTER, DROP, EXECUTE, SELECT, INSERT, and UPDATE privileges to the target schema of the flowgraph.

3. In the *Mappings* tab, use the *Mapping Editor* to remove or reorder input and output anchors and their attributes.
4. In the *Annotations* tab, use the *Annotations Editor* to edit the annotations of the flowgraph container.

Results

The settings made on the flowgraph container determine the type of runtime object generated during activation and the number and signatures of its inputs and outputs.

Related Information

[Flowgraphs \[page 427\]](#)

[Using the Mapping Editor \[page 462\]](#)

[Using the Annotation Editor \[page 464\]](#)

[Customizing the Node Palette \[page 470\]](#)

7.7.4.3 Adding an object from the Project Explorer

Drag and drop an object from the *Project Explorer* view to the *Editing Area*.

Prerequisites

You have opened a flowgraph in a project that has been shared with a HANA system.

Context

Nodes are the functional elements in a flowgraph. There are several types of nodes which represent data sources, data sinks, and operators in the flowgraph.

The following database objects are represented by nodes in a flowgraph.

- Development objects in the project:
 - Flowgraphs with no inputs and one output as Data Source nodes
 - Attribute Views as Data Source nodes
 - Calculation Views as Data Source nodes

Note

Flowgraphs that represent procedures with inputs or with more than one output cannot be directly inserted in other flowgraphs. However, it is possible to add the procedure generated by activating one flowgraph to another flowgraph. This is done via drag and drop from the catalog (see below) or by adding a Procedure node from the [Node Palette](#).

- Runtime objects in the catalog:
 - Tables as Data Source nodes, and as Data Sink nodes
 - Views as Data Source nodes
 - Procedures without scalar parameters and INOUT parameters as Procedure nodes
 - Table Types and Tables as anchors

→ Tip

You can also drag a Table Type or a Table to an anchor region to create a new anchor.

Procedure

- In the Project Explorer, select an object and drag it to the canvas of the flowgraph container.

If the dragged object is a table, a pop-up dialog lets you choose if this table is used as a data source or a data sink in the flowgraph.

A new node is added to the flowgraph. The type of the node matches the selected object in the Project Explorer. The flowgraph container is re-sized so that the new node is contained in the canvas of the flowgraph container.

Note

You need to be granted SELECT access rights on the schema that contains the object.

Note

In order to activate the flowgraph, database user `_SYS_REPO` needs to be granted SELECT object privileges for objects that are used as data sources and INSERT object privileges for objects that are used as data sinks.

⚠ Caution

The validation of the SAP HANA Application Function Modeler does not recognize when the signature of an input or output of a table or view has changed. In this case the signature of the respective input or output of the added node is inconsistent with that of the object. Consequently, the flowgraph activation fails.

- In the Project Explorer, select a table type or a table and drag it to an anchor region.

A new anchor with the same signature as the table type or the table is added to the anchor region at the position where the object was dropped.

ℹ Note

You need to be granted SELECT access rights on the object.

ℹ Note

Dragging a table to the anchor region only transfers the signature of the table to the anchor. No reference to the table or its content is stored in the flowgraph. Accordingly, no additional object privileges have to be granted to the database user `_SYS_REPO`.

Related Information

[Setting up the SAP HANA Application Function Modeler \[page 425\]](#)

[Attribute Views \[page 337\]](#)

[Calculation Views \[page 347\]](#)

7.7.4.4 Adding a node from the Node Palette

Drag a node template from the *Node Palette* to the canvas of the flowgraph container in the *Editing Area*.

Prerequisites

You have opened a flowgraph in a project that has been shared with a HANA system.

ℹ Note

The *Node Palette* is generated according to the functionality provided by the system. If you work in a project that is not shared with a system or the system is offline, the content of the *Node Palette* is restricted to a few basic relational operators. For example, the Data Source node and Data Sink node will be missing the *General* tab.

Context

Nodes are the functional elements in a flowgraph. There are several different types of nodes which represent data sources, data sinks, and operators in the flowgraph.

Procedure

In the Node Palette, select the entry you want to add and drag it to the canvas area of the flowgraph container.

Results

A new node is added to the flowgraph. The type of the node matches the selected node template in the [Node Palette](#). The flowgraph container is resized such that the new node is contained in its canvas.

Related Information

[Setting up the SAP HANA Application Function Modeler \[page 425\]](#)

7.7.4.5 Editing a node

Edit the properties of a node.

Context

The nodes in a flowgraph usually need to be configured. Relational nodes need configurations such as join conditions, filter predicates, and attribute sets for projection. Edit the configuration of a node by selecting the node and navigating to its [Properties](#) view. The selection of tabs and the configuration options in the [Properties](#) view depend on the type of node.

Tabs in the Properties view of a node

Tab name	Description	Optional
<i>General</i>	This tab always contains the following elements: • <i>Name</i>: name of the node (editable), • <i>Display Name</i>; name of the node template entry (read-only), • <i>Description</i>: description of the node template entry (read-only). In addition, this tab contains most configuration options that are specific to the particular node type.	No
<i>Script</i>	This tab is only relevant if an optional additional cost component offers Script node functionality (for example, R Integration).	Yes
<i>Mappings</i>	If the node defines a mapping of its inputs to its outputs or contains an anchor region, this mapping is displayed and can be edited in the <i>Mapping Editor</i>.	Yes
<i>INPUT (I) / OUTPUT (O)</i>	These tabs correspond to the input and output anchors of the node. They have the same names as the respective anchors and the same contents as the <i>All</i> tabs in the <i>Properties</i> views of the anchors.	Yes
<i>Annotations</i>	This tab contains the annotations of the node.	No
<i>All</i>	This tab is a summary of all tabs in this view except for the input and output anchor tabs.	No

Procedure

1. Select a node or add a new node.
2. Select the name of the node.
The name field becomes active for editing.

Note

The name of a node may contain only upper-case letters, underscores, and digits. It must be unique within the flowgraph.

3. In the *Annotations* tab of the *Properties* view, use the *Annotations Editor* to edit the annotations of the node.
4. Edit the remaining properties of the node in the *Properties* view. In particular, specify the type-specific properties of the node in the *General* tab.

Related Information

[Using the Mapping Editor \[page 462\]](#)

[Using the Annotation Editor \[page 464\]](#)

7.7.4.5.1 Data Source [Application Function Modeler]

Edit nodes that represent data sources.

Prerequisites

You added a Data Source node to the flowgraph.

Procedure

1. Drag the Data Source node onto the canvas.
You can click the magnifying glass icon to preview the existing data in the table or view.
2. In the [Select an Object](#) dialog, type the name of the object to add, or browse the object tree to select one or more objects, and click [OK](#).
3. In the [General](#) tab of the [Properties](#) view, use the drop-down menus [Authoring Schema](#) and [Catalog Object](#) to specify the data source.

Note

The check-box [Realtime Behavior](#) is only relevant if the flowgraph uses the additional cost SAP HANA smart data integration and SAP HANA smart data quality optional component and if a task plan is generated.

Tip

You can configure the authoring schema by choosing [Schema Mapping](#) in the [Quick](#) view of the [SAP HANA Modeler](#) perspective.

Results

The signature of the output anchor is set automatically.

Note

To activate the flowgraph, the database user `_SYS_REPO` needs SELECT object privileges for the chosen data source.

Next Steps

If a catalog object specified in a Data Source node changes, you can reconcile the differences between the structure of the catalog object and the structure in the node.

In the flowgraph editor, click the diff button to compare the catalog object with the structure in the Data Source node. Click [Reconcile](#) to update the node with the structure from the catalog object.

Note

Reconciling the Data Source node updates the flowgraph, but you must still save the flowgraph to make the changes permanent.

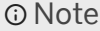
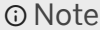
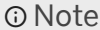
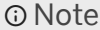
Related Information

[Setting up the SAP HANA Application Function Modeler \[page 425\]](#)

7.7.4.5.1.1 Data Source Options [Application Function Modeler]

Description of the options in the Data Source node.

Option	Description
Name	The name for the node.
Display Name	Note AFM only. The name shown in the Palette pane. Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	Note AFM only. (Optional.) Provides a comment about the source. For example, "West Region Sales Q1."
Type	Lists whether the data source is a view or table.
Authoring Schema	Lists the system or folder where the view or table is located.

Option	Description
Catalog Object	Lists the repository where the table or view is located
Realtime Behavior	Select to run in batch or real-time mode.
Partition Type	 Note Web-based Development Workbench only. Choose one of the following: None: does not partition the table Range: divides the table data into sets based on a range of data in a row. List: divides the table into sets based on a list of values in a row.
Attribute	 Note Web-based Development Workbench only. The column name used for the partition.
Partition name	 Note Web-based Development Workbench only. The name for the partition such as "region".
Value	 Note Web-based Development Workbench only. The range or list.

7.7.4.5.2 Data Sink [Application Function Modeler]

Edit nodes that represent data sinks.

Procedure

1. Drag the Data Sink node onto the canvas.
2. In the [Select an Object](#) dialog, type the name of the object to add, or browse the object tree to select one or more objects, and click [OK](#).
3. (Optional) You can click the magnifying glass icon to preview the existing data (if any) in the table. The data will change after the flowgraph runs.
4. In the [General](#) tab of the [Properties](#) view use the drop-down menus [Authoring Schema](#) and [Catalog Object](#) to specify the data sink.

→ Tip

You can configure the authoring schema by choosing [Schema Mapping](#) in the [Quick](#) view of the [SAP HANA Modeler](#) perspective.

5. Select [Truncate Table](#) to clear the table before inserting data. Otherwise, all inserted data is appended to the table.
6. Optionally, if the node is a Data Sink (Template Table) node, specify in the same tab in the drop-down menu [Data Layout](#) whether a table with row or column layout is created.
7. To optionally create a separate target table that tracks the history of changes, set the [History Table Settings](#) options.

Results

The signature of the input anchor is set automatically.

ⓘ Note

To activate the flowgraph, the database user `_SYS_REPO` needs INSERT and in case of truncation also DELETE object privileges for the chosen data sink.

Related Information

[Setting up the SAP HANA Application Function Modeler \[page 425\]](#)

7.7.4.5.2.1 Data Sink Options [Application Function Modeler]

Description of options for the Data Sink node.

Option	Description
Enter table or view name	ⓘ Note AFM only. Enter the name of the table or view.
Matching items	ⓘ Note AFM only. Shows matching tables or views as you begin typing in the previous option.

Option	Description
Name	The name for the output target.
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the target. For example, "West Region Sales Q1."
Type	Lists whether it is a view or table.
Authoring Schema	Lists the system or folder where the view or table is located.
Catalog Object	Lists the table or view.
Truncate Behavior	Limits the amount of data written to the Data Sink. In the SAP HANA Web-based Development Workbench, for the Truncate option, select it to clear the table before inserting data. Otherwise, all inserted data is appended to the table.
Writer Type	Choose from the following options: insert: adds new records to a table. upsert: if a record doesn't currently exist, it is inserted into a table. If the record exists, then it is updated. update: includes additional or more current information in an existing record.
Key Generation Attribute	Generates new keys for target data starting from a value based on existing keys in the column you specify.
Sequence Schema	When generating keys, select the schema where the externally created sequence file is located.
Sequence Name	When generating keys, select the externally created sequence to generate the new key values.
Change time column name	Select the target column that will be set to the time that the row was committed. The data type must be TIMESTAMP.
Change type column name	Select the target column that will be set to the row change type. The data type is VARCHAR(1).

7.7.4.5.3 Aggregation [Application Function Modeler]

An [Aggregation](#) node represents a relational group-by and aggregation operation.

Prerequisites

You have added an Aggregation node to the flowgraph.

Note

The Aggregation node is available for realtime processing.

Procedure

1. Select the Aggregation node.
2. Map the input columns and output columns by dragging them to the output pane. You can add, delete, rename, and reorder the output columns, as needed. To multiselect and delete multiple columns use CTRL/Shift keys, and then click [Delete](#).
3. In the [Aggregations](#) tab, specify the columns that you want to have the aggregate or group-by actions taken upon. Drag the input fields and then select the action from the drop-down list.
4. (Optional) Select the [Having](#) tab to run a filter on an aggregation function. Enter the expression. To view the options in the expression editor, click [Load Elements & Functions](#). You can drag and drop the input and output columns from the [Elements](#) pane, then drag an aggregation function from the [Functions](#) pane. Click or type the appropriate operators. For example, if you want to find the transactions that are over \$75,000 based on the average sales in the 1st quarter, your expression might look like this:
`AVG("Aggregation1_Input"."SALES") > 75000`.

Option	Description
Avg	Calculates the average of a given set of column values.
Count	Returns the number of values in a table column.
Group-by	Use for specifying a list of columns for which you want to combine output. For example, you might want to group sales orders by date to find the total sales ordered on a particular date.
Max	Returns the maximum value from a list.
Min	Returns the minimum value from a list.
Sum	Calculates the sum of a given set of values.

5. (Optional) Select the [Filter Node](#) tab to compare the column name against a constant value. Click [Load Elements & Functions](#) to populate the Expression Editor. Enter the expression by dragging the column names, the function, and entering the operators from the pane at the bottom of the node. For example, if you want to the number of sales that are greater than 10000, your expression might look like this:
`"Aggregation1_input"."SALES" > 10000`. See the *SAP HANA SQL and System Views Reference* for more information about each function.

6. Click [Save](#) to return to the Flowgraph Editor.

Related Information

[Using the Mapping Editor \[page 462\]](#)

[Alphabetical List Of Functions](#)

7.7.4.5.3.1 Aggregation Options

Description of options for the Aggregation node.

Option	Description
Name	The name of the node.
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the operation. For example, "Calculate total sales in May."
Column/Attribute	The input column name that you want to use in an Aggregation operation.
Aggregation/Action	Choose one of the following: Avg: calculates the average of a given set of column values. Count: returns the number of values in a table column. Group-by: use for specifying a list of columns for which you want to combine output. For example, you might want to group sales orders by date to find the total sales ordered on a particular date. Max: returns the maximum value from a list. Min: returns the minimum value from a list. Sum: calculates the sum of a given set of values.

7.7.4.5.4 Filter [Application Function Modeler]

A *Filter* node represents a relational selection combined with a projection operation. It also allows calculated attributes to be added to the output.

Prerequisites

You have added a Filter node to the flowgraph.

Note

The Filter node is available for real-time processing.

Context

Web-based Development Workbench

1. Drag the Filter node onto the canvas, and connect the source data or the previous node to the Filter node.
2. Double-click the Filter node.
3. (Optional) Enter a name for this Filter node in the *Node Name* option.
4. (Optional) Select *Distinct* to output only unique records.
5. (Optional) To copy any columns that are not already mapped to the output target, drag them from the Input pane to the Output pane. You may also remove any output columns by clicking the pencil icon or the trash icon, respectively. You can multi-select the columns that you do not want output by using the CTRL or Shift key, and then Delete.
6. (Optional) Click *Load Elements & Functions* to populate the Expression Editor. Drag input columns into the *Mapping* tab to define the output mapping and perform some sort of calculation. Choose the functions and the operators. For example, you might want to calculate the workdays in a quarter, so you would use the *Workdays_Between* function in an expression like this: `WORKDAYS_BETWEEN (<factory_calendar_id>, <start_date>, <end_date> [, <source_schema>])`. Click *Validate Syntax* to ensure that the expression is valid.
7. Click the *Filter node* tab and then click *Load Elements & Functions* to populate the Expression Editor. You can use the Expression Editor or type an expression to filter the data from the input to the output. Drag the input columns, select a function and the operators. For example, if you want to move all the records that are in Canada, your filter might look like this: `"Filter1_input"."COUNTRY" = "Canada"`. See the "SQL Functions" topic in the *SAP HANA SQL and System Views Reference* for more information about each function.
8. Click *Save* to return to the flowgraph.

Application Function Modeler

1. Select the Filter node.
2. Select the *General* tab of the *Properties* view.
3. Select the *Value Help* and use the *Expression Editor* to configure the *Filter Expression*.
4. Add additional attributes for calculated outputs in the *Output* tab.

5. Select the *Mappings* tab. In the *Mapping Editor*, define the output mapping of the node. In addition you can define the calculated attributes by first selecting the attribute in the *Target* list and then selecting *Edit Expression*.

The *Expression Editor* opens to edit the expression that calculates the attribute.

Note

You need to manually set the type of the calculated attribute.

6.

Example

Let's say that you have a single input source, and connected it to a Match node. You selected Most Recent as your survivor rule, so that the output from Match has a Group_Master column. Those duplicate records with the most recent Last_Updated date are marked with a value of "M". After connecting the Match node to the Filter node, you can use the following expression to output only the master and unique records:

Sample Code

```
("Filter1_Input"."GROUP_ID" is null) OR ("Filter1_Input"."GROUP_ID" is not null and "Filter1_Input"."GROUP_MASTER" = 'M')
```

Prior to the Filter node, some example data might look like the following.

Data input to the Filter node

GROUP_ID	RE-VIEW_GROUP	CON-FLICT_GROUP	LAST_UP-DATED	ADDRESS	ADDRESS2	GROUP_MAS-TER
<null>	<null>	<null>	<null>	1411 Broadway	New York 10018	<null>
<null>	<null>	<null>	<null>	3 Fleetwood Dr	Newberg NY 12550	<null>
<null>	<null>	<null>	<null>	300 Cliffside Dr	Atlanta GA 30350	<null>
1	N	C	01/01/16	332 Front St	La Crosse WI 54601	M
1	N	C	03/10/11	332 Front St	La Crosse WI 54601	<null>
1	N	C	07/04/15	332 Front St	La Crosse WI 54601	<null>
<null>	<null>	<null>	<null>	3738 North Fraser Way	Burnaby BC V3N 1E4	<null>

After the Filter node, you can see that two duplicate entries were removed, and only the master record and the other four unique records are output.

Data output from the Filter node

GROUP_ID	RE-VIEW_GROUP	CON-FLICT_GROUP	LAST_UP-DATED	ADDRESS	LASTLINE	GROUP_MAS-TER
<null>	<null>	<null>	<null>	1411 broadway	new york 10018	<null>
<null>	<null>	<null>	<null>	3 Fleetwood Dr	Newberg NY 12550	<null>
<null>	<null>	<null>	<null>	300 the cliffsup	atlanta 30350	<null>
1	N	C	01/01/16	332 Front st	La Crosse 54601	M
<null>	<null>	<null>	<null>	3738 NORTH FRASER WAY TH 6203	BURNABY BC	<null>

Related Information

[Using the Mapping Editor \[page 462\]](#)

[Using the Expression Editor \[page 463\]](#)

7.7.4.5.4.1 Filter Options [Application Function Modeler]

Description of options for the Filter node.

Option	Description
Name	The name for the node.
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the node. For example, "Only European Data."

Option	Description
Distinct	 Note Web-based Development Workbench only. (Optional). Select to output only unique records. The records must match exactly. If you know that you have duplicates, but have a ROW_ID column, or another column that has a unique identifier for each record, then you will want to suppress that column in the Filter node. The Distinct option is not available for CLOB, NCLOB, BLOB, TEXT datatypes
Filter Node	Enter an expression so that only the valid records are output based on the expression criteria. You can enter some SQL statements to set the value of the target column. Any of the SAP HANA SQL functions can be used. See the <i>SAP Hana SQL and System Views Reference</i>.  Note In AFM, you can use the Expression Editor to assist in creating the expression.

7.7.4.5.5 Join [Application Function Modeler]

A Join node represents a relational multi-way join operation.

Prerequisites

You have added a Join node to the flowgraph.

Note

The Join node is not available for real-time processing.

Context

The Join node can perform multiple step joins on two or more inputs.

Procedure

1. Select the Join node.
2. (Optional) Add additional input anchors.

- (Optional) Remove any output columns by clicking the pencil icon or the trash icon, respectively. You can multi-select the columns that you do not want output by using the CTRL or Shift key, and then Delete. The Mapping column shows how the column has been mapped with the input source.
- In the *Properties* view, select the *General* tab to configure the type of the join (inner join, left outer join, or right outer join).
- In the table defined in the *General* tab, use the *Table Editor* to define the *Left* join partner, the *Join Type*, the *Right* join partner and the *Join Condition* of each join step. In this, only the first entry in the join condition consists of a *Left* join partner and a *Right* join partner. Every subsequent join condition has the previous join tree as *Left* join partner.
The *Expression Editor* opens and lets you specify the *Join Condition*.
- In the *Mappings* tab, use the *Mapping Editor* to edit the output attributes of the join.

Related Information

[Using the Table Editor \[page 461\]](#)

[Using the Mapping Editor \[page 462\]](#)

[Using the Expression Editor \[page 463\]](#)

[Adding an anchor \[page 457\]](#)

7.7.4.5.5.1 Join Options [Application Function Modeler]

Description of options for the Join node.

Option	Description
Name	The name for the node.
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the node. For example, "Employee_v8 and Employee_v12."
Left	The left source of a join.

Option	Description
Join Type	Choose from one of these options: Inner: use when each record in the two tables has matching records. Left_Outer: output all records in the left table, even when the join condition does not match any records in the right table. Right_Outer: output all records in the right table, even when the join condition does not match any records in the left table.
Right	The right source of a join.
Join Condition	The expression that specifies the criteria of the join condition. Note In AFM, you can use the Expression Editor to assist in creating the expression.
Add	A join condition is created.
Remove	The highlighted join condition is deleted.

7.7.4.5.6 Sort [Application Function Modeler]

A Sort node represents a relational sort operation.

Prerequisites

You have added a Sort node to the flowgraph.

Context

The Sort node performs a sort by one or more attributes of the input.

Note

The Sort node is available for real-time processing.

Procedure

1. Select the Sort node.

2. In the *Properties* View, select the *General* tab to configure the sort order.
3. In the *General* tab, use the *Table Editor* to define the *Attributes* and the *Sort Order* by which the input is sorted. It is possible to specify several *Attributes* with descending priority.

Related Information

[Using the Table Editor \[page 461\]](#)

7.7.4.5.6.1 Sort Options [Application Function Modeler]

Description of options for the Sort node.

Option	Description
Name	The name for the node.
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the node. For example, "Sort ascending sales order."
Column/Attribute	The column used for sorting.
Sort Type/Sort Order	How to sort the data. Ascending: When sorting numerical data, put the smallest number first. When sorting alphabetically, start with the first letter. Descending: When sorting numerical data, put the largest number first. When sorting alphabetically, start with the last letter.
Add	A row is configured to be used for sorting.
Remove	The highlighted entry is deleted, so that it will not be used in sorting.
Up	The entry is moved up so that it is sorted before any entries below it.
Down	The entry is moved down so that it is sorted after any entries above it.

7.7.4.5.7 Union [Application Function Modeler]

A Union node represents a relational union operation.

Prerequisites

You have created a Union node in the flowgraph.

Context

The union operator forms the union from two or more inputs with the same signature. This operator can either select all values including duplicates (UNION ALL) or only distinct values (UNION).

ⓘ Note

The Union node is available for real-time processing.

Procedure

1. Select the Union node.
2. (Optional) Add additional input anchors.
3. In the *General* tab of the *Properties* view define whether the operator is a UNION ALL or a UNION operator by selecting or unselecting the checkbox *Create Union All*.

Related Information

[Adding an anchor \[page 457\]](#)

7.7.4.5.7.1 Union Options [Application Function Modeler]

Description of options for the Union node.

Option	Description
Name	The name for the node.

Option	Description
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the node. For example, "Combine HR2015 and HR2010."
Create Union All	The option to merge all of the input data (including duplicate entries) into one output, when selected.

7.7.4.5.8 Procedure [Application Function Modeler]

Use procedures from the catalog in the flowgraph.

Context

 **Note**
The Procedure node is not available for real-time processing.

Procedure

1. Drag the Procedure node onto the canvas.
2. In the [Select an Object](#) dialog, type the name of the object to add, or browse the object tree to select one or more objects, and click *OK*.
3. Select the Procedure node.
4. The following step applies only if you added the Procedure node from the [Node Palette](#).
 - In SAP HANA studio, in the [General](#) tab of the [Properties](#) view, select the drop-down menus for the [Schema](#) and the [Procedure](#) that is represented by the node.

- In SAP HANA Web-based Development Workbench, open the node and select a *Schema Name* and the *Procedure Name* for the node.
5. To activate the flowgraph, the database user `_SYS_REPO` needs EXECUTE object privileges for all procedures represented by Procedure nodes.

7.7.4.5.8.1 Procedure options [Application Function Modeler]

Description of options for the Procedure node.

Option	Description
Name	The name for the node.
Display Name	 Note AFM only. The name shown in the Palette pane.  Note This option can only be changed when creating a template. It cannot be changed when using the node outside of a template.
Description	 Note AFM only. (Optional.) Provides a comment about the node. For example, "Run schedule."
Schema	The location and definition of the procedure.
Procedure	The stored procedure that you want to run in the flowgraph.

7.7.4.5.9 AFL Function [Application Function Modeler]

Access functions of the Application Function Library.

Prerequisites

You have added an AFL Function node to the flowgraph.

Context

Use this node to model functions of the Application Function Library (AFL) that are registered with the system. AFL functions are grouped by function areas.

Note

You can retrieve the list of all AFL areas and functions registered in a HANA system by viewing the content of the views "SYS"."AFL_AREAS" and "SYS"."AFL_FUNCTIONS".

Many AFL areas are optional components for HANA. For some of these optional components the SAP HANA Application Function Modeler (AFM) provides preconfigured node templates. In this case, the AFM automatically displays a separate compartment for this area in the *Node Palette*.

Note

You can refresh the *Node Palette* by choosing *Refresh* in its context-sensitive menu.

Note

The AFL Function node is not available for real-time processing.

Procedure

1. Select the AFL Function node.
2. In the *General* tab in the *Properties* view, select the drop-down menus for *Area* and the *Function*.
The AFM changes the inputs and outputs of the node according to the existing meta-data for the function on the server.

Note

For some AFL areas there exists a preconfigured *Node Palette* compartment. You cannot change the *Area* or the *Function* of a node added from one of these compartments.

3. If applicable, change the *Category* of the function.
4. Specify the inputs and the outputs of the function by editing the signature and the fixed content of its anchors.

Note

For some AFL areas there exists a preconfigured node template for this function. In this case, the fixed content of the inputs that define parameters is preconfigured.

Related Information

[Using the Table Editor \[page 461\]](#)

7.7.4.6 Adding an anchor

Add an anchor to an anchor region of the flowgraph container or a node.

Context

The flowgraph container and some nodes (for example, the Join node and the Union node) can have a variable number of input or output anchors. In the flowgraph, this is represented by the existence of an anchor region for the corresponding set of anchors. New anchors can be added to the anchor region.

Procedure

1. Right-click on the anchor region at the position you want to add the new anchor.
2. In the context-sensitive menu, choose *Add Input* or *Add Output* (depending on whether you selected an anchor region for inputs or outputs).

Results

A new anchor with an empty signature is added to the anchor region at the mouse pointer position where the context menu is opened.

Note

Instead of adding an anchor via the context-sensitive menu, you can also copy an existing anchor to an anchor region. This has the advantage that the new anchor has a fully defined signature.

Note

Alternatively, you can add a new anchor while creating a connection. In this case the new anchor inherits the signature from the source anchor of the connection.

Note

A third option to add an anchor with a predefined signature is by dragging a table or a table type from the catalog to the anchor region.

Note

You can also delete an anchor that you added to an anchor region. Some anchor regions have a minimum number of anchors (for example, the anchor regions for the inputs of the Join node and the Union node each have to contain at least two anchors). In this case, if the anchor region contains the minimum number of anchors, then no anchor in the anchor can be deleted.

Related Information

[Adding an object from the Project Explorer \[page 434\]](#)

[Editing the flowgraph container \[page 433\]](#)

[Join \[Application Function Modeler\] \[page 449\]](#)

7.7.4.7 Editing an anchor

Change and define input and output table types.

Context

Anchors define inputs and outputs to the flowgraph container and to nodes.

Tabs in the Properties view of an anchor.

Tab name	Description	Optional
General	This tab contains the following entries: • <i>Name</i>: name of the anchor • <i>Kind</i>: kind of the anchor (Table, Column, Scalar).	No
Signature	In this tab, you can use the Table Editor to change the signature of the anchor. Anchors of the kind Scalar or Column are considered to be tables with one column.	No
Fixed Content	This tab exists only for input anchors. While the checkbox <i>Fixed Content</i> is selected, the anchor cannot be the target of a connection. Instead, a table providing the input is stored in the flowgraph with the anchor. The table can be edited using the Table Editor in this tab.	Yes
All	This tab is a summary the other tabs.	No

📘 Note

Most anchors have a fixed kind that cannot be changed. Currently, the anchor kinds "Column" and "Scalar" are only supported for input anchors of AFL Function nodes.

📘 Note

Many anchors either have a fixed signature or obtain their signature via an automatic table mapping.

Procedure

1. Select the anchor.
2. Select the name of the anchor and edit it in the direct editing area.
The name field becomes active for editing.

Note

The name of an anchor must consist of upper-case letters, underscores, and digits. It must be unique in the flowgraph.

3. Use the [Table Editor](#) to edit the signature of the anchor in the [Signature](#) tab of the [Properties](#) view.
4. Select the [Fixed Content](#) tab in the [Properties](#) view.
5. If you want to embed the content of the anchor with the flowgraph, select the checkbox [Fixed Content](#).
If the checkbox [Fixed Content](#) is selected, the embedded table is shown in the [Fixed Content](#) tab. Use the [Table Editor](#) to edit the table.

Note

For some areas of the Application Function Library the SAP HANA application function modeler provides template AFL Function nodes in separate compartments of the [Node Palette](#). These template nodes are preconfigured with fixed signature tables if the respective input is a design-time parameter of the node.

Related Information

[Flowgraphs \[page 427\]](#)

[Using the Table Editor \[page 461\]](#)

[Using the Mapping Editor \[page 462\]](#)

7.7.4.8 Creating a connection

Create a new connection between two nodes or a node and the flowgraph container.

Context

A connection represents the directed flow of data from a source to a target. The source and the target of a connection are anchors. The connection defines a table mapping between these table types defined by its source and target. The source of a connection is either an input anchor of the flowgraph container or an output anchor of a node. The target of a connection is either an output anchor of the flowgraph container or an input anchor of a node. An anchor can be the source of several connections. It can only be the target of one connection. A fixed content anchor cannot be the target of any connection.

Procedure

1. Select without releasing the *Connect* button  in the context button pad of the source anchor of the connection.
2. Drag a connection to the target anchor.

Note

Depending on the node of the target anchor, the *Create Input Table Mapping* wizard may open. This wizard helps you to choose the right mapping for the connection. You can still change this mapping in the *Mapping Editor* after completing the wizard. To open the wizard again, you have to remove the connection and create it again.

Results

A new connection between the source anchor and the target anchor is created. If possible, the signature of the source anchor is copied to the target anchor and propagated forward through the flowgraph.

Note

You can also add a new anchor to an anchor region and create a connection to this anchor in a single action. Instead of dragging the connection to an anchor, drag it to a free position in an anchor region. A new target anchor with the same signature as the source anchor is added before the connection is created.

Related Information

[Adding an anchor \[page 457\]](#)

7.7.4.9 Editing a connection

Edit the mapping represented by a connection.

Prerequisites

You have created a connection.

Context

A connection represents a mapping between the table types defined by the source anchor and the target anchor. The SAP HANA application function modeler tries to auto-generate a suitable mapping depending on the types of nodes connected by the mapping. The mapping can also be configured manually using the [Mapping Editor](#).

Tabs in the Properties view of a connection.

Tab name	Description	Optional
Mappings	This tab displays the mapping represented by the connection in the Mapping Editor .	No

Procedure

Select the connection and use the [Mapping Editor](#) in the [Mappings](#) tab of the [Properties](#) view to edit the mapping defined by the connection.

Related Information

[Flowgraphs \[page 427\]](#)

[Using the Mapping Editor \[page 462\]](#)

[Editing an anchor \[page 458\]](#)

7.7.4.10 Using the Table Editor

Edit embedded table-like anchor signatures and fixed content tables.

Context

Embedded tables appear in various flowgraph elements. For example, anchors have signature tables and may have fixed content tables. Specialized nodes may have tables in the [General](#) tab of the [Properties](#) view. The SAP HANA Application Function Modeler provides a [Table Editor](#) to edit these tables.

Procedure

- Add, remove, and re-order rows of the embedded table by selecting the respective operations on the right side of the [Table Editor](#).

- Edit an entry in the table by double-clicking the respective cell.

Related Information

[Flowgraphs \[page 427\]](#)

[Editing an anchor \[page 458\]](#)

7.7.4.11 Using the Mapping Editor

Edit the mappings between table types in the [Mappings](#) tab of the [Properties](#) view of a flowgraph element.

Prerequisites

You have selected the [Mappings](#) tab of the [Properties](#) view of a flowgraph element.

Context

A mapping is a projection between table types. The [Mapping Editor](#) allows you to edit mappings between a number of source and target table types. The left side of the editor shows the source table types, the right side shows the target table types. A binding of two attributes is indicated by a line between them.

ⓘ Note

The mapping editor is used to define the mappings of connections and possible projections within nodes (for example, the Filter node, the Join Node, and the Union Node). It is also used to edit this inputs and outputs of the flowgraph container and of nodes which do not define a projection. In this case, no lines are drawn between the attributes.

ⓘ Note

Not all flowgraph elements allow free editing of all their mappings and table types. In this case the functionality of the [Mapping Editor](#) is restricted to the permitted editing operations.

Procedure

- (Optional) To remove a table type, select it and press the minus sign on the right side of the [Mapping Editor](#).
- To re-order the source or target table types, click on a table type and use the up/down arrows on the right side of the [Mapping Editor](#).

- (Optional) To remove an attribute, select it and press the minus sign on the right side of the [Mapping Editor](#).
- (Optional) To re-order the source or target attributes, click on an attribute and use the up/down arrows on the right side of the [Mapping Editor](#).
- (Optional) To add an attribute from the source type to the target type, drag the source attribute and drop it on the root of the target tree.
The attribute is appended at the end of the target attribute list. If the [Mapping Editor](#) defines a mapping, it is connected by a line with the source attribute indicating an attribute binding.
- (Optional, only available if the [Mapping Editor](#) defines a mapping) To re-assign a source attribute to a target attribute that is already assigned, drag the source attribute to the target attribute.
The old binding is replaced by the new one.

Related Information

[Flowgraphs \[page 427\]](#)

[Editing the flowgraph container \[page 433\]](#)

[Editing a node \[page 437\]](#)

[Editing a connection \[page 460\]](#)

7.7.4.12 Using the Expression Editor

Compose expressions for filters, join conditions, and calculated attributes.

Context

The [Expression Editor](#) allows you to compose SQL expressions based on table type attributes and functions. It consists of an [Function Palette](#) on the top, an [Attribute Palette](#) on the left and a [Text Field](#) on the right.

❗ Note

The expression validation is disabled in the SAP HANA Application Function Modeler.

Procedure

- Type the expression in the [Text Field](#).

❗ Note

Press CTRL + Space bar for auto-completion.

- Select operators and functions in the [Function Palette](#) to add them to the [Text Field](#).

- Drag attributes from the [Attribute Palette](#) to the [Text Field](#).

Related Information

[Aggregation \[Application Function Modeler\] \[page 444\]](#)

[Filter \[Application Function Modeler\] \[page 446\]](#)

[Join \[Application Function Modeler\] \[page 449\]](#)

7.7.4.13 Using the Annotation Editor

Add arbitrary annotations to the flowgraph container or a node.

Context

The flowgraph container and all nodes have an [Annotation](#) tab in their [Properties](#) view. Annotations are nested key-value pairs. The SAP HANA Application Function Modeler (AFM) provides an [Annotation Editor](#) to edit existing annotations like the `sap.afm.palette` annotation or to add your own annotations.

ⓘ Note

When the flowgraph is activated, all annotations are exposed in a table with the name extension `.META` in the flowgraph target schema. This way, they can be consumed at runtime.

ⓘ Note

For some nodes, the annotations `sap.afm.displayName` and `sap.afm.description` are visible in the [Annotation Editor](#). These annotations are for internal use of the AFM and not supposed to be modified.

Procedure

- Add, remove, and re-order annotations by selecting the respective operations on the right side of the [Annotation Editor](#).
- Edit the Key and the Value of an annotation by double-clicking the respective cell.
- Add nested annotations by first selecting an annotation row and then the [Add Child](#) operation on the right side of the [Annotation Editor](#).
A nested annotation appears below the selected annotation.
- Collapse and expand nested annotations by selecting the triangle to the left of an annotation key.

Related Information

[Flowgraphs \[page 427\]](#)

[Customizing the Node Palette \[page 470\]](#)

7.7.5 Tutorial: Creating a Runtime Procedure using Application Function Modeler (AFM)

At the end of this tutorial, you will have created and tested a runtime procedure with the AFM

Prerequisites

- You have access to a running SAP HANA development system.
- You have a valid user account in the SAP HANA database on that system.
- Your user has been granted the MODELING role.
- Your user has been granted the EXECUTE privilege for the object SYS.REPOSITORY_REST.
- Your user has been granted the following repository package privileges:
 - repo.read
 - repo.activate_native_objects
 - repo.edit_native_objects
 - repo.maintain_native_packages
- The system user _SYS_REPO has SELECT and ALTER privileges on the schema of your user.
- You have access to SAP HANA Studio and opened the [SAP HANA Development](#) perspective.
- You have created a system in the [System](#) view in the and logged on to this system with your user.
- You have created a repository workspace for the system.
- You have created a project in the [Project Explorer](#) view and shared it with the system via the workspace.

→ Tip

To share a project, right-click on the project and choose ► [Team](#) ► [Share](#) ► [Project](#) ► in the context-sensitive menu. In the [Share Project](#) wizard, choose [SAP HANA Repository](#) on the first page and choose your repository workspace on the second page.

Context

This tutorial leads you through the most common steps of using the SAP HANA Application Function Modeler (AFM). At the end of this tutorial, you will have created and tested a runtime procedure with the AFM.

Procedure

1. Open the SQL console of the system and create the table type WEATHER and the two tables NORTH and SOUTH in your user's schema by executing the following script.

```
CREATE TYPE "WEATHER" AS TABLE ("REGION" VARCHAR(50), "SEASON" VARCHAR(50),  
"TEMPERATURE" INTEGER);  
CREATE COLUMN TABLE "NORTH" LIKE "WEATHER";  
INSERT INTO "NORTH" VALUES ('North', 'Spring', 10);  
INSERT INTO "NORTH" VALUES ('North', 'Summer', 23);  
INSERT INTO "NORTH" VALUES ('North', 'Autumn', 12);  
INSERT INTO "NORTH" VALUES ('North', 'Winter', 2);  
CREATE COLUMN TABLE "SOUTH" LIKE "WEATHER";  
INSERT INTO "SOUTH" VALUES ('South', 'Spring', 18);  
INSERT INTO "SOUTH" VALUES ('South', 'Summer', 34);  
INSERT INTO "SOUTH" VALUES ('South', 'Autumn', 23);  
INSERT INTO "SOUTH" VALUES ('South', 'Winter', 12);
```

After refreshing the catalog, the table type WEATHER with the three attributes REGION, SEASON, and TEMPERATURE appears in the directory [Procedures > Table Types](#) of your user's schema. The two tables NORTH and SOUTH with the same signature appear in the directory [Tables](#) your user's schema.

NORTH

REGION	SEASON	TEMPERATURE
North	Spring	10
North	Summer	23
North	Autumn	12
North	Winter	2

SOUTH

REGION	SEASON	TEMPERATURE
South	Spring	18
South	Summer	34
South	Autumn	23
South	Winter	12

2. In the [Project Explorer](#) view, right-click on the existing project and choose [New > Other](#) in the context-sensitive menu.
The [New](#) wizard appears.
3. Choose [SAP HANA > Database Development > Flowgraph Model](#), and then click [Next](#).
The [New Flowgraph Model](#) wizard appears.
4. In the text field [File Name](#) enter `avg_temp` as name of the new flowgraph and select [Finish](#).
The system automatically adds the file extension `.hdbf` lowgraph. The AFM opens and in the [Editing Area](#) the empty flowgraph container is displayed.
5. Select the flowgraph container and enter the schema of your user to the [Target Schema](#) field in the [Properties](#) view.
6. Add the table NORTH from the [Node Palette](#) to the flowgraph. For this, drag the Data Source entry from the [General](#) tab of the [Node Palette](#) (located on the right side of the AFM) to the flowgraph (choose any free

space inside the canvas of the flowgraph container). Choose the table NORTH from the schema of your user in the dialog that appears.

The node NORTH is added to the flowgraph.

7. Add the table SOUTH from the catalog to the flowgraph. For this, navigate in the catalog to the directory *Tables* in your schema (either in the *Project Explorer* view or in the *Systems* view). Drag the table SOUTH from the catalog to the flowgraph (place it below the NORTH node). Choose *Data Source* in the dialog that appears.

The node SOUTH is added to the flowgraph.

8. Add a Union node to the flowgraph. For this, drag the Union entry from the *General* tab of the *Node Palette* to the flowgraph (place it right of the other two nodes).

The node UNION is added to the flowgraph.

9. Create a connection between the DATA anchor of the NORTH node and the INPUT1 anchor of the UNION node. Click the *Connect* button  in the context button pad of the DATA anchor and drag a connection to the INPUT1 anchor.

A connection between the NORTH node and the UNION node is created.

10. Create a second connection between the DATA_2 anchor of the SOUTH node and the INPUT2 anchor of the UNION node.

A connection between the SOUTH node and the UNION node is created.

11. Create a connection between the OUTPUT anchor of the UNION node and the output anchor region of the flowgraph container (the light-blue area at its right boundary).

The output anchor OUTPUT_2 is added to the output anchor region of the flowgraph container and a connection between the UNION node and the new anchor is created.

12. Save the flowgraph. Select **File > Save** in the HANA Studio main menu.

13. Activate the flowgraph. For this, right-click the flowgraph object in the *Project Explorer* view and choose **Team > Activate** from the context-sensitive menu.

A new procedure is generated in the schema of your user.

Caution

If the system user `_SYS_REPO` does not have SELECT and ALTER privileges then the activation fails.

14. Execute the generated procedure. For this, select the *Execute* button  in the top right corner of the AFM. The *Data Preview* view opens. It contains a tab with the SQL command that calls the generated procedure (with no input and one output) and a tab with the result of the procedure. This result is the union of the tables NORTH and SOUTH.

15. Return to the *AFM* view for the `avg_temp` flowgraph.

16. Add an Aggregation node from the *GeneralNode Palette* to the flowgraph (place it right of the UNION node).

17. The node AGGREGATION is added to the flowgraph.

18. Connect the OUTPUT anchor of the UNION node with the INPUT anchor of the AGGREGATION node. compartment of the

The *Mapping Editor* for the connection is shown in the *Properties* view.

19. In the *Target* area of the *Mapping Editor* for the new connection, select the attribute SEASON of the target INPUT. Remove this attribute by clicking the *Remove* button on the right side of the *Mapping Editor*.

The attribute SEASON and the corresponding mapping are deleted.

20. Select the AGGRAGATION node. In the *General* tab of its *Properties* view double-click the action of the attribute TEMPERATURE and change it to the value AVG.

21. Create a connection between the OUTPUT_3 anchor of the AGGREGATION node and the output anchor region of the flowgraph container.
The output anchor OUTPUT_4 is added to the output anchor region of the flowgraph container and a connection between the AGGREGATION node and the new anchor is created.
22. Save and activate the flowgraph. Execute the generated procedure.
The *Data Preview* view opens again. It contains a tab with the SQL command that calls the generated procedure (with no input and two outputs) and two tabs with the results of the procedure. One result is still the union of the tables NORTH and SOUTH. The other result shows in two rows the average temperatures for the regions North and South.
23. Return to the *AFM* view for the *avg_temp* flowgraph.
24. Delete the OUTPUT_2 anchor of the flowgraph container by choosing *Delete* in its context menu (or the respective button in the context button pad).
25. Save and activate the flowgraph. Execute the generated procedure.
The *Data Preview* view opens again. It contains a tab with the SQL command that calls the generated procedure (with no input and one output) and a second result tab that again shows in two rows the average temperatures for the regions North and South.
26. Return to the *AFM* view for the *avg_temp* flowgraph.
27. Delete the SOUTH node from the flowgraph.
The SOUTH node and its connection to the UNION node is deleted.
28. Create an additional input anchor for the flowgraph by adding the table type WEATHER from the catalog.
For this, navigate to the directory ► *Procedures* ► *Table Types* ► in the catalog and drag the entry WEATHER to the input anchor region of the flowgraph container.
The input anchor DATA_2 is added to the flowgraph.
29. Create a connection between the new DATA_2 anchor and the INPUT_2 anchor of the UNION node.
A new connection between the DATA_2 anchor and the UNION node is created.
30. Save and activate the flowgraph. Execute the generated procedure.
A dialog appears where you can choose the free input DATA_2. Enter the table SOUTH in your user's schema to the Catalog Object field. The *Data Preview* view opens. Again, it contains a tab with the SQL command that calls the generated procedure (with one input and one output) and a second result tab that shows in two rows the average temperatures for the regions North and South.
31. Close the flowgraph. Select ► *File* ► *Close* ► in the HANA Studio main menu.

Results

You have created a stored procedure that has one input table of the table type WEATHER and one output table that is produced by first forming the union of the table NORTH with the input table and then calculating the average temperature of each season. This procedure can now be used in any application that consumes stored procedures.

Related Information

[Modeling a flowgraph \[page 429\]](#)

[Tutorial: Add an SAP HANA System \[page 38\]](#)

7.7.6 Node palette flowgraphs

A node palette flowgraph represents a node palette or a node palette compartment.

The *Node Palette* of the SAP HANA application function modeler is customizable. A custom node palette is represented by a node palette flowgraph. These flowgraphs have the file extension `.hdbflowgraphtemplate` in the *Project Explorer* view.

A node palette flowgraph contains Palette Container and template nodes. These represent the compartments or sub-compartments and the node templates of the corresponding node palette. The Palette Container nodes and template nodes have a nested structure. This structure represents the hierarchy of the corresponding node palette. Moreover, all nodes in a node palette flowgraph are aligned on a horizontal line. Their order (from left to right) represents the order of the node palette entries (from top to bottom).

The *Node Palette* hierarchy can have up to three levels.

1. The first level contains the compartments (for example, the *General* compartment of the application function modeler *Node Palette*). Nodes are not permitted on this level.
2. The second level contains nodes (for example, the Filter node) and sub-compartments.
3. The third level contains only nodes.

A node palette flowgraph represents either a complete node palette or a compartment of the node palette. In the first case, the nesting depth of the node palette flowgraph is at least two and at most three, in the second case, the nesting depth is at most two.

Each flowgraph can be assigned its own custom node palette. This is specified either on creation of the flowgraph or in the *Annotations* tab of the *Properties* view of the flowgraph container.

Related Information

[Flowgraphs \[page 427\]](#)

[Creating a flowgraph \[page 431\]](#)

[Editing the flowgraph container \[page 433\]](#)

7.7.6.1 Exporting the Node Palette

Export the *Node Palette* as a node template flowgraph.

Procedure

1. Right-click the *Node Palette* and choose *Export entire palette* from the context-sensitive menu. The *Save As* wizard appears.
2. Navigate to the directory of your project and save the node template flowgraph file with the extension `.hdbflowgraphtemplate` in this project.

Refresh the [Project Explorer](#) view, and then the node template flowgraph is available in your project.

Note

The standard location of HANA projects on your local system is the directory `hana_work` in the home directory of your local user. There you find a sub-directory corresponding to the system shared with your project. The directory of the project is then located in the sub-directory `__empty__`.

7.7.6.2 Customizing the Node Palette

Customize the node palette of a flowgraph by adding a reference to a node palette flowgraph to the annotations of its flowgraph container.

Context

A flowgraph can be assigned a custom node palette. This can be done in three ways.

- Add additional compartments to the existing AFM node palette.
- Add additional compartments to an empty node palette.
- Add additional compartments to a custom node palette.

Note

The recommended way to customize the node palette of a flowgraph is via the [New Flowgraph Wizard](#) during the creation of the flowgraph. The following procedure of directly editing the annotations of the flowgraph container is only advised if you actually need to change the node palette of an existing flowgraph.

Procedure

1. Open the [Annotations](#) tab in the [Properties](#) view of the flowgraph container of a flowgraph.
2. If the annotation does not exist, add the annotation with the key `sap.afm.palette`.
3. (Optional) Insert the name of a node palette flowgraph (with the extension `.hdbflowgraphtemplate`) to the [Value](#) of this annotation. This replaces the default AFM node palette with the custom node palette defined by the specified node palette flowgraph.
4. If the nested annotation with the key `isDefaultUsed` does not exist, add it as a child to the annotation `sap.afm.palette`.
The [Value](#) of this annotation determined if the default AFM node palette is shown.
5. If the nested annotation with the key `additions` does not exist, add it as a child to the annotation `sap.afm.palette`.
6. (Optional) Insert a comma-separated list of names of node palette flowgraphs (with the extension `.hdbflowgraphtemplate`) to the [Value](#) of this annotation. This adds the compartments defined by the specified node palette flowgraph to the node palette of the flowgraph.

Related Information

[Creating a flowgraph \[page 431\]](#)

7.7.6.3 Editing a node palette flowgraph

Edit a node palette flowgraph to model a custom node palette.

Prerequisites

You have exported the [Node Palette](#) of the SAP HANA application function modeler to a node palette flowgraph `Template.hdbflowgraphtemplate`.

In addition, you have created a new (standard) flowgraph `Custom.hdbflowgraph` with the advanced option of choosing the node palette flowgraph `Template.hdbflowgraphtemplate` as the Custom Node Palette.

Context

A node palette flowgraph represents a custom node palette for the application function modeler. Node palette flowgraphs can be edited with the application function modeler like standard flowgraphs. The behavior of the application function modeler when editing node palette flowgraphs differs in two aspects from the editing of standard flowgraphs.

1. All nodes in the node palette flowgraph are automatically aligned on a horizontal line. By this, the order of the nodes (left to right) represents the order of the custom node palette entries (top to bottom).
2. The node palette flowgraph contains nested Palette Container nodes. These nodes represent the hierarchical structure of the custom node palette. These nodes look and behave similar to the flowgraph container.

In the following step by step tutorial, we use the application function modeler to customize the node palette of the Custom flowgraph by editing the Template node palette flowgraph. We cover only those aspects of modeling node palette flowgraphs that differ from modeling standard flowgraphs.

Procedure

1. Open the Custom flowgraph with the application function modeler.
The application function modeler displays the empty Custom flowgraph with a custom node palette defined by the Template node palette flowgraph. At this point, this is still the default application function modeler node palette.
2. Open the Template node palette flowgraph with the application function modeler.

The Template node palette flowgraph is displayed in a separate tab. The flowgraph container contains Palette Container nodes representing the top compartments of the node palette for the Custom flowgraph.

Note

A node palette flowgraph contains no connections. Therefore the flowgraph container has no anchor regions. Creating connections is disabled when editing node palette flowgraphs.

3. Right-click the GENERAL node and choose [Collapse/Expand](#) in the context-sensitive menu. The GENERAL node expands. It contains the template nodes of the General compartment of the application function modeler node palette.

Note

You can collapse a Palette Container node by choosing again [Collapse/Expand](#) in the context-sensitive menu.

4. Drag the JOIN node to a position between the SORT node and the UNION node. The auto-layout function of the application function modeler rearranges the nodes such that the JOIN node and the SORT node have effectively swapped positions.
5. Switch to the editing tab of the Custom flowgraph. Refresh the custom [Node Palette](#) by right-clicking the [Node Palette](#) and choosing [Refresh](#) in the context-sensitive menu. The Join node template and the Sort node template have swapped places in the General compartment of the [Node Palette](#).
6. Switch to the editing tab of the Template node palette flowgraph. Add a Palette Container node to the GENERAL node by dragging the corresponding node template from the [General](#) compartment of the [Node Palette](#) to the canvas of the GENERAL node. A nested Palette Container node named COMPARTMENT is added to the GENERAL node.
7. Add an object from the [Project Explorer](#) view to the canvas of the COMPARTMENT node.
8. Switch to the editing tab of the Custom flowgraph. Refresh the custom [Node Palette](#). The sub-compartment [Palette Container](#) is added to the [General](#) compartment of the custom [Node Palette](#). It contains the node template for the object from the [Project Explorer](#) view added to the COMPARTMENT node in the previous step.
9. Switch to the editing tab of the Template node palette flowgraph. Add a Filter node from the [Node Palette](#) to the COMPARTMENT node. Edit the Display Name and the Description in the [General](#) tab of the [Properties](#) view of the Filter node. In addition, edit the signatures of the input and the output of the Filter node and define a filter expression.
10. Switch to the editing tab of the Custom flowgraph. Refresh the custom [Node Palette](#). A new node template with the chosen display name and description (tool-tip) was added to the Palette Container sub-compartment.
11. Add node template of the new filter node from the custom [Node Palette](#) to the Custom flowgraph. The added Filter node has received the modified input and output signatures and the filter expression of the Filter node in the Template node palette flowgraph.
12. Switch to the editing tab of the Template node palette flowgraph. Move the COMPARTMENT node from the canvas of the GENERAL node to the canvas of the flowgraph container.
13. Switch to the editing tab of the Custom flowgraph. Refresh the custom [Node Palette](#). The previous Palette Container sub-compartment in the General compartment is now a new top level compartment of the [Node Palette](#).

Related Information

[Modeling a flowgraph \[page 429\]](#)

8 Defining Web-based Data Access in XS Classic

SAP HANA extended application services (SAP HANA XS) provide applications and application developers with access to the SAP HANA database using a consumption model that is exposed via HTTP.

In addition to providing application-specific consumption models, SAP HANA XS also host system services that are part of the SAP HANA database, for example: search services and a built-in Web server that provides access to static content stored in the SAP HANA repository.

The consumption model provided by SAP HANA XS focuses on server-side applications written in JavaScript and making use of a powerful set of specially developed API functions. However, you can use other methods to provide access to the data you want to expose in SAP HANA. For example, you can set up the Web-based data access for **XS classic** applications using the following services:

- OData (v2)
You can map the persistence and consumption models with the Open Data Protocol (OData), a resource-based Web protocol for querying and updating data.
- SAP HANA REST API
SAP HANA REST API supports the Orion protocol 1.0, which allows development tools to access the SAP HANA Repository (XS classic) in a convenient and standards-compliant way.

Related Information

[Defining OData v2 Services for XS Advanced JavaScript Applications](#)

[SAP Note 3537501](#) 

8.1 Data Access with OData in SAP HANA XS

In SAP HANA Extended Application Services (SAP HANA XS), the persistence model (for example, tables, views, and stored procedures) is mapped to the consumption model that is exposed to clients - the applications you write to extract data from the SAP HANA database.

You can map the persistence and consumption models with the Open Data Protocol (OData), a resource-based Web protocol for querying and updating data. An OData application running in SAP HANA XS is used to provide the consumption model for client applications exchanging OData queries with the SAP HANA database.

Note

SAP HANA XS currently supports OData version 2.0, which you can use to send OData queries (for example, using the HTTP GET method). Language encoding is restricted to UTF-8.

You can use OData to enable clients to consume authorized data stored in the SAP HANA database. OData defines operations on resources using RESTful HTTP commands (for example, **GET**, **PUT**, **POST**, and **DELETE**) and specifies the URI syntax for identifying the resources. Data is transferred over HTTP using either the Atom (XML) or the JSON (JavaScript) format.

Note

For modification operations, for example, **CREATE**, **UPDATE**, and **DELETE**, SAP HANA XS supports only the JSON format (`"content-type: application/json"`).

Applications running in SAP HANA XS enable accurate control of the flow of data between the presentational layer, for example, in the Browser, and the data-processing layer in SAP HANA itself, where the calculations are performed, for example, in SQL or SQLScript. If you develop and deploy an OData service running in SAP HANA XS, you can take advantage of the embedded access to SAP HANA that SAP HANA XS provides; the embedded access greatly improves end-to-end performance.

8.1.1 OData in SAP HANA XS

OData is a resource-based web protocol for querying and updating data. OData defines operations on resources using HTTP commands (for example, **GET**, **PUT**, **POST**, and **DELETE**) and specifies the uniform resource indicator (URI) syntax to use to identify the resources.

Data is transferred over HTTP using the Atom or JSON format:

Note

OData makes it easier for SAP, for partners, and for customers to build standards-based applications for many different devices and on various platforms, for example, applications that are based on a lightweight consumption of SAP and non-SAP business application data.

The main aim of OData is to define an abstract data model and a protocol which, combined, enable any client to access data exposed by any data source. Clients might include Web browsers, mobile devices, business-intelligence tools, and custom applications (for example, written in programming languages such as PHP or Java); data sources can include databases, content-management systems, the Cloud, or custom applications (for example, written in Java).

The OData approach to data exchange involves the following elements:

- **OData data model**
Provides a generic way to organize and describe data. OData uses the Entity 1 Data Model (EDM).
- **OData protocol**
Enables a client to query an OData service. The OData protocol is a set of interactions, which includes the usual REST-based create, read, update, and delete operations along with an OData-defined query language. The OData service sends data in either of the following ways:
 - XML-based format defined by Atom/AtomPub
 - JavaScript Object Notation (JSON)
- **OData client libraries**
Enables access to data via the OData protocol. Since most OData clients are applications, pre-built libraries for making OData requests and getting results reduces and simplifies work for the developers who create those applications.

A broad selection of OData client libraries are already widely available, for example: Android, Java, JavaScript, PHP, Ruby, and the best known mobile platforms.

- **OData services**

Exposes an end point that allows access to data in the SAP HANA database. The OData service implements the OData protocol (using the OData Data Services runtime) and uses the Data Access layer to map data between its underlying form (database tables, spreadsheet lists, and so on) and a format that the requesting client can understand.

8.1.2 Define the Data an OData Service Exposes

An OData service exposes data stored in database tables or views as OData collections for analysis and display by client applications. However, first of all, you need to ensure that the tables and views to expose as an OData collection actually exist.

Context

To define the data to expose using an OData service, you must perform at least the following tasks:

Procedure

1. Create a database schema.
2. Create a simple database table to expose with an OData service.
3. Create a simple database view to expose with an OData service.

This step is optional; you can expose tables directly. In addition, you can create a modeling view, for example, analytic, attribute, or calculation.

4. Grant select privileges to the tables and views to be exposed with the OData service.

After activation in the repository, schema and tables objects are 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.

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

8.1.3 OData Service Definitions

The OData service definition is the mechanism you use to define what data to expose with OData, how, and to whom. Data exposed as an OData collection is available for analysis and display by client applications, for example, a browser that uses functions provided by an OData client library running on the client system.

To expose information by means of OData to applications using SAP HANA XS, you must define database views that provide the data with the required granularity. Then you create an OData service definition, which is a file you use to specify which database views or tables are exposed as OData collections.

Note

SAP HANA XS supports OData version 2.0, which you can use to send OData queries (for example, using the HTTP GET method). Language encoding is restricted to UTF-8.

An OData service for SAP HANA XS is defined in a text file with the file suffix `.xsodata`, for example, `OdataSrvDef.xsodata`. The file must contain at least the entry `service {}`, which would generate a completely operational OData service with an empty service catalog and an empty metadata file. However, usually you use the service definition to expose objects in the database catalog, for example: tables, SQL views, or calculation rules.

In the OData service-definition file, you can use the following ways to name the SAP HANA objects you want to expose by OData:

Note

The syntax to use in the OData service-definition file to reference objects depends on the object type, for example, repository (design-time) or database catalog (runtime).

- **Repository objects**

Expose an object using the object's repository (design-time) name in the OData service-definition file. This method of exposing database objects using OData enables the OData service to be automatically updated if the underlying repository object changes. Note that a design-time name can be used to reference analytic and calculation views; it cannot be used to reference SQL views. The following example shows how to include a reference to a table in an OData service definition using the table's *design-time* name.

```
service {  
    "acme.com.odata::myTable" as "myTable";  
}
```

Note

Calculation views are only accessible from within `xsodata` files by referring to the design-time name. However, it is recommended to use design-time names whenever possible for calculation views or common tables. With design-time names, the cross references are recreated during activation (for example, for where-used), which means changes are visible automatically.

- **Database objects**

Expose an object using the object's database catalog (runtime) name. The support for database objects is mainly intended for existing or replicated objects that do not have a repository design-time representation. The following example shows how to include a reference to a table in an OData service definition using the table's *runtime* name.

```
service {
```

```
} "mySchema"."myTable" as "MyTable";
```

Note

It is strongly recommended not to use catalog (runtime) names in an OData service-definition. The use of catalog object names is only enabled in a service-definition because some objects do not have a design-time name. If at all possible, use the design-time name to reference objects in an OData service-definition file.

By default, all entity sets and associations in an OData service are writeable, that is they can be modified with a CREATE, UPDATE, or DELETE requests. However, you can prevent the execution of a modification request by setting the appropriate keyword (*create*, *update*, or *delete*) with the `forbidden` option in the OData service definition. The following example of an OData service definition for SAP HANA XS shows how to prevent any modification to the table `myTable` that is exposed by the OData service. Any attempt to make a modification to the indicated table using a CREATE, UPDATE, or DELETE request results in the HTTP response status **403 FORBIDDEN**.

```
service {  
    "sap.test::myTable"  
        create forbidden  
        update forbidden  
        delete forbidden;  
}
```

For CREATE requests, for example, to add a new entry to either a table or an SQL view exposed by an OData service, you must specify an explicit key (not a generated key); the key must be included in the URL as part of the CREATE request. For UPDATE and DELETE requests, you do not need to specify the key explicitly (and if you do, it will be ignored); the key is already known, since it is essential to specify which entry in the table or SQL view must be modified with the UPDATE or DELETE request.

Note

Without any support for IN/OUT table parameters in SQLScript, it is not possible to use a sequence to create an entry in a table or view exposed by an OData service. However, you can use XS JavaScript exits to update a table with a generated value.

Related Information

[Tutorial: Creating a Modification Exit with XS JavaScript \[page 507\]](#)

8.1.3.1 OData Service-Definition Type Mapping

During the activation of the OData service definition, SQL types defined in the service definition are mapped to EDM types according to a mapping table.

For example, the SQL type `"Time"` is mapped to the EDM type `"EDM.Time"`; the SQL type `"Decimal"` is mapped to the EDM type `"EDM.Decimal"`; the SQL types `"Real"` and `"Float"` are mapped to the EDM type `"EDM.Single"`.

Note

The OData implementation in SAP HANA Extended Application Services (SAP HANA XS) does not support all SQL types.

In the following example, the SQL types of columns in a table are mapped to the EDM types in the properties of an entity type.

```
{name = "ID"; sqlType = INTEGER; nullable = false;}, {name = "RefereeID";  
sqlType = NVARCHAR; length = 10; nullable = true;}
```

```
<Property Name="ID" Type="Edm.Int32" Nullable="false"/> <Property  
Name="RefereeID" Type="Edm.String" MaxLength="10" Nullable="true"/>
```

Related Information

[OData Service Definition: SQL-EDM Type Mapping \(XS Advanced\) \[page 511\]](#)

[OData Service Definitions \[page 477\]](#)

8.1.3.2 OData Service-Definition Features

The OData service definition provides a list of keywords that you use in the OData service-definition file to enable important features. For example, the following list illustrates the most-commonly used features used in an OData service-definition and, where appropriate, indicates the keyword to use to enable the feature:

- **Aggregation**
The results of aggregations on columns change dynamically, depending on the grouping conditions. As a result, aggregation cannot be done in SQL views; it needs to be specified in the OData service definition itself. Depending on the type of object you want to expose with OData, the columns to aggregate and the function used must be specified explicitly (**explicit aggregation**) or derived from metadata in the database (**derived aggregation**). Note that aggregated columns cannot be used in combination with the `$filter` query parameter, and aggregation is only possible with generated keys.
- **Association**
Define associations between entities to express relationships between entities. With associations it is possible to reflect foreign key constraints on database tables, hierarchies and other relations between database objects.
- **Key Specification**
The OData specification requires an Entity Type to denote a set of properties forming a unique key. In SAP HANA, only tables can have a unique key, the primary key. All other (mostly view) objects require you to specify a key for the entity. The OData service definition language (OSDL) enables you to do this by denoting a set of existing columns or by generating a **local key**. Bear in mind that local keys are transient; they exist only for the duration of the current session and cannot be dereferenced.

Note

OSDL is the language used to define a service definition; the language includes a list of keywords that you use in the OData service-definition file to enable the required features.

- **Parameter Entity Sets**

You can use a special parameter entity set to enter input parameters for SAP HANA calculation views and analytic views. During activation of the entity set, the specified parameters are retrieved from the metadata of the calculation (or analytical) view and exposed as a new **EntitySet** with the name suffix "Parameters", for example "CalcViewParameters".

- **Projection**

If the object you want to expose with an OData service has more columns than you actually want to expose, you can use SQL views to restrict the number of selected columns in the **SELECT**. However, for those cases where SQL views are not appropriate, you can use the **with** or **without** keywords in the OData service definition to **include** or **exclude** a list of columns.

Related Information

[OData Service-Definition Examples \[page 486\]](#)

8.1.4 Create an OData Service Definition

The OData service definition is a configuration file you use to specify which data (for example, views or tables) is exposed as an OData collection for analysis and display by client applications.

Prerequisites

The following prerequisites apply when you create an OData service definition:

- SAP HANA studio (and client) is installed and configured
- An SAP HANA database user is available with repository privileges (for example, to add packages)
- An SAP HANA development system is added to (and available in) SAP HANA studio, for example, in either the [Systems](#) view or the [Repositories](#) view
- A working development environment is available including: a repository workspace, a package structure for your OData application, and a shared project to enable you to synchronize changes to the OData project files in the local file system with the repository
- You have defined the data to expose with the OData application, for example, at least the following:
 - A database schema
 - A database table

Context

An OData service for SAP HANA XS is defined in a text file with the file suffix `.xsodata`, for example, `OdataSrvDef.xsodata`. The file resides in the package hierarchy of the OData application and must contain at least the entry service `{}`, which would generate an operational OData service with an empty service catalog and an empty metadata file.

→ 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. In the shared project you are using for your OData application, use the [Project Explorer](#) view to locate the package where you want to create the new OData service definition.

📘 Note

The file containing the OData service definition must be placed in the root package of the OData application for which the service is intended.

2. Create the file that will contain your OData service definition.

In the [Project Explorer](#) view, right-click the folder where you want to create the new OData service definition file and choose **New > Other > SAP HANA > Application Development > XS OData Service** in the context-sensitive popup menu.

3. Enter or select the parent folder, where the new OData service definition is to be located.
4. Enter a name for the new OData service definition.
5. Select a template to use. Templates contain sample source code to help you.
6. Choose [Finish](#).

📘 Note

If you are using the SAP HANA Studio to create artifacts in the SAP HANA Repository, the file creation wizard adds the required file extension `.xsodata` automatically and opens the new file in the appropriate editor.

7. Define the OData service.

The OData service definition uses the OData Service Definition Language (OSDL), which includes a list of keywords that you specify in the OData service-definition file to enable important features.

The following example shows a simple OData service definition exposing a simple table:

```
service namespace "my.namespace" {
    "sample.odata::table" as "MyTable";
}
```

This service definition exposes a table defined in the file `sample.odata:table.hdbtable` and creates an EntitySet for this entity named `MyTable`. The specification of an alias is optional. If omitted, the default name of the EntitySet is the name of the repository object file, in this example, `table`.

8. Save and activate the OData service definition in the SAP HANA repository.

→ Tip

To run an OData service, right-click the OData service file in the *Project Explorer* view and choose  *Run As* > *XS Service* > in the context-sensitive menu.

Related Information

[OData Service Definitions \[page 477\]](#)

8.1.5 Tutorial: Use the SAP HANA OData Interface

The package you put together to test the SAP HANA OData interface includes all the artifacts you need to use SAP HANA Extended Application Services (SAP HANA XS) to expose an OData collection for analysis and display by client applications.

Prerequisites

Since the artifacts required to get a simple OData application up and running are stored in the repository, it is assumed that you have already performed the following tasks:

- Create a development workspace in the SAP HANA repository
- Create a project in the workspace
- Share the new project

Context

To create a simple OData application, perform the following steps:

→ Tip

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

Procedure

1. Create a root package for your OData application, for example, `helloodata` and save and activate it in the repository.
 - a. Click the [Content](#) directory with the alternate mouse button and choose [New](#) [Package](#).
 - b. Enter the required information for the package in the dialog box and choose [OK](#).

Note

The namespace `sap` is restricted. Place the new package in your own namespace, which you can create alongside the `sap` namespace.

2. Create a schema, for example, `HELLO_ODATA.hdbschema`.

The schema is required for the table that contains the data to be exposed by your OData service-definition. The schema is defined in a flat file with the file extension `.hdbschema` that you save in the repository and which you must activate.

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 file in the appropriate editor.

Enter the following code in the `HELLO_ODATA.hdbschema` file:

```
schema_name="HELLO_ODATA";
```

3. Create the database table that contains the data to be exposed by your OData service definition, for example, `otable.hdbtable`.

In the [Project Explorer](#) view, right-click the folder where you want to create the new OData service definition file and choose [New](#) [Other](#) [SAP HANA](#) [Database Development](#) [Database Table](#) in the context-sensitive popup menu.

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 file in the appropriate editor.

Enter the following code in the `otable.hdbtable` file:

Note

If the editor underlines the keywords `nullable` and `Defaultvalue` in red, you can safely ignore this.

```
table.schemaName = "HELLO_ODATA";
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 = 12;
scale = 3;}};
table.primaryKey.pkcolumns = ["Col1", "Col2"];
```

4. 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>');
```

5. Create an application descriptor for your new OData application in your root OData package `helloodata`. The application descriptor (`.xsapp`) is the core file that you use to define an application's availability within SAP HANA application. The `.xsapp` file sets the point in the application-package structure from which content will be served to the requesting clients.

Note

The application-descriptor file has no content and no name; it only has the extension `.xsapp`. 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 file in the appropriate editor.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new application descriptor and choose **New > Other > SAP HANA > Application Development > XS Application Descriptor File** in the context-sensitive popup menu.
- b. Save and activate the application-descriptor file in the repository.

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.

6. Create an application-access file for your new OData application and place it in your root OData package `helloodata`.

The application-access file enables you to specify who or what is authorized to access the content exposed by the application.

Note

The application-access file has no name; it only has the extension `.xsaccess`. 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 file in the appropriate editor.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new application descriptor and choose ► *New* ► *Other* ► *SAP HANA* ► *Application Development* ► *XS Application Access File* in the context-sensitive popup menu.
- b. Enter the following content in the .xsaccess file for your new OData application:

```
{
  "exposed" : true,
  "prevent_xsrf" : true
}
```

Note

It is highly recommended to always use the `prevent_xsrf` keyword to help protect your application against attacks that use cross-site request forgery.

- c. Save and activate the application-access file in the repository.
7. Create an OData service-definition file and place it in your root OData package `helloodata`. The Odata service-definition file has the file extension `.xsodata`, for example, `hello.xsodata` and for the purposes of this tutorial should be located in the root package of the OData application:

→ 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 file in the appropriate editor.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new application descriptor and choose ► *New* ► *Other* ► *SAP HANA* ► *Application Development* ► *XS OData File* in the context-sensitive popup menu.
- b. Enter the following content in the `hello.xsodata` OData service-definition file:

```
service {
  "helloodata::otable";
}
```

- c. Save and activate the OData service-definition file in the repository.
8. Test the new OData service.

Open a browser and enter the following URL.

Note

If you are using Internet Explorer, press `F12` and set *compatibility mode = IE10* and *document mode = Standards*.

`http://<hana.server.name>:80<HANA_instance_number>/helloodata/hello.xsodata/otable`

→ Tip

You can also run the service directly from the *Project Explorer* view where you activated it; right-click the object in the Project Explorer view and chose *Run As...* in the context-sensitive popup menu.

8.1.6 OData Service-Definition Examples

The OData service definition describes how data exposed in an end point can be accessed by clients using the OData protocol.

Each of the examples listed below is explained in a separate section. The examples show how to use the OData Service Definition Language (OSDL) in the OData service-definition file to generate an operational OData service that enables clients to use SAP HANA XS to access the OData end point you set up.

- Empty Service
- Namespace Definition
- Object Exposure
- Property Projection
- Key Specification
- Associations
- Aggregation
- Parameter Entity Sets
- ETag Support
- Nullable Properties

8.1.6.1 OData Empty Service

An OData service for SAP HANA XS is defined by a text file containing at least the following line:

Service definition `sample.odata:empty.xsodata`

```
service {}
```

A service file with the minimal content generates an empty, completely operational OData service with an empty service catalog and an empty metadata file:

Note

- Examples and graphics are provided for illustration purposes only; some URLs may differ from the ones shown.

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata`

```
{
  "d" : {
    "EntitySets" : []
  }
}
```

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```

<edmx:Edmx Version="1.0"
xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices m:DataServiceVersion="2.0"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <Schema Namespace="sample.odata.empty"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://schemas.microsoft.com/ado/2007/05/edm">
      <EntityContainer Name="empty" m:IsDefaultEntityContainer="true"/>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>

```

An empty service metadata document consists of one Schema containing an empty EntityContainer. The name of the EntityContainer is the name of the .xsodata file, in this example "empty".

8.1.6.2 OData Namespace Definition

Every .xsodata file must define its own namespace by using the namespace keyword:

Service definition sample.odata:namespace.xsodata

```
service namespace "my.namespace" {}
```

The resulting service metadata document has the specified schema namespace:

Note

Examples and graphics are provided for illustration purposes only; some URLs may differ from the ones shown.

http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/\$metadata

```

<edmx:Edmx Version="1.0"
xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices m:DataServiceVersion="2.0"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <Schema Namespace="my.namespace"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://schemas.microsoft.com/ado/2007/05/edm">
      <EntityContainer Name="namespace" m:IsDefaultEntityContainer="true"/>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>

```


8.1.6.3 OData Object Exposure

In the examples provided to illustrate object exposure, the following definition of a table applies:

Table definition `sample.odata:table.hdbtable`

```
COLUMN TABLE "sample.odata::table" (  
    "ID" INTEGER,  
    "Text" NVARCHAR(1000),  
    "Time" TIMESTAMP,  
    PRIMARY KEY ("ID")  
);
```

Database Objects

Similar to the exposure of an object by using the repository design-time name is the exposure by the database name:

Service definition `sample.odata:db.xsodata`

```
service {  
    "sample.odata::table" as "MyTable";  
}
```

The service exposes the same table by using the database catalog name of the object and the name of the schema where the table is created in.

8.1.6.4 OData Property Projection

If the object you want to expose with an OData service has more columns than you actually want to expose, you can use SQL views to restrict the number of selected columns in the `SELECT`.

Nevertheless, SQL views are sometimes not appropriate, for example with calculation views, and for these cases we provide the possibility to restrict the properties in the OData service definition in two ways. By providing an including or an excluding list of columns.

Including Properties

You can specify the columns of an object that have to be exposed in the OData service by using the `with` keyword. Key fields of tables must not be omitted.

Service definition `sample.odata:with.xsodata`

```
service {  
    "sample.odata::table" as "MyTable" with ("ID", "Text");  
}
```

The resulting `EntityType` then contains only the properties derived from the specified columns:

Note

Examples and graphics are provided for illustration purposes only; some URLs may differ from the ones shown.

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```
<EntityType Name="MyTableType">
  <Key>
    <PropertyRef Name="ID"/>
  </Key>
  <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
  <Property Name="Text" Type="Edm.String" Nullable="true" MaxLength="1000"/>
</EntityType>
```

Excluding Properties

The opposite of the `with` keyword is the `without` keyword, which enables you to specify which columns you do NOT want to expose in the OData service:

Service definition sample.odata:without.xsodata

```
service {
  "sample.odata::table" as "MyTable" without ("Text","Time");
}
```

The generated `EntityType` then does NOT contain the properties derived from the specified columns:

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```
<EntityType Name="MyTableType">
  <Key>
    <PropertyRef Name="ID"/>
  </Key>
  <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
</EntityType>
```

8.1.6.5 OData Key Specification

The OData specification requires an `EntityType` to denote a set properties forming a unique key. In HANA only tables may have a unique key, the primary key. For all other (mostly view) objects you need to specify a key for the entity.

In OSDL, you can specify a key for an entity/object by denoting a set of existing columns or by generating a key.

Note

Key attributes are not evaluated.

For the examples illustrating key specification, we use the following SQL view, which selects all data from the specified table.

View definition `sample.odata:view.hdbview`

```
{  
  VIEW "sample.odata::view" as select * from "sample.odata::table"  
}
```

Existing Key Properties

If the object has set of columns that may form a unique key, you can specify them as key for the entity. These key properties are always selected from the database, no matter if they are omitted in the `$select` query option. Therefore explicit keys are not suitable for calculation views and analytic views as the selection has an impact on the result.

Service definition `sample.odata:explicitkeys.xsodata/$metadata`

```
service {  
  "sample.odata::view" as "MyView" key ("ID", "Text");  
}
```

The metadata document for the exposure of the view above is almost equal to the metadata document for repository objects. Only the key is different and consists now of two columns:

Note

Examples and graphics are provided for illustration purposes only; some URLs may differ from the ones shown.

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```

<edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices m:DataServiceVersion="2.0"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <Schema Namespace="sample.odata.explicitkeys"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
      xmlns="http://schemas.microsoft.com/ado/2007/05/edm">
      <EntityType Name="MyViewType">
        <Key>
          <PropertyRef Name="ID"/>
          <PropertyRef Name="Text"/>
        </Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Text" Type="Edm.String" Nullable="true" MaxLength="1000"/>
        <Property Name="Time" Type="Edm.DateTime" Nullable="true"/>
      </EntityType>
      <EntityContainer Name="explicitkeys" m:IsDefaultEntityContainer="true">
        <EntitySet Name="MyView"
          EntityType="sample.odata.explicitkeys.MyViewType"/>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>

```

⚠ Caution

The OData infrastructure cannot check whether your specified keys are unique, so be careful when choosing keys.

Generated Local Key

For objects that do not have a unique key in their results, for example, calculation views or aggregated tables, you can generate a locally valid key. This key value numbers the results starting with 1 and is not meant for dereferencing the entity; you cannot use this key to retrieve the entity. The key is valid only for the duration of the current session and is used only to satisfy OData's need for a unique ID in the results. The property type of a generated local key is `Edm.String` and cannot be changed.

Service definition `sample.odata:generatedkeys.xsodata`

```

service {
  "sample.odata::view" as "MyView" key generate local "GenID";
}

```

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```

<EntityType Name="MyViewType">
  <Key>
    <PropertyRef Name="GenID"/>
  </Key>
  <Property Name="GenID" Type="Edm.String" Nullable="false" MaxLength="2147483647"/>
  <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
  <Property Name="Text" Type="Edm.String" Nullable="true" MaxLength="1000"/>
  <Property Name="Time" Type="Edm.DateTime" Nullable="true"/>
</EntityType>
<EntityContainer Name="generatedkeys" m:IsDefaultEntityContainer="true">
  <EntitySet Name="MyView" EntityType="sample.odata.generatedkeys.MyViewType"/>
</EntityContainer>

```

As a consequence of the transient nature of generated local keys, it is not possible to define navigation properties on these entities or use them in filter or order by conditions.

8.1.6.6 OData Associations

You can define associations between entities to express relationships between entities. With associations it is possible to reflect foreign key constraints on database tables, hierarchies and other relations between database objects. OSDL supports simple associations, where the information about the relationship is stored in one of the participating entities, and complex associations, where the relationship information is stored in a separate association table.

Associations themselves are freestanding. On top of them you can specify which of the entities participating in the relationship can navigate over the association to the other entity by creating `NavigationProperty` objects.

For the examples used to illustrate OData associations, we use the tables `customer` and `order`:

Table definition: `sample.odata:customer.hdbtable`

```

COLUMN TABLE "sample.odata::customer" (
  "ID" INTEGER NOT NULL,
  "OrderID" INTEGER,
  PRIMARY KEY ("ID")
);

```

Table definition: `sample.odata:order.hdbtable`

```

COLUMN TABLE "sample.odata::order" (
  "ID" INTEGER NOT NULL,
  "CustomerID" INTEGER,
  PRIMARY KEY ("ID")
);

```

There is one relationship `order.CustomerID` to `customer.ID`.

Simple Associations

The definition of an association requires you to specify a name, which references two exposed entities and whose columns keep the relationship information. To distinguish the ends of the association, you must use the keywords `principal` and `dependent`. In addition, it is necessary to denote the multiplicity for each end of the association.

Service definition: `sample.odata:assocsimple.xsodata`

```
service {
    "sample.odata::customer" as "Customers";
    "sample.odata::order" as "Orders";
    association "Customer_Orders" with referential constraint principal
    "Customers"("ID") multiplicity "1" dependent "Orders"("CustomerID") multiplicity
    "*";
}
```

The association in the example above with the name `Customer_Orders` defines a relationship between the table `customer`, identified by its EntitySet name `Customers`, on the `principal` end, and the table `order`, identified by its entity set name `Orders`, on the `dependent` end. Involved columns of both tables are denoted in braces (`{}`) after the name of the corresponding entity set. The `multiplicity` keyword on each end of the association specifies their cardinality - in this example, one-to-many.

The `with referential constraint` syntax ensures that the referential constraint check is enforced at design time, for example, when you activate the service definition in the SAP HANA repository. The referential constraint information appears in the metadata document.

Note

SAP strongly recommends that you use the `with referential constraint` syntax.

The number of columns involved in the relationship must be equal for both ends of the association, and their order in the list is important. The order specifies which column in one table is compared to which column in the other table. In this simple example, the column `customer.ID` is compared to `order.CustomerID` in the generated table join.

As a result of the generation of the service definition above, an `AssociationSet` named `Customer_Orders` and an `Association` with name `Customer_OrdersType` are generated:

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```

<EntityType Name="CustomersType">
  <Key>
    <PropertyRef Name="ID"/>
  </Key>
  <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
  <Property Name="RecruitID" Type="Edm.String" Nullable="true" MaxLength="1"/>
</EntityType>
<EntityType Name="OrdersType">
  <Key>
    <PropertyRef Name="ID"/>
  </Key>
  <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
  <Property Name="CustomerID" Type="Edm.Int32" Nullable="false"/>
</EntityType>
<Association Name="Customer_OrdersType">
  <End Type="sample.odata.assocsimple.CustomersType" Role="CustomersPrincipal" Multiplicity="1"/>
  <End Type="sample.odata.assocsimple.OrdersType" Role="OrdersDependent" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="CustomersPrincipal">
      <PropertyRef Name="ID"/>
    </Principal>
    <Dependent Role="OrdersDependent">
      <PropertyRef Name="CustomerID"/>
    </Dependent>
  </ReferentialConstraint>
</Association>
<EntityContainer Name="assocsimple" m:IsDefaultEntityContainer="true">
  <EntitySet Name="Customers" EntityType="sample.odata.assocsimple.CustomersType"/>
  <EntitySet Name="Orders" EntityType="sample.odata.assocsimple.OrdersType"/>
  <AssociationSet Name="Customer_Orders"
    Association="sample.odata.assocsimple.Customer_OrdersType">
    <End Role="CustomersPrincipal" EntitySet="Customers"/>
    <End Role="OrdersDependent" EntitySet="Orders"/>
  </AssociationSet>
</EntityContainer>

```

The second association is similar to the first one and is shown in the following listing:

```

association "Customer_Recruit" with referential constraint principal
"Customers"("ID") multiplicity "1" dependent "Customers"("RecruitID")
multiplicity "*";

```

Complex Associations

For the following example of a complex association, an additional table named `knows` is introduced that contains a relationship between customers.

Table definition: `sample.odata:knows.hdbtable`

```

COLUMN TABLE "sample.odata::knows" (
  "KnowingCustomerID" INTEGER NOT NULL,
  "KnowCustomerID" INTEGER NOT NULL,
  PRIMARY KEY ("KnowingCustomerID", "KnowCustomerID")
);

```

Relationships that are stored in association tables such as knows can be similarly defined as simple associations. Use the keyword `over` to specify the additional table and any required columns.

Service definition: `sample.odata:assoccomplex.xsodata`

```
service {
  "sample.odata::customer" as "Customers";
  "sample.odata::order" as "Orders";
  association "Customer_Orders"
    principal "Customers"("ID") multiplicity "*"
    dependent "Customers"("ID") multiplicity "*"
    over "sample.odata::knows" principal ("KnowingCustomerID") dependent
    ("KnownCustomerID");
}
```

With the keywords `principal` and `dependent` after `over` you can specify which columns from the association table are joined with the `principal` respectively `dependent` columns of the related entities. The number of columns must be equal in pairs, and their order in the list is important.

The generated Association in the metadata document is similar to the one created for a simple association except that the `ReferentialConstraint` is missing:

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```
<Association Name="Customer_OrdersType">
  <End Type="sample.odata.assoccomplex.CustomersType" Role="CustomersPrincipal" Multiplicity="*" />
  <End Type="sample.odata.assoccomplex.CustomersType" Role="CustomersDependent" Multiplicity="*" />
</Association>
<EntityContainer Name="assoccomplex" m:IsDefaultEntityContainer="true">
  <EntitySet Name="Customers" EntityType="sample.odata.assoccomplex.CustomersType" />
  <EntitySet Name="Orders" EntityType="sample.odata.assoccomplex.OrdersType" />
  <AssociationSet Name="Customer_Orders"
    Association="sample.odata.assoccomplex.Customer_OrdersType">
    <End Role="CustomersPrincipal" EntitySet="Customers" />
    <End Role="CustomersDependent" EntitySet="Customers" />
  </AssociationSet>
</EntityContainer>
```

Navigation Properties

By only defining an association, it is not possible to navigate from one entity to another. Associations need to be bound to entities by a `NavigationProperty`. You can create them by using the keyword `navigates`:

Service definition: `sample.odata:assocnav.xsodata`

```
service {
  "sample.odata::customer" as "Customers" navigates ("Customer_Orders" as
  "HisOrders");
  "sample.odata::order" as "Orders";
  association "Customer_Orders" principal "Customers"("ID") multiplicity "1"
  dependent "Orders"("CustomerID") multiplicity "*";
}
```

The example above says that it is possible to navigate from `Customers` over the association `Customer_Order` via the `NavigationProperty` named `"HisOrders"`.

The right association end is determined automatically by the entity set name. But if both ends are bound to the same entity, it is necessary to specify the starting end for the navigation. This is done by specifying either `from principal` or `from dependent` which refer to the principal and dependent ends in the association.

Service definition: `sample.odata:assocnavself.xsodata`

```
service {
    "sample.odata::customer" as "Customers"
        navigates ("Customer_Orders" as "HisOrders", "Customer_Recruit" as
"Recruit" from principal);
    "sample.odata::order" as "Orders";
    association "Customer_Orders" principal "Customers"("ID") multiplicity "1"
dependent "Orders"("CustomerID") multiplicity "*";
    association "Customer_Recruit" principal "Customers"("ID") multiplicity
"1" dependent "Customers"("RecruitID") multiplicity "*";
}
```

In both cases a `NavigationProperty` is added to the `EntityType`.

`http://<myHANAServer>:<port>/odata/services/<myService>.xsodata/$metadata`

```
<EntityType Name="CustomersType">
  <Key>
    <PropertyRef Name="ID"/>
  </Key>
  <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
  <Property Name="RecruitID" Type="Edm.String" Nullable="true" MaxLength="1"/>
  <NavigationProperty Name="HisOrders"
Relationship="sample.odata.assocnavself.Customer_OrdersType" FromRole="CustomersPrincipal"
ToRole="OrdersDependent"/>
  <NavigationProperty Name="Recruit"
Relationship="sample.odata.assocnavself.Customer_RecruitType" FromRole="CustomersPrincipal"
ToRole="CustomersDependent"/>
</EntityType>
```

8.1.6.7 OData Aggregation

The results of aggregations on columns change dynamically depending on the grouping conditions. This means that aggregation cannot be performed in SQL views; it needs to be specified in the OData service definition itself. Depending on the type of object to expose, you need to explicitly specify the columns to aggregate and the function to use or derived them from metadata in the database.

In general, aggregations do not have consequences for the metadata document. It just effects the semantics of the concerning properties during runtime. The grouping condition for the aggregation contain all selected non-aggregated properties. Furthermore, aggregated columns cannot be used in `$filter`, and aggregation is only possible with generated keys.

Derived Aggregation

The simplest way to define aggregations of columns in an object is to derive this information from metadata in the database. The only objects with this information are calculation views and analytic views. For all other

object types, for example, tables and SQL views, the activation will not work. To cause the service to use derived information, you must specify the keywords *aggregates always*, as illustrated in the following example:

```
service {
  "sample.odata:calc" as "CalcView"
    keys generate local "ID"
    aggregates always;
}
```

Explicit Aggregation

The example for the explicit aggregation is based on the following table definition:

`sample.odata:revenues.hdbtable`

```
COLUMN TABLE "sample.odata:revenues" (
  "Month" INTEGER NOT NULL,
  "Year" INTEGER NOT NULL,
  "Amount" INTEGER,
  PRIMARY KEY ("Month", "Year")
);
```

You can aggregate the columns of objects (without metadata) that are necessary for the derivation of aggregation by explicitly denoting the column names and the functions to use, as illustrated in the following example of a service definition: `sample.odata:aggrexpl.xsodata`

```
service {
  "sample.odata:revenues" as "Revenues"
    keys generate local "ID"
    aggregates always (SUM of "Amount");
}
```

The results of the entity set `Revenues` always contain the aggregated value of the column `Amount`. To extract the aggregated revenue amount per year, add `$select=Year, Amount` to your requested URI.

8.1.6.8 OData Parameter Entity Sets

SAP HANA calculation views can interpret input parameters. For OData, these parameters can be entered by using a special parameter *entity set*.

Parameter entity sets can be generated for calculation views by adding *parameters via entity* to the entity, as illustrated in the following service-definition example:

```
service {
  "sample.odata:calc" as "CalcView"
    keys generate local "ID"
    parameters via entity;
}
```

During loading of the service, parameters specified in `sample.odata/calc.calculationview` are retrieved from the metadata of the calculation view and exposed as a new `EntitySet` named after the entity set name and the suffix `Parameters`, for example, `CalcViewParameters`. A `NavigationProperty` named `Results` is generated to retrieve the results from the parameterized call.

The name of the generated parameter entity set and the navigation property can be customized, as illustrated in the following service-definition example:

```
service {
  "sample.odata::calc" as "CalcView"
    keys generate local "ID"
    parameters via entity "CVPParams" results property "Execute";
}
```

With the definition above, the name of the parameter entity set is `CVPParams`, and the name of the `NavigationProperty` for the results is `Execute`.

Navigating to Entities via Parameters

In an OData service definition, you can enable navigation between an entity and a parameterized entity. This feature is particularly useful if you need to have access to individual entries in a parameterized entity set, for example, a calculation view with parameters. If you need to access individual entries in an entity set that has parameters, you must expose the parameters as keys. If you do not need to have access to individual entries in an entity set, you can use the `key generate local` option to generate a pseudo key.

To enable navigation between an entity and a parameterized entity, you must perform the following steps:

1. Specify the parameters as part of the key of the target entity
2. Define the association between the entities

Enabling navigation between an entity and a parameterized entity is only possible if the parameters are part of the entity-type key in the OData service definition file. To make the parameters part of the key of the target entity, use the `via key` syntax, as illustrated in the following example:

```
service {
  "sap.test::calcview" key ("theKeyColumns") parameters via key and entity;
}
```

You also have to define an **association** between the source and target entities, for example, with additional entries introduced by the `via parameters` keyword, as illustrated in the following example:

```
service {
  "sap.test::table" as "Tab" navigates ("avp" as "ViewNav");
  "sap.test::calcview" as "View" key ("theKeyColumns") parameters via key and entity;

  association via parameters "avp"
    principal "Tab"("paramValue") multiplicity "*";
    dependent "View"("parameter") multiplicity "*";
}
```

Note

The order of the property list of the dependent end is crucial.

The parameters you define in the dependent end of the association **must** be the first properties in the list. In addition, the parameters specified **must** be given in the same order as they are specified in the view, as illustrated in the following example:

```
association via parameters "avp"
```

```
principal "Tab"("col1", "col2", "col3") multiplicity "*"
dependent "View"("parameter1", "parameter2", "colA") multiplicity "*";
```

In the example immediately above, the principal “Tab” has three columns that contain the information that is required to navigate to the dependent “View” in the association.

- “col1”
The value of “col1” should be set for “parameter1”
- “col2”
The value of “col2” should be set for “parameter2”
- “col3”
The parameter “col3” contains additional information that is not passed as an input parameter, but as part of a WHERE condition.

The generated SQL statement would look like the following:

```
select ... from "sap.test::calcview"(placeholder."$$parameter1$$"=>?,
placeholder."$$parameter2$$"=>?)
where "colA"=?
```

ⓘ Note

This navigation property cannot be used in combination with the OData query options \$expand, \$filter and \$orderby.

8.1.6.9 OData ETag Support

This feature allows a service to define the fields that are to be included in the concurrency check.

You can now use entity tags (ETags) for optimistic concurrency control. If you choose to use this feature, then you must enable it per entity in the .xsodata file. Enabling this feature per entity allows for the concurrency control to be applied to multiple fields. The following code example provides information about how to do this.

↗ Sample Code

```
service
{
  entity "sap.test.odata.db.views::Etag" as "EtagAll"
  key ("KEY_00") concurrencytoken;
  entity "sap.test.odata.db.views::Etag" as "EtagNvarchar"
  key ("KEY_00") concurrencytoken ("NVARCHAR_01", "INTEGER_03");
}
```

If you specify concurrencytoken only, then all properties, except the key properties, are used to calculate the ETag value. If you provide specific properties, then only those properties are used for the calculation.

ⓘ Note

You **cannot** specify concurrencytoken on aggregated properties that use the AVG (average) aggregation method.

8.1.6.10 OData Nullable Properties

You can create a service to enable nullable properties in OData.

During the 'Create' phase, the XSODATA layer generates all entity properties automatically. Since the properties are not nullable, consumers of the code are forced to pass dummy values into them. However, OData supports `$filter` and `$orderby` conditions on the `null` value. This means that it is now possible to treat `null` as a value, if you enable it. You can enable this behavior for the entire service only, not per entity.

The following code example provides information about how you can do this.

Sample Code

```
service {
  ...
}
settings {
  support null;
  content cache-control "no-store";
  metadata cache-control "max-age=86401,must-revalidate";
  hints
    "NO_CALC_VIEW_UNFOLDING";
  limits
    max_records = 10,
    max_expanded_records = 30;
}
```

If you enable support for `null`, then `$filter` requests, such as `$filter=NVARCHAR_01 eq null`, are possible. Otherwise `null` is rejected with an exception. If you do not enable the `null` support, then the default behavior applies.

Note

`null` values are "ignored" in comparisons: "ignored" in the sense that if you compare a column with `null` and the columns contain per definition no `null` values, no record passes the filter.

Related Information

[OData Configurable Cache Settings \[page 500\]](#)

[OData Hints for SQL Select Statements](#)

[OData Entity Limits](#)

8.1.6.11 OData Configurable Cache Settings

You can create a service to configure the cache settings for the `$metadata` request to optimize performance.

When calling OData services, the services make repeated requests for the `$metadata` document. Since changes to the underlying entity definitions occurs rarely, SAP has enabled the option to configure caching for

these `$metadata` documents. By configuring the cache, you can avoid many redundant queries to process the metadata.

The following code example provides information about how you can do this.

Sample Code

```
service {  
  ...  
}  
settings {  
  support null;  
  content cache-control "no-store";  
  metadata cache-control "max-age=86401,must-revalidate";  
  hints  
    "NO_CALC_VIEW_UNFOLDING";  
  limits  
    max_records = 10,  
    max_expanded_records = 30;  
}
```

content cache-control

You can use the `content cache-control` parameter to set the HTTP header "value" that is used for cache control in the data responses, for example:

```
content cache-control "no-store";
```

The value you specify must be enclosed in double quotes (for example, "`<value>`"), and multiple parameters must be separated by a comma.

→ Tip

You can include any value supported by the HTTP specification for cache-control. "no-store" indicates that the cache should not store any details of the client request or server response.

metadata cache-control

You can use the `metadata cache-control` parameter to set the header HTTP "value" that is used for the cache control in the metadata response, for example:

```
metadata cache-control "max-age=86401,must-revalidate";
```

The value you specify for `metadata cache-control` must be enclosed in double quotes (for example, "`<value>`"), and multiple elements must be separated by a comma, as illustrated in the example above.

→ Tip

You can include any value supported by the HTTP specification for cache-control. In the example, above, "must-revalidate" indicates that the cache must verify the status of resources (fresh or stale) and not

use any stale resource whose validity has expired; "max-age" specifies the amount of time a resource is considered fresh (and still usable).

Related Information

[OData Nullable Properties \[page 500\]](#)

[OData Hints for SQL Select Statements](#)

[OData Entity Limits](#)

8.1.6.12 Custom Exits for OData Write Requests

SAP HANA XS enables you to execute custom code at defined points of an OData write request.

If you provide a custom exit for an OData write request, the code has to be provided in form of an SQLScript procedure with signatures that follow specific conventions. The following type of write exits are supported for OData write requests in SAP HANA XS:

- **Validation Exits**
These exits are for validation of input data and data consistency checks. They can be registered for create, update, and delete events and executed before or after the change operation, or before or after the commit operation. You can specify a maximum of four validation exits per change operation; the exit is registered for the corresponding event with the respective keyword: "before", "after", "precommit" or "postcommit".
- **Modification Exits**
You can define custom logic to create, update, or delete an entry in an entity set. If a modification exit is specified, it is executed instead of the generic actions provided by the OData infrastructure. You use the [using](#) keyword to register the exit.

If registered, the scripts for the exits are executed in the order shown in the following table:

Execution Order of Exit Validation/Modification Scripts

OData Insert Type	Script Execution Order
Single Insert	before, using, after, precommit, postcommit
Batch Insert	before(1), using(1), after(1), before(2), using(2), after(2), ... , precommit(1), precommit(2), postcommit(1), postcommit(2)

The signature of a registered script has to follow specific rules, depending on whether it is registered for [entity](#) or [link](#) write operations and depending on the operation itself. The signature must also have table-typed parameters for both input and output:

- Entity Write Operations
- Link Write Operations

For **entity** write operations, the methods registered for the CREATE operation are passed a table containing the new entry that must be inserted into the target table; the UPDATE operation receives the entity both before and after the modification; the DELETE operation receives the entry that must be deleted. The table type of the parameters (specified with the [EntityType](#) keyword in the table below) corresponds to the types of the exposed entity.

Entity Write Operations

Script Type	Create	Update	Delete
before, after, precommit, using	IN new EntityType, OUT error ErrorType	IN new EntityType, IN old EntityType, OUT error ErrorType	IN old EntityType, OUT error ErrorType
postcommit	IN new EntityType	IN new EntityType, IN old EntityType	IN old EntityType

For **link** write operations, all exits that are executed before the commit operation take two table-typed input parameters and one table-typed output parameter. The first parameter must correspond to the structure of the entity type at the **principal** end of the association; the second parameter must correspond to the **dependent** entity type.

Link Write Operations

Script Type	Create, Update, Delete
before, after, precommit, using	IN principal PrincipalEntityType, IN dependent DependentEntityType, OUT error ErrorType
postcommit	IN principal PrincipalEntityType, IN dependent DependentEntityType

Note

Parameter types (**IN**, **OUT**) are checked during activation; the data types of table type columns are **not** checked.

The **OUT** parameter enables you to return error information. The first row in the **OUT** table is then serialized as **inner error** in the error message. If no error occurs, the **OUT** table must remain empty. The structure of the table type **ErrorType** is not restricted. Any columns with special names **HTTP_STATUS_CODE** and **ERROR_MESSAGE** are mapped to common information in the OData error response. Content of columns with other names are serialized into the **inner error** part of the error message that allows the return of custom error information.

Error Message Content

Column Name	Type	Value Range	Error Response Information
HTTP_STATUS_CODE	INTEGER	400-417 (default: 400)	The HTTP response status code
ERROR_MESSAGE	NVARCHAR		The error message (<message>)

Note

If the SQLScript procedure throws an exception or writes an error messages to the **OUT** parameter table, the OData write operation is aborted. If more than one error message is added, only the content of the first row is returned in the resulting error message. Any scripts registered for the **postcommit** event must not have an **OUT** parameter as the write operation cannot be aborted at such a late stage, even in the event of an error.

The following example illustrates a typical error-type table type, which is defined in a design-time file that must have the **.hdbtabletype** file suffix, for example **error.hdbtabletype**:

```
"sample.odata::error" AS TABLE (
```



```

    "HTTP_STATUS_CODE" INTEGER,
    "ERROR_MESSAGE" NVARCHAR(100),
    "DETAIL" NVARCHAR(100)
)

```

The following example shows how information is extracted from the error table if an error occurs during the execution of a create procedure for an OData write operation:

```

create procedure "sample.odata::createmethod"(IN new "sample.odata::table", OUT
error "sample.odata::error")
language sqlscript
sql security invoker as
    id INT;
begin
    select ID into id from :new;
    if :id < 1000 then
        error = select 400 as http_status_code,
            'invalid ID' error_message,
            'value must be >= 1000' detail from dummy;
    else
        insert into "sample.odata::table" values (:id);
    end if;
end;

```

8.1.6.13 Tutorial: Creating a Validation Exit with SQLScript

Use SQLScript to create a custom validation exit which runs server-side verification and data-consistency checks for an OData **update** operation.

Prerequisites

To perform this task, you need the following objects:

- A table to expose, for example, `sample.odata::table.hdbtable`
- An error type, for example, `sample.odata::error.hdbtabletype`

Context

In this tutorial, you see how to register an SQL script for an OData update operation; the script verifies, before the execution of the update operation, that the updated value is larger than the previous one. In the example shown in this tutorial, you define the table to be updated and a table type for the error output parameter of the exit procedure.

Procedure

1. Create a table definition file using `.hdbtable` syntax.

The table to expose is defined in `sample.odata:table.hdbtable`, which should look like the following example:

```
COLUMN TABLE "table" (  
    "ID" INTEGER NOT NULL,  
    PRIMARY KEY ("ID")  
);
```

2. Create a table type for the error output parameter of the exit procedure.

The error type file `sample.odata:error.hdbtabletype` should look like the following example:

```
"sample.odata::error" AS TABLE (  
    "HTTP_STATUS_CODE" INTEGER,  
    "ERROR_MESSAGE" NVARCHAR(100),  
    "DETAIL" NVARCHAR(100)  
)
```

3. Create a procedure that runs before the UPDATE event.

The procedure script for the `before UPDATE` event must have two table input parameters and one output parameter, for example:

- IN new "sample.odata::table"
- IN old "sample.odata::table"
- OUT error "sample.odata::error"

The procedure `sample.odata:beforeupdate.hdbprocedure` would look like the following example:

```
procedure "sample.odata::beforeupdate"  
    (IN new "sample.odata::table", IN old "sample.odata::table", OUT error  
    "sample.odata::error")  
language sqlscript  
sql security invoker as  
    idnew INT;  
    idold INT;  
begin  
    select ID into idnew from :new;  
    select ID into idold from :old;  
    if :idnew <= :idold then  
        error = select 400 as http_status_code,  
            'invalid ID' error_message,  
            'the new value must be larger than the previous' detail from dummy;  
    end if;  
end;
```

4. Register the procedure to be executed at the `before` event.

You use the `update events` (`before "..."`) keywords to register the procedure, as illustrated in the following example of an OData service file:

```
service {  
    "sample.odata::table"  
        update events (before "sample.odata::beforeupdate");  
}
```

8.1.6.14 Tutorial: Creating a Modification Exit with SQLScript

Register an SQL script as a modification exit for an OData **create** operation for an entity.

Prerequisites

To perform this task, you need the following objects:

- A table to expose for the OData create operation, for example, `sample.odata:table.hdbtable`
- An error type, for example, `sample.odata:error.hdbstructure`

Note

These objects are used as **types** in the procedure.

Context

SAP HANA XS enables you to register custom code that handles the OData write operation for non-trivial cases. In this tutorial, you see how to register a modification exit for an OData **CREATE** operation for an entity. The procedure you register verifies the data to insert, refuses the insertion request if the specified ID is less than 1000, and in the event of an error, inserts a row with error information into the output table.

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 table definition file using `.hdbtable` syntax.

The table you create in this step is used in the procedure you create later in the tutorial. The table to expose is defined in `sample.odata:table.hdbtable`, which should look like the following example:

```
table.schemaName = "ODATASAMPLES";
table.columns = [{name = "ID"; sqlType = INTEGER; nullable = false;});
table.primaryKey.pkcolumns = ["ID"];
```

2. Create a table type for the error output parameter of the exit procedure.

The error type you create in this step is used in the procedure you create later in the tutorial. The error type file `sample.odata:error.hdbstructure` should look like the following example:

```
table.schemaName = "ODATASAMPLES";
```

```
table.columns = [
  {name = "HTTP_STATUS_CODE"; sqlType = INTEGER;},
  {name = "ERROR_MESSAGE"; sqlType = NVARCHAR; length = 100;},
  {name = "DETAIL"; sqlType = NVARCHAR; length = 100;}
];
```

3. Create a procedure that runs before the UPDATE event.

The table and error type objects you created in the previous steps are used as **types** in the procedure created here. The procedure also performs a verification on the data, rejects the insertion in case of an ID below 1000, and inserts a row with error information into the output table.

The procedure `sample.odata:createmethod.hdbprocedure` should look like the following example:

```
procedure "ODATA_TEST"."sample.odata::createmethod"
  (IN new "sample.odata::table", OUT error "sample.odata::error")
language sqlscript
sql security invoker as
  id INT;
begin
  select ID into id from :new;
  if :id < 1000 then
    error = select 400 as http_status_code,
      'invalid ID' error_message,
      'value must be >= 1000' detail from dummy;
  else
    insert into "sample.odata::table" values (:id);
  end if;
end;
```

4. Register the procedure to be executed at the CREATE event.

You use the `create using` keywords to register the procedure, as illustrated in the following OData service file:

```
service {
  "sample.odata::table"
  create using "sample.odata::createmethod";
}
```

8.1.6.15 Tutorial: Creating a Modification Exit with XS JavaScript

You can use server-side JavaScript to write a script which you register as a modification exit for an OData **update** operation for an entity.

Prerequisites

To perform this task, bear in mind the following prerequisites:

- A table to expose for the OData create operation, for example, `sample.odata::table.hdbtable`

Context

SAP HANA XS enables you to register custom code that handles the OData write operation. In this tutorial, you see how to use server-side JavaScript (XSJS) to write a script which you register as a modification exit for OData UPDATE operations for an entity. The script you register verifies the data to insert, and throws a defined error string in the event of an error, for example, `Could not update table; check access permissions`.

To register an XS JavaScript function as an OData modification exit, perform the following steps:

Procedure

1. Create a table definition file, for example, using the `.hdbtable` syntax.

The table you create in this step is used in the XS JavaScript function you create later in the tutorial. The table to expose is defined in `sample.odata.table.hdbtable`, which should look like the following example:

```
COLUMN TABLE "table" (  
  "ID" INTEGER NOT NULL,  
  PRIMARY KEY ("ID")  
);
```

2. Create the XS JavaScript function that you want to register for OData modification events.

Note

The XS JavaScript function that you want to register for OData modification events must be created in the form of an XSJS library, for example, with the file extension `.xsjslib`; the XS JavaScript function cannot be an `.xsjs` file.

The function you register has one parameter, which can have the properties listed in the following table:

Property	Type	Description
connection	Connection	The SQL connection used in the OData request
beforeTableName	String	The name of a temporary table with the single entry before the operation (UPDATE and DELETE events only)
afterTableName	String	The name of a temporary table with the single entry after the operation (CREATE and UPDATE events only)

The XS JavaScript function `jsexit.xsjslib` could look like the following example:

```
function update_instead(param) {  
  $.trace.debug("entered function");  
  let before = param.beforeTableName;  
  let after = param.afterTableName;  
  let pStmt = param.connection.prepareStatement('select * from "' + after  
+ '"');  
  // ...  
  if (ok) {  
    // update
```

```

    } else {
        throw "an error occurred; check access privileges"
    }
}

```

3. Bind the XS JavaScript function to the entity specified in the OData service definition.

To bind the XS JavaScript function to a specified entity, use the syntax:

<Package.Path>:<file>.<suffix>::<XSJS_FunctionName> as illustrated in the following example:

```

service {
    "sample.odata::table" as "Table" update using
    "sap.test:jsexit.xsjslib::update_instead";
}

```

8.1.7 OData Service Definition Language Syntax (XS Advanced)

The OData Service Definition Language (OSDL) provides a set of keywords that enable you to set up an ODATA service definition file that specifies what data to expose, in what way, and to whom.

The following list shows the syntax of the OData Service Definition Language (OSDL) in an EBNF-like format; conditions that apply for usage are listed after the table.

definition	:=service [annotations]
service	:= 'service' [namespace] body
namespace	:= 'namespace' quotedstring
quotedstring	:= quote string quote
string	:=UTF8
quote	:= '"'
body	:= '{' content '}'
content	:= entry [content]
entry	:= (entity association) ';'
entity	:= object [entityset] [with] [keys] [navigates]
[aggregates] [parameters]	[modification]
object	:= ['entity'] (repoobject catalogobject)
repoobject	:= quote repopackage '/' reponame '.' repoextension quote
repopackage	:= string
reponame	:= string
repoextension	:= string
catalogobject	:= catalogobjectschema '.' catalogobjectname
catalogobjectschema	:= quotedstring
catalogobjectname	:= quotedstring
entityset	:= 'as' entitysetname
entitysetname	:= quotedstring
with	:= ('with' 'without') propertylist
propertylist	:= '(' columnlist ')'
columnlist	:= columnname [',' columnlist]
columnname	:= quotedstring
keys	:= 'keys' (keylist keygenerated)
keylist	:= propertylist
keygenerated	:= 'generate' (keygenlocal)
keygenlocal	:= 'local' columnname
navigates	:= 'navigates' '(' navlist ')'
navlist	:= naventry [',' navlist]
naventry	:= assocname 'as' navproprname [fromend]
assocname	:= quotedstring
navproprname	:= quotedstring
fromend	:= 'from' ('principal' 'dependent')
aggregates	:= 'aggregates' 'always' [aggregatestuple]

```

aggregatestuple      :=(' aggregateslist ')
aggregateslist       :=aggregate ['',' aggregateslist]
aggregate            :=aggregatefunction 'of' columnname
aggregatefunction    :=( 'SUM' | 'AVG' | 'MIN' | 'MAX' )
parameters           :='parameters' 'via' [parameterskeyand] 'entity'
[parameterentitysetname] [parametersresultsprop]
parameterskeyand     :='key' 'and'
parameterentitysetname :=quotedstring
parametersresultsprop :='results' 'property' quotedstring
modification         :=[create] [update] [delete]
create               :='create' modificationspec
update               :='update' modificationspec
delete               :='delete' modificationspec
modificationspec     :=( modificationaction [events] | events | 'forbidden' )
modificationaction   :='using' action
action               :=quotedstring
events               :='events' '(' eventlist ')'
eventlist            :=eventtype action ['',' eventlist]
eventtype            :=( 'before' | 'after' | 'precommit' | 'postcommit' )
association           :=associationdef [assocrefconstraint] principalend
dependentend [( associatable | storage | modification )]
associationdef       :='association' assocname
assocrefconstraint   :='with' 'referential' 'constraint'
principalend         :='principal' end
dependentend         :='dependent' end
end                  :=endref multiplicity
endref               :=endtype [joinpropertieslist]
endtype              :=entitysetname
joinpropertieslist   :=(' joinproperties ')
joinproperties       :=columnlist
multiplicity         :='multiplicity' quote multiplicityvalue quote
multiplicityvalue    :=( '1' | '0..1' | '1..*' | '*' )
associatable         :='over' repoobject overprincipalend overdependentend
[modification]
overprincipalend     :='principal' overend
overdependentend     :='dependent' overend
overend              :=propertylist
storage              :=( nostorage | storageend [modification] )
nostorage            :='no' 'storage'
storageend           :='storage' 'on' ( 'principal' | 'dependent' )
annotations          :='annotations' annotationsbody
annotationsbody      :='{ ' annotationscontent '}'
annotationscontent   :=annotationconfig [annotationscontent]
annotationconfig     :='enable' annotation
annotation           :='OData4SAP'

```

Note

Support for OData annotations is currently not available in SAP HANA XS Advanced.

Conditions

The following conditions apply when using the listed keywords:

1. If the **namespace** is not specified, the schema namespace in the EDMX metadata document will be the repository package of the service definition file concatenated with the repository object name. E.g. if the repository design time name of the `.xsodata` file is `sap.hana.xs.doc/hello.xsodata` the namespace will implicitly be `sap.hana.xs.doc.hello`.
2. **keylist** must not be specified for objects of type 'table'. They must only be applied to objects referring a view type. **keygenerated** in turn, can be applied to table objects.

3. If the `entityset` is not specified in an entity, the `EntitySet` for this object is named after the repository object name or the `catalogobjectname`. For example, if `object` is `"sap.hana.xs.doc/odata_docu"`, then the `entitysetname` is implicitly set to `odata_docu`, which then can also be referenced in associations.
4. The `fromend` in a `naventry` must be specified if the `endtype` is the same for both the `principalend` and the `dependentend` of an association.
5. The number of `joinproperties` in the `principalend` must be the same as in the `dependentend`.
6. Ordering in the `joinproperties` of `ends` is relevant. The first `columnname` in the `joinproperties` of the `principalend` is compared with the first `columnname` of the `dependentend`, the second with the second, and so on.
7. The `overprincipalend` corresponds to the `principalend`. The number of properties in the `joinproperties` and the `overproperties` must be the same and their ordering is relevant. The same statement is true for the dependent end.
8. `aggregates` can only be applied in combination with `keygenerated`.
9. If `aggregatestuple` is omitted, the aggregation functions are derived from the database. This is only possible for calculation views and analytic views.
10. Specifying `parameters` is only possible for calculation views and analytic views.
11. The default `parameterentitysetname` is the `entitysetname` of the entity concatenated with the suffix `"Parameters"`.
12. If the `parametersresultsprop` is omitted, the navigation property from the parameter entity set to the entity is called `"Results"`.
13. Support for OData annotations is currently under development. For more information about the SAP-specific metadata annotations that become available with the `enable OData4SAP` statement in an `.xsodata` file, see the Related Links below. Note that not all annotations allowed by OData are supported by SAP HANA XS.

Related Information

[SAP Annotations for OData Version 2.0](#) ➡

[Open Data Protocol](#) ➡

8.1.8 OData Service Definition: SQL-EDM Type Mapping (XS Advanced)

During the activation of the OData service definition, the SAP HANA SQL types are mapped to the required OData EDM types according to the rules specified in a mapping table.

The following mapping table lists how SAP HANA SQL types are mapped to OData EDM types during the activation of an OData service definition.

📘 Note

The OData implementation in SAP HANA XS supports only those SQL types listed in the following table.

SAP HANA SQL Type	OData EDM Type
Time	Edm.Time
Date	Edm.DateTime
SecondDate	Edm.DateTime
LongDate	Edm.DateTime
Timestamp	Edm.DateTime
TinyInt	Edm.Byte
SmallInt	Edm.Int16
Integer	Edm.Int32
BigInt	Edm.Int64
SmallDecimal	Edm.Decimal
Decimal	Edm.Decimal
Real	Edm.Single
Float	Edm.Single
Double	Edm.Double
Varchar	Edm.String
NVarchar	Edm.String
Char	Edm.String
NChar	Edm.String
Binary	Edm.Binary
Varbinary	Edm.Binary

Example SQL Type Mapping

The following examples shows how SAP HANA SQL types (name, integer, NVarchar) of columns in a table are mapped to the OData EDM types in the properties of an entity type.

SAP HANA SQL:

```
{name = "ID"; sqlType = INTEGER; nullable = false;},
{name = "RefereeID"; sqlType = NVARCHAR; length = 10; nullable = true;}
```

The following example illustrates how the SAP HANA SQL types illustrated in the previous example are mapped to EDM types:

```
<Property Name="ID" Type="Edm.Int32" Nullable="false"/>
<Property Name="RefereeID" Type="Edm.String" Nullable="true" MaxLength="10"/>
```

8.1.9 OData Security Considerations

Enabling access to data by means of OData can create some security-related issues that you need to consider and address, for example, the data you want to expose, who can start the OData service, and so on.

If you want to use OData to expose data to users and clients in SAP HANA application services, you need to bear in mind the security considerations described in the following list:

- **Data Access**
Restrict user select authorization for tables/views exposed by the OData service
- **OData Service**
Restrict authorization rights to start the OData service
- **OData Statistical content**
Restrict access to the URL/Path used to expose OData content in the Web browser

8.1.10 OData Batch Requests (XS Advanced)

The OData standard allows the collection of multiple individual HTTP requests into one single batched HTTP request.

Clients using a defined OData service to consume exposed data can collect multiple, individual HTTP requests, for example, retrieve, create, update and delete (GET, POST, PUT, DELETE), in a single “batch” and send the batched request to the OData service as a single HTTP request. You can compile the batch request manually (by creating the individual requests in the batch document by hand) or automatically, for example, with an AJAX call that adds requests to a queue and loops through the queues to build the batch request. In both cases, the OData standard specifies the syntax required for the header and body elements of a valid batch request document.

SAP HANA XS supports the OData \$batch feature out-of-the-box; there is nothing to configure in SAP HANA XS to use \$batch to perform operations in SAP HANA using an OData service. To understand how the \$batch feature works, you need to look at the following phases of the operation:

- Batch Request
- Batch Response

A batch request is split into two parts: the request header and the request body. The body of a batch request consists of a list of operations in a specific order where each operation either retrieves data (for example, using the HTTP GET command) or requests a change. A change request involves one or more insert, update or delete operations using the POST, PUT, or DELETE commands.

Note

A change request must not contain either a **retrieve** request or any nested change requests.

The batch request must contain a Content-Type header specifying the value “multipart/mixed” and a boundary ID boundary=batch_#; the batch boundary ID is then used to indicate the start of each batch request, as illustrated in the following example.

```
POST /service/$batch HTTP/1.1
Host: host
Content-Type: multipart/mixed;boundary=batch_8219-6895 // Define batch ID
```

```

--batch_8219-6895                                     // Batch 1 start
Content-Type: multipart/mixed; boundary=changeset_a4e3-a738 // Define
changeset ID
--changeset_a4e3-a738                                   // Changeset 1
start
Content-Type: application/http
Content-Transfer-Encoding: binary
[PUT...]

--changeset_a4e3-a738                                   // Changeset 2
start
Content-Type: application/http
Content-Transfer-Encoding: binary
[POST...]
--changeset_a4e3-a738--                                 // Changeset (all)
end
--batch_8219-6895                                     // Batch part 2
start
Content-Type: application/http
Content-Transfer-Encoding: binary
[GET...]
--batch_8219-6895--                                     // Batch (all) end

```

Within the batch request, changeset is defined by another boundary ID (for example, `boundary=changeset_123`), which is then used to indicate the start and end of the change requests. The batch request must be closed, too.

Note

In the following example of a simple OData batch request, some content has been removed to emphasize the structure and layout.

```

POST http://localhost:8002/sap/sample/odata/syntax.xsodata/$batch HTTP/1.1
Host: localhost:8002
Connection: keep-alive
Content-Length: 471
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/30.0.1599.101 Safari/537.36
Cache-Control: no-cache
Content-Type: multipart/mixed; boundary=batch_123
Accept: */*
Accept-Encoding: identity
Accept-Language: en-US,en;q=0.8
x-sap-request-language: en-US
--batch_123
Content-Type: multipart/mixed; boundary=changeset_456
Content-Transfer-Encoding: binary
--changeset_456
Content-Type: application/http
Content-Transfer-Encoding: binary
POST BatchSample HTTP/1.1
Content-Type: application/json
Content-Length: 11
{"ID" : 14}
--changeset_456
Content-Type: application/http
Content-Transfer-Encoding: binary
POST BatchSample HTTP/1.1
Content-Type: application/json
Content-Length: 11
Accept: application/json
{"ID" : 15}
--changeset_456--
--batch_123--

```

The batch response includes a response for each of the retrieve or change operations included in the corresponding batch request. The order of the responses in the response body must match the order of requests in the batch request. In the context of the batch response, the following is true:

- The response to a retrieve request is always formatted in the same way regardless of whether it is sent individually or as part of batch.
- The body of the collected response to a set of change-requests is one of the following:
 - A response for **all** the successfully processed change requests within the change set, in the correct order and formatted exactly as it would have appeared outside of a batch
 - A single response indicating the failure of the entire change set

The following example shows the form and syntax of the OData batch response to the request illustrated above.

```
HTTP/1.1 202 Accepted
content-type: multipart/mixed; boundary=0CDF14D90919CC8B4A32BD0E0B330DA10
content-length: 2029
content-language: en-US
cache-control: no-cache
expires: Thu, 01 Jan 1970 00:00:00 GMT
--0CDF14D90919CC8B4A32BD0E0B330DA10
Content-Type: multipart/form-data; boundary=0CDF14D90919CC8B4A32BD0E0B330DA11
Content-Length: 1843
--0CDF14D90919CC8B4A32BD0E0B330DA11
Content-Type: application/http
Content-Length: 1118
content-transfer-encoding: binary
HTTP/1.1 201 Created
Content-Type: application/atom+xml;charset=utf-8
location: http://localhost:8002/sap/sample/odata/syntax.xsodata/BatchSample(14)/
Content-Length: 943
<?xml version="1.0" encoding="utf-8"
standalone="yes"?><entry xml:base="http://localhost:8002/sap/sample/
odata/syntax.xsodata/" xmlns:d="http://schemas.microsoft.com/ado/
2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata" xmlns="http://www.w3.org/2005/Atom"><id>http://
localhost:8002/sap/sample/odata/syntax.xsodata/BatchSample(14)</id><title
type="text"></title><author><name /></author><link rel="edit"
title="BatchSample" href="BatchSample(14)"/><link rel="http://
schemas.microsoft.com/ado/2007/08/dataservices/related/Ref" type="application/
atom+xml;type=entry" title="Ref" href="BatchSample(14)/Ref"></
link><category term="sap.sample.odata.syntax.BatchSampleType"
scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" /><content
type="application/xml"><m:properties><d:ID m:type="Edm.Int32">14</d:ID><d:SELFID
m:type="Edm.Int32" m:null="true"></d:SELFID></m:properties></content></entry>
--0CDF14D90919CC8B4A32BD0E0B330DA11
Content-Type: application/http
Content-Length: 427
content-transfer-encoding: binary
HTTP/1.1 201 Created
Content-Type: application/json
location: http://localhost:8002/sap/sample/odata/syntax.xsodata/BatchSample(15)
Content-Length: 271
{"d":{"__metadata": {"uri":"http://localhost:8002/sap/sample/odata/
syntax.xsodata/
BatchSample(15)","type":"sap.sample.odata.syntax.BatchSampleType"},"ID":15,"SELFID":null,"Ref":{"__deferred":{"uri":"http://localhost:8002/sap/sample/odata/
syntax.xsodata/BatchSample(15)/Ref"}}}}
--0CDF14D90919CC8B4A32BD0E0B330DA11--
--0CDF14D90919CC8B4A32BD0E0B330DA10--
```

OData Batch Requests in SAPUI5 Applications

If you are developing a UI client using SAPUI5, you can make use of the ODataModel tools to ensure that the data requests generated by the various UI controls bound to an OData service are collected and sent in batches. The SAPUI5 ODataModel toolset includes a large selection of tools you can use to configure the use of the OData batch feature, for example:

- **setUseBatch**
Enable or disable batch processing for all requests (read and change)
- **addBatchChangeOperations**
Appends the change operations to the end of the batch stack, which is sent with the **submitBatch** function
- **addBatchReadOperations**
Appends the read operations to the end of the batch stack, which is sent with the **submitBatch** function
- **submitBatch**
Submits the collected changes in the batch which were collected via **addBatchReadOperations** or **addBatchChangeOperations**.

Related Information

[Open Data Protocol](#) 

[SAPUI5 ODataModel Reference](#)

8.2 Using the SAP HANA REST API

The SAP HANA REST Application Programming Interface (REST API) is based on and extends the Orion server and client APIs.

SAP HANA REST API supports the Orion protocol 1.0 which allows development tools to access the SAP HANA Repository in a convenient and standards-compliant way. This not only makes access to the Repository easier for SAP HANA tools, but it also enables the use of Orion-based external tools with the SAP HANA Repository. For SAP tools, the Orion server protocol has been extended with the following SAP HANA-specific features

- Activate design-time artifacts in the Repository
- Perform change-tracking operations (assuming change-tracking is enabled in the target SAP HANA system)
- Searching the database catalog

SAP HANA REST Application Programming Interfaces

API	Description
File	Enables access to services that you to browse and manipulate files and directories via HTTP
Workspace	Enables you to create and manipulate workspaces and projects via HTTP

API	Description
Transfer	Enables you to import and export packages and files
Metadata	Enables access to services that support search and auto-completion scenarios, for example, to retrieve metadata from runtime, design-time, and other metadata locations
Change Tracking	Enables the use of specific lifecycle-management features included with the SAP HANA Repository via HTTP
Info	Enables access to information about the current version of the SAP HANA REST API

The SAP HANA REST API uses an additional parameter called **SapBackPack** to send request parameters that are specific to SAP HANA; the **SapBackPack** parameter is added to the HTTP header. The value of the **SapBackPack** parameter is a JSON object with the attributes and values of the additional SAP-specific parameters. For example, when you create or update the content of a design-time artifact, you can use the **SapBackPack** value `{"Activate": true}` to request that the new version of the file is immediately activated in the SAP HANA Repository. If you only want to create an inactive version of a design-time artifact, you can use the `"workspace"` attribute to specify the name of the the Repository workspace where the inactive version is to be stored.

Related Information

[SAP HANA REST Info API \[page 517\]](#)

[SAP HANA REST File API \[page 518\]](#)

[SAP HANA REST Change-Tracking API \[page 523\]](#)

[SAP HANA REST Metadata API \[page 524\]](#)

[SAP HANA REST Transfer API \[page 526\]](#)

[SAP HANA REST Workspace API \[page 527\]](#)

[SAP HANA REST API Reference](#)

8.2.1 SAP HANA REST Info API

The SAP HANA REST API includes an Info API that can be used to display information about the current version of the REST API.

```
GET /sap/hana/xs/dt/base/info
Orion-Version: 1.0
```

The information displayed by the Info API includes a description of the current version of the delivery unit and the number of commands (API entry points) that are currently supported by the REST API.

```
HTTP/1.1 200 OK
{
  "DeliveryUnit": {
    "name": "HANA_DT_BASE",
    "version": "1",
    "responsible": "x###007, x###077",
    "vendor": "sap.com",
```

```

"version_sp": "0",
"version_patch": "8",
"ppmsID": "",
"caption": "",
"lastUpdate": 1386163749544,
"sp_PPMS_ID": "",
"ach": ""
},
"Commands": [
"/sap/hana/xs/dt/base/file",
"/sap/hana/xs/dt/base/workspace",
"/sap/hana/xs/dt/base/xfer/import",
"/sap/hana/xs/dt/base/metadata",
"/sap/hana/xs/dt/base/change",
"/sap/hana/xs/dt/base/info"
]
}

```

Related Information

[Using the SAP HANA REST API \[page 516\]](#)
[SAP HANA REST API Reference](#)

8.2.2 SAP HANA REST File API

The SAP HANA REST API includes a File API which uses the basic HTTP methods GET, PUT, and POST to send requests. JSON is used as the default representation format.

The File API enables you to perform the following actions:

Action	Description
Actions on files [page 518]	Get, set, or change file content and metadata
Actions on directories [page 519]	Get and change directory metadata and list directory contents
Create files and directories [page 520]	Create files and directories with or without content
Copy and move files [page 521]	Copy, move, or delete files and directories
Mass transfer actions [page 521]	Get multiple files (or file metadata) from a list, repository package, or a workspace
Change tracking [page 522]	Activate selectively the latest approved versions of repository objects

Actions on Files

```

GET /sap/hana/xs/dt/base/file/MyProj/myfile.txt
Orion-Version: 1.0

```

```
If-Match: "358768768767"  
SapBackPack: "{\"Workspace\": 'ABC', 'Version': 12}"
```

The REST File API enables you to retrieve the content of a specific file, for example, `myfile.txt`.

Note

In the request illustrated in the example above, the parameters `Version`, `If-Match`, and `SapBackPack` are optional

The response to the retrieval request is displayed in the following example:

```
HTTP/1.1 200 OK  
Content-Type: text/plain  
Content-Length: 22  
This is the content
```

The REST File API enables you to retrieve the metadata associated with a specific file, for example, `myfile.txt`.

Note

In the request illustrated in the example below, the parameters `Version`, `If-Match`, and `SapBackPack` are optional

```
GET /sap/hana/xs/dt/base/file/MyProj/myfile.txt?parts=meta  
Orion-Version: 1.0  
SapBackPack: "{\"History\": 'false', 'Version': 12}"  
If-Match: "35987989879"
```

The response to the retrieval request for metadata is displayed in the following example:

```
{  
  "Name": "myfile.txt",  
  "Location": "/sap/hana/xs/dt/base/file/MyProj/myfile.txt",  
  "RunLocation": "/MyProj/myfile.txt",  
  "ETag": "35987989879",  
  "Directory": false,  
  "LocalTimeStamp": 01234345009837,  
  "Attributes": {  
    "ReadOnly": false,  
    "Executable": false,  
    "SapBackPack" : {'Activated' : true}  
  }  
  "SapBackPack": {  
    "Version": 60,  
    "ActivatedAt": 1397644007537,  
    "ActivatedBy": "User"  
  }  
}
```

Actions on Directories

You can use the REST File API to retrieve and change directory (repository package) metadata as well as list the contents of a directory. The following example shows how to list the contents of a single directory, for example, `myfolder`.

→ Tip

To list all files from a directory recursively, use `depth=infinity` or `-1`. For security reasons the depth is limited to 1000.

```
GET /sap/hana/xs/dt/base/file/MyProj/myfolder?depth=1
```

The following example shows the response to the directory listing request:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 132
{
  "Name": "myfolder",
  "Location": "/sap/hana/xs/dt/base/file/MyProj/myfolder",
  "ContentLocation": "/MyProj/myfolder",
  "LocalTimeStamp": 01234345009837,
  "Directory": true
  "Attributes": {
    "ReadOnly": false,
    "Executable": false
  },
  "Children": [
    {
      "Name": "myfile.txt",
      "Location": "/sap/hana/xs/dt/base/file/MyProj/myfolder/myfile.txt",
      "RunLocation": "/MyProj/myfolder/myfile.txt",
      "Directory": false
    }
  ]
}
```

File and Directory Creation

You can use the REST File API to create files and directories (repository packages) with or without content. The following example shows how to create a new directory, for example, `myfolder`.

ⓘ Note

If a parent directory (in which the new directory is created) is already assigned to a delivery unit, the created directory will be assigned automatically to the same delivery unit.

```
POST /sap/hana/xs/dt/base/file/MyProj/
Content-Type: application/json
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"
Slug: myfolder
{
  "Name": "myfolder",
  "Directory": "true"
}
```

The following example shows the response to the directory creation request:

```
HTTP/1.1 201 OK
{
  "Name": "myfolder",
  "Location": "/sap/hana/xs/dt/base/file/MyProj/myfolder",
  "ContentLocation": "/MyProj/myfolder",
  "ETag": "35fd43td3",
}
```

```
"LocalTimeStamp": 01234345009837,  
"Directory": true  
"Attributes": {  
  "ReadOnly": false,  
  "Executable": false  
}
```

Copying and Moving Files

You can use the REST File API to copy, move, or delete files and directories (repository packages). You can also use the File API to delete the workspace that contains files and directories used for development work. The following example shows how to delete a directory, for example, `myfolder`.

```
DELETE /sap/hana/xs/dt/base/file/MyProj/myfile.txt  
Orion-Version = 1.0  
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"  
If-Match: "35" (optional)
```

The following example shows how to delete a workspace.

Note

You need to include the parameters `ProcessWorkspace=true` and `Workspace` in the `SapBackPack` parameter.

```
DELETE /sap/hana/xs/dt/base/file/MyProj/myfile.txt  
Orion-Version = 1.0  
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"  
SapBackPack: "{ 'Workspace': 'ABC', 'ProcessWorkspace': true }"
```

Both requests should receive the following response:

```
HTTP/1.1 204 OK
```

Mass File Transfer

Mass transfer with the REST File API enables you to apply GET and PUT operations to multiple files in a single HTTP request.

Note

The mass-transfer feature is not a part of the Orion specification; it was developed to optimize the performance of GET and PUT requests when dealing with a large number of files.

There are different ways of specifying the file paths. One way is to point the request's URL to the root of the file repository, as illustrated in the request example below. In this case, you must specify the complete path from the root of the repository for each file. Another possibility is to point the request's URL to a specified sub-package in the Repository, which is then considered to be the root package for the files to be retrieved in the request. To request a file's meta-data, use the parameter `parts=meta`; the response contains a list of file

metadata formatted as a JSON string. If the request does not contain the parameter `parts=meta`, a multipart response is returned.

```
GET /sap/hana/xs/dt/base/file?parts=meta
Orion-Version = 1.0
SapBackPack: '{"MassTransfer":true, "MassTransferData": [
{"Pkg":"MyProj/myfolder", "Name":"destination1.txt", "Dir":false}, ...]}'
```

The response expected should look like the following example:

```
HTTP/1.1 200 OK
Content-Type: application/json
[
{
  "Name": "destination1.txt",
  "Location": "/sap/hana/xs/dt/base/file/MyProj/myfolder/destination1.txt",
  "RunLocation": "/MyProj/myfolder/destination1.txt",
  "ETag": "351234567",
  "LocalTimeStamp": 01234345009837,
  "Directory": false
  "Attributes": { "ReadOnly": false, "Executable": true, "SapBackPack" :
  {'Activated' : true}}
},
{
  "Name": "destination2.txt",
  "Location": "/sap/hana/xs/dt/base/file/MyProj/myfolder/destination2.txt",
  "RunLocation": "/MyProj/myfolder/destination2.txt",
  "ETag": 251237891,
  "LocalTimeStamp": 01234345009837,
  "Directory": false,
  "Attributes": { "ReadOnly": false, "Executable": true, "SapBackPack" :
  {'Activated' : true}}
}
]
```

Change Tracking

Use can use the REST File API to perform change-tracking operations. Change tracking enables you to activate selectively the latest approved versions of objects.

Note

This feature of the REST File API assumes that the change-tracking feature is enabled in the SAP HANA repository.

```
PUT /sap/hana/xs/dt/base/file/PATH?parts=meta
Orion-Version = 1.0
X-CSRF-Token: securityToken
SapBackPack: '{"MassTransfer":true, "Activate":true, "ChangeId": "ABC//11111",
"ChangeIdList": [{"Path": "PATH/file1.txt", "ChangeId" = "ABC//12345"},
{"Path": "PATH/file2.txt", "ChangeId" = "ABC//12345"}]}'
```

If an object (or a set of objects) is activated using the default change-tracking handling (for example, without setting `SapBackPack.ChangeTrackingMode` or by setting `SapBackPack.ChangeTrackingMode` explicitly to 0), a dynamic change list is created, and the file(s) are activated in the SAP HANA Repository using the generated change list.

In the explicit handling of change tracking the user is allowed to activate files that are already assigned to a change list. Files can also be activated using an explicitly provided change list ID. In the change-tracking request above, the files `PATH/file1.txt` and `PATH/file2.txt` are assigned to the change list `ABC//12345`. All other files will be activated using the change list `ABC//11111`.

The response to the change-tracking request would look like the following example:

```
HTTP/1.1 200 OK
Content-Type: application/json
...
```

Related Information

[Using the SAP HANA REST API \[page 516\]](#)

[SAP HANA REST API Reference](#)

8.2.3 SAP HANA REST Change-Tracking API

The SAP HANA REST API includes a Change Tracking API which enables you to make use of specific lifecycle-management features that are included with the SAP HANA Repository via HTTP.

Change Tracking is integrated with the SAP HANA XS Repository transport tool set; with change tracking enabled, you can ensure that an export operation (to build a delivery unit) includes only the latest approved versions of repository objects.

Note

To use the Change-Tracking API, change tracking must be enabled in the SAP HANA system whose repository you are accessing.

To obtain the current status of change tracking in the system, for example, enabled or disabled, you can send a GET request to the **change** entry point of the REST API.

```
GET /sap/hana/xs/dt/change
```

If the change tracking feature is **enabled** in the target system, the resulting response is `true`. If change tracking is **disabled** in the target system or not supported by the system, the response to the GET status request is `false`.

```
HTTP/1.1 200 OK
{
  "ChangeTrackingStatus": true
}
```

You can also use the REST Change-Tracking API to manage change lists and track changes made to repository objects. For example, to display all change lists, for which a specified user ("XYZ") is a contributor:

```
GET /sap/hana/xs/dt/base/change
SapBackPack: {'User': 'XYZ', 'Status': 1}
```

The response would look like the following example:

```
HTTP/1.1 200 OK
[
  {
    "changeID": "ABC//1234",
    "status": 1,
    "description": "",
    "createdOn": "2014-04-09T13:26:58.868Z",
    "createdBy": "XYZ"
  },
  {
    "changeID": "ABC//1235",
    "status": 1,
    "description": "",
    "createdOn": "2014-04-09T14:08:53.024Z",
    "createdBy": "XYZ"
  }
]
```

To display the change status of a single file `SomeFile.txt`, use the following command:

```
GET /sap/hana/xs/dt/base/change/MyProj/SomeFile.txt
```

The response would look like the following example, which shows the change ID and the user responsible for the change:

```
HTTP/1.1 200 OK
{
  "ChangeId": "ABC//1234",
  "User": "XYZ"
}
```

Related Information

[Using the SAP HANA REST API \[page 516\]](#)

[SAP HANA REST API Reference](#)

8.2.4 SAP HANA REST Metadata API

The SAP HANA REST API includes a Metadata API which provides services to support search and autocompletion scenarios

The REST-based Metadata API enables you to retrieve metadata from runtime and design-time objects as well as other metadata locations. The typical location of runtime metadata is the SAP HANA database catalog. It is possible to retrieve metadata for tables, views, procedures, functions, sequences, and schemas. The design-time location for metadata is the SAP HANA Repository. Also accessible is the metadata location used by Core Data Services (CDS).

The following services are provided with the Metadata API in the default location `/sap/hana/xs/dt/base/metadata`; the services are called by setting the HTTP parameter `Service-Name` to the appropriate value:

- `checkMetadataExistence`
Checks for the existence of a provided set of entities and returns an array of entries which indicates if a specified entity exists or not.
- `checkMetadataExistence URI`
Checks for the existence of a specific resource (entity) uniquely expressed as an HTTP universal resource indicator (URI). `checkMetadataExistence URI` returns an array of entries which indicates if a given entity exists or not.
- `getMetadataSuggestion`

Note

This part of the interface only supports HTTP GET requests.

The following example shows how to use `checkMetadataExistence URI` to check for the existence of a specific URI resource.

```
var strPayloadFromJava = "{}";
var strHeaderServiceName = "checkMetadataExistence";
var strSapBackPack = strPayloadFromJava;
var strAccessPath = cMetaDataAccessP + '/VIEW/RT/TABLES';
var request = new $.net.http.Request($.net.http.GET, strAccessPath);
request.headers.set('SapBackPack', strSapBackPack);
request.headers.set('Service-Name', strHeaderServiceName);
var response = client.request(request, destination).getResponse();
```

`checkMetadataExistence URI` returns an array of entries which indicates if a given entity exists or not.

```
List<metadata>
  localName
  isExist
  List<exist> [6]
    namespace
    separator [7]
    baseLocalName
    baseType
    type
    mode [8]
    desc
```

Related Information

[Using the SAP HANA REST API \[page 516\]](#)

[SAP HANA REST API Reference](#)

8.2.5 SAP HANA REST Transfer API

The SAP HANA REST Transfer API is used to import and export packages and files.

You can use the Transfer API to perform both import and export operations:

- [Import \[page 526\]](#)
Upload files to the SAP HANA Repository, for example, using POST, PUT, or FTP
- [Export \[page 526\]](#)
Download files from the SAP HANA Repository to a client

Importing Files

The following example shows how to use the Transfer API to start an operation to upload files to the SAP HANA Repository. The request URL uses the POST command to perform the action and must indicate the target location of the uploaded file when the upload operation is complete. The request must also indicate the total size of the file the server should expect to receive during the upload operation.

```
POST /sap/hana/xs/dt/base/xfer/import/MyProj/SomeFile.jpg
Orion-Version: 1.0
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"
Slug: MyFile.jpg
X-Xfer-Content-Length: 901024
X-Xfer-Options: raw
```

The response to the request would look as follows:

```
HTTP/1.1 200 OK
Location: /sap/hana/xs/dt/base/xfer/import/fks3kjd7hf
ContentLocation: /xfer/fks3kjd7hf
```

After initiating the transfer, uploads are performed as many times as required using PUT actions.

```
PUT /sap/hana/xs/dt/base/xfer/import/fks3kjd7hf
Orion-Version: 1.0
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"
Content-Length: 32768
Content-Type: image/jpeg
Content-Range: bytes 0-32767/901024
```

For each successful upload operation, you should see the following response:

```
HTTP/1.1 200 success
Range: bytes 0-32767
```

Exporting Files

You can use the REST Transfer API to export (download) files and packages to a designated client in a zip archive, as illustrated in the following example:

```
GET /sap/hana/xs/dt/base/xfer/export/MyProj/SomeFolder.zip
```

```
Orion-Version: 1.0
```

For each successful download operation, you should see the following response:

```
HTTP/1.1 201 OK
Content-Type: application/zip
File contents.
```

Related Information

[Using the SAP HANA REST API \[page 516\]](#)

[SAP HANA REST API Reference](#)

8.2.6 SAP HANA REST Workspace API

The Workspace API enables you to create and manipulate Repository workspaces and projects via HTTP.

With the Workspace API, you can perform the following types of operation on workspaces and projects:

- Workspaces
List available workspaces, create or delete a workspace, and display or change workspace metadata
- Projects
Add projects to a workspace, move (or rename) a project, remove a project from a workspace

Workspace Actions

You can use the REST Workspace API to create a new workspace called “My Dev Workspace”, as illustrated in the following example:

```
POST sap/hana/xs/dt/base/workspace
EclipseWeb-Version: 1.0
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"
Slug: My Dev Workspace
```

The response to the workspace-creation request should look like the following example:

```
HTTP/1.1 201 Created
Location: [http://example.com/sap/hana/xs/dt/base/file/sap/hana/xs/dt/base/
content/workspace/SAM_My_Dev_workspace_0]
ETag: "1"
Content-Type: application/json
{
  "Id": "SAM_My_Dev_workspace_0",
  "Name": "My Dev Workspace",
  "Location": "http://example.com/sap/hana/xs/dt/base/file/sap/hana/xs/dt/base/
content/workspace/SAM_My_Dev_workspace_0",
  "Projects": [],
  "Children": []
}
```


Projects

You can also use the REST Workspace API to create a new SAP HANA XS project ("My Project") and add it to an existing workspace ("My Dev Workspace"), as illustrated in the following example. The Workspace API creates the new project as an SAP HANA XS subpackage in the specified workspace package. The new project is assigned to the list of projects in the specified workspace's metadata.

Note

The new project is **not** an SAP HANA XS application package.

```
POST /sap/hana/xs/dt/base/workspace/SAM_My_Dev_workspace_0
X-CSRF-Token: "65ABA3082325A3408FBE71C87929102B"
EclipseWeb-Version: 1.0
Slug: "My Project"
```

The response to the project-creation request should look like the following example:

```
{
  "Id": "SAM_My_Dev_Workspace_0_My_Project_0",
  "Location": "http://localhost:8080/sap/hana/xs/dt/base/file/sap/hana/xs/dt/base/
content/workspace/SAM_My_Dev_Workspace_0/My Project",
  "ContentLocation": "http://localhost:8080/sap/hana/xs/dt/base/file/sap/
hana/xs/dt/base/content/workspace/SAM_My_Dev_Workspace_0/My Project",
  "Name": "My Project"
}
```

Related Information

[Using the SAP HANA REST API \[page 516\]](#)

[SAP HANA REST API Reference](#)

9 Writing Server-Side JavaScript Code

SAP HANA Extended Application Services (SAP HANA XS) provide applications and application developers with access to the SAP HANA database using a consumption model that is exposed via HTTP.

In addition to providing application-specific consumption models, SAP HANA XS also host system services that are part of the SAP HANA database, for example: search services and a built-in Web server that provides access to static content stored in the SAP HANA repository.

The consumption model provided by SAP HANA XS focuses on server-side applications written in JavaScript. Applications written in server-side JavaScript can make use of a powerful set of specially developed API functions, for example, to enable access to the current request session or the database. This section describes how to write server-side JavaScript code that enables you to expose data, for example, using a Web Browser or any other HTTP client.

9.1 Data Access with JavaScript in SAP HANA XS

In SAP HANA Extended Application Services, the persistence model (for example, tables, views and stored procedures) is mapped to the consumption model that is exposed via HTTP to clients - the applications you write to extract data from SAP HANA.

You can map the persistence and consumption models in the following way:

- **Application-specific code**

Write code that runs in SAP HANA application services. Application-specific code (for example, server-side JavaScript) is used in SAP HANA application services to provide the consumption model for client applications.

Applications running in SAP HANA XS enable you to accurately control the flow of data between the presentational layer, for example, in the Browser, and the data-processing layer in SAP HANA itself, where the calculations are performed, for example in SQL or SQLScript. If you develop and deploy a server-side JavaScript application running in SAP HANA XS, you can take advantage of the embedded access to SAP HANA that SAP HANA XS provides; the embedded access greatly improves end-to-end performance.

9.2 Using Server-Side JavaScript in SAP HANA XS

SAP HANA application services (XS server) supports server-side application programming in JavaScript. The server-side application you develop can use a collection of JavaScript APIs to expose authorized data to client requests, for example, to be consumed by a client GUI such as a Web browser or any other HTTP client.

The functions provided by the JavaScript APIs enable server-side JavaScript applications not only to expose data but to update, insert, and delete data, too. You can use the JavaScript APIs to perform the following actions:

- Interact with the SAP HANA XS runtime environment
- Directly access SAP HANA database capabilities
- Interact with services on defined HTTP destinations.

JavaScript programs are stored in the repository along with all the other development resources. When the programs are activated, the code is stored in the repository as a runtime object.

→ Tip

To enable the Web Browser to display more helpful information if your JavaScript code causes an HTTP 500 exception on the SAP HANA XS Web server, ask someone with administrator privileges to start the SAP HANA studio's [Administration Console](#) perspective and add the parameter `developer_mode` to the `xsengine.ini` > `httpserver` section of the [Configuration](#) tab and set it to `true`.

Related Information

[Write XS Server-Side JavaScript \[page 530\]](#)

[JavaScript Security Considerations \[page 533\]](#)

9.2.1 Tutorial: Write Server-Side JavaScript Application Code

SAP HANA Extended Application Services (SAP HANA XS) supports server-side application programming in JavaScript. The server-side application you develop uses JavaScript APIs to expose authorized data to client requests, for example, for consumption by a client GUI such as a Web browser, SAPUI5 applications, or mobile clients.

Prerequisites

- Access to a running SAP HANA system.
- Access to SAP HANA studio
- Access to an SAP HANA Repository workspace

- Access to a shared project in the SAP HANA Repository where you can create the artifacts required for this tutorial.

Context

Since JavaScript programs are stored in the SAP HANA Repository, the steps in this task description assume that you have already created a workspace and a project (of type *XS Project*), and that you have shared the project with other members of the development team. To write a server-side JavaScript application, you must perform the following high-level steps.

→ Tip

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

Procedure

1. Create a root package for your application, for example, `helloxsjs`.
2. Create an application descriptor for your application and place it in the root package you created in the previous step.

The application descriptor is the core file that you use to describe an application's availability within SAP HANA XS. The application-descriptor file has no contents and no name; it only has the file extension `.xsapp`.

ⓘ Note

For backward compatibility, content is allowed in the `.xsapp` file but ignored.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new application descriptor and choose ► *New* ► *Other* ► *SAP HANA* ► *Application Development* ► *XS Application Descriptor File* ► in the context-sensitive popup menu.
- b. Save the application-descriptor file.

ⓘ 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.

- c. Activate the application-descriptor file in the repository.
Locate and right-click the new application-descriptor file in the *Project Explorer* view. In the context-sensitive pop-up menu, choose ► *Team* ► *Activate* ►.
3. Create an application-access file and place it in the package to which you want to grant access.

The application-access file does not have a name; it only has the file extension `.xsaccess`. The contents of the `.xsaccess` file must be formatted according to JavaScript Object Notation (JSON) rules and associated with the package the file belongs to. The rules defined in the `.xsaccess` file apply to the package it resides in as well as any subpackages lower in the package hierarchy.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new application-access file and choose ► *New* ► *Other* ► *SAP HANA* ► *Application Development* ► *XS Application Access File* ► in the context-sensitive popup menu.
- b. Enter the following content in the `.xsaccess` file for your new XSJS application:

```
{
  "exposed" : true,
  "authentication" : { "method": "Form" },
  "prevent_xsrp" : true
}
```

Note

These settings allows data to be exposed, require logon authentication to access the exposed data, and help protect against cross-site request-forgery (XSRF) attacks.

- c. Save and activate the application-access file in the repository.
4. Create the server-side JavaScript (XSJS) files that contain the application logic.

Server-side JavaScript files have the file suffix `.xsjs`, for example, `hello.xsjs` and contain the code that is executed when SAP HANA XS handles a URL request.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new XSJS file and choose ► *New* ► *Other* ► *SAP HANA* ► *Application Development* ► *XS JavaScript File* ► in the context-sensitive popup menu.
- b. Using the wizard, enter the following content in the `.xsjs` file for your new XSJS application:

```
$.response.contentType = "text/plain";
$.response.setBody( "Hello, world!");
```

- c. Save and activate the XSJS file in the repository.
 5. Check the layout workspace.
- Your application package structure should have a structure that looks like the following example:

```
.
└─ helloxsjs
   └─ .xsapp
      └─ .xsaccess
         └─ .xsprivileges // optional
            └─ hello.xsjs
```

6. Save and activate the changes and additions you made.
7. View the results in a Web browser.

The SAP HANA XS Web server enables you to view the results immediately after activation in the repository, for example: `http://<SAPHANA_hostname>:80<DB_Instance_Number>/helloxsjs/hello.xsjs`

9.2.1.1 JavaScript Editor

You can write server-side JavaScript using the SAP HANA studio JavaScript editor, which provides syntax validation, code highlighting and code completion.

The SAP HANA studio's JavaScript editor includes the JSLint open-source library, which helps to validate JavaScript code. The editor highlights any code that does not conform to the JSLint standards.

To configure the JSLint library and determine which validations are performed, go to: ► [Window](#) ► [Preferences](#) ► [SAP HANA](#) ► [Application Development](#) ► [JSLint](#) . In the preferences window, each JSLint setting is followed by the corresponding JSLint command name, which you can use to lookup more information on the JSLint Web site.

→ Tip

To disable all JSLint validations for files in a specific project, right-click the project and choose [Disable JSLint](#).

Related Information

<http://www.jshint.com/lint.html> ➤

9.2.1.2 Server-Side JavaScript Security Considerations

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) external attacks such as cross-site scripting and forgery, and insufficient authentication.

The following list illustrates the areas where special attention is required to avoid security-related problems when writing server-side JavaScript. Each of the problems highlighted in the list is described in detail in its own dedicated section:

- **SSL/HTTPS**
Enable secure HTTP (HTTPS) for inbound communication required by an SAP HANA application.
- **Injection flaws**
In the context of SAP HANA Extended Application Services (SAP HANA XS) injection flaws concern SQL injection that modifies the URL to expand the scope of the original request.
- **Cross-site scripting (XSS)**
Web-based vulnerability that involves an attacker injecting JavaScript into a link with the intention of running the injected code on the target computer.
- **Broken authentication and session management**
Leaks or flaws in the authentication or session management functions allow attackers to impersonate users and gain access to unauthorized systems and data.
- **Insecure direct object references**
An application lacks the proper authentication mechanism for target objects.

- Cross-site request forgery (XSRF)
Exploits the trust boundaries that exist between different Web sites running in the same web browser session.
- Incorrect security configuration
Attacks against the security configuration in place, for example, authentication mechanisms and authorization processes.
- Insecure cryptographic storage
Sensitive information such as logon credentials is not securely stored, for example, with encryption tools.
- Missing restrictions on URL Access
Sensitive information such as logon credentials is exposed.
- Insufficient transport layer protection
Network traffic can be monitored, and attackers can steal sensitive information such as logon credentials or credit-card data.
- Invalid redirects and forwards
Web applications redirect users to other pages or use internal forwards in a similar manner.
- XML processing issues
Potential security issues related to processing XML as input or to generating XML as output

Related Information

[SSL/HTTPS \[page 535\]](#)

[Injection flaws \[page 535\]](#)

[Cross-site scripting \(XSS\) \[page 537\]](#)

[Broken authentication and session management \[page 538\]](#)

[Insecure direct object references \[page 539\]](#)

[Cross-site request forgery \(XSRF\) \[page 539\]](#)

[Incorrect security configuration \[page 542\]](#)

[Insecure cryptographic storage \[page 542\]](#)

[Missing restrictions on URL Access \[page 543\]](#)

[Insufficient transport layer protection \[page 544\]](#)

[XML processing issues \[page 545\]](#)

9.2.1.2.1 Server-Side JavaScript: SSL/HTTPS

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) external attacks such as cross-site scripting and forgery, and insufficient authentication. You can set up SAP HANA to use secure HTTP (HTTPS).

SSL/HTTPS Problem

Incoming requests for data from client applications use secure HTTP (HTTPS), but the SAP HANA system is not configured to accept the HTTPS requests.

SSL/HTTPS Recommendation

Ensure the SAP Web Dispatcher is configured to accept incoming HTTPS requests. For more information, see the *SAP HANA Security Guide*.

❗ Note

The HTTPS requests are forwarded internally from the SAP Web Dispatcher to SAP HANA XS as HTTP (clear text).

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.2 Server-Side JavaScript: Injection Flaws

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) injection flaws. Typically, injection flaws concern SQL injection and involve modifying the URL to expand the scope of the original request.

The XS JavaScript API provides a number of different ways to interact with the SAP HANA database by using SQL commands. By default, these APIs allow you to read data, but they can also be used to update or delete data, and even to grant (or revoke) access rights at runtime. As a general rule, it is recommended to write a query which is either a call to an SQLScript procedure or a prepared statement where all parameters specified in the procedure or statement are escaped by using either `setString` or `setInt`, as illustrated in the examples provided in this section. Avoid using dynamic SQL commands with parameters that are not escaped.

Injection Flaws Problem

In the context of SAP HANA XS, injection flaws mostly concern SQL injection, which can occur in the SAP HANA XS JavaScript API or SQL script itself (both standard and dynamic). For example, the URL `http://xsengine/customer.xsjs?id=3` runs the code in the JavaScript file `customer.xsjs` shown below:

```
var conn = $.db.getConnection();
var pstmt = conn.prepareStatement( " SELECT * FROM accounts WHERE custID='" +
$.request.parameters.get("id"));
var rs = pstmt.executeQuery();
```

By modifying the URL, for example, to `http://xsengine/customer.xsjs?id=3 'OR 1=1'`, an attacker can view not just one account but **all** the accounts in the database.

Note

SAP HANA XS applications rely on the authorization provided by the underlying SAP HANA database.

Users accessing an SAP HANA XS based application require the appropriate privileges on the database objects to execute database queries. The SAP HANA authorization system will enforce the appropriate authorizations. This means that in those cases, even if the user can manipulate a query, he will not gain more access than is assigned to him through roles or privileges. Definer mode SQL script procedures are an exception to this rule that you need to take into consideration.

Injection Flaws Recommendation

To prevent injection flaws in the JavaScript API, use prepared statements to create a query and place-holders to fill with results of function calls to the prepared-statement object; to prevent injection flaws in standard SQL Script, use stored procedures that run in **caller** mode; in caller mode, the stored procedures are executed with the credentials of the logged-on HANA user. Avoid using dynamic SQL if possible. For example, to guard against the SQL-injection attack illustrated in the problem example, you could use the following code:

```
var conn = $.db.getConnection();
var pstmt = conn.prepareStatement( " SELECT * FROM accounts WHERE custID=?' );
pstmt.setInt(1, $.request.parameters.get("id"), 10);
var rs = pstmt.executeQuery();
```

Prepared statements enable you to create the actual query you want to run and then create several placeholders for the query parameters. The placeholders are replaced with the proper function calls to the prepared statement object. The calls are specific for each type in such a way that the SAP HANA XS JavaScript API is able to properly escape the input data. For example, to escape a string, you can use the `setString` function.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide* and the *SAP HANA SQL System Views and Reference*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.3 Server-Side JavaScript: Cross-Site Scripting

If you use server-side JavaScript to write your application code, bear in mind the potential for (and risk of) cross-site scripting (XSS) attacks. Cross-site scripting is a Web-based vulnerability that involves an attacker injecting JavaScript into a link with the intention of running the injected code on the target computer.

Cross-Site Scripting Problem

The vulnerability to cross-site scripting attacks comes in the following forms:

- Reflected (non-persistent)
Code affects individual users in their local Web browser
- Stored (persistent)
Code is stored on a server and affects all users who visit the served page

A successful cross-site scripting attack could result in a user obtaining elevated privileges or access to information that should not be exposed.

Cross-Site Scripting Recommendation

Since there are currently no libraries provided by the standard SAP HANA XS JavaScript API to provide proper escaping, we recommend not to write custom interfaces but to rely on well-tested technologies supplied by SAP, for example, OData or JSON together with SAPUI5 libraries.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.4 Server-Side JavaScript: Broken Authentication

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) attack against authentication infrastructure. Leaks or flaws in the authentication or session management functions allow attackers to impersonate users and gain access to unauthorized systems and data.

Authentication Problem

Leaks or flaws in the authentication or session management functions allow attackers to impersonate users; the attackers can be external as well as users with their own accounts to obtain the privileges of those users they impersonate.

Authentication Recommendation

Use the built-in SAP HANA XS authentication mechanism and session management (cookies). For example, use the "authentication" keyword to enable an authentication method and set it according to the authentication method you want implement, for example: SAP logon ticket, form-based, or basic (user name and password) in the application's .xsaccess file, which ensures that all objects in the application path are available only to authenticated users.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.5 Server-Side JavaScript: Insecure Object Reference

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) attacks using insecure references to objects.

Object Reference Problem

An SAP HANA XS application is vulnerable to insecure direct object reference if the application lacks the proper authentication mechanism for target objects.

Object Reference Recommendation

Make sure that only authenticated users are allowed to access a particular object. In the context of SAP HANA XS, use the "authentication" keyword to enable an authentication method and set it according to the authentication method you implement, for example: SAP logon ticket, form-based, or basic (user name and password) in the application's `.xsaccess` file, which ensures that all objects in the application path are available only to authenticated users.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.6 Server-Side JavaScript: Cross-Site Request Forgery

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) cross-site request forgery (XSRF). Cross-site scripting is a web-based vulnerability

that exploits the trust boundaries that exist between different websites running in the same web browser session.

Cross-Site Request-Forgery Problem

Since there are no clear trust boundaries between different Web sites running in the same Web-browser session, an attacker can trick users (for example, by luring them to a popular Web site that is under the attacker's control) into clicking a specific hyperlink. The hyperlink displays a Web site that performs actions on the visitor's behalf, for example, in a hidden iframe. If the targeted end user is logged in and browsing using an account with elevated privileges, the XSRF attack can compromise the entire Web application.

Cross-Site Request-Forgery Recommendation

SAP HANA XS provides a way to include a random token in the POST submission which is validated on the server-side. Only if this token is non-predictable for attackers can one prevent cross-site, request-forgery attacks. The easiest way to prevent cross-site, request-forgery attacks is by using the standard SAP HANA XS cookie. This cookie is randomly and securely generated and provides a good random token which is unpredictable by an attacker (`$.session.getSecurityToken()`).

To protect SAP HANA XS applications from cross-site request-forgery (XSRF) attacks, make sure you always set the `prevent_xsrp` keyword in the application-access (`.xsaccess`) file to `true`, as illustrated in the following example:

```
{
  "prevent_xsrp" : true
}
```

The `prevent_xsrp` keyword prevents the XSRF attacks by ensuring that checks are performed to establish that a valid security token is available for given Browser session. The existence of a valid security token determines if an application responds to the client's request to display content. A security token is considered to be valid if it matches the token that SAP HANA XS generates in the backend for the corresponding session.

Note

The default setting is false, which means there is no automatic prevention of XSRF attacks. If no value is assigned to the `prevent_xsrp` keyword, the default setting (false) applies.

The following client-side JavaScript code snippet show how to use the HTTP request header to fetch, check, and apply the XSRF security token required to protect against XSRF attacks.

```
<html>
<head>
  <title>Example</title>
  <script id="sap-ui-bootstrap" type="text/javascript"
    src="/sap/ui5/1/resources/sap-ui-core.js"
    data-sap-ui-language="en"
    data-sap-ui-theme="sap_goldreflection"
    data-sap-ui-libs="sap.ui.core,sap.ui.commons,sap.ui.ux3,sap.ui.table">
  </script>
```

```

<script type="text/javascript" src="/sap/ui5/1/resources/jquery-sap.js"></script>
<script>
    function doSomething() {
        $.ajax({
            url: "logic.xsjs",
            type: "GET",
            beforeSend: function(xhr) {
                xhr.setRequestHeader("X-CSRF-Token", "Fetch");
            },
            success: function(data, textStatus, XMLHttpRequest) {
                var token = XMLHttpRequest.getResponseHeader('X-CSRF-Token');
                var data = "somePayload";
                $.ajax({
                    url: "logic.xsjs",
                    type: "POST",
                    data: data,
                    beforeSend: function(xhr) {
                        xhr.setRequestHeader("X-CSRF-Token", token);
                    },
                    success: function() {
                        alert("works");
                    },
                    error: function() {
                        alert("works not");
                    }
                });
            }
        });
    }
</script>
</head>
<body>
    <div>
        <a href="#" onClick="doSomething();">Do something</a>
    </div>
</body>
</html>

```

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.7 Server-Side JavaScript: Security Misconfiguration

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) attacks against the security configuration in place, for example, authentication mechanisms and authorization processes.

Insecure Configuration Problem

No or an inadequate authentication mechanism has been implemented.

Insecure Configuration Recommendation

Applications should have proper authentication in place, for example, by using SAP HANA built-in authentication mechanisms and, in addition, the SAP HANA XS cookie and session handling features. Application developers must also consider and control which paths are exposed by HTTP to the outside world and which of these paths require authentication.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.8 Server-Side JavaScript: Insecure Storage

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) attacks against the insecure or lack of encryption of data assets.

Storage-Encryption Problem

Sensitive information such as logon credentials is exposed.

Storage-Encryption Recommendation

To prevent unauthorized access, for example, in the event of a system break-in, data such as user logon credentials must be stored in an encrypted state.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.9 Server-Side JavaScript: Missing URL Restrictions

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) unauthorized access to URLs.

URL Access Problem

Unauthenticated users have access to URLs that expose confidential (unauthorized) data.

URL Access Recommendation

Make sure you have addressed the issues described in "Broken Authentication and Session Management" and "Insecure Direct Object References". In addition, check if a user is allowed to access a specific URL before actually executing the code behind that requested URL. Consider putting an authentication check in place for each JavaScript file before continuing to send any data back to the client's Web browser.

→ Tip

For more information about Security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.10 Server-Side JavaScript: Transport Layer Protection

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) insufficient protection of the transport layer.

Transport Layer Protection Problem

Without transport-layer protection, the user's network traffic can be monitored, and attackers can steal sensitive information such as logon credentials or credit-card data.

Transport Layer Protection Recommendation

Turn on transport-layer protection in SAP HANA XS; the procedure is described in the SAP HANA security guide.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.11 Server-Side JavaScript: Invalid Redirection

If you use server-side JavaScript to write your application code, bear in mind the potential for (and risk of) redirection and internal forwarding from the requested Web page.

Invalid Redirection Problem

Web applications frequently redirect users to other pages or use internal forwards in a similar manner. Sometimes the target page is specified in an invalid (not permitted) parameter. This enables an attacker to choose a destination page leading to the possibility of phishing attacks or the spamming of search engines.

Invalid Redirection Recommendation

To prevent invalidated redirects or forwards, application developers should validate the requested destination before forwarding, for example, by checking if the destination is present in an allow list. If the URL specified in the redirection request is not present in the list of allowed destinations, the redirection is refused.

→ Tip

Avoid using redirection if you cannot control the final destination.

Alternatively, you can refuse to allow any direct user input; instead, the input can be used to determine the final destination for the redirection, as illustrated in the following example:

```
var destination = $.request.parameters.get("dest");
switch (destination) {
  case "1": $.response.headers.set("location", "http://
FirstAllowlistedURL.com"); break;
  case "2": $.response.headers.set("location", "http://
SecondAllowlistedURL.com"); break;
  default: $.response.headers.set("location", "http://
DefaultAllowlistedURL.com");
}
```

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.2.1.2.12 Server-Side JavaScript: XML Processing Issues

If you choose to use server-side JavaScript to write your application code, you need to bear in mind the potential for (and risk of) attacks aimed at the process used to parse XML input and generate the XML output.

XML Processing Problem

There are several potential security issues related to processing XML as input or to generating XML as output. In addition, problems with related technologies (for example, XSL Transformations or XSLT) can enable the inclusion of other (unwanted) files.

XML Processing Recommendation

Turn on transport-layer protection in SAP HANA XS; the procedure is described in the SAP HANA security guide.

Bear in mind the following rules and suggestions when processing or generating XML output:

- When processing XML that originates from an untrusted source, disable DTD processing and entity expansion unless strictly required. This helps prevent Billion Laugh Attacks (Cross-Site Request Forgery), which can bring down the processing code and, depending on the configuration of the machine, an entire server.
- To prevent the inclusion (insertion) of unwanted and unauthorized files, restrict the ability to open files or URLs even in requests included in XML input that comes from a trusted source. In this way, you prevent the disclosure of internal file paths and internal machines.
- Ensure proper limits are in place on the maximum amount of memory that the XML processing engine can use, the amount of nested entities that the XML code can have, and the maximum length of entity names, attribute names, and so on. This practice helps prevent the triggering of potential issues.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

[SAP HANA Security Guide for SAP HANA Platform](#)

9.3 Using Server-Side JavaScript Libraries

The elements defined in normal server-side JavaScript programs cannot be accessed from other JavaScript programs. To enable the reuse of program elements, SAP HANA Extended Application Services support server-side JavaScript libraries.

Server-side JavaScript libraries are a special type of JavaScript program that can be imported and called in other JavaScript programs. You can use JavaScript libraries to perform simple, repetitive tasks, for example, to handle forms and form data, to manipulate date and time strings, to parse URLs, and so on.

ⓘ Note

JavaScript libraries are internally developed extensions for SAP HANA.

The following example shows how to import a JavaScript mathematics library using the import function:

```
// import math lib
$.import("sap.myapp.lib","math");
// use math lib
var max_res = $.sap.myapp.lib.math.max(3, 7);
```

The import function requires the following parameters:

- **Package name**
Full name of the package containing the library object you want to import, for example, `sap.myapp.lib`
- **Library name**
Name of the library object you want to import, for example, `math`

Note

Restrictions apply to the characters you can use in the names of JavaScript libraries and application packages. Permitted characters are: upper- and lower-case letters (Aa-Zz), digits 0-9, and the dollar sign (\$).

The standard JavaScript limitations apply to the characters you can use in either the name of the XSJS library you create or the name of the package where the library is deployed. For example, you cannot use the hyphen (-) in the name of an XSJS library or, if you are referencing the library, the name of a package in the application package path. To prevent problems with activation of the object in the SAP HANA repository, you must follow the standard rules for accessing JavaScript property objects by name. The following example, shows how to use square brackets and quotes (`[ "<STRING>" ]`) to access an object whose name uses non-permitted characters such as a hyphen (-):

```
// import math lib
$.import("sap.myapp.lib.XS-QGP-SPS7", "math");
// use math lib
var max_res = $.sap.myapp.lib["XS-QGP-SPS7"].math.max(3, 7);
```

Related Information

[Import Server-Side JavaScript Libraries \[page 547\]](#)

[Write Server-Side JavaScript Libraries \[page 549\]](#)

9.3.1 Import Server-Side JavaScript Libraries

Server-side JavaScript libraries are a special type of JavaScript program that can be imported and called in other JavaScript programs. You can use JavaScript libraries to perform simple, repetitive tasks, for example: handle forms and form data, manipulate date and time strings, parse URLs, and so on.

Context

JavaScript libraries are internally developed extensions for SAP HANA. The libraries exist in the context of a package, which is referenced when you import the library. The following example of a JavaScript library displays the word "Hello" along with a name and an exclamation mark as a suffix.

```
var greetingPrefix = "Hello, ";
var greetingSuffix = "!";
```

```
function greet (name) {  
    return greetingPrefix + name + greetingSuffix;  
}
```

Note

This procedure uses the illustrated example JavaScript library to explain what happens when you import a JavaScript library, for example, which objects are created, when, and where. If you have your own library to import, substitute the library names and paths shown in the steps below as required.

To import a JavaScript library for use in your server-side JavaScript application, perform the following tasks

Procedure

1. Import the JavaScript library into a JavaScript application.

Open the server-side JavaScript file into which you want to import the JavaScript library.

Use the `$.import` function, as follows:

```
$.import("<path.to.your.library.filename>", "greetLib");  
var greeting = $.<path.to.your.library.filename>.greet("World");  
$.response.setBody(greeting);
```

2. Save and activate the changes to the JavaScript file.

Although the operation is simple, bear in mind the following points:

- Additional objects in the package hierarchy
The import operation generates a hierarchy of objects below `$` that resemble the library's location in the repository, for example, for the library `path/to/your/library/greetLib.xsjslib`, you would see the following additional object:

```
$.path.to.your.library.greetLib
```

- Additional properties for the newly generated library object:

```
$.path.to.your.library.greetLib.greet()  
$.path.to.your.library.greetLib.greetingSuffix  
$.path.to.your.library.greetLib.greetingPrefix
```

- Pre-import checks:
 - It is not possible to import the referenced library if the import operation would override any predefined runtime objects.
 - Do not import the referenced library if it is already present in the package.
- Library context
Imported libraries exist in the context defined by their repository location.

9.3.2 Write Server-Side JavaScript Libraries

Server-side JavaScript libraries are a special type of JavaScript program that can be imported and called in other JavaScript programs. You can use JavaScript libraries to perform simple, repetitive tasks, for example, to handle forms and form data, to manipulate date and time strings, to parse URLs, and so on.

Context

JavaScript libraries are internally developed extensions for SAP HANA. However, you can write your own libraries, too. JavaScript libraries exist in the context of a package, which is referenced when you import the library. To write a JavaScript library to use in your server-side JavaScript application, perform the following steps:

Procedure

1. Create the file that contains the JavaScript library you want to add to the package and make available for import.

In SAP HANA XS, server-side JavaScript libraries have the file extension `.xsjslib`, for example `greetLib.xsjslib`.

- a. In the *Project Explorer* view, right-click the folder where you want to create the new XSJS file and choose **New > Other > SAP HANA > Application Development > XS JavaScript Library File** in the context-sensitive popup menu.
- b. Type a name for the new XS JavaScript library file, for example `greetLib` and choose *Finish*.

If you are using SAP HANA studio to create artifacts in the SAP HANA Repository, the file creation wizard adds a separator (`.`) and the required file extension automatically, for example, `.xsjslib`.

- c. Enter the following content in the `greetLib.xsjslib` XSJS library file for your new XSJS application. The following example creates a simple library that displays the word "Hello" along with a supplied name and adds an exclamation point (!) as a suffix.

```
var greetingPrefix = "Hello, ";
var greetingSuffix = "!";
function greet (name) {
    return greetingPrefix + name + greetingSuffix;
}
```

2. Save the new JavaScript library.

It is important to remember where the JavaScript library is located; you have to reference the package path when you import the library.

3. Activate your new library in the repository so that it is available for import by other JavaScript applications.

9.4 Using the Server-Side JavaScript APIs

SAP HANA Extended Application Services (SAP HANA XS) provides a set of server-side JavaScript application programming interfaces (API) that enable you to configure your applications to interact with SAP HANA.

The *SAP HANA XS JavaScript Reference* lists all the functions that are available for use when programming interaction between your application and SAP HANA. For example, you can use the database API to invoke SQL statements from inside your application, or access details of the current HTTP request for SAP HANA data with the request-processing API. SAP HANA XS includes the following set of server-side JavaScript APIs:

XS JavaScript Application Programming Interfaces

API	Description
Database	Enables access to the SAP HANA by means of SQL statements. For example, you can open a connection to commit or rollback changes in SAP HANA, to prepare stored procedures (or SQL statements) for execution or to return details of a result set or a result set's metadata.
Outbound connectivity	Enables outbound access to a defined HTTP destination that provides services which an application can use. For example, you can read the connection details for an HTTP destination, request data, and set details of the response body. You can also set up an SMTP connection for use by outgoing multipart e-mails.
Request processing	Enables access to the context of the current HTTP request, for example, for read requests and write responses. You can use the functions provided by this API to manipulate the content of the request and the response.
Session	Enables access to the SAP HANA XS session, for example, to determine the language used in the session or if a user has the privileges required to run an application.
Job Schedule	Enables access to the job-scheduling interface which allows you to define and trigger recurring tasks that run in the background. The XS jobs API allows you to add and remove schedules from jobs.
Security	Enables access to the <code>\$.security.crypto</code> namespace and the classes <code>AntiVirus</code> and <code>Store</code> , which provide tools that allow you to configure a secure store, set up anti-virus scans, and generate hashes..
Trace	Enables access to the various trace levels you can use to generate and log information about application activity. You can view trace files in the <i>diagnosis Files</i> tab of the SAP HANA studio's <i>Administration</i> perspective.
Utilities	Enables access to utilities that you can use to parse XML and manipulate Zip archives, for example, to zip and unzip files, add and remove entries from Zip archives, and encrypt Zip archives with password protection.
XS Data Services	Provides access to a library of JavaScript utilities, which can be used to enable server-side JavaScript applications to consume data models that are defined using Core Data Services.
XS Procedures	Provides access to a library of JavaScript utilities, which can be used to enable server-side JavaScript applications to call SAP HANA stored procedures as if the procedures were JavaScript functions.

▲ Restriction

`XSPROC` is intended only for use with database connections made with the old `$.db` API. It is not recommended to use `XSPROC` with `$.hdb` connections. For `$.hdb` connections, use `$hdb.loadProcedure` instead.

Database API

The SAP HANA XS Database API (`$.hdb`) provides tools that enable simple and convenient access to the database.

⚠ Caution

The `$.hdb` namespace is intended as a replacement for the older `$.db` namespace. Since different database connections are used for the `$.hdb` and `$.db` APIs, avoid using both APIs in a single http-request, for example, to update the same tables as this can lead to problems, including deadlocks.

You can use the Database API for the following operations

- `$.hdb.Connection`
Establish a connection to the SAP HANA database
- `$.hdb.ProcedureResult`
Represents the result of a stored procedure call to the SAP HANA database
- `$.hdb.ResultSet`
Represents the result of a database query

The following example shows how to use the database API to connect to the SAP HANA database, commit some changes, and end the current transaction.

📌 Note

By default, auto-commit mode is disabled, which means that all database changes must be explicitly committed.

```
var connection = $.hdb.getConnection();
connection.executeUpdate('UPDATE "DB_EXAMPLE"."ICECREAM" SET QUANTITY=? WHERE
FLAVOR=?', 9, 'CHOCOLATE');
connection.commit();
```

The following example of usage of the SAP HANA XS database API shows how to establish a connection with SAP HANA and return a result set from the specified procedure call. The example code assumes that a procedure exists with the following signature:

```
PROCEDURE 'DB_EXAMPLE'.icecream.shop::sell(
  IN flavor VARCHAR,
  IN quantity INTEGER,
  IN payment DECIMAL,
  OUT change DECIMAL)
```

Note that the result can be accessed as if it were a JSON object with a structure similar to the following example: `{change: 1.50, $resultSets:[...]}` .

→ Tip

`$resultSets` is not enumerable; it does not show up in a `for-each` loop.

```
var fnSell = connection.loadProcedure('DB_EXAMPLE', 'icecream.shop::sell');
var result = fnSell('CHOCOLATE', 3, 30.0);
// value of output parameter 'change'
var change = result['change'];
// array of $.hdb.ResultSet returned by the stored procedure
```



```

var resultSets = result['$resultSets'];
// iterate over all output parameters.
var params;
for (var outputParam in result) {
    params += outputParam + ' ';
}

```

Outbound API

The Outbound API (\$.net) provides tools that you can use to perform the following actions:

- `$.net.SMTPConnection`
For sending `$.net.Mail` objects by means of an SMTP connection
- `$.net.Mail`
For constructing and sending multipart e-mails
- `$.net.http`
HTTP(s) client (and request) classes for outbound connectivity and an HTTP(s) destination class that hold metadata, for example: host, port, useSSL.

The following example shows how to use the `$.net.SMTPConnection` class to send e-mail objects (`$.net.Mail`) by means of an SMTP connection object:

```

subscribers = ["kofi@sap.com", "kwaku@sap.com"];
smtpConnection = new SMTPConnection();
var mail = new $.net.Mail({ sender: "manager@sap.com",
    subject: "Promotion Notice",
    subjectEncoding: "UTF-8",
    parts: [new $.net.Mail.Part({
        type: $.net.Mail.Part.TYPE_TEXT,
        contentType: "text/html",
        encoding: "UTF-8"
    })]
});
for (var i = 0; i < subscribers.length; ++i) {
    mail.to = subscribers[i];
    mail.parts[0].text = "Dear " + subscribers[i].split("@")[0] + ", \
        you have been promoted. Congratulations!";
    smtpConnection.send(mail);
}
smtpConnection.close();

```

The following example shows how to use the `$.net.Mail` class to create an e-mail from an XS JavaScript object and send it to the named recipients:

Note

If mandatory information is missing or an error occurs during the send operation, the `mail.send()` call fails and returns an error.

```

var mail = new $.net.Mail({
    sender: {address: "sender@sap.com"},
    to: [{ name: "John Doe", address: "john.doe@sap.com", nameEncoding: "US-ASCII"}], \
        {name: "Jane Doe", address: "jane.doe@sap.com"}],
    cc: ["cc1@sap.com", {address: "cc2@sap.com"}],
    bcc: [{ name: "Jonnie Doe", address: "jonnie.doe@sap.com"}],
    subject: "subject",

```

```

        subjectEncoding: "UTF-8",
        parts: [ new $.net.Mail.Part({
            type: $.net.Mail.Part.TYPE_TEXT,
            text: "The body of the mail.",
            contentType: "text/plain",
            encoding: "UTF-8",
        })]
    });
    var returnValue = mail.send();
    var response = "MessageId = " + returnValue.messageId + ", final reply = " +
    returnValue.finalReply;
    $.response.status = $.net.http.OK;
    $.response.contentType = "text/html";
    $.response.setBody(response);

```

The following example of server-side JavaScript shows how to use the outbound API to get (read) an HTTP destination. You can also set the contents of the response, for example, to include details of the header, body, and any cookies. For HTTPs connections you need to maintain a certificate (CA or explicit server certificate) in a Trust Store; you use the certificate to check the connection against.

```

var dest = $.net.http.readDestination("inject", "ipsec");
var client = new $.net.http.Client();
var req = new $.web.WebRequest($.net.http.GET, "");
client.request(req, dest);
var response = client.getResponse();
var co = [], he = [];
for(var c in response.cookies) {
    co.push(response.cookies[c]);
}
for(var c in response.headers) {
    he.push(response.headers[c]);
}
var body = undefined;
if(response.body)
    var body = response.body.asString();
$.response.contentType = "application/json";

```

→ Tip

You define the HTTP destination in a text file using `keyword=value` pairs. You must activate the HTTP destination in the SAP HANA repository. After activation, you can view details of the HTTP destination in the SAP HANA XS Administration tool.

Request-Processing API

The Request-Processing API (\$.web) provides access to the body of HTTP request and response entities. For example, you can use the following classes:

- **\$.web.Body**
Represents the body of an HTTP request entity and provides access to the data included in the body of the HTTP request entity
- **\$.web.EntityList**
Represents a list of request or response entities; the EntityList holds WebEntityRequest or WebEntityResponse objects.
- **\$.web.TupleList**

Represents a list of name-value pairs. The `TupleList` is a container that provides tuples for cookies, headers, and parameters. A “tuple” is a JavaScript object with the properties “name” and “value”.

- `$.web.WebRequest`
Enables access to the client HTTP request currently being processed
- `$.web.WebResponse`
Enables access to the client HTTP response currently being processed for the corresponding request object (
- `$.web.WebEntityRequest`
Represents an HTTP request entity and provides access to the entity's metadata and (body) content.
- `$.web.WebEntityResponse`
Represents the HTTP response currently being populated

The following example shows how to use the request-processing API to display the message “Hello World” in a browser.

```
$.response.contentType = "text/plain";
$.response.setBody( "Hello, World !");
```

In the following example, you can see how to use the request-processing API to get the value of parameters describing the name and vendor ID of a delivery unit (DU) and return the result set in JSON-compliant form.

```
var duName = $.request.parameters.get("du_name");
var duVendor = $.request.parameters.get("du_vendor");
result = {
    content_id : contentId.toString()
};
$.response.status = $.net.http.OK;
$.response.contentType = 'application/json';
$.response.setBody(JSON.stringify(result));
```

In the following example of use of the request-processing API, we show how to access to the request's meta data (and body) and, in addition, how to set and send the response.

```
if($.request.method === $.net.http.GET) {
    // get query parameter named id
    var qpId = $.request.parameters.get("id");

    // handle request for the given id parameter...
    var result = handleRequest(qpId);

    // send response
    $.response.contentType = "plain/test";
    $.response.setBody("result: " + result);
    $.response.status = $.net.http.OK;
} else {
    // unsupported method
    $.response.status = $.net.http.INTERNAL_SERVER_ERROR;
}
```

Session API

Enables access to the SAP HANA XS session, for example, to determine the language used in the session or check if a user has the privileges required to run an application.

You can use the XS JavaScript `$.session` API to request and check information about the currently open sessions. For example, you can find out the name of a user who is currently logged on to the database or get the session-specific security token. The `$.session` API also enables you to check if a user has sufficient privileges to call an application. The following example checks if the user has the `execute` privilege that is required to run an application. If the check reveals that the user does not have the required privilege, an error message is generated indicating the name of the missing privilege.

```
if (!$.session.hasAppPrivilege("sap.xse.test::Execute")) {  
    $.response.setBody("Privilege sap.xse.test::Execute is missing");  
    $.response.status = $.net.http.INTERNAL_SERVER_ERROR;  
}
```

Job Schedule API

In SAP HANA XS, a scheduled job is created by means of an `.xsjob` file, a design-time file you commit to (and activate in) the SAP HANA repository. The `.xsjob` file can be used to define recurring tasks that run in the background; the Job Schedule API allows developers to add and remove schedules from such jobs.

The Job Schedule API provides the following tools:

- **Job**
`$.jobs.Job` represents a scheduled XS job
- **JobLog**
`$.jobs.JobLog` provide access to the log entries of a scheduled job
- **JobSchedules**
`$.jobs.JobSchedules` enables control of an XS job's schedules.

Note

It is not possible to call the `$.request` and `$.response` objects as part of an XS job.

The XS jobs API `$.jobs.Job` enables you to add schedules to (and remove schedules from) jobs defined in an `.xsjob` file.

The following example of server-side JavaScript shows how to use the Job Schedule API to add a schedule to a existing job and delete a schedule from an existing job.

```
var myjob = new $.jobs.Job({uri:"myJob.xsjob", sqlcc:"sqlcc/otheruser.xssqlcc"});  
// add schedule to a job  
var id = myjob.schedules.add({  
    description: "Added at runtime, run every 10 minutes",  
    xscron: "* * * * * /10 0",  
    parameter: {  
        a: "c"  
    }  
});  
// delete a schedule from a job  
myjob.schedules.delete({id: id});
```

If the XS job file referred to in the URI is not in the same package as the XS JavaScript or SQLScript function being called, you must add the full package path to the XS job file specified in the URI illustrated in line 1 of the example above, for example, `</path/to/package.>MyXSjob.xsjob`.

Note

The path specified in `</path/to/package.>` can be either absolute or relative.

In addition, the SQL connection defined in `sqlcc/otheruser.xssqlcc` is used to modify the job; it is not used to execute the job specified in `myJob.xsjob`.

To understand the cron-like syntax required by the `xscron` job scheduler, use the following examples:

- `2013 * * fri 12 0 0`
Run the job every Friday in 2013 at 12:00.
- `* * 3:-2 * 12:14 0 0`
Run every hour between 12:00 and 14:00 every day between the third and second-to-last day of the month.
- `* * * -1.sun 9 0 0`
Run the job on the last Sunday of every month at 09:00.

Security API

The SAP HANA XS JavaScript security API `$.security` includes the `$.security.crypto` namespace and the following classes:

- `$.security.AntiVirus`
Scan data with a **supported** external anti-virus engine
- `$.security.Store`
Store data securely in name-value form

The `$.security.crypto` namespace includes methods (for example, `md5()`, `sha1()`, and `sha256()`) that enable you to compute an MD5 or SHA1/256 hash (or HMAC-MD5, HMAC-SHA1, and HMAC-SHA256).

The `AntiVirus` class includes a method `scan()` that enables you to set up a scan instance using one of the supported anti-virus engines. The `Store` class enables you to set up a secure store for an SAP HANA XS application; the secure store can be used to store sensitive information either at the application level (`store()`) or per user (`storeForUser()`).

The following code example shows how to use the SAP HANA XS virus-scan interface (VSI) to scan a specific object type: a Microsoft Word document.

Note

For more information about which antivirus engines SAP HANA supports, see *SAP Note 786179*.

```
var data = //Some data to be checked
var av = new $.security.AntiVirus();
//AV scan data as Word document
av.scan(data, "myDocument.docx");
```

The following code example shows how to set up a simple scan for data uploads using the SAP HANA XS virus-scan interface.

```
//scan a buffer with own "upload" profile
var av = new $.security.AntiVirus("upload");
```

```
av.scan(buffer);
```

The SAP HANA XS \$.security.Store API can be used to store data safely and securely in name-value form. The security API enables you to define a secure store (in a design-time artifact) for each application and refer to this design time object in the application coding.

Note

The design-time secure store is a file with the file extension ".xssecurestore", for example, `localStore.xssecurestore`; the secure-store file must include only the following mandatory content: `{}`.

SAP HANA XS looks after the encryption and decryption of data and also ensures the persistency of the data. For the stored data, you can choose between the following visibility options:

- Application-wide data visibility
Use `store(<parameters>)` to ensure that all users of the corresponding application have access to one secure store where they can share the same data and can decrypt or encrypt data, for example, passwords for a remote system.
- Application-wide data visibility but with user-specific stores separation
Use `storeForUser(<parameters>)` to ensure that **each** user of the corresponding application has a separate container to securely store personal, encrypted data, for example, credit card numbers or personal-information-number (PIN) codes; the encrypted data can only be decrypted by the owner of the secure store; the user who encrypted it.

```
function store() {  
  var config = {  
    name: "foo",  
    value: "bar"  
  };  
  var aStore = new $.security.Store("localStore.xssecurestore");  
  aStore.store(config);  
}  
function read() {  
  var config = {  
    name: "foo"  
  };  
  try {  
    var store = new $.security.Store("localStore.xssecurestore");  
    var value = store.read(config);  
  }  
  catch(ex) {  
    //do some error handling  
  }  
}
```

Trace API

Enables access to the various trace levels you can use to generate and log information about application activity. The specified error message is written to the appropriate trace file.

```
$.trace.error("This is an error message")
```

You can set the following trace levels:

- `$.trace.debug(message)`
Writes the string defined in (message) to the application trace with **debug** level
- `$.trace.error(message)`
Writes the string defined in (message) to the application trace with **error** level
- `$.trace.fatal(message)`
Writes the string defined in (message) to the application trace with **fatal** level
- `$.trace.info(message)`
Writes the string defined in (message) to the application trace with **info** level
- `$.trace.warning(message)`
Writes the string defined in (message) to the application trace with **warning** level

Note

If tracing is enable, messages generated by the `$.trace` API are logged in the SAP HANA trace file `xsengine_<host>_<Instance>_<#>.trc` on the SAP HANA server, for example, in `<installation_path>/<SID>/HDB<nn>/<hostname>/trace`. Trace messages with severity status “warning”, “error” and “fatal” are also written to a similarly named alert file, for example, `xsengine_alert_<host>.trc`.

Utilities API

The SAP HANA XS JavaScript Utilities API includes the `$.util` namespace, which contains the following classes

- `$.util.SAXParser`
Tools for parsing XML content (for example, strings, array buffers, and the content of Web response body objects)
- `$.util.Zip`
Compression tools for building, modifying, extracting, and encrypting archives

With the XS JavaScript Utilities API `$.util.SAXParser` class, you can create a new parser object and parse the XML content of an XMLstring, an XML array buffer, or a `$.web.Body` object. The following example shows how to use the XML parsing capabilities of the `$.util.SAXParser` class:

Note

You can **stop**, **reset**, and **resume** a parsing operation. If the content to be parsed does not contain XML, the parser throws an error.

```
var parser = new $.util.SAXParser();
var xml = "<?xml version='1.0' encoding='UTF-8' standalone='yes'?'>\n\
  <!-- this is a note -->\n\
  <note noteName='NoteName'>\n\
    <to>To</to>\n\
    <from>From</from>\n\
    <heading>Note heading</heading>\n\
    <body>Note body</body>\n\
  </note>\n";
var startElementHandlerConcat = "";
var endElementHandlerConcat = "";
var characterDataHandlerConcat = "";
```

```

parser.startElementHandler = function(name, atts) {
    startElementHandlerConcat += name;
    if (name === "note") {
        startElementHandlerConcat += " noteName = '" + atts.noteName + "'";
    }
    startElementHandlerConcat += "\n";
};
parser.endElementHandler = function(name) {
    endElementHandlerConcat += name + "\n";
};
parser.characterDataHandler = function(s) {
    characterDataHandlerConcat += s;
};
parser.parse(xml);
...

```

The following code snippet shows how to use the `$.util.SAXParser` tools to parse the content of a `$.web.Body` object.

```

var body = $.request.body
var parser = new $.util.SAXParser()
//... set handlers
parser.parse(body);

```

The following encodings are supported:

- UTF-8 (**default**)
- UTF-16
- US-ASCII

The SAP HANA XS JavaScript Utilities API also includes the `$.util.Zip` tool, which enables you to perform a series of actions on Zip archives, for example:

- Compress files into (zip) and extract files from (unzip) a Zip archive
- Add new entries to, update existing entries in, and remove entries from a Zip archive
- Encrypt Zip archives with password protection

The following code illustrates a simple usage of the Zip tool:

```

var zip = new $.util.Zip("myPassword");
zip["entry.txt"] = "Two fish are in a tank. One turns to the other and asks 'How do you drive this thing?'";
$.response.status = $.net.http.OK;
$.response.contentType = "application/zip";
$.response.headers.set("Content-Disposition", "attachment; filename = Encrypted.zip");
$.response.setBody(zip.asArrayBuffer());

```

The following code snippets show how to use the `$.util.Zip` tools to work with Zip file content, for example, by adding, updating, extracting, and deleting entries. When modeling folder hierarchies, the Zip object behaves like an associative array; the entry names are the [keys](#) (the full paths to the indicated files). In the following example, we add an entry to a Zip file:

ⓘ Note

“`zip["entry1"]`” is equivalent to “`zip.entry1`”.

```

var zip = new $.util.Zip();
zip["entry1"] = "old entry";

```


In the following example, we **update** an entry in a Zip file:

```
var zip = new $.util.Zip();
zip["entry1"] = "new entry";
```

In the following example, we **extract** an entry from a Zip file: if the entry does not exist, this returns undefined.

```
var zip = new $.util.Zip();
var content = zip["entry1"];
```

In the following example, we **delete** an entry from a Zip file: if the entry does not exist, nothing happens.

```
var zip = new $.util.Zip();
delete zip["entry1"];
```

Note

There is a restriction on the amount of uncompressed data that can be extracted from a Zip archive using the XS JS utilities API.

When using the XS JS utilities API to extract data from a Zip archive, the maximum amount of uncompressed data allowed during the extraction process is defined with the parameter `max_uncompressed_size_in_bytes`, which you can set in the `zip` section of the `xsengine.ini` configuration file for a given SAP HANA system. If the `zip` section does not already exist, you must create it and add the parameter to it, for example, using the [SAP HANA Administration Console](#) in SAP HANA studio. If the parameter `max_uncompressed_size_in_bytes` is **not** set, a default value is assumed. The default value is the value assigned to the property `max_runtime_bytes` in section `jsvm` section of the `xsengine.ini` file.

You can deactivate the global check on the amount of uncompressed data. If the global system parameter `max_uncompressed_size_in_bytes` is set to **-1**, no check is performed on the amount of uncompressed data generated by an extraction process using the Utilities API, unless there is a specific user limitation in the XS JavaScript code, for example, with the `maxUncompressedSizeInBytes` parameter.

With the `$.util.Zip` class or the `$.util.compression` namespace, you can use the property `maxUncompressedSizeInBytes` to override the global setting and reduce the amount of uncompressed data allowed.

Note

Note that the parameter `max_uncompressed_size_in_bytes` cannot be used to increase the amount of uncompressed data allowed beyond the value specified in the global setting.

XS Data Services API

SAP HANA XS Data Services (XSDS) is a collection of tools that includes a native client for Core Data Services (CDS) and a query builder for SAP HANA Extended Application Services (SAP HANA XS) JavaScript. The XSDS API provides a high-level abstraction of the database API (`$.db`, `$.hdb`) and gives access to SAP HANA artifacts such as CDS entities or stored procedures. XSDS enables server-side JavaScript applications to consume data models that are defined using Core Data Services more efficiently.

The following example shows how to import a CDS entity and how to update a given entity instance in XSDS **managed** mode.

```
// import CDS client library
var XSDS = $.import("sap.hana.xs.libs.dbutils", "xsds");
// import CDS entity
var MyEntity = XSDS.$importEntity("cds.namespace", "cds_context.cds_entity");
// retrieve entity instance
var instance = MyEntity.$get({ id: 69 });
// update instance
instance.stringProp = "new value";
instance.intProp++;
instance.assocProp.dateProp = new Date();
// persist changes
instance.$save();
```

The following example shows how to query the database using CDS model data in XSDS **unmanaged** mode.

```
// import CDS client library
var XSDS = $.import("sap.hana.xs.libs.dbutils", "xsds");
// import CDS entity
var MyEntity = XSDS.$importEntity("cds.namespace", "cds_context.cds_entity");
// build query
var query = MyEntity.$query();
var projection = query.$project({
    stringProp: true,
    aliasProp: "aliasName",
    assocProp: { dateProp: true }
});
var filter = query.$where({ stringProp: { $like: "A%" } });
// retrieve result
var result = projection.$execute();
// process result
for (var i = 0; i < result.length; i++) {
    var diff = result[i].assocProp.dateProp - Date.now();
    // ...
}
```

XS Procedures API

SAP HANA XS Procedures is a library of JavaScript tools which enable you to call SAP HANA stored procedures from server-side JavaScript (XS JS) as if the stored procedures were native JavaScript functions.

⚠ Restriction

XSPProc is intended only for use with database connections made with the old `$.db` API. It is not recommended to use XSPProc with `$.hdb` connections. For `$.hdb` connections, use `$hdb.loadProcedure` instead.

The following example shows how to consume a stored procedure using the XS Procedures API.

```
// import XS Procedures library
var XSPProc = $.import("sap.hana.xs.libs.dbutils", "procedures");
// set a schema where temporary tables can be created for passing table-valued
parameters to the procedure
XSPProc.setTempSchema($.session.getUsername().toUpperCase());
// load the procedure
var proc = XSPProc.procedure("schema", "namespace", "procedureName");
// call the procedure
```

```
var result = proc(1, [{col1: 0, col2:1}, {col1: 1, col2:2}]);  
// result is a JavaScript object
```

Related Information

[SAP HANA XS JavaScript API Reference](#)

[Maintaining HTTP Destinations \[page 112\]](#)

[XS Job File Keyword Options \[page 595\]](#)

[SAP Note SAP Note 786179](#) 

9.4.1 Tutorial: Use the XSJS Outbound API

The application package you put together in this tutorial includes all the artifacts you need to enable your server-side JavaScript application to use the Outbound Connectivity API to request and obtain data via HTTP from a service running on a remote host.

Prerequisites

Since the artifacts required to get the JavaScript application up and running are stored in the repository, it is assumed that you have already performed the following tasks:

- Create a development workspace in the SAP HANA repository
- Create a project in the workspace
- Share the new project
- The [HTTPDestViewer](#) SAP HANA user role

Context

SAP HANA Extended Application Services (SAP HANA XS) includes a server-side JavaScript API that enables outbound access to a defined HTTP destination. The HTTP destination provides services which an application can use, for example, to read live data. In this tutorial, you create a JavaScript application that queries financial services to display the latest stock values. The financial services are available on a remote server, whose details are specified in an HTTP destination configuration.

Procedure

1. Create a package for the SAP HANA XS application that will use the HTTP destination you define in this tutorial.

For example, create a package called `testApp`. Make sure you can write to the schema where you create the new application.

- a. Start the SAP HANA studio and open the *SAP HANA Development* perspective.
- b. In the *SAP HANA Systems* view, right-click the node in the package hierarchy where you want to create the new package and, in the pop-up menu that displays, choose *Packages...*
- c. In the *New Package* dialog that displays, enter the details of the new package (`testApp`) that you want to add and click *OK*.

2. Define the details of the HTTP destination.

You define the details of an HTTP destination in a configuration file that requires a specific syntax. The configuration file containing the details of the HTTP destination must have the file extension `.xshttpdest`.

⚠ Caution

Place the HTTP destination configuration in the same package as the application that uses it. An application cannot reference an HTTP destination configuration that is located in another application package.

- a. Create a plain-text file called `yahoo.xshttpdest` and open it in a text editor.

You can use the file-creation wizard in the *Project Explorer* view to create this file, for example, **► New ► Other ► XS HTTP Destination Configuration ►**.

- b. Enter the following code in the new file `yahoo.xshttpdest`.

```
host = "download.finance.yahoo.com";
port = 80;
description = "my stock-price checker";
useSSL = false;
pathPrefix = "/d/quotes.csv?f=a";
authType = none;
useProxy = false;
proxyHost = "";
proxyPort = 0;
timeout = 0;
```

- c. Save and activate the file.

📌 Note

Saving a file in a shared project automatically commits the saved version of the file to the repository.

3. View the activated HTTP destination.

You can use the *SAP HANA XS Administration Tool* to check the contents of an HTTP destination configuration.

📌 Note

To make changes to the HTTP Destination configuration, you must use a text editor, save the changes and reactivate the file.

- a. Open a Web browser.
- b. Start the *SAP HANA XS Administration Tool*.

The *SAP HANA XS Administration Tool* tool is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/admin/`.

Note

Access to details of HTTP destinations in the *SAP HANA XS Administration Tool* requires the credentials of an authenticated database user and one of the following SAP HANA roles:

- *HTTPDestViewer*
- *HTTPDestAdministrator*

- In the *XS Artifact Administration* screen, expand the nodes in the *Application Objects* tree to locate the application `testApp`.
 - Choose `yahoo.xshttpdest` to display details of the HTTP destination.
 - Check the details displayed and modify if required.
4. Create a server-side JavaScript application that uses the HTTP destination you have defined.
- The XSJS file must have the file extension `.xsjs`, for example, `sapStock.xsjs`.

Caution

You must place the XSJS application and the HTTP destination configuration it references in the same application package. An application cannot use an HTTP destination configuration that is located in another application package.

- Create a plain-text file called `sapStock.xsjs` and open it in a text editor.
- Enter the following code in the new file `sapStock.xsjs`.

In this example, you define the following:

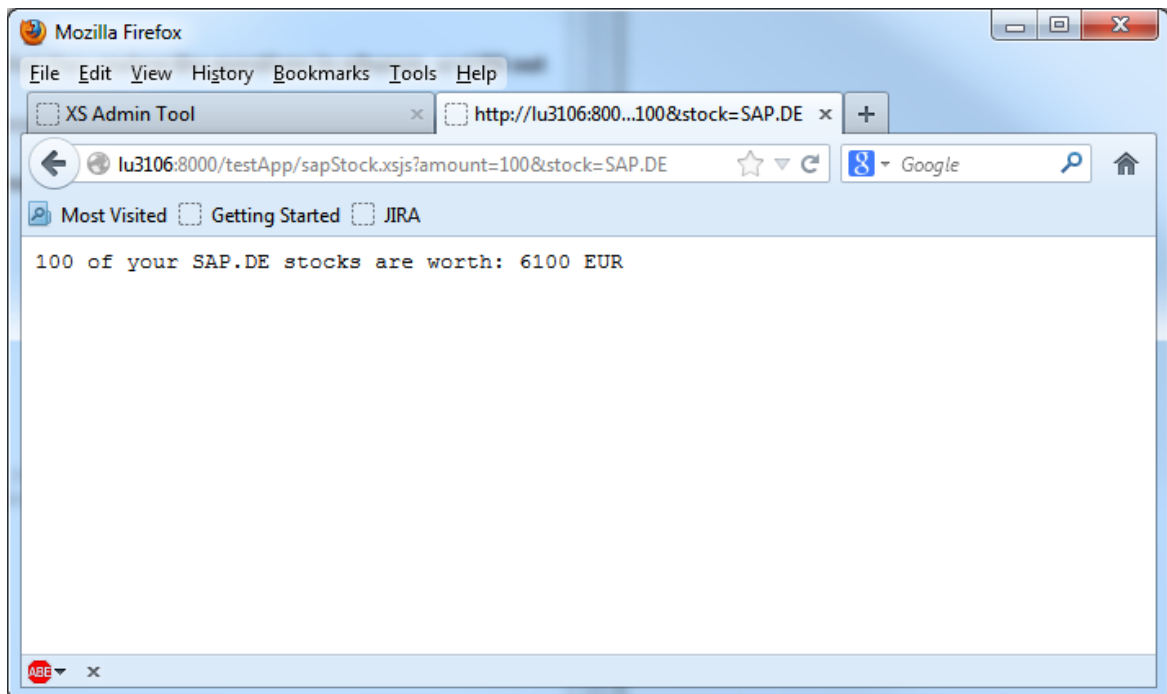
- A variable (`<stock>`) that defines the name of the stock, whose value you want to check, for example `SAP.DE`
- A variable (`<amount>`) that defines the number of stocks you want to check, for example, `100`
- A variable (`<dest>`) that retrieves metadata defined for the specified HTTP(S) destination, for example: host, port, useSSL...
- A variable (`<client>`) that creates the client for the outbound connection
- A variable (`<req>`) that enables you to add details to the request URL
- A variable (`<res>`) that calculates the value of the stock/amount
- The format and content of the response body displayed in the browser

```
var stock = $.request.parameters.get("stock");
var amount = $.request.parameters.get("amount");
var dest = $.net.http.readDestination("testApp", "yahoo");
var client = new $.net.http.Client();
var req = new $.web.WebRequest($.net.http.GET, "&s=" + stock);
client.request(req, dest);
var response = client.getResponse();
var co = [], he = [];
for(var c in response.cookies) {
    co.push(response.cookies[c]);
}
for(var c in response.headers) {
    he.push(response.headers[c]);
}
var body = undefined;
if(response.body)
    var body = response.body.asString();
$.response.contentType = "application/json";
var res = parseInt(response.body.asString()) * amount;
$.response.setBody(amount + " of your " + stock + " are worth: " + res);
```

- c. Save and activate the file.
5. Call the service provided by the application `sapStock.xsjs`.
 - a. Open a Web browser.
 - b. Enter the URL that calls your `sapStock.xsjs` application.

`http://<XS_Webserver>:80<SAPHANA_InstanceNr>/testApp/sapStock.xsjs?amount=100&stock=SAP.DE`

- **`<XS_Webserver>`**
Name of the system hosting the Web server for the SAP HANA XS instance where your `sapStock.xsjs` application is located.
- **`<SAPHANA_InstanceNr>`**
Number of the SAP HANA instance where the SAP HANA XS Web server is running, for example, **00**



6. Change the details specified in the URL used to run the application.
You can enter different values for the parameters `&amount` and `&stock` in the URL:

- **`amount=250`**
Change the number of stocks to check from 100 to 250
- **`&stock=SAP.DE`**
Change the name of stock to check from **SAP.DE** to **MCRO.L**

Related Information

[Maintaining HTTP Destinations \[page 112\]](#)
[SAP HANA XS JavaScript API Reference](#)

9.4.2 Tutorial: Call an XS Procedure with Table-Value Arguments

You can use the XS Procedures library to call stored procedures as if they were JavaScript functions.

Prerequisites

- The delivery unit `HANA_XS_DBUTILS` contains the XS procedures library. The content is available in the package `sap.hana.xs.libs.dbutils`.

⚠ Restriction

`XSPROC` is intended only for use with database connections made with the old `$.db` API. It is not recommended to use `XSPROC` with `$.hdb` connections. For `$.hdb` connections, use `$hdb.loadProcedure` instead.

- Create a new (or use an existing) development workspace in the SAP HANA repository.
- Create a new (or use an existing) shared project in the workspace.
- Create a new (or use an existing) stored procedure.
This tutorial refers to the stored procedure `get_product_sales_price`, which is included in the demonstration content provided with the SAP HANA Interactive Education (SHINE) delivery unit (DU). The SHINE DU is available for download in the SAP Software Download Center.

📌 Note

Access to the SAP Software Download Center is only available to SAP customers and requires logon credentials.

Context

You can call stored procedures by using the contents of the XS Procedures library as if they were JavaScript functions. For example, the library allows you to pass arguments as a JavaScript object to a stored procedure that expects table arguments; XS Procedures manages the creation and use of the temporary tables needed to pass arguments to a table-valued procedure. You can use the functions provided with the XS procedures library to enable programmatic access to stored procedures in the SAP HANA database from an XS JavaScript service; the access is provided by binding the stored procedure to a JavaScript function. The result of the call to the bound function is a JavaScript object, whose properties are the outbound parameters of the procedure.

Procedure

1. Import the XS procedures library.

In your server-side (XS) JavaScript code, ensure that the XS procedures are made available.

```
var XSProc = $.import("sap.hana.xs.libs.dbutils", "procedures");
```

2. Specify a schema where temporary tables can be created and filled with the values that are passed as arguments to the stored procedure.

XS procedures use temporary tables to pass table-valued parameters. As a user of XS procedures you must specify the name of a schema where these temporary tables reside, for example, a user's own schema.

Note

The application code using XS procedures must ensure that the necessary privileges have been granted to enable the creation and update of (and selection from) temporary tables in the specified schema.

```
XSProc.setTempSchema($.session.getUsername().toUpperCase());
```

3. Bind the stored procedure to a JavaScript function.

This step creates one or more JavaScript functions which can later be used to call the stored procedure. You can also define functions which map your call arguments to the parameters of the stored procedure.

```
var createPurchaseOrder = XSProc.procedure("SAP_HANA_DEMO",  
"sap.hana.democontent.epm.Procedures", "poCreate", {connection: conn});
```

Note

XS procedures uses the connection {connection: conn} passed in a configuration object as a parameter. If no connection object is passed, the XS procedure library opens a separate connection for the call and closes the connection after the call completes.

4. Call the procedure.

Use the imported procedure like a normal JavaScript function using JavaScript object argument lists.

```
var result = createPurchaseOrder([  
  "PURCHASEORDERID": '0300009001',  
  "HISTORY.CREATEDBY": '0000000044',  
  "HISTORY.CREATEDAT": new Date(),  
  "HISTORY.CHANGEDBY": '0000000044',  
  "HISTORY.CHANGEDAT": new Date()  
]);
```

Table-valued input arguments are passed to the stored procedure using a Javascript array that corresponds to the rows of the table containing the values to pass. The row objects should contain the properties of the name of the columns. Skipped columns are filled with NULL; properties without a same-named column are ignored.

Example

```
var XSProc = $.import("sap.hana.xs.libs.dbutils", "procedures");  
XSProc.setTempSchema($.session.getUsername().toUpperCase());  
var conn = $.db.getConnection();
```



```

var createPurchaseOrder = XSProc.procedure(
    "SAP_HANA_DEMO", "sap.hana.democontent.epm.Procedures",
    "poCreate", {connection: conn}
);
var result = createPurchaseOrder([
    "PURCHASEORDERID": '0300009001',
    "HISTORY.CREATEDBY": '0000000044',
    "HISTORY.CREATEDAT": new Date(),
    "HISTORY.CHANGEDBY": '0000000044',
    "HISTORY.CHANGEDAT": new Date()
]);
if (result && result.ERROR.length > 0) {
    $.response.setBody(result.ERROR.length + " errors occurred.");
} else {
    $.response.setBody("no error occurred");
}

```

Related Information

http://help.sap.com/hana/SAP_HANA_XS_DBUTILS_JavaScript_API_Reference_en/index.html

9.4.2.1 Accessing Stored Procedures from XS JavaScript

Call stored SAP HANA procedures from XS server-side JavaScript (XSJS) and process the results of the calls in JavaScript.

XS procedures provide a convenient way to call stored procedures in SAP HANA from XS server-side JavaScript (XSJS) and process the results of the calls in JavaScript. The XS procedures library extends the features already available with the SAP HANA XS JavaScript database API. Using XS procedures, SAP HANA stored procedures can be considered as simple XS JavaScript functions for anyone developing XS JavaScript services.

For example, where an SAP HANA stored procedure uses a table as input parameter and a table as output, XS Procedures use JavaScript objects (or an array of objects) which can be passed to the procedure. Similarly, the result of the procedure call is provided as an array of JavaScript objects. You declare a stored procedure as an XS JavaScript function and then call the stored procedure as if it were a JavaScript function delivering a JavaScript object.

To use a stored procedure as an XS JavaScript function, the following steps are required:

Step	Action	Description
1	Import the XS Procedures library	Provide access to the XS procedures

⚠ Restriction

XSProc is intended only for use with database connections made with the old `$.db` API. It is not recommended to use **XSProc** with `$.hdb` connections. For `$.hdb` connections, use `$hdb.loadProcedure` instead.

Step	Action	Description
2	Specify a schema for temporary tables	Temporary tables are used to store the JavaScript arguments provided for the function.
3	Import the procedure	Create the XS JavaScript functions, which can later be used to call the stored SAP HANA procedure. You can define functions which map your call arguments to the parameters of the stored procedure.
4	Call the procedure	Use the imported procedure in the same way as any normal JavaScript function, for example, using JavaScript object argument lists.
		▲ Restriction XSPROC is intended only for use with database connections made with the old \$.db API. For \$.hdb connections, use \$hdb.loadProcedure instead.
Use Arguments that Reference an Existing Table [page 569]		(Optional) Write the results or a procedure call into a physical table and pass the table as an argument rather than a JavaScript object
Use Table-Valued Arguments [page 570]		(Optional) Call a procedure with arguments stored as values in a table

Calling Procedures with Arguments that Reference an Existing Table

If you want to pass a table as an argument rather than a JavaScript object, you must specify the name of the table (as a string) in the call statement as well as the name of the schema where the table is located. The following example shows how to reference the table `rating_table`.

```
getRating('schema.rating_table', 3);
```

The SAP HANA database enables you to materialize the results of a procedure call; that is, to write the results into a physical table using the **WITH OVERVIEW** expression. In the **WITH OVERVIEW** expression, you pass a string value to the output parameter position that contains the result you want to materialize. The value returned is not the rating itself, but a reference to the table into which the results have been written. The results of the procedure call can now be retrieved from the specified table, in this example, `OUTPUT_TABLE`.

```
var resCall = getRating(rating, 3, "schema.output_table");
// {"RESULT": [{"variable":"RESULT","table":"\SCHEMA\\".\"OUTPUT_TABLE\""}]}
```

The **WITH OVERVIEW** expression also allows you to write the results of a procedure into a global temporary table; that is, a table that is truncated at session close. To use XS Procedures to write the results of a procedure into a global temporary table, you do not specify a name for the result table; you include an empty string (`' '`), as illustrated in the following example:

```
var conn = $.db.getConnection();
```

```
resCall = getRating(rating, 3, '', conn);
// {"RESULT":
[{"variable": "RESULT", "table": "\"SCHEMA\".\"RESULT_5270ECB8F7061B7EE10000000A379516\""}]}
```

The returned reference points to a global temporary table which can be queried for the procedure results with the same connection.

📌 Note

To ensure access to the global temporary table, it is necessary to specify the connection object `conn`.

Using Table-Valued Arguments

XS Procedures enables you to call procedures with arguments stored as values in a table, as illustrated in the following example. Table-valued input arguments are passed using a JavaScript array that corresponds to the rows of the table to pass. These row objects must contain properties that correspond to the name of the columns. Skipped columns are filled with NULL, and properties that do not correspond to an identically named column are ignored.

```
var XSProc = $.import("sap.hana.xs.libs.dbutils", "procedures");
XSProc.setTempSchema($.session.getUsername().toUpperCase());
var conn = $.db.getConnection();
var createPurchaseOrder = XSProc.procedure("SAP_HANA_DEMO",
    "sap.hana.democontent.epm.Procedures:poCreate", {
    connection: conn
});
var result = createPurchaseOrder([
    {
        "PURCHASEORDERID": '0300009001',
        "HISTORY.CREATEDBY": '0000000044',
        "HISTORY.CREATEDAT": new Date(),
        "HISTORY.CHANGEDBY": '0000000044',
        "HISTORY.CHANGEDAT": new Date()
    }
]);
if (result && result.ERROR.length > 0) {
    $.response.setBody(result.ERROR.length + " errors occurred.");
} else {
    $.response.setBody("no error occurred");
}
```

Related Information

[SAP HANA XS JavaScript API Reference](#)

9.4.3 Tutorial: Query a CDS Entity using XS Data Services

You can use the SAP HANA XS Data Services (XSDS) library to query CDS entities as if they were JavaScript objects.

Prerequisites

- A new (or an existing) development workspace in the SAP HANA repository
- A new (or an existing) shared project in the workspace
- This tutorial refers to CDS models that are included in the demonstration content provided with the SAP HANA Interactive Education (SHINE) delivery unit (DU). The SHINE DU is available for download in the SAP Software Download Center.

Note

Access to the SAP Software Download Center is only available to SAP customers and requires logon credentials.

Context

XS Data Service queries are used to build incrementally advanced queries against data models that are defined with Core Data Service. Query results are arrays of nested JSON objects that correspond to instances of CDS entities and their associations.

Procedure

1. Import the XS DS library and reference it through a variable.

```
var XSDS = $.import("sap.hana.xs.libs.dbutils", "xsds");
```

2. Import the CDS entities you want to query.

As a first step to working with CDS entities in SAP HANA XS JavaScript, you must import the CDS entities. The following example shows how to import the entities as defined in the SHINE demonstration content:

```
var soItem = XSDS.$importEntity("sap.hana.democontent.epm.data",  
    "EPM.SO.Item");  
var soHeader = XSDS.$importEntity("sap.hana.democontent.epm.data",  
    "EPM.SO.Header", {  
    items: {  
        $association: {  
            $entity: soItem,  
            $viaBacklink: "SALESORDERID"  
        }  
    }  
});
```

In addition to the basic CDS definition, the code in the example above shows how to extend the definition of `soHeader` by an explicit association called `items`. This is done by using the keyword `$association` together with the referenced entity (`soItem`) and the type of the association. In this case, `$viaBackLink` is used as type, that is; the items of `soHeader` stored in `soItem` have a foreign key `SALESORDERID` referencing the key of the `soHeader` table.

3. Add a query.

A general query related to an entity is built by calling the `$query()` method of the entity constructor.

```
var qOrders = soHeader.$query();
```

4. Refine the query if required.

You can refine the query object as necessary to suit your use case. For example, you can specify that the query returns only the first three (3) entries.

```
qOrders = qOrders.$limit(3);
```

5. Execute the query.

Use the `$execute` method to run the query.

```
var result = qOrders.$execute();
```

`result` contains an array of unmanaged values, each of which represents a row of the `Post` entity.

Note

In the refinements to the query, you must call `$execute` to send the query to the database.

6. Specify the fields the query should return.

Use the `$project()` method to create a query which specifies the fields the query should return. For example, you can return the IDs of the sales orders together with the net amount of the header and the net amount of all items.

```
var qOrderAndItemTitles = qOrders.$project({
  SALESORDERID: true,
  NETAMOUNT: "TotalNet",
  items: {
    NETAMOUNT: true
  }
});
```

The list of projected fields is a JavaScript object, where desired fields are marked by either `true` or a String literal such as `"TotalNet"` denoting an alias name. The query illustrated in the example above would return the following result.

```
[{
  "SALESORDERID": "0500000236",
  "TotalNet": 273.9,
  "items": {
    "NETAMOUNT": 29.9
  }
}, {
  "SALESORDERID": "0500000236",
  "TotalNet": 273.9,
  "items": {
    "NETAMOUNT": 102
  }
}]
```

```

}, {
  "SALESORDERID": "0500000236",
  "TotalNet": 273.9,
  "items": {
    "NETAMOUNT": 55
  }
}]

```

The actual database query automatically JOINS all required tables based on the associations involved. In the example above, the generated SQL looks like the following:

Note

In the following code example, the names of table are abbreviated to help readability.

```

SELECT "t0"."SALESORDERID" AS
  "t0.SALESORDERID",
  "t0"."NETAMOUNT" AS "t0.NETAMOUNT",
  "t0.items"."NETAMOUNT" AS "t0.items.NETAMOUNT"
FROM "Header" "t0"
LEFT OUTER JOIN "Item" "t0.items"
  ON "t0"."SALESORDERID"="t0.items"."SALESORDERID"
LIMIT 10

```

7. Use conditions to restrict the result set.

You can use the `$where()` method to set conditions that restrict the result set returned by the query. The following example show how to select all items with a net amount equal to a half (or more) of their order's net amount.

```

var qSelectedOrders =
  qOrderAndItemTitles.$where(soHeader.items.NETAMOUNT.$div(soHeader.NETAMOUNT).$
    gt(0.5))

```

References to fields and associations such as `items` are available as properties of the entity constructor function, for example, `soHeader.items`. As in the case with projections, XSDS generates all required JOINS for associations referenced by the conditions automatically, even if they are not part of the current projection. To build more complex expressions in `$where`, see the *SAP HANA XS Data Services JavaScript API Reference*.

8. Refine the query conditions to a specific matching pattern.

With the `$matching()` method you can specify conditional expressions using the JSON-like syntax of the `$find()` and `$findAll()` methods. The following code example shows how to further refine the selection returned by the result set, for example, to accept only those items with a EUR currency and quantity greater than 2.

```

qSelectedOrders = qSelectedOrders.$matching({
  items: {
    CURRENCY: 'EUR',
    QUANTITY: {
      $gt: 2
    }
  }
});

```

→ Tip

Unlike `$findAll()`, `$matching()` returns an unmanaged plain value and ignores all unpersistent changes to any entity instances.

9. Add arbitrary values to the result set.

You can add arbitrary calculated values to the result set by using the `$addField()` method. The following example shows how to query the days passed since the delivery of the sales item.

```
qSelectedOrders = qSelectedOrders.$addField({
  "DaysAgo": soHeader.items.DELIVERYDATE.$prefixOp("DAYS_BETWEEN", new
  Date())
});
```

Note

This query refers to the SQL function `DAYS_BETWEEN`, which is not a pre-defined function in XSJS. Instead, you can use the generic operator `$prefixOp`, which can be used for any SQL function `f`, for example, with the syntax `f(arg1, ... argN)`.

10. Use aggregations with calculated fields.

Aggregations are a special case of calculated fields that combine the `$addField()` operator with an additional `$aggregate()` method. The following example shows to retrieve the average quantity of the first 100 sales order IDs together with their product ID.

```
var qAverageQuantity = soItem.$query().$limit(100).$aggregate({
  SALESORDERID: true,
  PRODUCTID: true
}).$addField({
  averageQuantity: soItem.QUANTITY.$avg()
});
```

→ Tip

In SQL terms, the `$aggregate()` operator creates a GROUP BY expression for the specified paths and automatically projects the result.

If you need to use a more restrictive projection, you can replace `true` with `false` in the `$aggregate` call, as illustrated in the following example, which removes the sales order IDs for the result set.

```
var qAverageQuantity = soItem.$query().$limit(100).$aggregate({
  SALESORDERID: false,
  PRODUCTID: true
}).$addField({
  averageQuantity: soItem.QUANTITY.$avg()
});
```

11. Specify the order of the result set.

To specify the order in the result set, you can use the `$order()` method, including a number of order criteria as arguments. Each order criteria contains a property "by" with an expression that defines the desired order. Optionally each criterion can contain a flag `$desc` to require a descending order and a `$nullsLast` flag. The following example uses two criteria to display the result set first in descending order by the net amount in the header and then ascending order by the item net amount.

```
qSelectedOrders = qSelectedOrders.$order({$by: soHeader.NETAMOUNT,
  $desc:true},
  {$by: soHeader.items.NETAMOUNT});
```

12. Remove duplicates entries from the result set.

The `$distinct` operator removes duplicates from the result set. The following example shows how to display the set of all the currencies used in the sales orders.

```
var qAllCurrencies = soHeader.$query().$project({CURRENCY: true}).$distinct();
```

Related Information

[SAP HANA XS JavaScript API Reference](#)

[SAP HANA XS DB Utilities JavaScript API Reference](#)

[Creating Data Persistence Artifacts with CDS in XS Classic \[page 155\]](#)

9.4.4 Tutorial: Update a CDS Entity Using XS Data Services

You can use the XS Data Services (XSDS) library to update CDS entities as if they were JavaScript objects.

Prerequisites

- A new (or an existing) development workspace in the SAP HANA repository
- A new (or an existing) shared project in the workspace
- This tutorial refers to CDS models that are included in the demonstration content provided with the SAP HANA Interactive Education (SHINE) delivery unit (DU). The SHINE DU is available for download in the SAP Software Download Center.

Note

Access to the SAP Software Download Center is only available to SAP customers and requires logon credentials.

Context

For read-write scenarios, SAP HANA XS Data Services (XSDS) offer a managed mode with automatic entity management and additional consistency guarantees. Managed mode shares CDS imports and transaction handling with unmanaged mode but uses a different set of methods provided by the entity constructors.

Procedure

1. Import the XSDS library and the CDS entities into your application.

In your entity import, specify a SAP HANA sequence that is used to generate the required keys.

```
// import XSDS client library
var XSDS = $.import("sap.hana.xs.libs.dbutils", "xsds");
// import CDS entity as XSDS entity
var SOItem = XSDS.$importEntity("sap.hana.democontent.epm.data",
"EPM.SO.Item");
var SOHeader = XSDS.$importEntity("sap.hana.democontent.epm.data",
"EPM.SO.Header", {
  SALESORDERID: { $key:
  "\"SAP_HANA_DEMO\".\"sap.hana.democontent.epm.data::salesOrderId\"" },
  items: {
    $association: {
      $entity: SOItem,
      $viaBacklink: "SALESORDERID"
    }
  }
});
```

2. Retrieve an existing entity instance in managed mode.

The `$importEntity()` function returns a constructor for the entity imported. To retrieve an existing entity instance in managed mode, run a query using the entity's key (for example, using `$get`), or retrieve multiple instances that satisfy a given condition.

```
var order = SOHeader.$get({ SALESORDERID:
"0500000236" }); // by key
var orders = SOHeader.$findAll({ LIFECYCLESTATUS: "N", TAXAMOUNT: { $gt:
17000 } }); // by filter
```

3. Use or modify entity instances as required.

Instances of CDS entities are regular JavaScript objects which you can use and modify as required.

```
order.CURRENCY = "USD";
order.HISTORY.CHANGEDAT = new Date();
```

4. Ensure all changes are made persistent in the database.

Calling `$save()` flushes in-memory changes of the instance and all its reachable associated instances to the database. Only entity instances that have been changed will be updated in the database.

```
order.$save();
```

5. Use the entity constructor to create a new CDS instance.

The key is generated automatically by the SAP HANA sequence supplied during the import of the XSDS library and the CDS entities into your application.

```
var newOrder = new SoHeader ({
  TAXAMOUNT": 69.04,
  NETAMOUNT": 190.9,
  GROSSAMOUNT": 325.94,
  CURRENCY": "EUR",
  PARTNERID": "0100000044",
  DELIVERYSTATUS: "I",
  BILLINGSTATUS: "I",
  LIFECYCLESTATUS: "N",
  HISTORY": {
    CHANGEDAT": Date.now(),
    CHANGEDBY": "0000000033",
```

```

    CREATEDAT": Date.now(),
    CREATEDBY": "0000000033"
  },
  items: []
});
newOrder.$save();

```

6. Discard any unwanted instances of a CDS entity.

Retrieved CDS entities are stored in the entity manager cache and subject to general JavaScript garbage-collection rules. Use the `$discard()` function to permanently delete an entity instance from the database.

```

order.$discard();

```

7. Control how associations in a CDS document are followed.

By default, all associations are resolved, that is; association properties store a reference to their associated entity instance. For heavily connected data, this may lead to very large data structures in memory. A “lazy” association will delay the retrieval of the associated instances until the property is actually accessed. The first time the lazy association is accessed, the associated entity is queried from the entity cache or the database. After a lazy association has been resolved, it becomes a normal property of its parent entity instance.

To control how associations are being followed, declare “lazy” associations during the import operation, as shown in the following example:

```

var SOHeader = XSDS.$importEntity("sap.hana.democontent.epm.data",
  "EPM.SO.Header", {
    SALESORDERID: { $key:
      "\"SAP_HANA_DEMO\".\"sap.hana.democontent.epm.data::salesOrderId\"" },
    items: {
      $association: {
        $entity: SOItem,
        $viaBacklink: "SALESORDERID",
        $lazy: true
      }
    }
  });

```

The retrieval of “Lazy” associations is handled transparently by XSDS.

```

var order = SOHeader.$get({ SALESORDERID: "0500000236" }); // retrieve
single SO header
if (order.DELIVERYSTATUS != "D")
  return; // return without loading SO items from database
for (var item in order.items) { ... }; // now retrieve items for processing

```

8. Manually control transactions for your application where necessary.

Every SAP HANA XS application using XSDS is associated with one database connection and one transaction. This is also true if the application uses multiple imports of the XSDS library; XS libraries are single instances by default. Entities retrieved from the database are stored in the entity manager cache, and any updates need to be saved explicitly to the database. By default, database saves will automatically commit the changes to the database. However, you can manually control transactions for your application by disabling auto-commit and calling `$commit` and `$rollback` explicitly, as illustrated in the following example.

```

// disable auto-commit
XSDS.Transaction.$setAutoCommit(false);
var order = SOHeader.$get({ SALESORDERID: "0500000236" });
order.CURRENCY = "JPY";
order.$save(); // persist update

```

```

XSDS.Transaction.$commit();           // commit change
order.CURRENCY = "EUR";
order.$save();                         // persist update
order.HISTORY.CHANGEDAT = new Date();
order.$save();                         // persist update
XSDS.Transaction.$rollback();          // database rollback
// order #0500000236 now has currency JPY again

```

Related Information

[SAP HANA XS JavaScript API Reference](#)

[SAP HANA XS DB Utilities JavaScript API Reference](#)

[Creating Data Persistence Artifacts with CDS in XS Classic \[page 155\]](#)

9.5 Creating Custom XS SQL Connections

In SAP HANA Extended Application Services (SAP HANA XS), you use the SQL-connection configuration file to configure a connection to the database; the connection enables the execution of SQL statements from inside a server-side JavaScript application with credentials that are different to the credentials of the requesting user.

In cases where it is necessary to execute SQL statements from inside your server-side JavaScript application with credentials that are different to the credentials of the requesting user, SAP HANA XS enables you to define and use a specific configuration for individual SQL connections. Each connection configuration has a unique name, for example, `Registration` or `AdminConn`, which is generated from the name of the corresponding connection-configuration file (`Registration.xssqlcc` or `AdminConn.xssqlcc`) on activation in the repository. The administrator can assign specific, individual database users to this configuration, and you can use the configuration name to reference the unique SQL connection configuration from inside your JavaScript application code.

The following code example shows how to use the XS SQL connection `AdminConn.xssqlcc`.

```

function test() {
    var body;
    var conn;
    $.response.status = $.net.http.OK;
    try {
        conn = $.db.getConnection("sap.hana.sqlcon::AdminConn");
        var pStmt = conn.prepareStatement("select CURRENT_USER from dummy");
        var rs = pStmt.executeQuery();
        if (rs.next()) {
            body = rs.getNString(1);
        }
        rs.close();
        pStmt.close();
    } catch (e) {
        body = "Error: exception caught";
        $.response.status = $.net.http.BAD_REQUEST;
    }
    if (conn) {
        conn.close();
    }
    $.response.setBody( body );
}

```

```
}  
test();
```

To use the SQL connection from your application during runtime, you must bind the SQL connection configuration to a registered database user and assign the user the appropriate permissions, for example, by assigning a pre-defined role to the user. To maintain this user mapping, SAP HANA XS provides the Web-based [SAP HANA XS Administration Tool](#). When the run-time status of the XSSQLCC artifact is set to [active](#), SAP HANA generates a new auto user (with the name `XSSQLCC_AUTO_USER_[...]`). The new user is granted the permissions specified in a role, which can be assigned using the parameter `role_for_auto_user` - either in the design-time artifact or the run-time configuration.

Note

Access to the tools provided by the [XS Administration Tool](#) requires the privileges granted by one or more specific user roles.

To use the [SAP HANA XS Administration Tool](#) to view or maintain an XS SQL connection configuration, you need the privileges granted by the following SAP HANA XS roles:

- [sap.hana.xs.admin.roles::SQLCCViewer](#)
Required to display the available SQL Connections and the current user mapping
- [sap.hana.xs.admin.roles::SQLCCAdministrator](#)
Required to modify details of the user mapping; the [SQLCCAdministrator](#) role includes the role [SQLCCViewer](#).

Troubleshooting Tips

If you are having problems implementing the XS SQL connection feature using an `.xssqlcc` configuration, check the following points:

- User permissions
Make sure that you grant the necessary user the activated role (for example, [sap.hana.xs.admin.roles::SQLCCAdministrator](#)). You can use the developer tools to grant roles (or privileges), as follows:

Note

The granting user must have the object privilege EXECUTE on the procedure `GRANT_ACTIVATED_ROLE`.

- SAP HANA studio
In the [Systems](#) view of the [Administration Console](#) perspective, choose ► [Security](#) ► [Users](#) ►
- SAP HANA Web-based Development Workbench
In the [Security](#) tool, expand the [Users](#) node, choose the target (or add a new) user, and use the [Granted roles](#) tab.
- XS Administration Tools
In the [SQL Connection Details](#) tab of the XSSQLCC artifact's run time configuration. To edit user/role details here, you will need the role [SQLCCAdministrator](#) and, in addition, the appropriate administrator permissions required to set up (and assign roles to) a database user.

- **File location**
Make sure that the SQL-role configuration file (`.xssqlcc`) you create is located in the same package as the application that references it.
- **Logon dependencies**
If your application is using form-based logon (configured in the application's `.xsaccess` file), make sure the `libxsauthenticator` library is present and specified in the list of trusted libraries displayed in the SAP HANA studio's *Administration Console* perspective (► *Administration* ► *Configuration Tab* ► *xsengine.ini* ► *application_container* ► *application_list*). If the `libxsauthenticator` library is not in the list of authorized libraries, an SAP HANA system administrator must add it.

ⓘ Note

If you have to authorize `libxsauthenticator`, you might also need to refresh the Web page in your browser the next time you want to access `.xssqlcc` to display the logon dialog again.

9.5.1 Create an XS SQL Connection Configuration

The `.xssqlcc` file enables you to establish a database connection that you can use to execute SQL statements from inside your server-side JavaScript application with credentials that are different to the credentials of the requesting user.

Prerequisites

- Access to an SAP HANA system
- Access to a development workspace and a shared project.
- The application package structure in which to save the artifacts you create and maintain in this task
- The SQL connection configuration file (`.xssqlcc`) you create must be located in the same package as the application that uses it.
- You have the privileges granted in the following SAP HANA user roles:
 - `sap.hana.xs.admin.roles::SQLCCViewer`
 - `sap.hana.xs.admin.roles::SQLCCAdministrator`

ⓘ Note

This tutorial combines tasks that are typically performed by two different roles: the application developer and the database administrator. The developer would not normally require the privileges of the SAP HANA administrator or those granted by the `SQLCCAdministrator` user role.

Context

In this tutorial, you learn how to configure an SQL connection that enables you to execute SQL statements from inside your server-side JavaScript application with credentials that are different to the credentials of the user requesting the XSJS service.

To configure and use an XS SQL configuration connection file, perform the following steps:

Procedure

1. Start the SAP HANA studio.
 - a. Open the [SAP HANA Development](#) perspective.
 - b. Open the [Project Explorer](#) view.
2. Create the application descriptors for the new application.
 - a. In the SAP HANA studio's [Project Explorer](#) view, right-click the folder `acme.com.xs.testApp1` where you want to create the new (`.xsapp`) file.
 - b. In the context-sensitive popup menu, choose [New](#) > [Other...](#) .
 - c. In the [Select a Wizard](#) dialog, choose [SAP HANA](#) > [Application Development](#) > [XS Application Descriptor File](#) .

The file-creation wizard adds the required file extension `.xsapp` automatically.

- d. Choose [Finish](#).

→ Tip

Files with names that begin with the period (`.`), for example, `.xsapp` or `.xsaccess`, are sometimes not visible in the [Project Explorer](#). To enable the display of all files in the [Project Explorer](#) view, use the [Customize View](#) > [Available Customization](#)  option and clear all check boxes.

- e. Activate the application descriptor file.

In the SAP HANA studio's [Project Explorer](#) view, right-click the new (`.xsapp`) file and choose [Team](#) > [Activate](#) .
3. Create the application access file for the new application.
 - a. In the SAP HANA studio, open the [SAP HANA Development](#) perspective.
 - b. In the [Project Explorer](#) view, right-click the folder where you want to create the new (`.xsaccess`) file.
 - c. In the context-sensitive popup menu, choose [New](#) > [Other...](#) .
 - d. In the [Select a Wizard](#) dialog, choose [SAP HANA](#) > [Application Development](#) > [XS Application Access File](#) .

The file-creation wizard adds the required file extension `.xsaccess` automatically and enables direct editing of the file.

📌 Note

The default name for the core application-access file is `.xsaccess` and cannot be changed.

- e. Choose *Finish*.
- f. Check the contents of the `.xsaccess` file.

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form"},
  "prevent_xsrp" : true
}
```

The entries in the `.xsaccess` file ensure the following:

- Application data can be exposed to client requests
 - Username and password credentials are required for logon authentication
 - Protection against cross-site, request-forgery attacks is enabled
- g. Activate the application access file.
In the SAP HANA studio's *Project Explorer* view, right-click the new (`.xsaccess`) file and choose **Team > Activate** from the context-sensitive popup menu.

4. Create the XS SQL connection configuration file.

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

Note

The SQL connection configuration file (`.xssqlcc`) you create must be located in the same package as the application that references it.

- a. Right-click the folder where you want to save the XS SQL connection configuration file and choose **New > Other... > Application Development > XS SQL Connection Configuration File** in the context-sensitive popup menu.
- b. Enter the name of the SQL connection configuration file in the *File Name* box, for example, `AdminConn`.

→ Tip

The file-creation wizard adds the required file extension automatically (for example, `AdminConn.xssqlcc`) and, if appropriate, enables direct editing of the new file in the corresponding editor.

- c. Choose *Finish* to save the changes and commit the new XS SQL connection configuration file in the repository.
- #### 5. Configure the details of the SQL connection that the XS JavaScript service will use.
- a. Define the required connection details.

```
{
  "description" : "Admin SQL connection"
  "role_for_auto_user" : "com.acme.roles::JobAdministrator"
}
```

→ Tip

Replace the package path (`com.acme.roles`) and role name (`JobAdministrator`) with the suitable ones for your case.

- b. Activate the XS SQL connection configuration file.

In the SAP HANA studio's *Project Explorer* view, right-click the new (.xssqlcc) file and choose

► *Team* ► *Activate* ► from the context-sensitive popup menu.

Note

Activating the SQL connection configuration file `AdminConn.xssqlcc` creates a catalog object with the name `sap.hana.xs.testApp1::AdminConn`, which can be referenced in a XS JavaScript application.

6. Write an XS JavaScript application that calls the XS SQL connection configuration.

To create a preconfigured SQL connection using the configuration object `AdminConn`, for example, from inside your JavaScript application code, you must reference the object using the object name with the full package path, as illustrated in the following code example.

```
function test() {
    var body;
    var conn;
    $.response.status = $.net.http.OK;
    try {
        conn = $.db.getConnection("sap.hana.xs.testApp1::AdminConn");
        var pStmt = conn.prepareStatement("select CURRENT_USER from dummy");
        var rs = pStmt.executeQuery();
        if (rs.next()) {
            body = rs.getNString(1);
        }
        rs.close();
        pStmt.close();
    } catch (e) {
        body = "Error: exception caught";
        $.response.status = $.net.http.BAD_REQUEST;
    }
    if (conn) {
        conn.close();
    }
    $.response.setBody( body );
}
test();
```

7. Save the changes to the artifacts you have created.

Note

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

8. Activate the changes in the repository.

- a. In the *Project Explorer* view, locate and right-click the package containing the new XS SQL and XS JavaScript artifacts.
- b. In the context-sensitive pop-up menu, choose ► *Team* ► *Activate* ►.

9. Bind the SQL connection configuration to a user.

You use the Web-based *SAP HANA XS Administration Tool* to configure the runtime elements of the XS SQL connection.

- a. Start the *SAP HANA XS Administration Tool*.

The *SAP HANA XS Administration Tool* is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/admin/`.

Note

Access to details of HTTP destinations in the *SAP HANA XS Administration Tool* requires the credentials of an authenticated database user and one of the following SAP HANA roles:

- *sap.hana.xs.admin.roles::SQLCCViewer*
- *sap.hana.xs.admin.roles::SQLCCAdministrator*

- In the *XS Applications* tab, expand the nodes in the application tree to locate the application *testApp*.
- Choose *AdminConn* to display details of the XS SQL configuration connection.

10. Set the run-time status of the XS SQL connection configuration.

You must change the status run-time status of the XS SQL connection configuration to *Active*. This run-time status can only be changed by an administrator. When the run-time status of the XSSQL connection configuration is set to *active*, SAP HANA automatically generates a new user (*XSSQLCC_AUTO_USER_[...]*) for the XSSQL connection configuration object and assigns the role defined in *role_for_auto_user* to the new auto-generated user.

Related Information

[The SQL Connection Configuration File \[page 584\]](#)

[SQL Connection Configuration Syntax \[page 586\]](#)

9.5.1.1 The SQL Connection Configuration File

The SQL-connection configuration file specifies the details of a connection to the database that enables the execution of SQL statements from inside a server-side (XS) JavaScript application with credentials that are different to the credentials of the requesting user.

If you want to create an SQL connection configuration, you must create the configuration as a flat file and save the file with the suffix *.xsqlcc*, for example, *MYSQLconnection.xsqlcc*.

→ Tip

If you are using the SAP HANA studio to create artifacts in the SAP HANA Repository, the file creation wizard adds the required file extension automatically and enables direct editing of the file.

The new configuration file must be located in the same package as the application that references it.

Note

An SQL connection configuration can only be accessed from an SAP HANA XS JavaScript application (*.xsjs*) file that is in the same package as the SQL connection configuration itself. Neither subpackages nor sibling packages are allowed to access an SQL connection configuration.

The following example shows the composition and structure of a configuration file *AdminConn.xsqlcc* for an SAP HANA XS SQL connection called *AdminConn*. On activation of the SQL connection configuration file *AdminConn.xsqlcc* (for example, in the package *sap.hana.sqlcon*), an SQL connection configuration

with the name `sap.hana.sqlcon::AdminConn` is created, which can be referenced in your JavaScript application. In the `xssqlcc` artifact, you can set the following values:

- **description**
A short description of the scope of the xs sql connection configuration
- **role_for_auto_user**
The name of the role to be assigned to the auto user (if required) that the XSSQL connection uses, and the absolute path to the package where the role definition is located in the SAP HANA repository.

`sap.hana.sqlcon::AdminConn.xssqlcc`

```
{
  "description" : "Admin SQL connection"
  "role_for_auto_user" : "com.acme.roles::JobAdministrator"
}
```

The run-time status of an XSSQL connection configuration is *inactive* by default; the run-time status can only be activated by an SAP HANA user with administrator privileges, for example, using the [SAP HANA XS Administration Tools](#). When the run-time status of the XSSQLCC artifact is set to *active*, SAP HANA generates a new auto user (with the name `XSSQLCC_AUTO_USER_[ . . . ]`) and assigns the role defined in `role_for_auto_user` to the new auto-generated user.

→ Tip

In the [SAP HANA XS Administration Tools](#), it is possible to view and edit both the the user's parameters and the role's definition.

To create a preconfigured SQL connection using the configuration object `AdminConn`, for example, from inside your JavaScript application code, you reference the object using the object name and full package path, as illustrated in the following code example.

```
{
  conn = $.db.getConnection("sap.hana.sqlcon::AdminConn");
}
```

Related Information

[SQL Connection Configuration Syntax \[page 586\]](#)

[Create an XS SQL Connection Configuration \[page 580\]](#)

9.5.1.2 SQL Connection Configuration Syntax

The XS SQL connection-configuration file `.xssqlcc` uses pairs of keywords and values to define the SQL connection.

Example: The XS SQL Connection Configuration (`.xssqlcc`) File

Code Syntax

```
{
  "description" : "Admin SQL connection"
  "role_for_auto_user" : "com.acme.roles::JobAdministrator"
}
```

description

A short description of the selected SQL connection configuration.

Sample Code

```
"description" : "Admin SQL connection"
```

role_for_auto_user

The name of (and package path to) the role assigned to be assigned to the new user that is automatically generated on activation of the XSSQL connection-configuration artifact.

Sample Code

```
"role_for_auto_user" : "com.acme.roles::JobAdministrator"
```

Activating the design-time XSSQL connection configuration generates a run-time object whose status is “inactive” by default; the run-time status must be set to *active* by an SAP HANA user with administrator privileges, for example, using the [SAP HANA XS Administration Tools](#). When the run-time status of the XSSQLCC artifact is set to *active*, SAP HANA generates a new auto user (with the name `XSSQLCC_AUTO_USER_[...]`) and assigns the role defined in `role_for_auto_user` to the new auto-generated user.

Related Information

[The SQL Connection Configuration File \[page 584\]](#)

[Create an XS SQL Connection Configuration \[page 580\]](#)

9.6 Setting the Connection Language in SAP HANA XS

HTTP requests can define the language used for communication in the HTTP header `Accept - Language`. This header contains a prioritized list of languages (defined in the Browser) that a user is willing to accept. SAP HANA XS uses the language with the highest priority to set the language for the requested connection. The language setting is passed to the database as the language to be used for the database connection, too.

In server-side JavaScript, the `session` object's `language` property enables you to define the language an application should use for a requested connection. For example, your client JavaScript code could include the following string:

```
var application_language = $.session.language = 'de';
```

Note

Use the language-code format specified in BCP 47 to set the session language, for example: "en-US" (US English), "de-AT" (Austrian German), "fr-CA" (Canadian French).

As a client-side framework running in the JavaScript sandbox, the SAP UI5 library is not aware of the `Accept - Language` header in the HTTP request. Since the current language setting for SAPUI5 is almost never the same as the language specified in the SAP HANA XS server-side framework, SAPUI5 clients could have problems relating to text displayed in the wrong language or numbers and dates formatted incorrectly.

The application developer can inform the SAP UI5 client about the current server-side language setting, for example, by adding an entry to the `<script>` tag in the SAPUI5 HTML page, as illustrated in the following examples:

- Script tag parameter:

```
<script id="sap-ui-bootstrap"
  type="text/javascript"
  src="/sap/ui5/1/resources/sap-ui-core.js"
  data-sap-ui-theme="sap_goldreflection"
  data-sap-ui-libs="sap.ui.commons"
  data-sap-ui-language="de">
</script>
```

- Global `sap-ui-config` object:

```
<script>
  window["sap-ui-config"] = {
    "language": "de"
  }
</script>
[...]
```

```
<script id="sap-ui-bootstrap"
[...]
```

The `sap-ui-config` object must be created and filled before the `sap-ui-bootstrap` script.

It is important to understand that the session starts when a user logs on, and the specified language is associated with the session. Although the user can start any number of applications in the session, for example, in multiple Browser tabs, it is not possible to set a different language for individual applications called in the session,

Setting the Session Language on the Server side

The script tag for the SAPUI5 startup can be generated on the server side, for example, using the `$.session.language` property to set the `data-sap-ui-language` parameter. Applications that have the SAPUI5 `<script>` tag in a static HTML page can use this approach, as illustrated in the following example:

```
<script id="sap-ui-bootstrap"
  type="text/javascript"
  src="/sap/ui5/1/resources/sap-ui-core.js"
  data-sap-ui-theme="sap_goldreflection"
  data-sap-ui-libs="sap.ui.commons"
  data-sap-ui-language="$UI5_LANGUAGE$">
</script>
```

The called XSJS page can be instructed to replace the `$UI5_LANGUAGE$` parameter, for example, with the value stored in `$.session.language` when loading the static HTML page.

Setting the Session Language with an AJAX Call

You can include an HTTP call in the static HTML page to fetch the correct language from the server using some server-side JavaScript code, as illustrated in the following example:

```
<script>
  var xmlHttp = new XMLHttpRequest();
  xmlHttp.open( "GET", "getAcceptLanguage.xsjs", false );
  xmlHttp.send( null );
  window["sap-ui-config"] = {
    "language" : xmlHttp.getResponseHeader("Content-Language")
  }
</script>
<script id="sap-ui-bootstrap"
...
</script>
```

This approach requires an XSJS artifact (for example, `getAcceptLanguage.xsjs`) that responds to the AJAX call with the requested language setting, as illustrated in the following example:

```
$.response.contentType = "text/plain";
$.response.headers.set("Content-Language", $.session.language);
$.response.setBody("");
```

9.7 Scheduling XS Jobs

Scheduled jobs define recurring tasks that run in the background. The JavaScript API `$.jobs` allows developers to add and remove schedules from such jobs.

If you want to define a recurring task, one that runs at a scheduled interval, you can specify details of the job in a `.xsjob` file. The time schedule is configured using cron-like syntax. You can use the job defined in an `.xsjob` file to run an XS Javascript or SQLScript at regular intervals. To create and enable a recurring task using the xsjob feature, you perform the following high-level tasks:

Note

The tasks required to set up a scheduled job in SAP HANA XS are performed by two distinct user roles: the application developer and the SAP HANA administrator. In addition, to maintain details of an XS job in the [SAP HANA XS Administration Tool](#), the administrator user requires the privileges granted by the role template `sap.hana.xs.admin.roles::JobAdministrator`.

Setting up Scheduled Jobs in SAP HANA XS.

Step	Task	User Role	Tool
1	Create the function or script you want to run at regular intervals	Application developer	Text editor
2	Create the job file <code>.xsjob</code> that defines details of the recurring task	Application developer	Text editor
3	Maintain the corresponding runtime configuration for the xsjob	SAP HANA administrator	XS Job Dashboard
4	Enable the job-scheduling feature in SAP HANA XS	SAP HANA administrator	XS Job Dashboard
5	Check the job logs to ensure the job is running according to schedule.	SAP HANA administrator	XS Job Dashboard

Related Information

[The XSJob File \[page 594\]](#)

[Tutorial: Schedule an XS Job \[page 590\]](#)

[XS Job File Keyword Options \[page 595\]](#)

9.7.1 Tutorial: Schedule an XS Job

The `xsjob` file enables you to run a service (for example, an XS JavaScript or an SQLScript) at a scheduled interval.

Prerequisites

- You have access to an SAP HANA system.
- You have access to the XS Job Dashboard, which requires the role `JobSchedulerAdministrator`
- You have a role based on the role template `sap.hana.xs.admin.roles::JobAdministrator`.
- You have a role based on the role template `sap.hana.xs.admin.roles::HTTPDestAdministrator`.

Note

This tutorial combines tasks that are typically performed by two different roles: the application developer and the database administrator. The developer would not normally require the privileges granted to the `sap.hana.xs.admin.roles::JobAdministrator` role, the `sap.hana.xs.admin.roles::HTTPDestAdministrator` role, or the SAP HANA administrator.

Context

In this tutorial, you learn how to schedule a job that triggers an XS JavaScript application that reads the latest value of a share price from a public financial service available on the Internet. You also see how to check that the XS job is working and running on schedule.

To schedule an XS job to trigger an XS JavaScript to run at a specified interval, perform the following steps:

Procedure

1. Create the application package structure that contains the artifacts you create and maintain in this tutorial.
Create a root package called `yahoo`. You use the new `yahoo` package to contain the files and artifacts required to complete this tutorial.

```
/yahoo/  
  .xsapp           // application descriptor  
  yahoo.xsjob      // job schedule definition  
  yahoo.xshttpdest // HTTP destination details  
  yahoo.xsjs       // Script to run on schedule
```

2. Write the XS JavaScript code that you want to run at the interval defined in an XS job schedule.
The following XS JavaScript connects to a public financial service on the Internet to check and download the latest prices for stocks and shares.

Create an XS JavaScript file called `yahoo.xsjs` and add the code shown in the following example:

```
function readStock(input) {
    var stock = input.stock;

    var dest = $.net.http.readDestination("yahoo", "yahoo");
    var client = new $.net.http.Client();
    var req = new $.web.WebRequest($.net.http.GET, "/d/quotes.csv?f=a&s=" +
stock);
    client.request(req, dest);
    var response = client.getResponse();
    var stockValue;
    if(response.body)
        stockValue = parseInt(response.body.asString(), 10);
    var sql = "INSERT INTO stock_values VALUES (NOW(), ?)";
    var conn = $.db.getConnection();
    var pstmt = conn.prepareStatement(sql);
    pstmt.setDouble(1, stockValue);
    pstmt.execute();
    conn.commit();
    conn.close();
}
```

Save and activate the changes in the SAP HANA Repository.

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.

3. Create an HTTP destination file using the wizard to provide access to the external service (via an outbound connection).

Since the financial service used in this tutorial is hosted on an external server, you must create an HTTP destination file, which provides details of the server, for example, the server name and the port to use for HTTP access.

Note

To maintain the runtime configuration details using the Web-based *XS Administration Tool* you need the privileges granted in the SAP HANA user role `sap.hana.xs.admin.roles::HTTPDestAdministrator`.

Create a file called `yahoo.xshttpdest` and add the following content:

```
host = "download.finance.yahoo.com";
port = 80;
```

Save and activate the changes in the SAP HANA Repository.

4. Create the XS job file using the wizard to define the details of the schedule at which the job runs.

The XS job file uses a cron-like syntax to define the schedule at which the XS JavaScript must run. This job file triggers the script `yahoo.xsjs` on the 59th second of every minute and provides the name "SAP.DE" as the parameter for the stock value to check.

Create a file called `yahoo.xsjob` and add the following code:

```
{
    "description": "Read stock value",
    "action": "yahoo:yahoo.xsjs::readStock",
    "schedules": [
```



```

    {
      "description": "Read current stock value",
      "xscron": "* * * * * 59",
      "parameter": {
        "stock": "SAP.DE"
      }
    }
  ]
}

```

Save and activate the changes in the SAP HANA Repository.

5. Maintain the XS job's runtime configuration.

You maintain details of an XS Job's runtime configuration in the [XS Job Dashboard](#).

→ Tip

Access to the XS Job Dashboard tool is only available to users who have been assigned the XS classic administrator role "JobSchedulerAdministrator". For more information about XS classic administration roles, see *Related Information* below.

a. Start the [SAP HANA XS Administration Tool](#).

The [SAP HANA XS Administration Tool](#) is available on the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/admin/`.

b. Maintain the details of the XS job.

📌 Note

To maintain details of an XS job using the Web-based [XS Administration Tool](#) you need the privileges granted in the SAP HANA user role [sap.hana.xs.admin.roles::JobAdministrator](#).

You need to specify the following details:

- [User](#)
The user account in which the job runs, for example, **SYSTEM**
- [Password](#)
The password required for user, whose account is used to run the job.
- [Locale](#)
The language encoding required for the locale in which the job runs, for example, **en_US**
- [Start/Stop time](#)
An optional value to set the period of time during which the job runs. Enter the values using the syntax used for the SAP HANA data type `LocalDate` and `LocalTime`, for example, **2014-11-05 00:30:00** (thirty minutes past midnight on the 5th of November 2014).
- [Active](#)
Enable or disable the job schedule
- [Session timeout](#)
Specify the session timeout for this XSJob in seconds. If you specify a value of 0 (zero) seconds for the XSJob's session timeout, the XSJob checks if a value is defined for the `sessiontimeout` key in the section `scheduler` of the `xsengine.ini` file. If no such key exists, the default session timeout of 900 seconds is used. If you want to define a non-default value for the scheduler's `sessiontimeout` key, you must create the key in the `scheduler` section of the `xsengine.ini` file and supply the desired timeout value, for example, 600 seconds.

Caution

It is not recommended to specify a value of 0 (zero) for the `sessiontimeout` key; this disables the session-timeout feature for all jobs started by the scheduler.

- c. Save the job.

Choose [Save Job](#) to save and activate the changes to the job schedule.

6. Enable the job-scheduling feature in SAP HANA XS.

This step requires the permissions granted to the SAP HANA administrator.

Note

It is not possible to enable the scheduler for more than one host in a distributed SAP HANA XS landscape.

- a. In the [XS Job Dashboard](#) set the [Scheduler Enabled](#) toggle button to **YES**.

Toggling the setting for the [Scheduler Enabled](#) button in the [XS Job Dashboard](#) changes the value set for the SAP HANA configuration variable `xsengine.ini > scheduler > enabled`, which is set in the [Configuration](#) tab of the SAP HANA studio's [Administration](#) perspective.

7. Check the job logs to ensure the XS job is active and running according to the defined schedule.

You can view the xsjob logs in the [XS Job Dashboard](#) tab of the [SAP HANA XS Administration Tool](#).

Note

To maintain details of an XS job using the Web-based [XS Administration Tool](#) you need the privileges granted in the SAP HANA user role `sap.hana.xs.admin.roles::JobAdministrator`.

If the job does not run at the expected schedule, the information displayed in the xsjob logs includes details of the error that caused the job to fail.

Related Information

[The XS Job File \[page 594\]](#)

[XS Job-File Keyword Options \[page 595\]](#)

[The XS Job Dashboard](#)

[SAP HANA XS Classic Administration Roles](#)

9.7.1.1 The XS Job File

The `.xsjob` file defines the details of a task that you want to run (for example, an XS JavaScript or an SQLScript) at a scheduled interval.

The XS job file uses a cron-like syntax to define the schedule at which the service defined in an XS JavaScript or SQLScript must run, as you can see in the following example, which runs the specified job (the stock-price checking service `yahoo.xsjs`) on the 59th second minute of every minute.

```
{
  "description": "Read stock value",
  "action": "yahoo:yahoo.xsjs::readStock",
  "schedules": [
    {
      "description": "Read current stock value",
      "xscron": "* * * * * 59",
      "parameter": {
        "stock": "SAP.DE"
      }
    }
  ]
}
```

When defining the job schedule in the `xsjob` file, pay particular attention to the entries for the following keywords:

- **action**

Text string used to specify the path to the function to be called as part of the job.

```
"action": "<package_path>:<XSJS_Service>.xsjs::<FunctionName>",
```

Note

You can also call SQLScripts using the `action` keyword.

- **description**

Text string used to provide context when the XSjob file is displayed in the [SAP HANA XS Administration](#) tool.

- **xscron**

The schedule for the specified task (defined in the “`action`” keyword); the schedule is defined using cron-like syntax.

- **parameter**

A value to be used during the action operation. In this example, the parameter is the name of the stock `SAP.DE` provided as an input for the parameter (`stock`) defined in the `readStock` function triggered by the `xsjob` action. You can add as many parameters as you like as long as they are mapped to a parameter in the function itself.

The following examples illustrate how to define an `xscron` entry including how to use expressions in the various `xscron` entries (day, month, hour, minutes,...):

- `2013 * * fri 12 0 0`

Every Friday of 2013 at 12:00 hours

- `* * 3:-2 * 12:14 0 0`

Every hour between 12:00 and 14:00 hours on every day of the month between the third day of the month and the second-last day.

→ Tip

In the day field, third from the left, you can use a negative value to count days backwards from the end of the month. For example, `* * -3 * 9 0 0` means: three days from the end of every month at 09:00.

- `* * * * * */5 *`

Every five minutes (`*/5`) and at any point (`*`) within the specified minute.

ⓘ Note

Using the asterisk (`*`) as a wild card in the seconds field can lead to some unexpected consequences, if the scheduled job takes less than 59 seconds to complete; namely, the scheduled job restarts on completion. If the scheduled job is very short (for example, 10 seconds long), it restarts repeatedly until the specified minute ends.

To prevent short-running jobs from restarting on completion, schedule the job to start at a specific second in the minute. For example, `* * * * * */5 20` indicates that the scheduled job should run every five minutes and, in addition, at the 20th second in the specified minute.

- `* * * -1.sun 9 0 0`

Every last Sunday of a month at 09:00 hours

Related Information

[XS Job File Keywords \[page 595\]](#)

[Tutorial: Schedule an XS Job \[page 590\]](#)

9.7.1.2 XS Job File Keyword Options

The XS job file `.xsjob` uses a number of keywords to define the job that must be run at a scheduled interval.

Example: The XS Job (`.xsjob`) File

```
{
  "description": "Read stock value",
  "action": "yahoo:yahoo.xsjs::readStock",
  "schedules": [
    {
      "description": "Read current stock value",
      "signature_version": 1,
      "xscron": "* * * * * 59",
      "parameter": {
        "stock": "SAP.DE"
      }
    }
  ]
}
```

```
}
```

description

```
{  
  "description": "Read stock value",  
}
```

The [description](#) keyword enables you define a text string used to provide context when the XS job is displayed for maintenance in the *SAP HANA XS Administration Tool*. The text string is used to populate the [Description](#) field in the *SCHEDULED JOB* tab.

action

```
{  
  "action": "myapps.finance.yahoo:yahoo.xsjs::readStock",  
}
```

The [action](#) keyword enables you to define the function to run as part of the XS job, for example, an XS JavaScript or an SQLScript. The following syntax is required: `"action" : "<package.path>:<XSJS_Service>.xsjs::<functionName>"`.

Note

If you want to use the action to call an SQLScript, replace the name of the XSJS service in the example, with the corresponding SQLScript name.

schedules

```
{  
  "schedules": [  
    {  
      "description": "Read current stock value",  
      "xscron": "* * * * * 59",  
      "parameter": {  
        "stock": "SAP.DE"  
      }  
    }  
  ]  
}
```

The [schedules](#) keyword enables you define the details of the XS job you want to run. Use the following additional keywords to provide the required information:

- **description** (optional)
Short text string to provide context

- **xscron**
Uses cron-like syntax to define the schedule at which the job runs
- **parameter** (optional)
Defines any values to be used as input parameters by the (XSJS or SQLScript) function called as part of the job

signature_version

```
{
  "signature_version": 1,
}
```

The [signature_version](#) keyword enables you manage the version “signature” of an XS job. You change the XS job version if, for example, the parameter signature of the job action changes; that is, an XS job accepts more (or less) parameters, or the types of parameters differ compared with a previous version of an XS job. On activation in the SAP HANA Repository, the signature of an XS job is compared to the previous one and, if the job’s signature has changed, any job schedules created at runtime will be deactivated.

Note

The default value for [signature_version](#) is 0 (zero).

Deactivation of any associated runtime job schedules prevents the schedules from silently failing (no information provided) and enables you to adjust the parameters and reactivate the job schedules as required, for example, using the enhanced XS JS API for schedules. Schedules defined in a design-time XS Job artifact are replaced with the schedules defined in the new version of the XS job artifact.

→ Tip

Minor numbers (for example, 1.2) are not allowed; the job scheduler interprets “1.2” as “12”.

xscron

```
{
  "schedules": [
    {
      "description": "Read current stock value",
      "xscron": "** * * * * * 59",
      "parameter": {
        "stock": "SAP.DE"
      }
    }
  ]
}
```

The [xscron](#) keyword is used in combination with the `schedules` keyword. The [xscron](#) keyword enables you to define the schedule at which the job runs. As the name suggests, the `xscron` keyword requires a cron-like syntax.

The following table explains the order of the fields (*) used in the “xscron” entry of the .xsjob file and lists the permitted value in each field.

xscron Syntax in the XS Job File

xscron Field (* from left to right)	Meaning and Permitted Value
Year	4-digit, for example, 2013
Month	1 to 12
Day	-31 to 31
DayOfWeek	mon,tue,wed,thu,fri,sat,sun
Hour	0 to 23
Minute	0 to 59
Second	0 to 59

Note

Using the asterisk (*) as a wild card in the seconds field can lead to some unexpected consequences, if the scheduled job takes less than 59 seconds to complete; namely, the scheduled job restarts on completion. If the scheduled job is very short (for example, 10 seconds long), it restarts repeatedly until the specified minute ends.

To prevent short-running jobs from restarting on completion, schedule the job to start at a specific second in the minute. For example, * * * * * */5 20 indicates that the scheduled job should run every five minutes and, in addition, at the 20th second in the specified minute. The job starts at precisely 20 seconds into the specified minute and runs only once.

The following table illustrates the syntax allowed to define expressions in the “xscron” entry of the .xsjob file.

Expression	Where used...	Value
*	Anywhere	Any value
*/a	Anywhere	Any a-th value
a:b	Anywhere	Values in range a to b
a:b/c	Anywhere	Every c-th value between a and b
a.y	DayOfWeek	On the a-th occurrence of the weekday y (a = -5 to 5)
a,b,c	Anywhere	a or b or c

parameter

```
{
  "schedules": [
    {
      "description": "Read current stock value",
      "xscron": "** * * * * * 59",
```

```

        "parameter": {
          "stock": "SAP.DE",
          "share": "BMW.DE"
        }
      ]
    }
  }
}

```

The optional `parameter` keyword is used in combination with the `schedules` keyword. The `parameter` keyword defines values to be used as input parameters by the XSJS function called as part of the job. You can list as many parameters as you like, separated by a comma (,) and using the JSON-compliant syntax quotations (").

9.7.2 Add or Delete a Job Schedule during Runtime

The `$.jobs` application programming interface (API) enables you to manipulate the schedules for an XS job at runtime.

Context

You can use the `$.jobs.JobSchedules` API to add a schedule to (or delete a schedule from) a job defined in an `.xsjob` file at runtime.

Note

Schedules added at runtime are deleted when the `.xsjob` file is redeployed.

Procedure

1. Create an XS job file using the `.xsjob` syntax.

Note

If you have already created this XS job file, for example, in another tutorial, you can skip this step.

Create a file called `yahoo.xsjob` and add the following code:

```

{
  "description": "Read stock value",
  "action": "yahoo:yahoo.xsjs::readStock",
  "schedules": [
    {
      "description": "Read current stock value",
      "xscron": "* * * * * 59",
      "parameter": {
        "stock": "SAP.DE"
      }
    }
  ]
}

```


Save and activate the changes in the SAP HANA Repository.

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.

2. Create the XS JavaScript (.xsjs) file you want to use to define the automatic scheduling of a job at runtime.

Name the file `schedule.xsjs`.

3. Use the `$.jobs` JavaScript API to add or delete a schedule to a job at runtime.

The following example `schedule.xsjs` adds a new schedule at runtime for the XS job defined in `yahoo.xsjob`, but uses the `parameter` keyword to change the name of the stock price to be checked.

```
var myjob = new $.jobs.Job({uri:"yahoo.xsjob"});
var id = myjob.schedules.add({
  description: "Query another stock",
  xcron: "* * * * * */10",
  parameter: {
    stock: "APC.DE"
  }
});
// delete a job schedule
// myjob.schedules.delete( {id: id } );
```

4. Save and activate the changes in the SAP HANA Repository.
5. Call the XS JavaScript service `schedule.xsjs` to add the new job schedule at runtime.

Related Information

[SAP HANA XS JavaScript Reference](#)

[XS Job File Keyword Options \[page 595\]](#)

9.8 Tracing Server-Side JavaScript

The SAP HANA XS server-side JavaScript API provides tracing functions that enable your application to write predefined messages in the form of application-specific trace output in the xsengine trace files (`xsengine*.trc`) according to the trace level you specify, for example, "info" (information) or "error".

If you use the server-side JavaScript API to enable your application to write trace output, you can choose from the following trace levels:

- debug
- info
- warning

- error
- fatal

For example, to enable debug-level tracing for your JavaScript application, include the following code:

```
$.trace.debug("request path: " + $.request.path);
```

📌 Note

You can view the `xsengine*.trc` trace files in the [Diagnosis Files](#) tab page in the [Administration](#) perspective of the SAP HANA studio.

9.8.1 Trace Server-Side JavaScript Applications

The server-side JavaScript API for SAP HANA XS enables you to activate the writing of messages into a trace file; the following trace levels are available: debug, error, fatal, info, and warning.

Context

By default, applications write messages of severity level error to the `xsengine*.trc` trace files; you can increase the trace level manually, for example, to fatal. In SAP HANA XS, the following steps are required to enable trace output for your server-side JavaScript application:

Procedure

1. Open the SAP HANA studio.
2. In the [Systems](#) view, double-click the SAP HANA instance to open the [Administration](#) view for the repository where your server-side JavaScript source files are located.
3. Choose the [Trace Configuration](#) view.
4. In the [Database Trace](#) screen area, choose [Edit Configuration](#).

The [Edit Configuration](#) icon is only visible if you have the required privileges on the selected SAP HANA system.

Overview	Landscape	Alerts	Performance	Volumes	Configuration	System Information	Diagnosis Files	Trace Configuration

Database Trace

Configuration: **Default**

If the database trace is configured, the traces for the trace components (INDEXSERVER and NAMESERVER for example) of the system are written to files named <servicename>.trc. Some of these traces are always activated by default.

SQL Trace

 Edit Configuration

If the SQL trace is active, the database calls for application users are traced. The trace data is sqltrace_<servername>.

User-Specific Trace

Configuration: **Not Specified**

If the user trace is specified, the traces for the trace components (INDEXSERVER and NAMESERVER for example) for a specific database or application user are written to files named <servicename>_<servername>_<contextname>.trc.

Performance Trace

Status: **Inactive**

If the performance trace is running, the system trace data is saved to the file specified.

Note

If the *Database Trace* screen area is not displayed, check that you are using a version of SAP HANA studio that is compatible (the same as) with the SAP HANA server where you want to set up tracing.

5. Select the *Show All Components* checkbox.
6. Enter the partial or full name of your application into the search box.
7. Find the trace matching your application name and choose the trace level you want to use to generate output.

The application name is the location (package) of the .xsapp file associated with the application you are tracing. The trace topic is named xsa: <package.path> <appName>.

Note

To set the trace level, click the cell in the *System Trace Level* column that corresponds with the application you want to trace and choose the desired trace level from the drop-down list.

8. Choose *Finish* to activate the trace level changes.

9.8.2 View XS JavaScript Application Trace Files

The server-side JavaScript API for SAP HANA XS enables you to instruct your JavaScript applications to write application-specific trace messages in the `xsengine*.trc` trace files. You can view the trace files in the [Diagnosis Files](#) tab page of the [Administration](#) perspective in the SAP HANA studio.

Context

The trace levels “debug”, “error”, “fatal”, “info”, and “warning” are available. To view trace output for your server-side JavaScript application, perform the following steps:

Procedure

1. Open the SAP HANA studio.
2. In the [Systems](#) view, double-click the SAP HANA instance to open the [Administration](#) view for the repository where your server-side JavaScript source files are located.
3. Choose the [Diagnosis Files](#) tab page.
4. In the [Filter](#) box, enter a string to filter the list of search files displayed, for example, `xsengine*.trc`.
The timestamp displayed in the [Modified](#) column does not always reflect the precise time at which the trace file was written or most recently modified.
5. Locate the trace file for your SAP HANA XS application and doubleclick the entry to display the contents of the selected trace-file in a separate tab page.

9.9 Debugging Server-Side JavaScript

SAP HANA XS provides a set of dedicated tools to enable you to debug the XS JavaScript code that you write. To trigger debugging, you need an XS JavaScript configuration.

Overview

To prepare the system for debugging, you need to perform the following high-level steps:

- Ensure all prerequisites listed below are met.
- Create a debug configuration or choose an existing debug configuration to use.
- Set breakpoints in the file you want to debug.
- Execute XS JavaScript debugging.

To trigger debugging, you need to choose an XS JavaScript configuration; each configuration type represents a different starting point for debugging an XS JavaScript file. To debug XS JavaScript, you must choose one of the following types of configuration:

- [XS JavaScript](#)
Use to debug a stand-alone XS JavaScript service.
- [XS JavaScript: Manual Session](#)
Use to debug an XS JavaScript initiated from any remote client using that specific XS session.
- [XS JavaScript: HTML-based](#)
Use to debug an XS JavaScript initiated from HTML.
- [XS JavaScript: XS OData-based](#)
Use to debug an XS JavaScript initiated from an XS OData breakout.

Note

Before you start debugging server-side JavaScript on SAP HANA Extended Application Services (SAP HANA XS), first check that you have fulfilled the following prerequisites:

- Ensure the delivery unit for SAPHANA XS debugging tools is imported
To import the `HANA_XS_BASE` .tgz delivery unit (DU) that contains the XS JavaScript debugging tools, in SAP HANA Studio, choose the option  [New](#)  [Import](#)  [Delivery Unit](#) .

Import

Select File

Select the required file to import content objects.

Select file

☒ Server
☐ Client

File: /usr/sap/X4D/SYS/global/hdb/auto_content/HANA_XS_BASE.tgz

Actions

☒ Overwrite inactive versions
☒ Activate objects
☒ Bypass clientside validation rules

Object import simulation

Status	Object name	Package name
✓		sap.hana.xs.admin
✓		sap.hana.xs.admin
✓	config	sap.hana.xs.admin.config
✓		sap.hana.xs.debugger
✓		sap.hana.xs.debugger
✓		sap.hana.xs.debugger
✓	Debugger	sap.hana.xs.debugger
✓	Debugger	sap.hana.xs.debugger.api

?

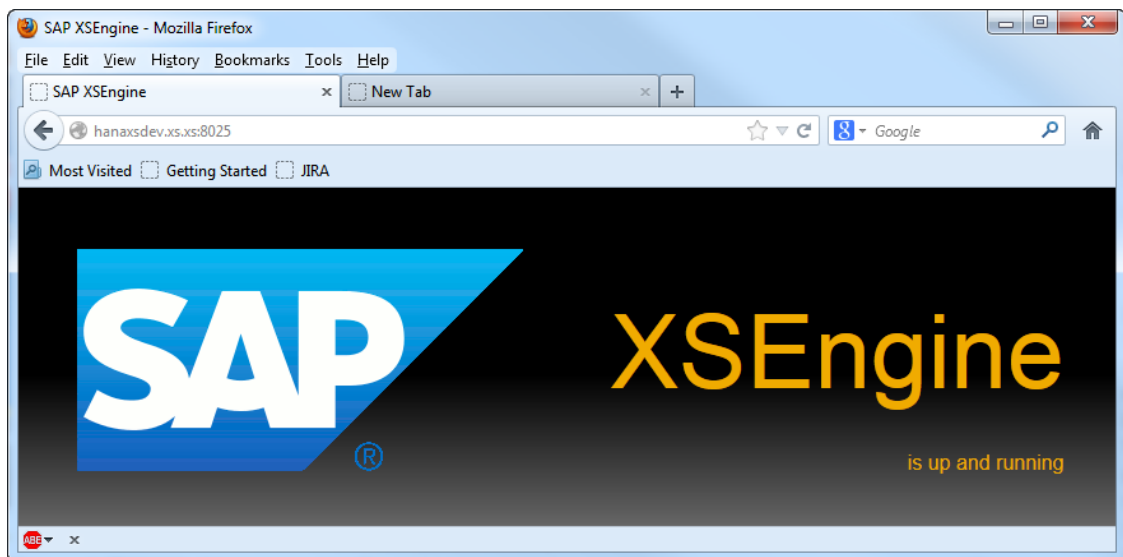
< Back

Next >

Finish

Cancel

- Enable debugging on the system level:
 1. Ensure the SAP HANA XS Web server is running, and that you have HTTP access to the following URL:
http:<SAPHANA_HOSTNAME>:<PortNumber>:/



2. Start SAP HANA Studio and open the *Administration* perspective.
3. In the *Configuration* tab, add a section called `xsengine.ini > debugger` (if it does not exist) and add (or set) the following parameter: `enabled=true`

xsengine.ini		
[] application_container		
[] communication		
[] debugger		
enabled		● true

- Assign the debugging role to your user
SAP HANA XS provides a dedicated debugger user role; the role must be assigned to any user who wants to start a debugging session for server-side JavaScript in SAP HANA XS.

Granted Roles			System Privileges	Object Privileges	Analytic Privileges	Package Privileges	Application Privilege
+	×						
Role	Grantor						
PUBLIC	SYS						
sap.hana.xs.debugger::Debugger	_SYS_REPO						
TEAM_XS	SYSTEM						

- Assign the debugging role to another user (optional)
You can grant a user global access to any of your debug sessions or grant access to a debug session that is flagged with a specified token. You can also restrict access to a debug session to a specified period of time.

Note

By default, other users do not have the permissions required to access your XS JavaScript debugging sessions. However, SAP HANA XS enables you to grant access to your debug sessions to other users, and vice versa.

1. Start SAP HANA Studio and open the *Administration* perspective.
2. In the *Systems* view, expand the *Security* node and double-click the user to whom you want to assign the debugger role.

3. In the [Granted Roles](#) view, choose the [+] icon and, in the [Select Role](#) dialog, enter **debugger** to search for the debugger role and choose [OK](#).

Note

Debugging can also be done in other settings, for example, when a server is cloud-based or when it is a secured server.

- Debugging with HANA Cloud Platform (HCP) (optional)
Debugging using HCP requires prerequisites to be fulfilled. For more information, see *Getting Started* in the SAP HANA Cloud Documentation.
- Debugging using a secure server (optional)
Debugging using a secure server requires specific prerequisites to be fulfilled. For more information, see *Configure SSL for SAP HANA Studio Connections* in the *SAP HANA Security Guide*.

Related Information

[Debug Session Access \[page 612\]](#)

[The XSJS Debugger Role \[page 611\]](#)

[Configure TLS/SSL for SAP HANA Studio Connections](#)

9.9.1 Create a Debug Configuration

Context

To create an XS JavaScript debug configuration, do the following:

Procedure

1. Open the [Debug](#) perspective.
2. Choose  and select [Debug Configurations](#).
3. Choose the debug configuration type you want to debug.

It can be one of the following:

- [XS JavaScript](#): Use to debug a standalone XS JavaScript service.
- [XS JavaScript: Manual Session](#): Use to debug an XS JavaScript initiated from any remote client using that specific XS session.
- [XS JavaScript: HTML-based](#): Use to debug an XS JavaScript initiated from HTML.

- *XS JavaScript: XS OData-based*: Use to debug an XS JavaScript initiated from an XS OData breakout.

Note

You can use an existing configuration, change it or create a new debug configuration by selecting the file type to use for debugging, and clicking on the [New](#) button.

4. In the *General* tab, enter a name for the new debug configuration.
5. The external browser is your default debug mode. You can also choose to debug using the internal SAP HANA Studio.
6. To build the URL, select the file and resource path or add parameters where relevant. Parameters can be entered using raw text or a table format.
 - When creating a manual session debug configuration, you only need to select the system to debug.
 - If a system is logged off, it will not show in the system dropdown list.
7. You can include stored procedures in your debug configuration which will enable SQL script to be debugged along with XS JavaScript. If your XS JavaScript code triggers stored procedures, you can set breakpoints and debug them using the same debug configuration. You do not need to create a separate, dedicated debug configuration for the stored procedures.
8. For configurations with an *Input Parameters* tab, select the method, and enter the header and body information as relevant. Body details can be entered as raw text or in the x-www-form-urlencoded format.
9. Choose *Apply*.
10. Choose *Close* to save the configuration for later use or *Debug* to start debugging.

9.9.2 Execute XS JavaScript Debugging

SAP HANA studio enables you to debug XS JavaScript files, including setting breakpoints and inspecting variables.

Context

To enable the display of more helpful and verbose information for HTTP 500 exceptions on the SAP HANA XS Web server, add the parameter `developer_mode` to the `xsengine.ini` `httpserver` section and set it to `true`. `xsengine.ini` is in the *Configuration* tab of the *Administration* perspective in SAP HANA studio.

Prerequisites

- Ensure that debugging is enabled on the SAP HANA server.
- You have the debugger role assigned to your user.
- User authentication is enabled. This is required to open the debugging session.

To start debugging, do the following:

Procedure

1. In a Web browser, run the XS JavaScript source file that you want to debug.
2. Create or choose a debug configuration for debug sessions for a specific SAP HANA installation.
 - a. Open the [Debug](#) view.
 - b. Choose a debug configuration.

You can also create a new configuration by doing one of the following:

- From the menu bar, click **Run > Run > Debug Configurations**.
 - Select the file to be debugged and right-click, choose **Debug As > Debug Configurations**.
- c. Choose [Apply](#).
 - d. Choose [Close](#).
3. Set Breakpoints

Set breakpoints in the JavaScript code by double-clicking the left vertical ruler.



4. Run the new debug configuration for your server by choosing  and selecting your debug configuration.

You can also run the debug configuration by doing one of the following:

- From the menu bar, click **Run > Debug Configurations**, then choose the debug configuration you want to use.
- Select the file to be debugged and right-click on it, and then choose [Debug As](#).
- From [Debug Configurations](#), click the debug configuration you want to use.
- For an HTML file, select the file to be debugged and right-click on it, then choose **Debug As > HTML**.

Note

When using the external debug mode, you can only have one open XS debug session per system. This is relevant for the following debug configurations:

- [XS JavaScript](#)
- [XS JavaScript: HTML-based](#)
- [XS JavaScript: XS OData-based](#)

Related Information

[Create a Debug Configuration \[page 607\]](#)

9.9.2.1 The Debug Perspective

SAP HANA studio includes a dedicated debug perspective, which provides the tools needed by a developer who wants to debug server-side JavaScript code.

Application developers can use the SAP HANA studio's [Debug](#) perspective to perform standard debugging tasks, for example: starting and resuming code execution, stepping through code execution, adding breakpoints to the code. Developers can also inspect variables and check the validity of expressions. The following views are available as part of the standard [Debug](#) perspective:

- [Debug](#)
Displays the stack frame for the suspended or terminated threads for each target you are debugging. Each thread in your program appears as a node in the tree. You can also see which process is associated with each target.
- [Breakpoints](#)
Displays a list of the breakpoints set in the source file you are currently debugging
- [Variables](#)
Displays a list of the variables used in the source file you are currently debugging
- [Expressions](#),
Displays global variables, such as `$ . request` and other SAP HANA XS JavaScript API objects
- [Outline](#)
Displays a structural view of the source file you are currently debugging. You can double-click an element to expand and collapse the contents.
- [Source-code editor](#)
SAP HANA studio uses the file extension (for example, `.js` or `.xsjs`) of the source file you want to debug and opens the selected file in the appropriate editor. For example, files with the `.js` or `.xsjs` file extension are displayed in the built-in JavaScript editor.

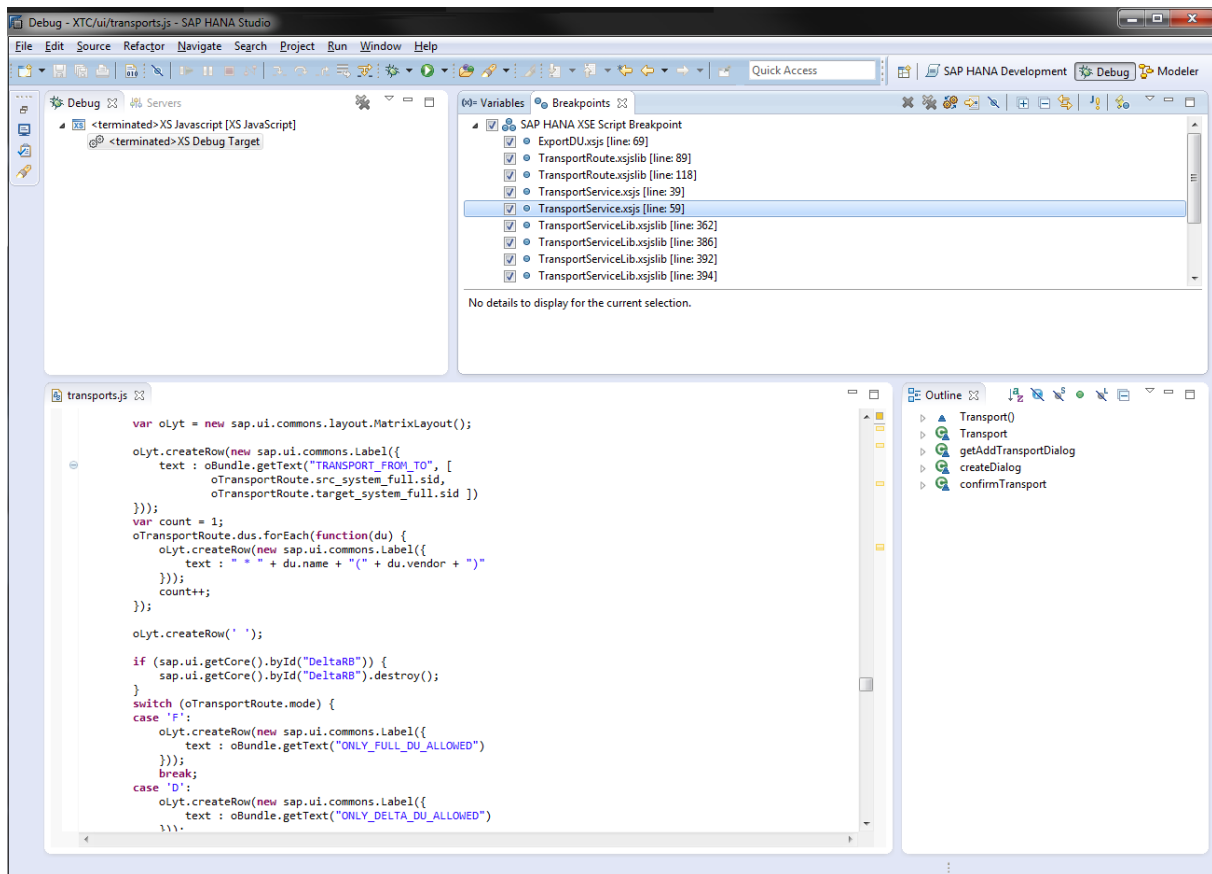
Note

Unified Debugger

In the unified debugger, if you choose to include the SQL script layer in the debugging session, you will see the targets of both the XS JavaScript and SQL script in the debug view.

If a breakpoint is set in the XS JavaScript or in an SQL script procedure, you will see the breakpoints in the breakpoint view. The debugger will stop at the breakpoints in the relevant XS JavaScript or in the SQL script as usual. SQL script debugging behavior is the same in the SQL script debugger as it is in the unified

debugger, with the exception of the call stack behavior. For more information about debugging SQL script procedures, see *Debugging Procedures*.



Related Information

[Debugging Procedures \[page 414\]](#)

9.9.2.2 The XSJS Debugger Role

The JavaScript debugger included with SAP HANA Extended Application Services (SAP HANA XS) requires user authentication to start a debug session. SAP HANA XS includes a dedicated debugger **role**, which defines the permissions needed by a developer who wants to debug server-side JavaScript code.

Debugging application code is an essential part of the application-development process. SAP HANA Extended Application Services (SAP HANA XS) includes a debug perspective, a debug view, and a dedicated debugger **role** that must be assigned to any developer who wants to debug XS JavaScript. The debugging role is named `sap.hana.xs.debugger::Debugger` and can be assigned to a user (or a role) either with the standard role-assignment feature included in SAP HANA studio (the *Application Privileges* tab in the *Security* area of the *Systems* view) or in a design-time, role-configuration file (`.hdbrole`).

Since a developer primarily needs to debug his own HTTP calls, the following limitations apply to a debug session:

- Only authenticated users can start a debug session, for example, by providing a user name and password when logging in to a debug session
- A user can debug his own sessions.
- A user can debug any session to which access has been explicitly granted, for example, by the owner of the session.

Note

It is also possible to use SSL for debugging. If SSL is configured, the server redirects the Web-socket connect call to the corresponding SSL (secure HTTP) URL, for example, if sent by plain HTTP.

SAP HANA studio includes a graphical user interface (GUI) which you can use to grant access to debug sessions at both the session level and the user level.

Related Information

[Custom Development Role \[page 707\]](#)

9.9.2.3 Debug Session Access

You can grant other developers access to the debug sessions you use for debugging server-side JavaScript on SAP HANA XS.

By default, other users are not allowed to access your XSJS debugging sessions. However, SAP HANA XS provides a tool that enables you to grant access to your debugging sessions to other users, too.

Note

You can grant a user global access to any of your sessions or grant access to a session that is flagged with a specified token. You can also restrict access to a debug session to a specified period of time.

The *XS Debugging* tool is available on the SAP HANA XS Web server at the following URL:

`<SAPHANAWebServer>80<SAPHANAinstance>/sap/hana/xs/debugger/`.

When you are grant access to your debugging session, the following options are available:

- *User Name*
The name of the database user who requires access to your debug session
- *Privilege Expires*
The point in time that marks the end of the period for which access to one or more debug sessions is allowed.
- *grant debug permission for all sessions*
You can grant a user **global** access to **any** of your debug sessions.

⚠ Restriction

The user you grant access to must already be registered and authenticated in the SAP HANA database.

- [grant debug permission for this session only](#)

You can grant access to a debug session that is flagged with a specific token:

⚠ Restriction

Unauthenticated users must use the token-based option.

The following rules apply to access to debug sessions flagged with a token:

- The session used for granting access to the debug sessions is flagged automatically.
- The session token is distributed by means of a session cookie; the cookie is inherited by any session created with the current browser session.
- [Session Name](#)
A freely definable name that can be used to distinguish your debug session in the context of multiple sessions.

Related Information

[The XSJS Debugger Role \[page 611\]](#)

[Debugging Server-Side JavaScript \[page 603\]](#)

9.9.3 Troubleshoot Server-Side JavaScript Debugging

When debugging your JavaScript code, you sometimes need to solve problems, not only with the code itself, but the configuration of the sessions and the tools you use to perform the debugging.

Prerequisites

- Start a Web-browser session with the SAP HANA server **before** starting a debug session.
Make sure you open a session with the SAP HANA server by calling an XS JavaScript file from your Web browser **before** starting the debug operation.
- Select the session ID.
Before starting to debug, select the session whose ID is specified in the `xsSessionId` cookie in your open Web-browser session.

Context

If you are having problems using the embedded debugging tools to debug your server-side XSJS (JavaScript) code, check the following solutions:

- **Breakpoints**
The execution of your XS JavaScript code is not stopping at a breakpoint.
- **Network connections**
Your SAP HANA server is behind a proxy or a firewall.

Procedure

1. Restart the SAP HANA studio with the `-clean` option.

Sample Code

```
hdbstudio.exe -clean
```

To determine if a clean restart of SAP HANA studio is required, check if the [Breakpoints](#) view in SAP HANA studio's [Debug](#) perspective displays the breakpoints as type `SAP HANA XSE Script`, as follows:

- a. In the [Breakpoints](#) view, choose the [View Menu](#).
 - b. Choose  [Group By](#) .
2. Remove breakpoints.
Try removing all the existing breakpoints from the debug session and recreating them.
 3. Create a new workspace.
If a restart of SAP HANA studio with the `-clean` option does not solve the problem of unrecognized breakpoints, it might be necessary to create a new Eclipse (not repository) Workspace.
 4. Set the [Active Provider](#) feature to **manual**.
If your SAP HANA server is behind a proxy or firewall, check that your [Network Connections](#) are configured for using a proxy, as follows:

Note

It is not recommended to run a debugging session without using the Secure Sockets Layer (SSL) protocol. The debugging session uses standard HTTP(S). The session either leverages an existing session or requests basic (HTTP) authentication on the connection request. The debugging session upgrades the HTTP connection to a WebSocket.

- a. In SAP HANA studio, choose  [Window](#)  [Preferences](#)  [General](#)  [Network Connections](#) .
 - b. Set the [Active Provider](#) to [Manual](#).
The default setting is [Native](#).
 - c. Update the schemas.
 - d. Add the relevant proxy host and port.
5. Configure the [Debug Configuration Connection](#) properties.

- a. Select and right-click your SAP HANA system.
- b. Choose ► *Properties* ► *XS Properties* ►.
- c. Check that your system's SAP HANA XS properties match the *Debug Configuration Connection* properties.

Related Information

[Execute XS JavaScript Debugging \[page 608\]](#)

9.10 Testing XS JavaScript Applications

SAP HANA provides a test framework called XSUnit that enables you to set up automatic tests for SAP HANA XS applications.

The test framework SAP HANA XSUnit (XSUnit) is a custom version of the open-source JavaScript test framework, Jasmine, adapted for use with SAP HANA XS. You can use the XSUnit test framework to automate the tests that you want to run for SAP HANA XS applications, for example, to test the following elements:

- Server side JavaScript code
- SQLScript code (stored procedures and views)
- Modeled calculation views

To use the tools and features provided with the XSUnit test framework, you must perform the following high-level steps:

1. Set up the client-side environment:
 - Install the latest version of SAP HANA studio (**optional**).
 - Ensure that the `hdbcli` tool is installed and running.

2. Set up the server-side environment.

The XSUnit test framework is included in the delivery unit *HANA_TEST_TOOLS*, which you must install manually, for example, using the SAP HANA studio or the *SAP HANA Application Lifecycle Management* tool. After the installation completes, the tools are available in the package `sap.hana.testtools`.

Note

Importing a delivery unit into an SAP HANA system requires the *REPO.IMPORT* privilege, which is normally granted only to the system administrator.

3. Maintain SAP HANA user privileges.

The system administrator must grant test users the privileges required to use the test tools. The privileges are defined in roles, which the SAP HANA administrator can assign to all developers by default.

4. Maintain the test schema (**optional**).

If you write XSUnit tests that are designed to test database content, you require a test schema in which you create test tables during your test execution and fill the tables with test data. To avoid conflicts when different users run the same test at the same time, it is recommended that individual developers place test tables in their corresponding user schema.

Note

You must ensure that `_SYS_REPO` has select permission to schema where the tables are located (for example, either your user schema or the test schema).

```
grant select on schema MY_TEST_SCHEMA to _SYS_REPO with grant option;
```

Related Information

[Automated Tests with XUnit in SAP HANA \[page 616\]](#)

[XUnit Test Examples \[page 624\]](#)

[SAP HANA XUnit JavaScript API Reference](#)

9.10.1 Automated Tests with XUnit in SAP HANA

XUnit is an integrated test environment that enables you to set up automatic tests for SAP HANA XS applications.

People developing applications in the context of the SAP HANA database need to understand how to implement a test-automation strategy. Especially for new applications which are designed to work exclusively with SAP HANA, it is a good idea to consider the adoption of best practices and tools.

If you want to develop content that is designed to run specifically in SAP HANA, it is strongly recommended to use the XUnit test framework that is integrated in SAP HANA XS; this is the only way to transport your test code with your SAP HANA content. The XUnit tools are based on a Java Script unit test framework that uses Jasmine as the underlying test library.

Test Isolation and Simulation

To write self-contained unit tests that are executable in any system, you have to test the various SAP HANA objects in isolation. For example, an SAP HANA view typically has dependencies to other views or to database tables; these dependencies pass data to the view that is being tested and must not be controlled or overwritten by the test. For this reason, you need to be able to simulate dependencies on the tested view. XUnit includes a test-isolation tool that provides this functionality; it allows you to copy a table for testing purposes.

Note

Although you cannot copy a view for testing purposes, you can create a table that acts like a view.

All (or specific) dependencies on any tables or views are replaced by references to temporary tables, which can be created, controlled, and populated with values provided by the automated test.

Test Data

Preparing and organizing test data is an important part of the process of testing SAP HANA content such as views and procedures; specific data constellations are required that have to be stable in order to produce reliable regression tests. In addition, test-isolation tools help reduce the scope of a test by enabling you to test a view without worrying about dependent tables and views. Limiting the scope of a test also helps to reduce the amount of data which needs to be prepared for the test.

Related Information

[XUnit Test Examples \[page 624\]](#)

[Test an SAP HANA XS Application with XUnit \[page 617\]](#)

9.10.2 Application Development Testing Roles

Dedicated roles enable developers to access and use the tools provided with the SAP HANA XS test framework (XUnit).

To grant access to the SAP HANA XS test framework that enables developers to set up automatic testing for SAP HANA applications, the SAP HANA system administrator must ensure that the appropriate roles are assigned. The following table lists the roles that are available; one (or more) of the listed roles must be assigned to the application developers who want to use the XUnit testing tools.

Default Developer Testing Roles

Role Name	Description
sap.hana.testtools.common::TestExecute	Enables you to view the persisted test results produced by the XUnit test framework and to execute the examples included in the demonstration package (<code>sap.hana.testtools.demo</code>).
sap.hana.xs.debugger::Debugger	Enables you to debug your server side JavaScript (test-)code
sap.hana.xs.ide.roles::Developer	Enables you to view source files in the SAP HANA Web-based Work Bench (Web IDE)

9.10.3 Test an SAP HANA XS Application with XUnit

Use the XUnit tools to set up automated testing of your applications in SAP HANA XS.

Prerequisites

The following prerequisites apply if you are using SAP HANA studio to set up and run tests with XUnit:

- SAP HANA studio
You will need access to a shared development project in the SAP HANA system where you plan to run the tests.

Context

If you want to develop content that is designed to run specifically with SAP HANA, you can use the XSUnit tools that are integrated in SAP HANA XS. The XS Unit tools are based on a Java Script unit test framework that uses Jasmine as the underlying test library.

Procedure

1. Create an Eclipse project.

If you want to create your first unit test, you need an XS Project that will contain the test code. You can either create a new shared XS Project or, if a shared project already exists, you can checkout and import the existing project from the SAP HANA Repository. Within that project you can structure your tests in folders.

To create a shared Eclipse project, start SAP HANA studio and, in the *SAP HANA Development* perspective, perform the following steps:

- a. In the *Systems* view, add the SAP HANA system you want to work and test on.
- b. In the *Repositories* view, add a repository workspace for your *SAP HANA* system
- c. Create and share a project of type *XS Project*.

→ Tip

You can also check out and import an existing project from the SAP HANA Repository.

2. Create an XSUnit test.

XSUnit test files are XSlibrary files (files with the `.xsjslib` suffix).

- a. Create an XSlibrary file, for example, called `<MyFirstTest>.xsjslib`.

You can use the file-creation Wizard in SAP HANA studio, for example, **File** > **New** > **Other** > *SAP HANA Development* > *XS JavaScript Library File* >

- b. Add the following content to the new XSlibrary test file `<MyFirstTest>.xsjslib`.

```
/*global jasmine, describe, beforeEach, it, xit, expect*/
describe("My First Test Suite using Jasmine", function() {

    it('should show an assertion that passes', function() {
        expect(1).toBe(1);
    });
    it("should show a negative assertion", function() {
        expect(true).not.toBe(false);
    });
    it("should throw an expected error", function() {
        expect(function() {
            throw new Error("expected error");
        }).toThrowError("expected error");
    });
});
```

```
});
//xit = this test case is excluded
xit("should show an assertion that fails", function() {
    expect(1).toBe(2);
});
});
```

The JSLint tool that SAP HANA studio uses to check your XSJS code tells you that functions (for example, `describe`, `it`, `expect`) do not exist. This is not true; the functions do exist but they are defined in another library. To ensure that JSLint considers functions to be globally available, add the following comment as the first line of the XUnit test file: `/*global jasmine, describe, beforeEach, afterEach, it, xit, expect*/`

Note

You can extend the list of globally available functions to include any additional functions that you use in your test.

- c. Save the test file.
- d. Activate the test file.

In the SAP HANA studio's *SAP HANA Development* perspective, open the *Project Explorer* view, right-click the test file, and choose **Team > Activate > .**

3. Execute the XUnit test.

How you execute an XUnit test depends on the development tool suite you are using, for example, SAP HANA studio.

You execute an XUnit test by entering the following URL in a Web Browser:

```
http://<hostname>:<port>/sap/hana/testtools/unit/jasminexs/TestRunner.xsjs?
package=<packageName>
```

Where `<hostname>` is the name of the SAP HANA system where you are running your application test, and `<port>` is the port number that the SAP HANA instance is available on.

The TestRunner tool recursively searches the package `<packageName>` for any files with the suffix `.xsjslib` whose names match the pattern `"*Test"`.

Note

If you want to search for a string other than `"*Test"`, you must pass a custom pattern to TestRunner using the parameter `pattern`.

Related Information

[XUnit Test Run Options \[page 622\]](#)

9.10.3.1 XUnit's Enhanced Jasmine Syntax

The XUnit test framework is a custom version of the JavaScript test framework Jasmine adapted to suit SAP HANA XS.

A test specifications begin with a call to the global Jasmine function `describe`. The `describe` functions define suites that enable you to group together related test suites and specifications. Test-suite specifications are defined by calling the global Jasmine function `it`. You can group several test suites in one test file. The following code snippet shows one test suite (introduced by the function “`describe`”) and two test specifications, indicated by the function “`it`”.

```
/*jslint undef:true */
describe('testSuiteDescription', function() {
    beforeOnce(function() {
        // beforeOnce function is called once for all specifications
    });
    beforeEach(function() {
        // beforeEach function is called before each specification
    });
    it('testSpecDescription', function() {
        expect(1).toEqual(1);
    });
    it('anotherTestSpecDescription', function() {
        expect(1).not.toEqual(0);
    });
});
```

To enable a test suite to remove any duplicate setup and teardown code, Jasmine provides the global functions `beforeEach` and `afterEach`. As the name implies the `beforeEach` function is executed **before** each specification in the enclosing suite and all sub-suites; the `afterEach` function is called after each specification. Similarly, the special methods `beforeOnce` and `afterOnce` are called once per test suite.

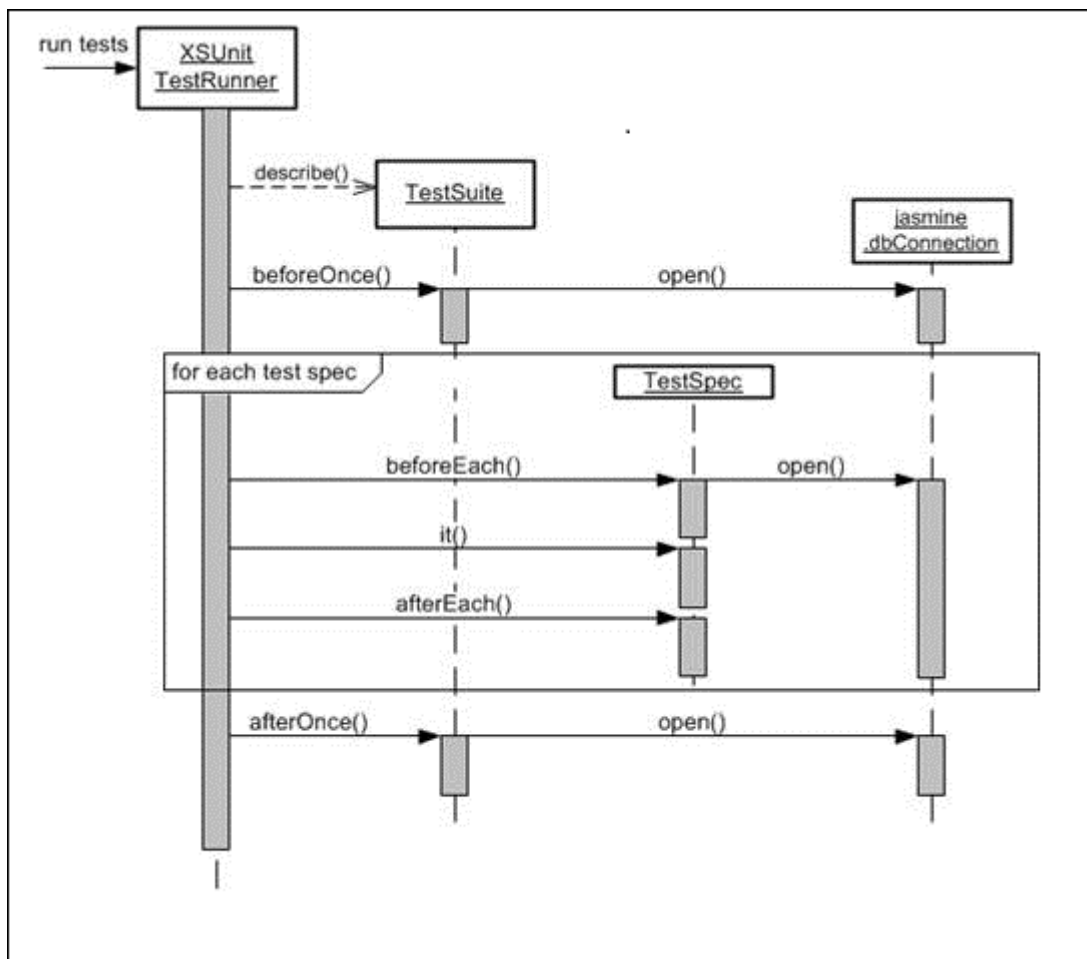
- `beforeOnce`
Executed once **before** all specifications of the test suite
- `afterOnce`
Executed once **after** all specifications of the test suite

Database Connection Setup

The XUnit framework provides a managed database connection called `jasmine.dbConnection`, which is globally available. You can use it in the following scenarios:

- Directly (in the function “`it`”)
- In the functions “`beforeEach`” and “`afterEach`”
- In other functions defined in your test libraries
- In imported libraries (if you have moved test code to external libraries)

One obvious advantage of this is that you no longer have to pass the database connection as a parameter or define it as a global variable. The `jasmine.dbConnection` is opened automatically and rolled back (and closed). However, if you want to persist your data, you have to `commit() jasmine.dbConnection` manually.



XSUnit TestRunner Tool Flow Chart

9.10.3.2 XSUnit Test Tools Syntax

Example syntax for the functions, assertions, and parameters required by the SAP HANA XSUnit test tools.

The following code example lists the most commonly used functions and assertions used in the XSUnit syntax. For more information about the assertions used, for example, `toBe`, `toBeTruthy`, or `toBeFalsy`, see *Assertions*.

```

/*global jasmine, describe, beforeEach, it, xit, expect*/
describe("My First Test Suite using Jasmine", function() {
  beforeEach(function() {
    // beforeEach is called only one time for all specs
  });
  beforeEach(function() {
    // beforeEach is called before each specs
  });
  // it = test case specification it("should show an assertion that passes",
  function() {
    var array = [{foo: 'bar', baz: 'quux'}, {bar: 'foo', quux: 'baz'}];
    expect(1).toBeTruthy();
    expect(12).toBe(jasmine.any(Number));
    expect(array).toContain(jasmine.objectContaining({foo: 'bar' }));
  });
  it("should show an negative assertion", function() {
    expect(true).not.toBe(false);
  });
});

```

```

});
it("should throw an expected error", function() {
    expect(function() {
        throw new Error("expected error");
    }).toThrowError("expected error");
});
// xit = this test case is excluded
xit("should show an assertion that fails", function() {
    expect(1).toBe(2);
});
});

```

XSUnit Assertions and Parameters

The following code example lists the most commonly used assertions, shows the required syntax, and the expected parameters.

```

expect(actual).toBe(expected);
expect(actual).toBeFalsy();
expect(actual).toBeTruthy();
expect(actual).toEqual(expected);
expect(actualArray).toContain(expectedItem);
expect(actual).toBeNull();
expect(actualNumber).toBeCloseTo(expectedNumber, precision);
expect(actual).toBeDefined();
expect(actual).toBeUndefined();
expect(actualString).toMatch(regExpression);
expect(actualFunction).toThrowError(expectedErrorMessage);
expect(actualFunction).toThrowError(expectedErrorType, expectedErrorMessage);
expect(actualTableDataSet).toMatchData(expected, keyFields);
expect(actual).toBeLessThan(expected);
expect(actual).toBeGreaterThan(expected);

```

9.10.3.3 XSUnit Test Run Options

The XSUnit tool suite includes a generic tool that you can use to run tests.

You can start the XSUnit test-running tool (`TestRunner.xsjs`) by entering the following URL in a Web Browser:

`http://<hostname>:80<HANAinstancenumber>/sap/hana/testtools/unit/jasminexs/TestRunner.xsjs?<parameters>`

The following table lists the parameters that you can use to control the behavior of test-runner tool. If you execute the test runner without specifying the `pattern` parameter, only the tests in `*Test.xsjslib` files are discovered (and run) within the package hierarchy.

📌 Note

You can specify multiple parameters by separating each `parameter=value` pair with the ampersand character (&), for example: `coverage=true&exclude=sap.hana.tests`

TestRunner.xsjs Parameters

Name	Mandatory	Description
package	yes	Package that acts as starting point for discovering the tests. If not otherwise specified by parameter "pattern" all .xsjslib files in this package and its sub-packages conforming to the naming pattern "*Test" will be assumed to contain tests and will be executed. package=sap.hana.testtools.demo
pattern	no	Naming pattern that identifies the .xsjslib files that contain the tests. If not specified, the pattern "*Test" is applied. You can use question mark (?) and asterisk (*) as wildcards to match a single or multiple arbitrary characters, respectively. To match all "Suite.xsjslib" files, use the following code: pattern=Suite
format	no	Specifies the output format the test runner uses to report test results. By default, the results will be reported as HTML document. This parameter has no effect if a custom reporter is provided via parameter "reporter". To display outputs results using the JSON format, use the following code: format=json
reporter	no	Complete path to module that provides an implementation of the Jasmine reporter interface. With this parameter a custom reporter can be passed to publish the test results in an application specific format. To specify the reporter interface, use the following code: reporter=sap.hana.testtools.unit.jasminexs.reporter.db.dbReporter Note format=db produces the same result
tags	no	Comma-separated list of tags which is used to define the tests to be executed. tags=integration, long_running
profile	no	Name of a "profile" defined in the test which filters the tests to be executed on the basis of tags. profile=end2end
coverage	no	Activate code coverage measurement for all server-side (XS) JavaScript code that is executed by the tests or which is in the scope of a specified package. coverage=true coverage=sap.hana.testtools.mockstar coverage=true&exclude=sap.hana.testtools.mockstar.tests

9.10.3.4 XSUnit Test Examples

XSUnit includes a selection of test packages that demonstrate the scope of tests you can perform on an SAP HANA XS application.

The following table lists the test packages included in the XSUnit test framework. The table also indicates the name of the test file and provides a quick overview of the scope of the test.

Note

If you want to have a look at the code in the tests, checkout the package `sap.hana.testtools.demo` as an XS project to your local workspace.

ExampleTest Units

Package Name	Test Name (.xsjslib)	Description
tests.getting_started	myFirstTest	Shows the usage of some basic Jasmine matchers as well as the usage of custom matchers <code>toMatchData</code> and <code>toEqualObject</code> that are supported by the extended Jasmine version.
tests.attribute_view_1	AT_PRODUCTS_Test	Shows how to configure mockstar in order to replace a CDS entity with a test table. Be aware that this test does not make sense, as this attribute test tests nothing at all - no logic, no joins,...
tests.graphic_calcview_1	CA_ORDERS_Test	Tests a copy of the graphical calculation view where the direct dependent tables are replaced by test tables.
tests.graphic_calcview_3	CA_OPEN_AMOUNT_Test	Tests the integration with the analytic view but replaces the dependencies to the tables with test tables. This example test shows how to upload data from a comma-separated-values (CSV) file into the test tables
tests.hdbprocedure_with_cds	CreateProductTest	Tests a non-read-only HDBProcedure with table in/out parameters while replacing the underlying Core Data Services (CDS) entities with test tables.
tests.hdbprocedure_with_hdbview	GetInvoicesTest	Tests an HDBProcedure with scalar in and view out parameters while replacing a dependent hdbview with a test table.
tests.hdbprocedure_with_hierarchy-view	HierarchyProcedureTest	Tests an HDBProcedure that includes a hierarchy view while replacing all underlying CDS entities with test tables.
tests.hdbprocedure_with_hdbprocedure	CreateProductTest	Tests an HDBProcedure while replacing a dependent hdbprocedure with an hdbprocedure that was created for testing.

Package Name	Test Name (.xsjslib)	Description
tests.http_service	whoAmIServiceTestE2E	Tests an http service and checks if it returns the expected value. This test is not automatically executed since the SAP HANA instance needs to be maintained by the system administrator.
tests.procedure_1	PR_OPEN_AMOUNT_Test	Tests a copy of the stored procedure where the directly dependent tables are replaced with test tables.
tests.scripted_calview_1	CA_ABC_PRODUCTS_Test	Tests a copy of the scripted calculation view where the directly dependent analytic view is replaced with a test table.
tests.scripted_calview_2	CA_OPEN_AMOUNT_SCRIPTED_W_PR OCEDURE_Test	Tests the integration with the called stored procedures but replaces the dependencies to the tables with test tables.
apps.rating.tests	validatorTest	Tests a simple server-side (XS) JavaScript.
	dataAccessorTest	Tests the database layer of server-side (XS) JavaScript using Jasmine spyOn() for testing in isolation.
	oDataTestE2E	Checks the accessibility of an OData service and tests an OData service without dependencies using mockstar.
	ratingServicesTestE2E	Tests an XS JavaScript service (end-to-end scenario test).
tests	myMockstarEnvironment	Shows how to enhance the mockstarEnvironment library to add further reuse functions or change the behaviour slightly to suit the context.

9.10.3.5 The Mockstar Test Environment

Mockstar is a tool that is designed to enable you to isolate SAP HANA content in tests run by an automated test suite.

To write self-contained unit tests that are executable in any system, it is essential to be able to test the selected SAP HANA objects in isolation. For a typical **unit** test using the XSUnit tools, you need to be able to change any direct dependencies between the tested objects and other views or tables with references to simple tables. For **integration** tests, rather than change the direct dependencies to a view or a table, you might need to change dependencies between the dependent views (deeper in the dependency hierarchy).

Mockstar is a tool that is specifically designed to enable you to isolate test objects, for example, a view or procedure. Mockstar allows you to create a copy of the tested view or procedure and substitute the dependency to a another view or table with a table that is stored in a test schema. It is strongly recommended to use a dedicated schema for the tests; in this test schema, you have write permissions and, as a result, full control over the data in the tables and views.

The Mockstar test-isolation tool provides the following features:

- Creates a copy of the SAP HANA object to test (for example, a view or database table); the copied object retains the same business logic as the original one object, but replaces some or all dependencies.
- Replaces the (static) dependencies to tables or views with temporary tables
- Supports deep dependency substitution
Mockstar can determine dependencies deep within a hierarchy of dependencies and copy only the necessary parts of the hierarchy.

Mockstar tools are included in the delivery unit [HANA_TEST_TOOLS](#), which you must install manually, for example, using the SAP HANA studio or the [SAP HANA Application Lifecycle Management](#) tool. After the installation completes, the Mockstar tools are available in the package `sap.hana.testtools.mockstar`.

Note

Importing a delivery unit into an SAP HANA system requires the [REPO.IMPORT](#) privilege, which is normally granted only to the system administrator.

9.10.3.6 Mockstar Environment Example Syntax

A basic example of the syntax required to set up the Mockstar test environment.

The following example shows a simple setup using standard locations.

Note

The names of schemas, tables, and views used in the following code example are intended to be for illustration purposes only.

```
var mockstarEnvironment = $.import('sap.hana.testtools.mockstar',
  'mockstarEnvironment');
describe('testSuiteDescription', function() {
  var testEnvironment = null;
  beforeEach(function() {
    var definition = {
      schema : 'SCHEMA',
      model : {
        schema : '_SYS_BIC',
        name : 'modelName' //e.g. package/MODEL
      },
      substituteTables : {
        "table" : { name : 'package::TABLE' }
      },
      substituteViews : {
        "view" : {
          schema : '_SYS_BIC',
          name : 'package/VIEW'
        }
      }
    };
    testEnvironment = mockstarEnvironment.defineAndCreate(definition);
  });
});
```

9.10.3.7 XSUnit Troubleshooting Solutions

Use trace files and other tools to fix problems with test operations.

The Mockstar test-isolation tools write helpful information in the SAP HANA trace files. You can adapt the trace level, for example, to **debug** to ensure the right amount and type of information is written during the test run. Note that you need the corresponding administration role to be able to change the trace-level settings in SAP HANA. The trace files are written in the trace component `xsa:sap.hana.testtools` (truncated to “xsa:sap.hana.tes” in the trace files).

→ Tip

As an alternative to reading the trace files directly, you can also use the SQL console to select data from the table `M_MERGED_TRACES`.

This section contains information about the problems that developers frequently encounter during test runs:

- [SAP HANA Test Tools Version \[page 627\]](#)
- [The Library is Not Part of an Application \[page 627\]](#)
- [Error for Cloned OData Service \[page 628\]](#)
- [Duplicate Entries When Inserting Test Data \[page 628\]](#)
- [Test Table Already Exists \[page 628\]](#)
- [Test Model Activation Fails \[page 629\]](#)
- [No Entries Returned From Copied Test Model \[page 629\]](#)
- [No Test Data Inserted into Test Table \[page 629\]](#)
- [TestRunner Tool Times Out \[page 630\]](#)
- [Test Model Creation is Aborted \[page 631\]](#)
- [Database Connections in XSUnit Test \[page 631\]](#)

SAP HANA Test Tools Version

Which version of the SAP HANA test tools suite is installed?

1. Start SAP HANA studio
2. Open the *SAP HANA Modeler* perspective.
3. In the *Quick Launch* window, choose *Delivery Units...*
4. Choose *HANA_TEST_TOOLS*.

Import Error: The Library is Not Part of an Application

If the test runner tool shows the following error:

```
import: the library is not part of an application
```

The JavaScript library you want to test can only be loaded when there is an application descriptor (`.xsapp` file) defined within the package hierarchy. The application descriptor is the core file that you use to describe an

application's framework within SAP HANA XS. If your tests are not part of your application package hierarchy, it is recommended you to create an `.xsapp` file in the context of the XS Project that contains the tests.

Error for Cloned OData Service

The following error message is displayed when testing access to an OData service in SAP HANA XS:

```
404 - Not found: Error for cloned OData Service (.xsodata)
```

Try the following solutions:

1. Try to access the generated service directly in a separate Web browser.
2. Check whether the file (`xsodata` service definition) exists, has been activated in the SAP HANA repository, and is in the expected target folder.
3. Ensure that the target folder or one of its parent folders contains the following activated artifacts:
 - `.xsapp` file
Application descriptor file required by an SAP HANA XS application
 - `.xsaccess` file
Application access file which enables access to an SAP HANA XS application

Duplicate Entries When Inserting Test Data

If you encounter problems concerning duplicate entries when running tests, try the following solutions:

1. When inserting records into a productive table, ensure that no `jasmine.dbConnection.commit()` call occurs during test execution.
2. When inserting records into a test table, ensure that the table entries are deleted (dropped) before they are (re)created.

```
var tableUtils = new TableUtils(jasmine.dbConnection);
tableUtils.clearTableInUserSchema(invoicesTestTable);
```

Test Table Already Exists

You encounter an error message that explains that a test table cannot be created during the test because the table already exists. You must ensure that the specified table is deleted before the test tries to create it during the test run.

```
var sqlExecutor = new SqlExecutor(jasmine.dbConnection);
var createTableString = 'CREATE COLUMN TABLE ' + <table name> + '...' );
sqlExecutor.execSingleIgnoreFailing('drop table ' + <table name> );
sqlExecutor.execSingle(createTableString );
```

You can also use the functions provided by the table utilities library, which enables you to ensure that the table is dropped at the right time:

```
var tableUtils = new TableUtils(jasmine.dbConnection);
testTable = tableUtils.copyIntoUserSchema(originSchema, originTable);
```

Test Model Activation Fails

Your test produces an error relating to a failed activation:

```
Error: Repository: Activation failed for at least one object [...]
identifier is too long:[...]
Maximum length is 127: ...
```

the name of the model is too long (including the package name). You can reduce the name by setting the `TruncOptions` option as shown in the following code snippet:

```
var mockstar = $.sap.hana.testtools.mockstar;
testView = mockstar.apiFacade.createTestModel(originalModel,
    targetPackage, dependencySubstitutions, mockstar.TruncOptions.FULL);
```

→ Tip

It's a good idea to analyze the created model before it is activated.

To generate a detailed and structured error log, in the SAP HANA [Systems](#) view in the SAP HANA studio and locate the test package and activate it manually.

No Entries Returned From Copied Test Model

1. Open the generated test model.
The generated model is located in a package with the name `tmp.unittest.<userName>.<originalPackage>`. If you have configured the `createTestModel()` function with the parameter `mockstar.TruncOptions.FULL`, the package name is `tmp.unittest.<userName>`.
2. Ensure that the dependencies have been replaced as expected.
To see if the tables are filled correctly by the test, see [No Test Data Inserted into Test Table \[page 629\]](#).
3. Check the test view itself.
If the tested view returns no data, but data are expected, check if the data are removed by a filter during extraction from the underlying data source.

No Test Data Inserted into Test Table

To test whether a test inserts data as expected into the created test table, implement a `jasmine.dbConnection.commit()` connection to ensure that the data created during the test is stored

persistently. Without the `jasmine.dbConnection.commit( )`, the test data is not persistent; the test deletes all test data when the database session is closed. Start the test again using the TestRunner tool. When the test completes, the test table should contain test data.

TestRunner Tool Times Out

The default timeout setting for the TestRunner tool is ten (10) minutes. If your test run for longer than ten minutes and cause a timeout, try splitting the test into smaller and shorter elements. If this is no possible, try running the test in three phases:

1. Prepare the test run.

`/sap/hana/testtools/unit/jasminexs/PrepareTestRun.xsjs`

This generates a new test-run ID; no test runs are executed.

- Response:
Returns the new test-run ID. If you request the answer in HTML format and provide all required parameters for the TestRunner tool, you receive the appropriate links you can use in the following steps (run the test and fetch the results).
- Parameters:
`format` (optional; default = "html")
Set this parameter to receive the test-run ID in the desired format. You can use any of the formats supported by the TestRunner `format` parameter.

2. Run the tests.

`/sap/hana/testtools/unit/jasminexs/TestRunner.xsjs`

This step is almost identical to the usual test execution with the addition of parameter `runid`.

- Response:
If the tests finish within the configured time frame, you receive the test results as expected. If the test are too long, a timeout occurs.
- Parameters:
`runid`. Required for this kind of (manual) execution: This is the test-run ID generated in the previous step.

3. Fetch the test results (optional: only required if the test run causes a timeout).

`/sap/hana/testtools/unit/jasminexs/GetTestResults.xsjs`

Fetches the test results for a given test-run ID. You can run this service multiple times for each test.

- Response:
Returns the test results in the requested format. If the tests are not yet finished, you receive a status message (either "PREPARED" or "STARTED"). If the run ID provided does not exist, an error message is displayed.
- Parameters:
`runid`. Required for this kind of (manual) execution.
`format` (optional; default = "html")

Test Model Creation is Aborted

This error sometimes occurs if you try to create a copy of the original view and replace some dependencies with test tables. The reason for the error is one of the following:

- You did not provide any dependency substitutions. For example, you passed an empty array as the third parameter of `mockstar.createTestModel()`.
- The view that you want to test does not depend on any of the original views specified in the dependency substitutions.
- For active schema mapping, you have written the dependencies with the **physical** schema whereas the view refers to the **authoring** schema. Provide the schema in the same way as it is written in the view (or stored procedure).

Database Connections in XUnit Test

The XUnit test framework provides a new “managed” database connection called `jasmine.dbConnection`, which is automatically opened and rolled back (and closed) after each test completes. You can use it in `beforeEach` or `afterEach` functions, in other functions defined in your test libraries, or even in imported libraries, in the event that you have moved test code into external libraries.

Related Information

[Managed Database Connection Setup \[page 620\]](#)

9.10.4 Testing JavaScript with XUnit

Test an XS JavaScript using XUnit test tools.

As the XUnit test tools are based on a custom version of the JavaScript test framework Jasmine, you can use XUnit to test JavaScript. XUnit provides tools that enable you to create and install a test “double” for one or more object methods. In the Jasmine framework, a test double is known as a “spy”. A spy can be used not only to stub any function but also to track calls to it and all arguments, too.

Note

XUnit includes special *matchers* that enable interaction with Jasmine spies.

The XUnit test tools delivery unit (DU) includes a small XS JavaScript demo “Ratings” application which comprises an SAPUI5 client front end on top of OData and XS JavaScript services; the Ratings application enables you to experiment with different test techniques. You can try out the application at the following URL:

```
http://<SAPHANA_host>:80<instancenumber>/sap/hana/testtools/demo/apps/rating/  
WebContent/
```


Related Information

[XUnit's Jasmine Spy Syntax \[page 632\]](#)

[Testing HTTP Services with XUnit \[page 633\]](#)

9.10.4.1 XUnit's Jasmine Spy Syntax

A command “cheat sheet” for the Jasmine Spy syntax.

The following code example provides a quick overview of commonly used commands that enable the use of Jasmine Spies. You can see how to perform the following actions:

- [Install a method double \[page 632\]](#)
- [Install an object double \[page 632\]](#)
- [Check a function call \(and values\) \[page 633\]](#)

Installing a Method Double

The following code example shows how install a method double (simple example).

```
spyOn(object, "method");  
expect(object.method).toHaveBeenCalled();
```

The following code example shows how install a method double (variant).

```
var spyMethod = spyOn(object, "method");  
expect(spyMethod).toHaveBeenCalled();
```

The following code example shows how install a method double (custom action for double).

```
spyOn(object, "method"); // delegates nowhere  
spyOn(object, "method").and.returnValue(3); // returns constant value  
spyOn(object, "method").and.callThrough(); // delegates to original function  
spyOn(object, "method").and.callFake(fakeFunction); // delegates to other  
function
```

Installing an Object Double

The following code example shows how install an object double.

```
var spyObject = jasmine.createSpyObj("spy name", [ "method1", "method2",  
"method3" ] );  
spyObject.method1.and.returnValue(3);  
expect(spyObject.method1).toHaveBeenCalled();
```

Checking Function Calls (and Values)

The following code example shows how to check whether the function has been called as expected, and if so, if the right values were used.

```
expect(spyObject.method).toHaveBeenCalled();
expect(spyObject.method).toHaveBeenCalledWith(expArgValue1, expArgValue2);
expect(spyObject.method.calls.allArgs()).toContain([ expArgValue1,
expArgValue2 ]);
expect(spyObject.method.calls.mostRecent().args).toEqual([ expArgVal1,
expArgVal2 ]);
expect(spyObject.method.calls.count()).toBe(2);
spyObject.method.calls.reset(); // reset all calls
```

9.10.4.2 Testing HTTP Services with XUnit

XS JavaScript files that can be accessed by performing an HTTP call against the service defined in the XS JavaScript file.

You can use the `TestRunner` tool to call an XS JavaScript service. The `TestRunner` service is part of the `test-tools` package `sap.hana.testtools.unit.jasminexs` and has one mandatory parameter, namely `package`. Since `TestRunner` is an HTTP GET service, you can execute the service in the browser using the following URL:

```
http://<hostname>:80<instancenumber>/sap/hana/testtools/unit/jasminexs/
TestRunner.xsjs?package=<mypackage>
```

Since it is not possible to import XS Javascript files (`.xsjs`) files into a JavaScript library (`.xsjslib`), the functions you implement inside the XS JavaScript file cannot be tested within an XUnit test. As a consequence, it is recommended to include only minimal logic within the XSJS files and delegate tasks to the functions implemented in corresponding JavaScript libraries; these libraries can be tested in isolation using XUnit tools (for example, `Mockstar`).

Note

XUnit enables you to perform an HTTP call to your XSJS services via HTTP. However, this is an end-to-end system test with no possibility to use test doubles during the test. These tests are not suitable for testing a JavaScript function.

Since you cannot insert test data into the test table during the test, the tests have no control over the data. This restriction reduces the scope of the tests you can perform for HTTP calls, for example, you can test the following scenarios:

- Service must return an error if mandatory parameters are missing
- Service must return an error if the chosen HTTP type is correct
- Service must return an error if the wrong input is provided
- End-to-end HTTP scenarios (CREATE, READ, UPDATE, and DELETE)

```
describe("example for http tests", function() {
  it("should receive answer from service", function() {
    var requestBody = '{"param1":42,"param2":"xyz"}';
    var headers = {
```

```

        "Content-Type" : "application/json"
    };
    var response = jasmine.callHTTPService("/path/to/your/app/Service.xsjs",
$.net.http.POST, requestBody, headers);
    expect(response.status).toBe($.net.http.OK);
    var body = response.body ? response.body.asString() : "";
    expect(body).toMatch(/regular expression that checks correct response/);
  });
});

```

SAP HANA Database Logon for XUnit

To ensure access to SAP HANA, you need to adapt the default HTTP destination file (`localhost.xshttpdest`) provided with the XUnit test tools. The default HTTP destination configuration file is located in `sap.hana.testtools.unit.jasminexs.lib:localhost.xshttpdest` to fit to your HANA instance. To access an HTTP destination configuration, you need the permissions granted in the user role `sap.hana.xs.admin.roles::HTTPDestAdministrator`.

⚠ Caution

To change the HTTP destination, create an HTTP **extension** of your own; do not make any changes to the file `localhost.xshttpdest`. Changes to `localhost.xshttpdest` are overwritten by updates to the XUnit test tools on your system.

Related Information

[Maintaining HTTP Destinations \[page 112\]](#)

9.10.4.3 Testing JavaScript Functions with XUnit

Use XUnit tools to test JavaScript code that depends on functions in your code, for example: dependencies on functions, libraries, or to database tables.

In JavaScript it is possible to overwrite anything that is visible in a context, for example: public data, public functions, or even the whole class. With XUnit, you can make use of a simulation framework that is included with Jasmine. The simulation framework provides a mechanism that enables you to create and install a test double (so-called Jasmine “Spy”), which can help you to reduce some of the basic code and keep the code more concise. Jasmine Spies should be created in the test setup, before you define any expectations. The Spies can then be checked, using the standard Jasmine expectation syntax. You can check if a Spy is called (or not) and find out what (if any) parameters were used in the call. Spies are removed at the end of every test specification.

📌 Note

Each dependency increases the complexity of testing involved for a function or a component.

The Average Component Dependency (ACD) is the number of dependencies to other components, averaged over all components; it indicates whether your system is loosely coupled. If you prefer to implement JavaScript in an object-oriented way, you can apply dependency management aspects by following object-oriented design principles (OOD).

The information in this topic covers the following test scenarios:

- [Dependencies on Function Libraries \[page 635\]](#)
- [Dependency on Database Table \[page 636\]](#)

Dependencies on Function Libraries

The following code snippet defines a controller that you want to test; the controller depends on a `Date` object. The accompanying code snippet shows how you can test this code.

```
var Controller = null;
(function() {
    //constructor function
    Controller = function(dataModel) {
        this.model = dataModel;
    };
    function updateModelWithTimestamp(newData) {
        this.model.updateData(newData, this.getCurrentDate());
    }
    Controller.prototype.updateModel = function(newData) {
        //bind 'this' to the private function
        updateModelWithTimestamp.call(this, newData);
    };
    Controller.prototype.getCurrentDate = function() {
        return new Date(Date.now());
    };
})();
function DataModel() {
    var modifiedAt = null;
    var modifiedBy = null;
    var data = null;
    this.updateData = function(newData, modifiedAtDate) {
        data = newData;
        modifiedAt = modifiedAtDate;
        modifiedBy = $.session.getUsername();
    };
    this.getModificationDate = function() {
        return modifiedAt;
    };
}
```

The following code snippets shows an example of the test code you could run; the code uses a Jasmine Spy ensures the dependencies on the `Date` object are replaced and tested as expected.

```
var Controller = $.import("sap.hana.testtools.demo.objects.xs_javascript",
"javascript00").Controller;
var DataModel= $.import("sap.hana.testtools.demo.objects.xs_javascript",
"javascript00").DataModel;
describe('Controller', function() {
    var controller = null;
    var model = null;
    var anyDate = new Date(2013, 8, 27, 11, 0, 0, 0);

    beforeEach(function() {
        model = new DataModel();
```

```

        controller = new Controller(model);
    });
    it('should set current date when data is modified (replace Date.now() using
jasmine spies)', function() {
        spyOn(Date, 'now').and.returnValue(anyDate.getTime());
        oController.updateModel({data : [1,2,3]});
        expect(model.getModificationDate()).toEqual(anyDate);
    });
});

```

Dependency on Database Table

It is important to try to avoid mixing business logic that is implemented in JavaScript with the data base interaction. We recommend moving the database persistency logic into a dedicated persistency class, so that just the business logic remains for testing. The goal of the test is to be able to test both normal and special cases without interacting with the data base at all.

To unit test the persistency class, you can parameterize the schema and use a schema for testing, for example, the user schema where you have all authorizations required to create, modify, and drop objects, and cannot mess things up with the test. Last of all, you can offer a small set of integration tests, that just ensure that the productive classes, the `AnyService` class, and the `Persistency` class, integrate well.

Note

For sake of conciseness, resource closing and error handling is missing from the following code example.

```

function Persistency(dbConnection, schema) {
    var dbSchema = schema !== undefined ? schema : 'SAP_HANA_TEST_DEMO';
    this.existsEntry = function(key) {
        var pstmt = dbConnection.prepareStatement('SELECT key FROM "' +
dbSchema + '"."Table" WHERE KEY=?');
        pstmt.setString(1, key);
        if (pstmt.executeQuery().next()) {
            return true;
        }
        return false;
    };
    this.insertEntry = function(newEntry) {
        var pstmt = dbConnection.prepareStatement('INSERT INTO "' + dbSchema +
'"."Table" VALUES(?,?)');
        pstmt.setString(1, newEntry.Id);
        pstmt.setString(2, newEntry.Value);
        pstmt.execute();
    };
}
function AnyService(persistency) {
    this.execute = function(input) {
        //validate input
        if (!persistency.existsEntry(input.Id)) {
            //calculate newEntry
            persistency.insertEntry(newEntry);
        }
    };
}

```

The following code snippets shows an example of the test code you could run to test the dependencies.

```

var Persistency = $.import("package.of.persistency", "persistency").Persistency;

```

```

describe('Persistency test', function() {
    var SqlExecutor = $.import('sap.hana.testtools.unit.util',
    'sqlExecutor').SqlExecutor;
    var TableUtils = $.import('sap.hana.testtools.unit.util',
    'tableUtils').TableUtils;
    var originTable = 'TableName';
    var testTable = null;
    var originSchema = 'SAP_HANA_TEST_DEMO';
    var userSchema = $.session.getUsername().toUpperCase();
    beforeOnce(function(){
        var tableUtils = new TableUtils(jasmine.dbConnection);
        testTable = tableUtils.copyIntoUserSchema(originSchema, originTable);
    });
    it('should insert one entry into table', function() {
        var persistency = new Persistency(jasmine.dbConnection, userSchema);
        persistency.insertEntry({ Id : '0815', Value : 1});
        expect(persistency.existsEntry('0815'));
        expect(selectAllFromTable().getRowCount()).toBe(1);
    });
    function selectAllFromTable() {
        var sqlExecutor = new SqlExecutor(jasmine.dbConnection);
        return sqlExecutor.executeQuery('select * from ' + testTable);
    }
});

```

Testing a Self-Contained JavaScript Function

The following code snippet show how to use XUnit to test a self-contained JavaScript function (**mathlib**); a self-contained function has no dependencies to other JavaScript functions, database tables or session parameters.

```

var mathlib = $.import("package.of.your.library", "math");
describe('The math XS JavaScript library', function() {
    it('should calculate "7" as maximum value of "3, 7"', function() {
        var maxValue = mathlib.max(3, 7);
        expect(maxValue).toBe(7);
    });
    it('should calculate "-10" as maximum value of "-10, -20"', function() {
        var maxValue = mathlib.max(-10, -20);
        expect(maxValue).toBe(-10);
    });
});

```

10 Building UIs

10.1 Building User Interfaces with SAPUI5 for SAP HANA

UI development toolkit for HTML5 (SAPUI5) is a user-interface technology that is used to build and adapt client applications based on SAP HANA. You can install SAPUI5 and use it to build user interfaces delivered by SAP HANA's Web server.

The SAPUI5 run time is a client-side HTML5 rendering library with a rich set of standard and extension controls. The SAPUI5 run time provides a lightweight programming model for desktop as well as mobile applications. Based on JavaScript, it supports Rich Internet Applications (RIA) such as client-side features. SAPUI5 complies with OpenAjax and can be used with standard JavaScript libraries.

SAPUI5 Demo Kit

The SAPUI5 Demo Kit contains the following components:

- [Documentation](#)
Information about the programming languages used, open source technology, development tools, and API usage. You can also find tutorials to help you get started and details about the new features delivered with each version of SAPUI5.
- [API reference](#)
The complete JavaScript documentation for the Framework and Control API, including featured namespaces such as `sap.m` (main controls), `sap.ui.layout` (layout controls), `sap.ui.table` (table controls), `sap.f` (SAP Fiori), and `sap.ui.core` (UI5 core run time).
- [Samples](#)
A detailed view of almost every control available in the kit, including detailed information about featured controls such as: user interaction elements, lists, tables, pop-up dialogs tiles, messages, maps and charts, smart controls, step-based interaction, and so on. You can also find detailed information about object pages, dynamic pages, and how to use flexible columns.
- [Demo Apps](#)
A selection of apps for download that are designed to showcase UI5 concepts and controls in real-life scenarios
- [Tools](#)
SAPUI5 comes with a built-in set of icons for use in your applications. The [Icon Explorer](#) helps you to find icons by allowing you to browse through categories or simply start a search. You can see a live preview of how the icon looks when being applied to a SAPUI5 control. You can also add icons to your favorites list to make them easier to find at a later stage.
In addition, you can download development tools such as as well as the *UI5 Inspector* which you can use for debugging and supporting both OpenUI5 and SAPUI5 applications.

Related Information

[SAPUI5 Demo Kit](#)

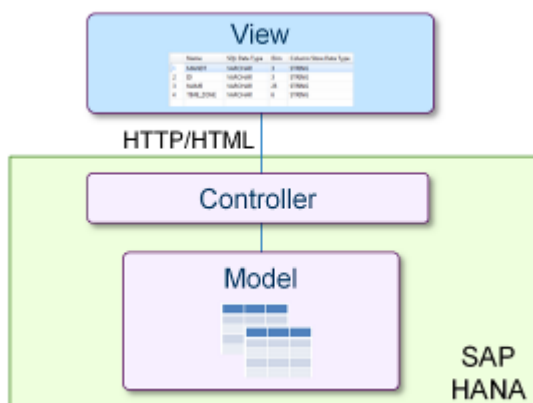
10.2 Consuming Data and Services with SAPUI5 for SAP HANA

SAP HANA Extended Application Services (SAP HANA XS) can be used to expose the database data model, with its tables, views and database procedures, to UI clients.

You can expose an SAP HANA model using OData services or by writing native server-side JavaScript code that runs in the SAP HANA context. You can also use SAP HANA XS to build dynamic HTML5 client applications, for example, using SAPUI5 for SAP HANA.

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

- **View**
UI rendering occurs completely in the client (SAPUI5, browser, mobile applications)
- **Controller**
Procedural (control-flow) logic is defined in (XS) JavaScript, SQLScript or an OData service
- **Model**
All application artifacts are stored in SAP HANA



SAP HANA Application Development with SAP HANA XS

Each of the levels illustrated in the graphic (view, control, model) is manifested in a particular technology and dedicated languages. After you have defined the data model with design-time artifacts and the equivalent run-time objects, you develop the control-flow logic to expose the data, for example, using server-side JavaScript or an OData service. With the data model and control-flow logic in place, you can build the presentation logic to view the exposed data in a UI client application using SAPUI5 for SAP HANA. For example, you can use an SAPUI5 client to request and display data exposed by an OData service; the UI could include buttons that trigger operations performed by SAP HANA XS JavaScript service; and the data displayed is retrieved from data end points defined in your data model, for example, SQLScript or CDS.

Related Information

[SAPUI5 Demo Kit \(version 1.32.7\)](#)

10.3 SAPUI5 for SAP HANA Development Tutorials

Tutorials are designed to extend task-based information to show you how to use real code and examples to build native SAP HANA applications. The tutorials provided here include examples of how to build simple SAPUI5 applications.

The tutorials provided here show you how to create your own simple SAPUI5-based applications. Some of the tutorials make use of sample data, design-time development objects, and functions provided by the SAP HANA Interactive Education (SHINE) demo application, for example: database tables, data views, server-side JavaScript (XSJS) and OData services, and user-interface elements.

Note

If the SHINE DU (HCODEMOCONTENT) is not already installed on your SAP HANA system, you can download the DU from the SAP Software Download Center in the SAP Support Portal at <http://service.sap.com/swdc>. On the [SAP HANA PLATFORM EDIT. 1.0](#) Web page, locate the download package ► [SAP HANA DEMO MODEL 1.0](#) ► [# OS independent](#) ► [SAP HANA database](#) ►.

The tutorials provided here cover the following areas:

- SAPUI5 clients
 - Hello world
Build a simple “Hello World” application using SAPUI5 tools; the exercise shows how the development process works and which components are required.
- Consuming Server-side JavaScript (XSJS) services with SAPUI5
Build an SAPUI5 application that calls an XSJS service in response to user interaction with the user interface, for example, clicking a button to perform an action. In this case, the XSJS service called by the UI request performs an action and returns a response, which is displayed in the SAPUI5 client.
- Consuming OData services with SAPUI5
Build an SAPUI5 application that calls an OData service in response to user interaction with the user interface, for example, clicking a graph or report chart. In this case, the OData service called by the UI request performs an action (collects data) and returns a response, which is displayed in the SAPUI5 client.
 - Bind a UI element in an SAPUI5 application to the data specified in an OData service. For example, you can populate the contents of a table column displayed in an SAPUI5 application by using the data stored in a database table defined in an OData service.
 - Build an SAPUI5 view that provides input fields, which you can use to create a new record or update an existing record in a database table, for example, using the OData create, update, and delete (CRUD) features.
- Localizing UI Strings in SAPUI5
Create a simple text-bundle file for translation purposes and re-import the translated text into SAP HANA for use with a specific language locale. Textbundles containing text strings that define elements of the user-interface (for example, buttons and menu options).

Related Information

[SAPUI5 Demo Kit \(version 1.28\)](#)

[Tutorial: Create a Hello World SAPUI5 Application \[page 641\]](#)

[Tutorial: Consume an XSJS Service from SAPUI5 \[page 645\]](#)

[Tutorial: Consume an OData Service from SAPUI5 \[page 651\]](#)

[Tutorial: Consume an OData Service with the CREATE Option \[page 658\]](#)

[Tutorial: Create and Translate Text Bundles for SAPUI5 Applications \[page 665\]](#)

10.3.1 Tutorial: Create a Hello-World SAP UI5 Application

SAPUI5 provides a client-side HTML5 rendering library with a comprehensive set of standard controls and extensions that you can use to build a UI quickly and easily.

Prerequisites

To complete this tutorial successfully, bear in mind the following requirements:

- You have installed the SAP HANA studio.
- You have installed the SAPUI5 tools included in the delivery unit (DU) [SAPUI5_1](#).

Context

SAPUI5 application development tools provides wizards to help you to create application projects and views according to the model-controller-view concept. The development tools include features such as editors with JavaScript code-completion, templates and code snippets, and application previews. To create a simple “Hello World” application in SAPUI5, perform the following steps:

Procedure

1. Create a base structure for your application packages and files.

Your application files must be placed in a package structure in the SAP HANA Repository, for example `/workshop/session/ui/HelloWorld/`.

2. Create the application-descriptor files that enable client access to the services and data exposed by the new application.

Each SAP HANA XS application requires two **mandatory** application descriptor files, which are located in the root package of the application they apply to.

Note

Application descriptors have a file extension, but no file name, for example, `.xsapp` or `.xsaccess`.

- a. In the SAP HANA studio's *Project Explorer* view, right-click the application package where you want to create the new application descriptors and, in the popup menu, choose, ► *New* ► *Other...* ►
- b. Create the XS application descriptor file (`.xsapp`).

In the *Select a Wizard* dialog, choose ► *SAP HANA* ► *Application Development* ► *XS Application Access File* ►.

→ Tip

The application descriptor has no content; its job is to mark the root package of the resources exposed to client requests by the application.

- c. Create the XS application-access file (`.xsaccess`).

In the *Select a Wizard* dialog, choose ► *SAP HANA* ► *Application Development* ► *XS Application Access File* ►.

→ Tip

The `.xsaccess` file controls who has access to the application (and how) and what data or services the application can expose.

- d. Select a template to use for the application-access file (for example, *Basic*) and choose *Finish*.

A basic `.xsaccess` file looks like the following example, which exposes your application data, requires logon credentials for authentication, and helps to prevent cross-site request-forgery (XSRF) attacks:

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form"},
  "prevent_xsrp" : true
}
```

- e. Activate the XS application descriptor files in the SAP HANA Repository.

Right-click the package containing the application descriptor files you have created and, in the context-sensitive menu, choose ► *Team* ► *Activate* ►.

You now have a basic package structure to hold your application files. The root package for your new application also contains the required application descriptors, which control access to the services and data exposed by the new application.

3. Create an SAPUI5 project.

- a. Start the *New Application Project* wizard.

In the SAP HANA studio's *Project Explorer* view, choose ► *New* ► *Other...* ►

- b. Select the application project.

SAP HANA studio provides dedicated wizards to help you set up an application project; here you choose the project ► *SAPUI5 Application Development* ► *Application Project* ► in the *New Project* wizard.

- c. Define details of the new project.

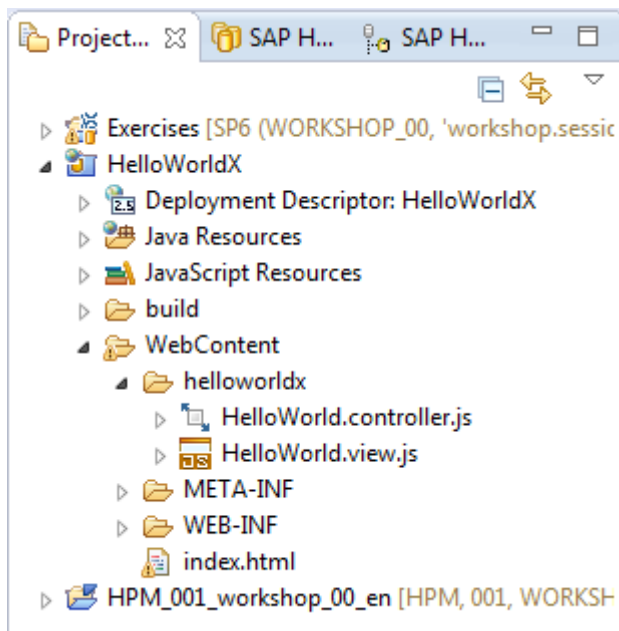
- Enter a name for the new SAPUI5 application project, for example, **HelloWorldX**.

- Check the [Use default location](#) option.
- d. Define details of the new SAPUI5 application view and choose [Finish](#).
 - Check the folder for the project; it should be `WebContent/helloworld`.
 - Provide a name for the base HTML page that the SAPUI5 application uses, for example, **HelloWorld**.
 - Choose [JavaScript](#) as the [Development Paradigm](#).

Note

If prompted, do not switch to the Java EE perspective.

You now have an Eclipse project with a bootstrap HTML (`index.html`) page in the `WebContent` folder and a `HelloWorld` controller and `HelloWorld` view in an sub-package called `helloworld`.



4. Share the new SAPUI5 project with the SAP HANA Repository.
 - a. Choose the appropriate repository **type**, for example, [SAP HANA Repository](#).
 - b. Specify the location in the SAP HANA repository where the new SAP UI5 application project should reside.
 In the [Share Project](#) wizard, choose [Browse...](#) to select the package in which you want to store the new SAPUI5 application artifacts.
 - c. Check the settings you made for the new SAPUI5 application project.
 - d. Activate the new SAPUI5 application project.

Note

Activate at the project level to ensure that all project artifacts are created and stored in the SAP HANA repository.

5. Modify the default settings for the SAPUI5 bootstrap location in the base `index.html`.
 The SAPUI5 project wizard inserts a default bootstrap location to the `index.html` file which is incorrect for SAP HANA. You must manually change the bootstrap location in the SAPUI5 application's `index.html`

file by adding `/sap/ui5/1` to the start of the default location `resources/sap-ui-core.js`, as illustrated in the following example:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <script src="/sap/ui5/1/resources/sap-ui-core.js"
            id="sap-ui-bootstrap"
            data-sap-ui-libs="sap.ui.commons"
            data-sap-ui-theme="sap_goldreflection">
    </script>
```

6. Add UI elements to the SAPUI5 application interface.

You define UI elements in the `createContent` function section of the `HelloWorld.view.js` file. In this example, you instantiate the `Button` UI element class as `myButton` and then return it at the end of the `createContent` function. The SAPUI5 application renders any UI element (or element group) returned from the `createContent` function function.

```
sap.ui.jsview("helloworldx.HelloWorld", {
  /** Specifies the Controller belonging to this View.
   * In the case that it is not implemented, or that "null" is returned,
   * this View does not have a Controller.
   * @memberOf helloworldx.HelloWorld
   */
  getControllerName : function() {
    return "helloworldx.HelloWorld";
  },
  /** Is initially called once after the Controller has been instantiated.
   * It is the place where the UI is constructed.
   * Since the Controller is given to this method, its event handlers can be
   * attached right away.
   * @memberOf helloworldx.HelloWorld
   */
  createContent : function(oController) {
    var myButton = new sap.ui.commons.Button("btn");
    myButton.setText("helloworld");
    myButton.attachPress(function(){$("#btn").fadeOut();});
    return myButton;
  }
});
```

7. Save and activate all changes to all SAPUI5 application artifacts.

Note

Activate at the project level to ensure that the changes made to **all** project artifacts are updated in the SAP HANA repository.

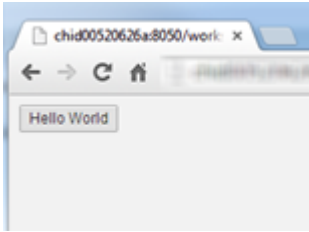
8. Test your “Hello World” SAPUI5 application in a Web browser.

The URL for the SAPUI5 application is: `http://<WebServerHost>:80<SAPHANAinstance>/workshop/session/ui/HelloWorld/WebContent/`.

Note

The content of the URL is case sensitive. Log on using your SAP HANA user name and password.

You should see the *Hello World* button shown in the following example:



10.3.2 Tutorial: Consume an XSJS Service from SAPUI5

An XS server-side JavaScript (XSJS) application can be used to perform an action linked to an element such as a button or a text box in an SAPUI5 application.

Prerequisites

To complete this tutorial successfully, bear in mind the following requirements:

- You have installed the SAP HANA studio.
- You have installed the SAPUI5 tools included in the delivery unit (DU) [SAPUI5_1](#).
- You have installed the [SHINE](#) (democontent) delivery unit; this DU contains the XSJS service you want to consume with the SAPUI5 application you build in this tutorial.
- You have generated data to populate the tables and views provided by the [SHINE](#) delivery unit and used in this tutorial. You can generate the data with tools included in the [SHINE](#) delivery unit.

Note

You might have to adjust the paths in the code examples provided to suit the folder/package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration tables and services) referenced in the tutorial.

Context

You can configure an SAPUI5 application to call an XSJS service in response to user interaction with the UI; the XSJS service performs an action and returns a response. This tutorial demonstrates how to trigger an XSJS service which performs a mathematical multiplication when numbers are typed in text boxes displayed in an SAPUI5 application.

Procedure

1. Create an SAPUI5 project.

- a. Start the *New Application Project* wizard.

In the SAP HANA studio's *Project Explorer* view, choose **New > Other...**

- b. Select the application project.

SAP HANA studio provides dedicated wizards to help you set up an application project; here you choose the project **SAPUI5 Application Development > Application Project** in the *New Project* wizard.

- c. Define details of the new project.

- Enter a name for the new SAPUI5 application project, for example, **xsjsMultiply**.
- Check the *Use default location* option.

- d. Define details of the new SAPUI5 application view and choose *Finish*.

- Provide a name for the base HTML page that the SAPUI5 application uses, for example, **xsjsMultiply**.
- Choose *JavaScript* as the *Development Paradigm*.

Note

If prompted, do not switch to the Java EE perspective.

You now have an Eclipse project for the new SAPUI5 application. The SAPUI5 application project has a bootstrap HTML page in the **WebContent** folder and an **xsjsMultiply** controller (and a view) in the sub-package **xsjsMultiply**.

2. Create the application-descriptor files that enable client access to the services and data exposed by the new application.

Each SAP HANA XS application requires two **mandatory** application descriptor files, which are located in the root package of the application they apply to. If the application-descriptor files already exist (for example, because they are created as part of the new-application Wizard), you can safely skip this step.

Note

Application descriptors have a file extension, but no file name, for example, **.xsapp** or **.xsaccess**.

- a. In the SAP HANA studio's *Project Explorer* view, right-click the application package where you want to create the new application descriptors and, in the popup menu, choose, **New > Other...**
- b. Create the XS application descriptor file (**.xsapp**).

In the *Select a Wizard* dialog, choose **SAP HANA > Application Development > XS Application Access File**.

Tip

The application descriptor has no content; its job is to mark the root package of the resources exposed to client requests by the application.

- c. Create the XS application-access file (**.xsaccess**).

In the *Select a Wizard* dialog, choose **SAP HANA > Application Development > XS Application Access File**.

→ Tip

The `.xsaccess` file controls who has access to the application (and how) and what data or services the application can expose.

- d. Select a template to use for the application-access file (for example, [Basic](#)) and choose [Finish](#).

A basic `.xsaccess` file looks like the following example, which exposes your application data, specifies that logon credentials are required for authentication, and helps to prevent cross-site request-forgery (XSRF) attacks:

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form"},
  "prevent_xsrp" : true
}
```

- e. Activate the XS application descriptor files in the SAP HANA Repository.

Right-click the package containing the application descriptor files you have created and, in the context-sensitive menu, choose **Team > Activate**.

You now have a basic package structure to hold your application files. The root package for your new application also contains the required application descriptors, which control access to the services and data exposed by the new application.

3. Share the new SAPUI5 project with the SAP HANA Repository.

In the SAP HANA studio's [Project Explorer](#) view, right-click the new SAPUI5 application project, and choose **Team > Share Project...**

- a. Choose the appropriate repository **type**, for example, [SAP HANA Repository](#).
b. Specify the package location in the SAP HANA repository where the new SAP UI5 application project should reside.

In the [Share Project](#) wizard, choose [Browse...](#) to select the package in which you want to store the new SAPUI5 application artifacts. Select the `ui` package in the SAPUI5 folder hierarchy.

- c. Check the settings you made for the new SAPUI5 application project.
d. Activate the new SAPUI5 application project.

In the SAP HANA studio's [Project Explorer](#) view, right-click the new SAPUI5 application project, and choose **Team > Activate**.

→ Tip

Remember to activate at the project level to ensure that all project artifacts are created and stored in the SAP HANA repository.

4. Modify the default settings for the SAPUI5 bootstrap location in the base SAPUI5 `index.html`.

The SAPUI5 project wizard inserts a default bootstrap location into the `index.html` file which is incorrect for SAP HANA. You must manually change the bootstrap location in the SAPUI5 application's `index.html` file by adding `/sap/ui5/1` to the beginning of the default path defined in the `script src=` tag, for example, `script src="/sap/ui5/1/resources/sap-ui-core.js` as illustrated in the following example:

Note

You must also declare any additional libraries you want the SAPUI5 application to use to render the data it consumes. For this tutorial, add `sap.ui.table` to the list of SAPUI5 libraries, as shown in the following example.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <script src="/sap/ui5/1/resources/sap-ui-core.js"
      id="sap-ui-bootstrap"
      data-sap-ui-libs="sap.ui.commons,sap.ui.table"
      data-sap-ui-theme="sap_goldreflection">
    </script>
```

5. Set up the SAPUI5 view displayed in the application user interface.

The SAPUI5 view for this tutorial is specified in the file `xsjsMultiply.view.js`; it displays a simple UI with two text boxes that you can use to specify the numbers to use for the multiplication action.

```
sap.ui.jsview("xsjsmultiply.xsjsMultiply", {
  getControllerName : function() {
    return "xsjsmultiply.xsjsMultiply";
  },
  createContent : function(oController) {
    var multiplyPanel = new sap.ui.commons.Panel().setText("XS Service
Test - Multiplication");

    var layoutNew = new
sap.ui.commons.layout.MatrixLayout({width:"auto"});
    multiplyPanel.addContent(layoutNew);
    var oVal1 = new sap.ui.commons.TextField("val1",{tooltip: "Value
#1", editable:true});
    var oVal2 = new sap.ui.commons.TextField("val2",{tooltip: "Value
#2", editable:true});
    var oResult = new sap.ui.commons.TextView("result",{tooltip:
"Results"});
    var oEqual = new sap.ui.commons.TextView("equal",{tooltip:
"Equals", text: " = "});
    var oMult = new sap.ui.commons.TextView("mult",{tooltip: "Multiply
by", text: " * "});

    //Attach a controller event handler to Value 1 Input Field
oVal1.attachEvent("liveChange", function(oEvent){
  oController.onLiveChange(oEvent,oVal2); });
    //Attach a controller event handler to Value 2 Input Field
oVal2.attachEvent("liveChange", function(oEvent){
  oController.onLiveChange(oEvent,oVal1); });

    layoutNew.createRow(oVal1, oMult, oVal2, oEqual, oResult );
    return multiplyPanel;
  }
});
```

6. Set up the SAPUI5 controller functions to handle the UI events.

The code described in this step must be added to the SAPUI5 view controller file `xsjsMultiply.controller.js`.

- a. Add the code that creates an event handler named `onLiveChange`.

The `onLiveChange` function has two parameters: `oEvent` and `oVal`, which are used in the `jQuery.Ajax` call to the XSJS service at the specified URL. This is the event which is triggered every time the value is changed in either of the text boxes displayed in the application UI.

```
onLiveChange: function(oEvent,oVal){
    var aUrl = '/sap/hana/democontent/epm/services/multiply.xsjs?
cmd=multiply'+ '&num1='
        +escape(oEvent.getParameters().liveValue)
    + '&num2='+escape(oVal.getValue());
    jQuery.ajax({
        url: aUrl,
        method: 'GET',
        dataType: 'json',
        success: this.onCompleteMultiply,
        error: this.onErrorCall });
}
```

If the AJAX call is successful, call a controller event named `onCompleteMultiply`; if the AJAX call is **not** successful, call a controller event named `onErrorCall`.

- b. Add the code that creates an event handler named `onCompleteMultiply`.

The `onCompleteMultiply` function accepts the response object as an input parameter called `myTxt`. This text box will contain the result of the multiplication in clear text. Use the `sap.ui.core.format.NumberFormat` to format the output as an integer and set the value back into the `oResult` `textView`.

```
onCompleteMultiply: function(myTxt){
    var oResult = sap.ui.getCore().byId("result");
    if(myTxt==undefined){ oResult.setText(0); }
    else{
        jQuery.sap.require("sap.ui.core.format.NumberFormat");
        var oNumberFormat =
        sap.ui.core.format.NumberFormat.getIntegerInstance({
            maxFractionDigits: 12,
            minFractionDigits: 0,
            groupingEnabled: true });
        oResult.setText(oNumberFormat.format(myTxt)); }
    },
```

- c. Add the code that produces an error dialog if the event produces an error.

The `onErrorCall` function displays a message dialog (`sap.ui.commons.MessageBox.show`) in the event of an error during the multiplication action provided by the XSJS service. The information displayed in the error message is contained in `jqXHR.responseText`.

```
onErrorCall: function(jqXHR, textStatus, errorThrown){
    sap.ui.commons.MessageBox.show(jqXHR.responseText,
        "ERROR",
        "Service Call Error" );
    return;
}
```

The complete `xsjsMultiply.controller.js` file should look like the following example:

```
sap.ui.controller("xsjsmultiply.xsjsMultiply", {
    onLiveChange: function(oEvent,oVal){
        var aUrl = '/sap/hana/democontent/epm/services/multiply.xsjs?
cmd=multiply'+ '&num1='
            +escape(oEvent.getParameters().liveValue)
        + '&num2='+escape(oVal.getValue());
        jQuery.ajax({
            url: aUrl,
            method: 'GET',
            dataType: 'json',
```

```

        success: this.onCompleteMultiply,
        error: this.onErrorCall });
    },
    onCompleteMultiply: function(myTxt){
        var oResult = sap.ui.getCore().byId("result");
        if(myTxt==undefined){ oResult.setText(0); }
        else{
            jQuery.sap.require("sap.ui.core.format.NumberFormat");
            var oNumberFormat =
sap.ui.core.format.NumberFormat.getIntegerInstance({
                maxFractionDigits: 12,
                minFractionDigits: 0,
                groupingEnabled: true });
            oResult.setText(oNumberFormat.format(myTxt)); }
    },
    onErrorCall: function(jqXHR, textStatus, errorThrown){
        sap.ui.commons.MessageBox.show(jqXHR.responseText,
            "ERROR",
            "Service Call Error" );
        return;
    }
});

```

7. Save and activate all changes to all SAPUI5 application artifacts.

ⓘ Note

Activate at the project level to ensure that the changes made to **all** project artifacts are updated in the SAP HANA repository.

8. Test your “xsjsMultiply” SAPUI5 application in a Web browser.

The URL for the SAPUI5 application is: <http://<WebServerHost>:80<SAPHANAinstance>/workshop/session/ui/xsjsMultiply/webContent/>.

ⓘ Note

The content of the URL is case sensitive. If prompted, log on using your SAP HANA user name and password.

XS Service Test - Multiplication

222 * 1144.8 = 254,145.59999999998

10.3.3 Tutorial: Consume an OData Service from SAPUI5

An OData service can be used to provide the data required for display in an SAPUI5 application.

Prerequisites

To complete this tutorial successfully, bear in mind the following requirements:

- You have installed the SAP HANA studio.
- You have installed the SAPUI5 tools included in the delivery unit (DU) [SAPUI5_1](#).
- You have installed the [SHINE](#) delivery unit (DU); this DU contains the views (`sap.hana.democontent.epm.models::AN_SALES_OVERVIEW_WO_CURR_CONV` and `sap.hana.democontent.epm.models::AT_BUYER`) specified in the OData service (`salesOrders.xsodata`) that you want to consume with the SAPUI5 application you build in this tutorial.
- You have generated data to populate the tables and views provided by the [SHINE](#) DU and used in this tutorial. You can generate the data with tools included in the [SHINE](#) DU.

Note

You might have to adjust the paths in the code examples provided to suit the folder/package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration tables and services) referenced in the tutorial.

Context

You can bind a UI element in an SAPUI5 application to the data specified in an OData service. For example, you can populate the contents of a table column displayed in an SAPUI5 application with the data stored in a database table defined in an OData service.

Procedure

1. Create an SAPUI5 project.
 - a. Start the [New Application Project](#) wizard.

In the SAP HANA studio's *Project Explorer* view, choose  [New](#)  [Other...](#) 
 - b. Select the application project.

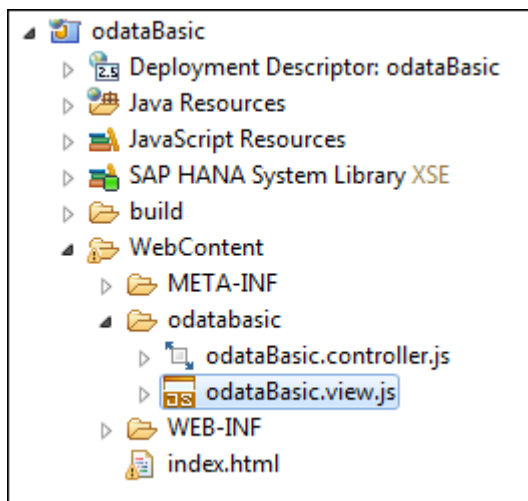
SAP HANA studio provides dedicated wizards to help you set up an application project; here you choose the project  [SAPUI5 Application Development](#)  in the [New Project](#) wizard.
 - c. Define details of the new project.
 - Enter a name for the new SAPUI5 application project, for example, **odataBasic**.

- Check the [Use default location](#) option.
- d. Define details of the new SAPUI5 application view and choose [Finish](#).
- Check the folder for the project; it should be `WebContent/odatabasicx`.
 - Provide a name for the base HTML page that the SAPUI5 application uses, for example, **odataBasic**.
 - Choose [JavaScript](#) as the [Development Paradigm](#).

Note

If prompted, do not switch to the Java EE perspective.

You now have an Eclipse project for the new SAPUI5 application. The SAPUI5 application project has a bootstrap HTML page (`index.html`) in the `WebContent` folder and an `odataBasic` controller (and view) in the sub-package `odatabasicx` as illustrated in the following example:



2. Create the application-descriptor files that enable client access to the services and data exposed by the new application.

Each SAP HANA XS application requires two **mandatory** application descriptor files, which are located in the root package of the application they apply to. If the application-descriptor files already exist (for example, because they are created as part of the new-application Wizard), you can safely skip this step.

Note

Application descriptors have a file extension, but no file name, for example, `.xsapp` or `.xsaccess`.

- a. In the SAP HANA studio's [Project Explorer](#) view, right-click the application package where you want to create the new application descriptors and, in the popup menu, choose, **New** > **Other...**
- b. Create the XS application descriptor file (`.xsapp`).

In the [Select a Wizard](#) dialog, choose **SAP HANA** > **Application Development** > **XS Application Access File**.

Tip

The application descriptor has no content; its job is to mark the root package of the resources exposed to client requests by the application.

- c. Create the XS application-access file (.xsaccess).

In the *Select a Wizard* dialog, choose ► **SAP HANA** ► *Application Development* ► *XS Application Access File* .

→ Tip

The .xsaccess file controls who has access to the application (and how) and what data or services the application can expose.

- d. Select a template to use for the application-access file (for example, *Basic*) and choose *Finish*.

A basic .xsaccess file looks like the following example, which exposes your application data, specifies that logon credentials are required for authentication, and helps to prevent cross-site request-forgery (XSRF) attacks:

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form"},
  "prevent_xsrp" : true
}
```

- e. Activate the XS application descriptor files in the SAP HANA Repository.

Right-click the package containing the application descriptor files you have created and, in the context-sensitive menu, choose ► **Team** ► *Activate* .

You now have a basic package structure to hold your application files. The root package for your new application also contains the required application descriptors, which control access to the services and data exposed by the new application.

3. Share the new SAPUI5 project with the SAP HANA Repository.

In the SAP HANA studio's *Project Explorer* view, right-click the new SAPUI5 application project, and choose ► **Team** ► *Share Project...* .

- a. Choose the appropriate repository **type**, for example, *SAP HANA Repository*.
- b. Specify the package location in the SAP HANA repository where the new SAP UI5 application project should reside.

In the *Share Project* wizard, choose *Browse...* to select the package in which you want to store the new SAPUI5 application artifacts. Select the `ui` package in the SAPUI5 folder hierarchy.

- c. Check the settings you made for the new SAPUI5 application project.
- d. Activate the new SAPUI5 application project.

In the SAP HANA studio's *Project Explorer* view, right-click the new SAPUI5 application project, and choose ► **Team** ► *Activate* ► .

→ Tip

Remember to activate at the project level to ensure that all project artifacts are created and stored in the SAP HANA repository.

4. Modify the default settings for the SAPUI5 bootstrap location in the base SAPUI5 `index.html`.

The SAPUI5 project wizard inserts a default bootstrap location into the `index.html` file which is incorrect for SAP HANA. You must manually change the bootstrap location in the SAPUI5 application's `index.html` file by adding `/sap/ui5/1` to the beginning of the default path defined in the `script src=` tag, for example, `script src="/sap/ui5/1/resources/sap-ui-core.js` as illustrated in the following example:

Note

You must also declare any additional libraries you want the SAPUI5 application to use to render the data it consumes. For this tutorial, add `sap.ui.table` to the list of SAPUI5 libraries, as shown in the following example.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <script src="/sap/ui5/1/resources/sap-ui-core.js"
      id="sap-ui-bootstrap"
      data-sap-ui-libs="sap.ui.commons,sap.ui.table"
      data-sap-ui-theme="sap_goldreflection">
    </script>
```

5. Connect the SAPUI5 table element to the OData service.

The code described in this step must be added to the SAPUI5 view controller file `odataBasic.view.js`.

- a. Add the code to create an object named `oModel` of type `sap.ui.model.odata.ODataModel`, as illustrated in the following code example:

```
var oModel = new sap.ui.model.odata.ODataModel("/sap/hana/democontent/epm/
services/salesOrders.xsodata/", true);
```

- b. Add the code to set the model named `oModel` to the UI table control named `oTable`.

The code you add creates a sorting mechanism (of type `sap.ui.model.Sorter`) which uses the column `SALESORDERID`. Bind the table to the entity `SalesOrderHeader` in the OData service definition and add the sorter object to the binding.

```
this.oSHTable.setModel(oModel);
var sort1 = new sap.ui.model.Sorter("SALESORDERID", true);

this.oSHTable.bindRows({
  path: "/SalesOrderHeader",
  parameters: {expand: "Buyer",
    select:
      "SALESORDERID,CURRENCY,GROSSAMOUNT,PARTNERID.PARTNERID,Buyer/COMPANYNAME"},
  sorter: sort1
});
```

These two steps connect the SAPUI5 table element to the OData service `salesOrders.xsodata`. The result in the `odataBasic.view.js` file should look like the code illustrated in the following example:

```
sap.ui.jsview("odatabasic.odataBasic", {
  /** Specifies the Controller belonging to this View.
   * In the case that it is not implemented, or that "null" is returned,
   this View does not have a Controller.
   * @memberOf databasic.odataBasic
   */
  getControllerName : function() {
    return "odatabasic.odataBasic";
  },
  /** Is initially called once after the Controller has been instantiated.
   It is the place where the UI is constructed.
   * Since the Controller is given to this method, its event handlers can be
   attached right away.
   * @memberOf databasic.odataBasic
   */
  createContent : function(oController) {
```

```

        var oLayout = new
sap.ui.commons.layout.MatrixLayout({width:"100%"});

        var oModel = new sap.ui.model.odata.ODataModel("/sap/hana/
democontent/epm/services/salesOrders.xsodata/", true);

        var oControl;
        this.oSHTable = new sap.ui.table.Table("soTable",{
            visibleRowCount: 10,
        });
        this.oSHTable.setTitle("SALES_ORDER_HEADERS");

        //Table Column Definitions
        oControl = new
sap.ui.commons.TextView().bindProperty("text", "SALESORDERID");
        this.oSHTable.addColumn(new sap.ui.table.Column({label:new
sap.ui.commons.Label({text: "SALES_ORDER_ID"}),
            template: oControl, sortProperty:
"SALESORDERID", filterProperty: "SALESORDERID", filterOperator:
sap.ui.model.FilterOperator.EQ, flexible: true }));

        oControl = new
sap.ui.commons.TextView().bindProperty("text", "PARTNERID.PARTNERID");
        this.oSHTable.addColumn(new sap.ui.table.Column({label:new
sap.ui.commons.Label({text: "PARTNER_ID"}),
            template: oControl, sortProperty: "PARTNERID", filterProperty:
"PARTNERID" }));

        oControl = new sap.ui.commons.TextView().bindProperty("text", "Buyer/
COMPANYNAME");
        this.oSHTable.addColumn(new sap.ui.table.Column({label:new
sap.ui.commons.Label({text: "COMPANY"}),
            template: oControl, sortProperty: "Buyer/
CompanyName", filterProperty: "Buyer/CompanyName", filterOperator:
sap.ui.model.FilterOperator.Contains }));

        oControl = new
sap.ui.commons.TextView().bindText("GROSSAMOUNT", oController.numericFormatter)
;
        oControl.setTextAlign("End");
        this.oSHTable.addColumn(new sap.ui.table.Column({label:new
sap.ui.commons.Label({text: "GROSS_AMOUNT"}),
            template: oControl, sortProperty: "GROSSAMOUNT",
filterProperty: "GROSSAMOUNT", hAlign: sap.ui.commons.layout.HAlign.End}));
        oControl = new
sap.ui.commons.TextView().bindProperty("text", "CURRENCY");
        this.oSHTable.addColumn(new sap.ui.table.Column({label:new
sap.ui.commons.Label({text: "CURRENCY"}),
            template: oControl, sortProperty: "CURRENCY", filterProperty:
"CURRENCY" }));
        this.oSHTable.setModel(oModel);
        var sort1 = new sap.ui.model.Sorter("SALESORDERID", true);

        this.oSHTable.bindRows({
            path: "/SalesOrderHeader",
            parameters: {expand: "Buyer",
                select:
"SALESORDERID,CURRENCY,GROSSAMOUNT,PARTNERID.PARTNERID,Buyer/COMPANYNAME"},
            sorter: sort1
        });

        this.oSHTable.setTitle("Sales Orders");
        oLayout.createRow(this.oSHTable);

        return oLayout;
    }
});

```


6. Save and activate all changes to all SAPUI5 application artifacts.

Note

Activate at the project level to ensure that the changes made to **all** project artifacts are updated in the SAP HANA repository.

7. Test your "odataBasic" SAPUI5 application in a Web browser.

The URL for the SAPUI5 application is: `http://<WebServerHost>:80<SAPHANAinstance>/workshop/session/ui/odataBasic/WebContent/`.

Note

The content of the URL is case sensitive. Log on using your SAP HANA user name and password.

SALES_ORDER_ID	PARTNER_ID	COMPANY	GROSS_AMOUNT	CURRENCY
0500001028	0100000044	Sorali	325.94	EUR
0500001027	0100000044	Sorali	325.94	EUR
0500001026	0100000044	Sorali	325.94	EUR
0500001025	0100000044	Sorali	325.94	EUR
0500001024	0100000044	Sorali	325.94	EUR
0500001023	0100000044	Sorali	325.94	EUR
0500001022	0100000044	Sorali	325.94	EUR
0500001021	0100000044	Sorali	325.94	EUR
0500001020	0100000044	Sorali	325.94	EUR
0500001019	0100000044	Sorali	325.94	EUR

8. **Optional:** Use the metadata that OData exposes to build the table columns dynamically.

You do not have to hard code the column definitions in the `*.view.js` file. To use Odata metadata to build the columns dynamically, replace the list of hard-coded table-column definitions in the `odataBasic.view.js` with the code that builds the table columns dynamically, as illustrated in the following example.

```

sap.ui.jsview("odatabasic.odataBasic", {
    /** Specifies the Controller belonging to this View.
     * In the case that it is not implemented, or that "null" is returned,
     * this View does not have a Controller.
     * @memberOf databasic.odataBasic
     */
    getControllerName : function() {
        return "odatabasic.odataBasic";
    },
    /** Is initially called once after the Controller has been instantiated.
     * It is the place where the UI is constructed.
     * Since the Controller is given to this method, its event handlers can be
     * attached right away.
     * @memberOf databasic.odataBasic
     */
    createContent : function(oController) {
        var oLayout = new sap.ui.commons.layout.MatrixLayout({width:"100%"});

        var oModel = new sap.ui.model.odata.ODataModel("/sap/hana/
democontent/epm/services/salesOrders.xsodata/", true);

```

```

var oControl;
this.oSHTable = new sap.ui.table.Table("soTable",{
    visibleRowCount: 10,
});
this.oSHTable.setTitle("SALES_ORDER_HEADERS");

//Table Column Definitions
var oMeta = oModel.getServiceMetadata();
var oControl;

for ( var i = 0; i < oMeta.dataServices.schema[0].entityType[0].property.length; i++) {
    var property = oMeta.dataServices.schema[0].entityType[0].property[i];

    oControl = new sap.ui.commons.TextField().bindProperty("value",property.name);
    oTable.addColumn(new sap.ui.table.Column({label:new sap.ui.commons.Label({text:
property.name}), template: oControl, sortProperty: property.name, filterProperty: property.name,
filterOperator: sap.ui.model.FilterOperator.EQ, flexible: true, width: "125px" }));
}

this.oSHTable.setModel(oModel);
var sort1 = new sap.ui.model.Sorter("SALESORDERID", true);

this.oSHTable.bindRows({
    path: "/SalesOrderHeader",
    parameters: {expand: "Buyer",
        select:
"SALESORDERID,CURRENCY,GROSSAMOUNT,PARTNERID,Buyer/COMPANYNAME"},
    sorter: sort1
});

this.oSHTable.setTitle("Sales Orders");
oLayout.createRow(this.oSHTable);

return oLayout;
}
});

```

The code you insert performs the following actions:

- Uses the function `getServiceMetadata()` to connect to the OData metadata object
- Inspects the OData metadata and extracts the columns of the service defined in the property `dataServices.schema[0].entityType[0].property`
- Loops over this collection of OData metadata and creates a column for each `property.name` in the service dynamically.

Sales Orders							
SALESORDERID	History.CREATEDBY	History.CREATEDAT	History.CHANGEDBY	History.CHANGEDAT	NOTEID	PARTNERID	CURRENCY
0500001028	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001027	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001026	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001025	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001024	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001023	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001022	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001021	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001020	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR
0500001019	0000000033	Wed Sep 11 02:00:00 U	0000000033	Wed Sep 25 02:00:00 L		0100000044	EUR

10.3.4 Tutorial: Consume an OData Service with the CREATE Option

An OData service can be used to provide the data required for display in an SAPUI5 application.

Prerequisites

To complete this tutorial successfully, bear in mind the following requirements:

- You have installed the SAP HANA studio.
- You have installed the SAPUI5 tools included in the delivery unit (DU) [SAPUI5_1](#).
- You have installed the [SHINE](#) delivery unit (DU); this DU contains the tables and OData services that you want to consume with the SAPUI5 application you build in this tutorial.
- You have generated data to populate the tables and views provided by the [SHINE](#) delivery unit and used in this tutorial. You can generate the data with tools included in the [SHINE](#) delivery unit.

Note

You might have to adjust the paths in the code examples provided to suit the folder/package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration tables and services) referenced in the tutorial.

Context

You can bind a UI element in an SAPUI5 application to the data specified in an OData service. For example, you can populate the contents of table columns displayed in an SAPUI5 application with the data stored in a database table defined in an OData service. In this tutorial, you learn how to build an SAPUI5 view that provides input fields, which you can use to create a new record or update an existing record in a database table, for example, using the OData create, update, and delete (CRUD) features.

Procedure

1. Create an SAPUI5 project.
 - a. Start the [New Application Project](#) wizard.

In the SAP HANA studio's [Project Explorer](#) view, choose  [New](#)  [Other...](#) 
 - b. Select the application project.

SAP HANA studio provides dedicated wizards to help you set up an application project; choose the project  [SAPUI5 Application Development](#)  in the [New Project](#) wizard.
 - c. Define details of the new project.
 - Enter a name for the new SAPUI5 application project, for example, **userCRUD**.

- Check the *Use default location* option.
- d. Define details of the new SAPUI5 application view and choose *Finish*.
- Provide a name for the base HTML page that the SAPUI5 application uses, for example, **userCRUD**.
 - Choose *JavaScript* as the *Development Paradigm*.

Note

If prompted, do not switch to the Java EE perspective.

You now have an Eclipse project for the new SAPUI5 application. The SAPUI5 application project has a bootstrap HTML page (`index.html`) in the `WebContent` folder and an `odataBasic` controller (and view) in the sub-package `odatacrudx`.

2. Create the application-descriptor files that enable client access to the services and data exposed by the new application.

Each SAP HANA XS application requires two **mandatory** application descriptor files, which are located in the root package of the application they apply to. If the application-descriptor files already exist (for example, because they are created as part of the new-application Wizard), you can safely skip this step.

Note

Application descriptors have a file extension, but no file name, for example, `.xsapp` or `.xsaccess`.

- In the SAP HANA studio's *Project Explorer* view, right-click the application package where you want to create the new application descriptors and, in the popup menu, choose, **New** > *Other...*
- Create the XS application descriptor file (`.xsapp`).

In the *Select a Wizard* dialog, choose **SAP HANA** > *Application Development* > *XS Application Access File*.

Tip

The application descriptor has no content; its job is to mark the root package of the resources exposed to client requests by the application.

- Create the XS application-access file (`.xsaccess`).

In the *Select a Wizard* dialog, choose **SAP HANA** > *Application Development* > *XS Application Access File*.

Tip

The `.xsaccess` file controls who has access to the application (and how) and what data or services the application can expose.

- Select a template to use for the application-access file (for example, *Basic*) and choose *Finish*.

A basic `.xsaccess` file looks like the following example, which exposes your application data, specifies that logon credentials are required for authentication, and helps to prevent cross-site request-forgery (XSRF) attacks:

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form"},
  "prevent_xsrf" : true
}
```

- e. Activate the XS application descriptor files in the SAP HANA Repository.

Right-click the package containing the application descriptor files you have created and, in the context-sensitive menu, choose **Team > Activate**.

You now have a basic package structure to hold your application files. The root package for your new application also contains the required application descriptors, which control access to the services and data exposed by the new application.

3. Share the new SAPUI5 project with the SAP HANA repository.

In the SAP HANA studio's *Project Explorer* view, right-click the new SAPUI5 application project, and choose **Team > Share Project...**

- a. Choose the appropriate repository **type**, for example, *SAP HANA Repository*.
- b. Specify the package location in the SAP HANA repository where the new SAP UI5 application project should reside.

In the *Share Project* wizard, choose *Browse...* to select the package in which you want to store the new SAPUI5 application artifacts. Select the `ui` package in the SAPUI5 folder hierarchy.

- c. Check the settings you made for the new SAPUI5 application project.
- d. Activate the new SAPUI5 application project.

In the SAP HANA studio's *Project Explorer* view, right-click the new SAPUI5 application project, and choose **Team > Activate**.

→ Tip

Remember to activate at the project level to ensure that all project artifacts are created and stored in the SAP HANA repository.

4. Modify the default settings for the SAPUI5 bootstrap location in the base SAPUI5 `index.html`.

The SAPUI5 project wizard inserts a default bootstrap location into the `index.html` file which is incorrect for SAP HANA. You must manually change the bootstrap location in the SAPUI5 application's `index.html` file by adding `/sap/ui5/1` to the beginning of the default path defined in the `script src=` tag, for example, `script src="/sap/ui5/1/resources/sap-ui-core.js` as illustrated in the following example:

ⓘ Note

You must also declare any additional libraries you want the SAPUI5 application to use to render the data it consumes. For this tutorial, add `sap.ui.table` to the list of SAPUI5 libraries, as shown in the following example.

```
<!DOCTYPE HTML>
<html><head><meta http-equiv="X-UA-Compatible" content="IE=edge">
    <script src="/sap/ui5/1/resources/sap-ui-core.js"
              id="sap-ui-bootstrap"
              data-sap-ui-libs="sap.ui.commons, sap.ui.table,
sap.ui.ux3, sap.viz"
              data-sap-ui-theme="sap_goldreflection">
    </script>
    <!-- add sap.ui.table, sap.ui.ux3 and/or other libraries to
'data-sap-ui-libs' if required -->
    <script>
        var version = sap.ui.version;
        var versionMinor = version.substring(2,4);
        if(versionMinor>=14){
            sap.ui.getCore().applyTheme("sap_bluecrystal")
        }
    </script>
</head></html>
```

```

        sap.ui.localResources("usercrud");
        var view = sap.ui.view({id:"iduserCRUD1",
viewName:"usercrud.userCRUD", type:sap.ui.core.mvc.ViewType.JS});
        view.placeAt("content");
    </script>
</head>
<body class="sapUiBody" role="application">
    <div id="content"></div>
</body>
</html>

```

5. Set up the SAPUI5 user interface and bind it to an OData service.

The code you need to add to the `userCRUD.view.js` performs the following actions:

- Adds three text-entry boxes (`sap.ui.commons.TextField`) to the SAPUI5 application interface (*First Name*, *Last Name*, and *Email*)
- Adds a *Create Record* button (`sap.ui.commons.Button`) to the SAPUI5 application interface
- Binds the SAPUI5 view to the OData service `user.xsodata`

```

sap.ui.jsview("usercrud.userCRUD", {
    getControllerName : function() {
        return "usercrud.userCRUD";
    },
    createContent : function(oController) {

        var oLayout = new sap.ui.commons.layout.MatrixLayout();
        this.oModel = new sap.ui.model.odata.ODataModel("/sap/hana/
democontent/epm/services/user.xsodata/", true);

        var updatePanel = new sap.ui.commons.Panel("updPanel").setText('New
User Record Details');
        var layoutNew = new
sap.ui.commons.layout.MatrixLayout({width:"auto"});

        var oVal1 = new sap.ui.commons.TextField("fName",{tooltip: "First
Name", width: "200px", editable:true});
        var oVal2 = new sap.ui.commons.TextField("lName",{tooltip: "Last
Name", width: "200px", editable:true});
        var oVal3 = new sap.ui.commons.TextField("email",{tooltip: "Email",
width: "200px", editable:true});
        var oExcButton = new sap.ui.commons.Button({
            text : "Create Record",
            press : oController.callUserService });
        layoutNew.createRow(new sap.ui.commons.Label({text: "First Name:
"}, oVal1 ); //oExcButton );
        layoutNew.createRow(new sap.ui.commons.Label({text: "Last Name:
"}, oVal2 ); //oExcButton );
        layoutNew.createRow(new sap.ui.commons.Label({text: "Email:
"}, oVal3, oExcButton );
        updatePanel.addContent(layoutNew);
        oLayout.createRow(updatePanel);

        oTable = new sap.ui.table.Table("userTbl",{tableId: "tableID",
            visibleRowCount: 10});
        oTable.setTitle("Users");

        //Table Column Definitions
        var oMeta = this.oModel.getServiceMetadata();
        var oControl;

        for ( var i = 0; i <
oMeta.dataServices.schema[0].entityType[0].property.length; i++) {

```

```

        var property =
oMeta.dataServices.schema[0].entityType[0].property[i];

        oControl = new sap.ui.commons.TextField({change:
oController.updateService } ).bindProperty("value",property.name);
        if(property.name === 'PERS_NO'){
            oControl.setEditable(false);
        }
        oTable.addColumn(new sap.ui.table.Column({label:new
sap.ui.commons.Label({text: property.name}), template: oControl,
sortProperty: property.name, filterProperty: property.name, filterOperator:
sap.ui.model.FilterOperator.EQ, flexible: true, width: "125px" }));
    }

    oTable.setModel(this.oModel);
    oTable.bindRows("/Users");
    oTable.setTitle("Users" );
    oTable.setEditable(true);

    oLayout.createRow(oTable);
    return oLayout;

});
}

```

The `userCRUD.view.js` file should display the UI view illustrated in the following example:

New User Record Details

First Name:

Last Name:

Email:

Users

	PERS_NO	FIRSTNAME	LASTNAME	E_MAIL
	1000000238	John	Smith	john.smith@sap.com
	1000000239	James	Doe	james.doe@sap.com

6. Set up the UI elements that the SAPUI5 application uses to handle create and update events.

The functions that handle the create and update events are defined in the `SAPUI5 controller.js` file.

- a. Add a declaration for the `oModel` and set it to `null`.

This code ensures that the model instance is passed from the SAPUI5 view to the SAPUI5 controller.

```

sap.ui.controller("usercrud.userCRUD", {
    oModel : null,
})

```

- b. Add the event handlers required to **create** and **update** a database record with OData CRUD operations.

The event handlers are empty at this point but, when finished, ensures that the functions `callUserService` (which creates new records in a table) and `updateService` (which updates records in a table) are available.

```

callUserService : function() {
},
updateService: function(Event) {
}

```

- c. Set up the `callUserService` function to handle create events.

The code required for this implementation of the `callUserService` function is illustrated in the following example:

```
callUserService : function() {
    var oModel = sap.ui.getCore().byId("userTbl").getModel();
    var oEntry = {};
    oEntry.PERS_NO = "0000000000";
    oEntry.FIRSTNAME = sap.ui.getCore().byId("fName").getValue();
    oEntry.LASTNAME = sap.ui.getCore().byId("lName").getValue();
    oEntry.E_MAIL = sap.ui.getCore().byId("email").getValue();
    oModel.setHeaders({"content-type" : "application/
json;charset=utf-8"});
    oModel.create('/Users', oEntry, null, function() {
        alert("Create successful");
    }, function() {
        alert("Create failed");
    });
},
```

In this example, the `callUserService` function performs the following actions:

- Provides access to the model object by means of the controller with a call to `var oModel = sap.ui.getCore().byId("userTbl").getModel();`.
 - Creates a JSON object with the service fields: `PERS_NO`, `FIRSTNAME`, `LASTNAME`, and `E_MAIL`. `PERS_NO` can have a fixed value `0000000000`. The other fields should be read from the screen with `sap.ui.getCore().byId("<insert field id>").getValue();`
 - Sets a custom header of "content-type" with the value "application/json;charset=utf-8" in the model. This enables a call to the `oModel.create` function for the entity `/Users`.
- d. Set up the `updateService` function to handle update events.

The code required for this implementation of the `updateService` function is illustrated in the following example:

```
updateService: function(Event) {
    var oModel = sap.ui.getCore().byId("userTbl").getModel();
    var index = Event.getSource().oParent.getIndex();
    var oEntry = {};
    oEntry.PERS_NO = sap.ui.getCore().byId("__field0-col0-
row"+index).getValue();
    switch (Event.mParameters.id){
        case "__field1-col1-row"+index:
            oEntry.FIRSTNAME = Event.mParameters.newValue; break;
        case "__field2-col2-row"+index:
            oEntry.LASTNAME = Event.mParameters.newValue; break;
        case "__field3-col3-row"+index:
            oEntry.E_MAIL = Event.mParameters.newValue;
    }
    break;

    var oParams = {};
    oParams.fnSuccess = function(){ alert("Update successful");};
    oParams.fnError = function(){alert("Update failed");};
    oParams.bMerge = true;
    oModel.setHeaders({"content-type" : "application/
json;charset=utf-8"});
    oModel.update("/Users('"+oEntry.PERS_NO+"')", oEntry, oParams);
}
};
```


The `updateService` performs the following actions:

- Accesses the model to read the index of the table for the changed record using `Event.getSource().oParent.getIndex()`.
- Creates a JSON object with the service fields `PERS_NO` and whichever field was modified or updated. You can access the fields in the table using the event parameter ID `"__field<index>-col<index>-row"+index`, where `index` is the table index you read earlier, for example, `__field1-col1-row"+index`.
- Sets a custom header of `"content-type"` with the value `"application/json; charset=utf-8"` in the model. Then you can call the `oModel.update` function for the entity `/Users`.

7. Save and activate all changes to all SAPUI5 application artifacts.

Note

Activate at the project level to ensure that the changes made to **all** project artifacts are updated in the SAP HANA repository.

8. Test your "userCRUD" SAPUI5 application in a Web browser.

The URL for the SAPUI5 application is: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/democontent/epm/ui/userCRUD/index.html`. You should test both the **create** and the **update** operations.

- a. Create a new record in the table referenced in the OData service.

The screenshot shows the 'New User Record Details' form with the following fields: First Name: Alan, Last Name: Parkar, Email: alan.parkar@sap.com. A 'Create Record' button is visible. Below the form is a table titled 'Users' with columns PERS_NO, FIRSTNAME, LASTNAME, and E_MAIL. The table contains two rows: (1000000238, John, Smith, john.smith@sap.com) and (1000000239, James, Doe, james.doe@sap.com). A modal dialog box is open, displaying the message: 'The page at sap.corp:8055 says: Create successful' with an 'OK' button.

PERS_NO	FIRSTNAME	LASTNAME	E_MAIL
1000000238	John	Smith	john.smith@sap.com
1000000239	James	Doe	james.doe@sap.com

- b. Update an existing record in the table referenced in the OData service.

The screenshot shows the 'New User Record Details' form with the same fields as before. A modal dialog box is open, displaying the message: 'The page at sap.corp:8055 says: Update successful' with an 'OK' button. The 'Users' table now has three rows: (1000000238, John, Smith, john.smith@sap.com), (1000000239, James, Doe, james.doe@sap.com), and (1000000240, Alan, Parkar, alan.parkarnew@sap.com). The new row is highlighted with a red rectangle.

PERS_NO	FIRSTNAME	LASTNAME	E_MAIL
1000000238	John	Smith	john.smith@sap.com
1000000239	James	Doe	james.doe@sap.com
1000000240	Alan	Parkar	alan.parkarnew@sap.com

10.3.5 Tutorial: Create and Translate Text Bundles for SAPUI5 Applications

Text bundles are used in the context of internationalization (i18n) to store text strings that are displayed in the user interface, for example, dialog titles, button texts, and error messages.

Prerequisites

To complete this tutorial successfully, bear in mind the following requirements:

- You have installed the SAP HANA studio.
- You have installed the SAPUI5 tools included in the delivery unit (DU) [SAPUI5_1](#).
- You have installed the [democontent](#) delivery unit; this DU contains the tables and OData services that you want to consume with the SAPUI5 application you build in this tutorial.
- You have generated data to populate the tables and views provided by the [democontent](#) delivery unit and used in this tutorial. You can generate the data with tools included in the [democontent](#) delivery unit.

Note

You might have to adjust the paths in the code examples provided to suit the folder/package hierarchy in your SAP HANA repository, for example, to point to the underlying content (demonstration services and tables) referenced in this tutorial.

Context

For applications running in production environments, you need to maintain text strings independently so the strings can easily be translated. For UI5 development in SAP HANA you create a so-called “text bundle” named `<FileName>.hdbtextbundle` which contains the text strings. If you need to provide text strings in an alternative language, you can use the Repository Translation Tool (`rtt`) to export the `hdbtextbundle` to a translation system. The translated text can then be imported back into the system for use in language-specific application sessions.

Note

In the SAP HANA repository, there is only a single `hdbtextbundle` file. However, if available, multiple language versions of the strings are stored in the SAP HANA database, and the appropriate string is selected and used automatically depending on the languages settings in the application.

Procedure

1. In an existing SAPUI5 application folder structure, create a dedicated folder (package) for the internationalization elements, for example, the text bundles.

Name the new folder `i18n`.

Note

For Translation purposes you must specify the *Translation Domain* and the *Text collection* in the *Translation* section of the package creation dialog.

2. Create a container for the text bundle.

The file containing the text bundle must have the file extension `.hdbtextbundle`, for example, `ErrorMessages.hdbtextbundle`

Create a file with the name `messagebundle.hdbtextbundle`.

3. Add content to the message bundle.

Add the text in the following code example to the `messagebundle.hdbtextbundle` file:

```
# TRANSLATE
helloworld=Hello World
```

4. Save the `messagebundle.hdbtextbundle` file and activate it in the SAP HANA repository.
5. Add a reference to the `hdbtextbundle` in the core HTML file for the SAP UI5 “Hello World” application you are developing.

Open the file `<...>/WebContent/index.html` in the SAP UI5 “Hello World” project and add the following text (in **bold** font type in the example) to the Language Resource Loader section:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <script src="/sap/ui5/1/resources/sap-ui-core.js"
      id="sap-ui-bootstrap"
      data-sap-ui-libs="sap.ui.commons"
      data-sap-ui-theme="sap_goldreflection">
    </script>
    <!-- add sap.ui.table, sap.ui.ux3 and/or other libraries to 'data-sap-
  ui-libs' if required -->
    <script>
      /***** Language Resource Loader *****/
      jQuery.sap.require("jQuery.sap.resources");
      var sLocale =
        sap.ui.getCore().getConfiguration().getLanguage();
      var oBundle =
        jQuery.sap.resources({url : "../i18n/messagebundle.hdbtextbundle",
locale: sLocale});

      sap.ui.localResources("helloworldx");
      var view = sap.ui.view({id:"idHelloWorld1",
viewName:"helloworldx.HelloWorld", type:sap.ui.core.mvc.ViewType.JS});
      view.placeAt("content");
    </script>
  </head>
  <body class="sapUiBody" role="application">
    <div id="content"></div>
  </body>
</html>
```

6. Save the `index.html` file and activate it in the Repository.
7. Add a reference to the `hdbtextbundle` in the core JavaScript file for the SAP UI5 “Hello World” application you are developing.

In this step, you tell the `setText` function of the *Hello World* button in the UI to use the information in the specified text bundle to display the required text.

Open the file `<...>/ui/HelloWorldX/helloworldx/HelloWorld.view.js` in the SAP UI5 “Hello World” project and add the following text `myButton.setText(oBundle.getText("helloworld"))`; to the `createContent` section:

Note

The additional text is indicated in **bold** font type in the example.

```
sap.ui.jsview("helloworldx.HelloWorld", {
    /** Specifies the Controller belonging to this View.
     * In the case that it is not implemented, or that "null" is returned,
     * this View does not have a Controller.
     * @memberOf helloworldx.HelloWorld
     */
    getControllerName : function() {
        return "helloworldx.HelloWorld";
    },
    /** Is initially called once after the Controller has been instantiated.
     * It is the place where the UI is constructed.
     * Since the Controller is given to this method, its event handlers can be
     * attached right away.
     * @memberOf helloworldx.HelloWorld
     */
    createContent : function(oController) {
        var myButton = new sap.ui.commons.Button("btn");
        myButton.setText(oBundle.getText("helloworld"));
        myButton.attachPress(function(){$("#btn").fadeOut();});
        return myButton;
    }
});
```

8. Save the changes to the `HelloWorld.view.js` file and activate the file in the SAP HANA repository.
9. Test the changes in a Web browser.

`http://<hostname>:<port>/<...>/ui/HelloWorld/WebContent/.`

Note

The URL path and resource names are case sensitive. If prompted, enter your SAP HANA user name and password.

The text string “Hello World” should appear in the Web browser.

10. Export the text bundle for translation.

You use the repository translation tool (`rtt`) included with the SAP HANA client to produce an XML document with the XLIFF format required for upload to an SAP translation system.

Note

One XML document is used for each language pair in the translation process, for example, English to German.

Open a command shell on the machine running the SAP HANA studio/client, and type the following command:

```
rtt -export -p <package containing hdbtextbundle>
```

The XML document generated by the export process specifies the source language (English) and the source text to translate.

```
<?xml version="1.0" encoding="UTF-8"?>
<xliff xmlns="urn:oasis:names:tc:xliff:document:1.2" version="1.2">
  <file datatype="plaintext" original="bla.test.hdbtextbundle" source-
language="en">
    <header>
      <sxmd:metadata xmlns:sxmd="urn:x-sap:mlt:xliff12:metadata:1.0"
xmlns="urn:x-sap:mlt:tsmetadata:1.0">
        <object-name>1.bla.test.hdbtextbundle</object-name>
        <collection>coll</collection>
        <domain>1A</domain>
        <developer>SYSTEM</developer>
        <description>n/a</description>
        <origin>bla.test.hdbtextbundle</origin>
      </sxmd:metadata>
    </header>
    <body>
      <group resname="c.test.hdbtextbundle" restype="x-objectContentTexts">
        <trans-unit xmlns:sap="urn:x-sap:sls-mlt" id="TEST" maxwidth="20"
sap:sc="XTIT" size-unit="char">
          <source>hello world</source>
        </trans-unit>
      </group>
    </body>
  </file>
</xliff>
```

11. Add the translated version of the text string to the XLIFF document.

Typically, the XML document containing the translated text strings is generated by a translation system. However, for the purposes of this tutorial, you can manually add the required information to the `hdbtextbundle.xlf` file:

- Language information
The translated language is defined in the XML metadata using the `target-language="de-DE"` option, for example, `<file [...] target-language="de-DE">` tag.
- The translated text:
The translated text is specified in the body of the XML document using the `<target>` tag, as illustrated in the following example:
- The `hdbtextbundle` file name
The name of the XLIFF `hdbtextbundle` file with language-specific content must include the following characters in the file suffix: a dash ("-"), the appropriate ISO 639 language key (for example, "de"), an underscore ("_"), and an ISO 3166 country code (for example, "DE").
 - Target language=German
`messagebundle.hdbtextbundle-de_DE.xlf`
 - Target language=Chinese
`messagebundle.hdbtextbundle-zh_ZH.xlf`

The XML document you use to import translated version of text strings specifies both the original source language (English) and the target (translated) text, which in this example is German (DE).

```
<?xml version="1.0" encoding="UTF-8"?>
<xliff xmlns="urn:oasis:names:tc:xliff:document:1.2" version="1.2">
  <file datatype="plaintext" date="2013-09-05T13:57:13Z"
original="bla.test.hdbtextbundle" source-language="en" target-language="de-DE">
    <header>
      <sxmd:metadata xmlns:sxmd="urn:x-sap:mlt:xliff12:metadata:1.0"
xmlns="urn:x-sap:mlt:tsmetadata:1.0">
        <object-name>1.bla.test.hdbtextbundle</object-name>
```

```

    <collection>coll</collection>
    <domain>1A</domain>
    <developer>SYSTEM</developer>
    <description>n/a</description>
    <origin>bla.test.hdbtextbundle</origin>
  </saxml:metadata>
</header>
<body>
  <group resname="c.test.hdbtextbundle" restype="x-objectContentTexts">
    <trans-unit xmlns:sap="urn:x-sap:sls-mlt" id="TEST" maxwidth="20"
sap:sc="XTIT" size-unit="char">
      <source>hello world</source>
      <target>Hallo Welt</target>
    </trans-unit>
  </group>
</body>
</file>
</xliff>

```

12. Import the XLF file containing the text strings for the source and target languages into the SAP HANA repository.

You use the repository translation tool (rtt) included with the SAP HANA client to import the .XLF file.

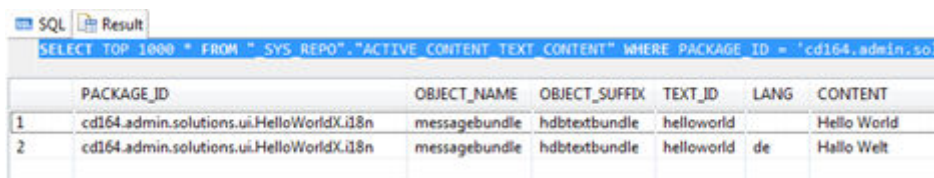
Open a command shell on the machine running the SAP HANA studio/client, and type the following command:

```
rtt -import -p <package containing hdbtextbundle>
```

13. Activate the package containing the XLF file with the translated text strings.

The import operation inserts the translated strings into the appropriate table in the SAP HANA database. You can check which language versions of which text strings are stored in the SAP HANA repository by looking in the table `_SYS.REPO.ACTIVE_CONTENT_TEXT_CONTENT`, for example, with the following SQL command:

```
SELECT TOP 1000 * FROM "_SYS.REPO"."ACTIVE_CONTENT_TEXT_CONTENT" WHERE PACKAGE_ID = <path>.ui.HelloWorld.i18n
```



	PACKAGE_ID	OBJECT_NAME	OBJECT_SUFFIX	TEXT_ID	LANG	CONTENT
1	cd164.admin.solutions.ui.HelloWorldX.i18n	messagebundle	hdbtextbundle	helloworld		Hello World
2	cd164.admin.solutions.ui.HelloWorldX.i18n	messagebundle	hdbtextbundle	helloworld	de	Hallo Welt

14. Change the language setting of your Web browser to German.

You can set the language of the Web browser session either by adding the string `sap-ui-language=de` or changing the language setting in the Web Browser itself.

The request still points at the original text bundle `messagebundle.hdbtextbundle`, but the button in the simple SAPUI5 application now displays the text *Hallo Welt*.

10.4 Using UI Integration Services

SAP HANA UI Integration Services is a set of Eclipse-based tools that enable you to integrate standalone SAP HANA client applications into Web user interfaces to support end-to-end business scenarios.

These Web user interfaces are referred to as SAP Fiori launchpad sites.

Prerequisites

- SAP HANA studio is installed on your local system. The SAP HANA studio version must match the current SAP HANA version.
- A supported browser is installed on your local system. At design time, the following browsers are supported on desktop:
 - Windows: Internet Explorer 9 or higher
 - Linux: latest version of Firefox

Note

For end users at runtime, the following browsers are supported:

- On desktop:
 - Windows: Internet Explorer 9 or higher, latest versions of Chrome, Firefox
 - Linux: latest version of Firefox
 - Mac: latest versions of Safari, Chrome and Firefox
 - On mobile devices: Safari on iOS
- You are assigned the `sap.hana.uis.db:SITE_DESIGNER` role. End users are assigned the `sap.hana.uis.db:SITE_USER` role, and are assigned privileges for the relevant sites.

Note

Make sure you have the appropriate repository package privileges to read, edit and activate files in the package of your project.

- You have set up an SAP HANA application project.

Related Information

[Creating Content for Application Sites \[page 670\]](#)

[SAP Fiori Launchpad Sites \[page 682\]](#)

[Setting Up Roles and Privileges \[page 692\]](#)

[Using SAP HANA Projects \[page 62\]](#)

10.4.1 Creating Content for Application Sites

You can create content for launchpad application sites.

Launchpad sites use tiles, which serve as entry points to SAP Fiori applications running on SAP HANA.

Tiles are used to launch apps from launchpad sites.

Tile catalogs are collections of logically related tiles, which are created and configured by administrators. Site designers choose tiles from available catalogs and add them to the launchpad application sites.

Related Information

[Create a Tile Catalog \[page 671\]](#)

[Configuring Tiles \[page 673\]](#)

[SAP Fiori Launchpad Sites \[page 682\]](#)

10.4.1.1 Create a Tile Catalog

In the SAP HANA studio, you can create tile catalogs from which site designers and users choose tiles for application sites.

Procedure

1. In *Project Explorer*, in the project's context menu, choose ► *New* ► *Other...* .
2. In the *New* dialog box, choose ► *SAP HANA* ► *Application Development* ► *UIS Catalog* , and choose *Next*.
3. In the *New Catalog* dialog box, choose the parent folder, and enter the file name and the name of the catalog.
4. Choose *Finish*.

The newly created catalog is added to the project.

Related Information

[Edit a Tile Catalog \[page 671\]](#)

10.4.1.2 Edit a Tile Catalog

You can edit tile catalogs in the browser-based design environment embedded in the SAP HANA studio.

Context

When planning tile catalogs, consider that access to a catalog is assigned to a role and applies to all the tiles within the catalog. Therefore, do not place tiles that require different permission levels in the same catalog, for example, tiles for managers and for employees.

Procedure

1. To open a catalog for editing in the embedded browser, in [Project Explorer](#), double-click the catalog's `.xswidget` file.

You can perform the following tasks:

Task	Instructions
Edit the catalog's title	Click  (Edit title). In the dialog box that opens, edit the title and choose Save .
Add a tile	Click  (Add tile). In the page that opens, click a tile template that you want to add to the catalog.
Delete a tile	Drag a tile to the trash can image in the lower-left corner of the page.
Configure a tile	Tiles are added to catalogs as generic templates. To make a tile usable, you need to configure it. Double-click a tile to open the configuration page and edit its properties as required.

2. When you have finished editing, save the catalog by choosing  [File](#) > [Save](#)  from the main menu.
3. To make the catalog available to users, activate it by choosing  [Team](#) > [Activate](#)  from the context menu of the catalog's `.xswidget` file.
4. If the site in which you want to use this catalog is open for editing, close and reopen the site to refresh the catalog.

Next Steps

- Configure each tile that you have added to the catalog.
- When you have completed editing the catalog, make it available to users by assigning the application privileges for this catalog to the relevant roles or users.

Related Information

[Configuring Tiles \[page 673\]](#)

10.4.1.3 Configuring Tiles

Tiles are added to catalogs as generic templates. To make a tile usable, you need to configure it to point to a specific application.

Procedure

1. Double-click a tile to open its configuration page.
2. Configure the properties of the tile, following the instructions for the specific tile type:

Tile Type	Instructions
Static, dynamic and custom app launchers	App Launcher Tiles [page 673]
Navigation target and target mapping	Navigation Target and Target Mapping [page 677]
News	News Tile [page 681]

3. Choose [Save](#), and choose [OK](#) in the confirmation dialog.

Results

The configured tile appears in the catalog.

Related Information

[Edit a Tile Catalog \[page 671\]](#)

10.4.1.3.1 App Launcher Tiles

App launcher tiles are used to launch applications. This topic describes how to configure properties of the app launcher tiles.

App launcher tiles come in three flavors: static, dynamic, and custom. All tile flavors are used to launch applications. In addition, dynamic tiles can display data that is updated at regular intervals, for example, KPIs. This data is retrieved from the back-end system using oData services. Custom tiles can display any content defined by a custom SAPUI5 application.

You need to configure the app launcher tile properties, which are divided into the following sections:

General

Property	Description
<i>Title</i>	Name of the tile.
<i>Subtitle</i>	Text displayed below the title.
<i>Keywords</i>	Keywords used to tag a tile so that users can find it more easily using the search function in the tile catalog at runtime.
<i>Icon</i>	Open the dropdown list to select an SAPUI5 icon. After you have selected an icon, the property is set to the icon's URL, preceded by sap-icon:// . For example, sap-icon://Fiori2/F0072 .
<i>Information</i>	Text displayed at the bottom of the tile.
<i>Number Unit</i> (for dynamic tiles only)	Unit displayed below the number. For example, USD .

Configuration (for custom tiles only)

Property	Description
<i>Module Type</i>	In the dropdown box, select the type of the SAPUI5 application module: <i>UIComponent</i> or one of the view types, such as <i>XMLView</i> .
<i>Module Name</i>	Name of the module. For example, test.ui5 .
<i>Module Name Prefix</i>	Name prefix to map to the server location of the module. For example, test .
<i>Module Path</i>	Relative path to the module on the server. For example, /content/ui5 .
<i>Custom Properties</i>	Custom properties as key-value pairs.

Dynamic Data (for dynamic tiles only)

Property	Description
Service URL	URL of an OData service from which data should be read. The response is expected in JSON format. When the service is called, the values that are provided by the service override the values that are configured manually in the tile. Note that the service is executed at runtime only. At design time, sample data is displayed. → Tip If you want to read only a number of entities dynamically from an OData service, and read all other content for the app launcher statically from the configuration, you can use the <code>\$count</code> parameter in the service URL. For more information on the OData service API for dynamic app launcher tiles, see Related Information.
Refresh Interval	Number of seconds after which dynamic content is reloaded from the data source and the display is refreshed. Note the following: • Default value and minimum value is 10 seconds. • If the entered value is between 1 and 9 inclusive, it is automatically modified to 10. • If the entered value is 0, the dynamic tile is updated only once upon loading. • If the entered value is 10 or greater, it is used as is.

Navigation

Property	Description
Use Semantic Object Navigation	Deselect this checkbox if you want to define the navigation target using a simple URL rather than a semantic object, and leave all properties empty, except for Target URL. Otherwise, configure intent-based navigation. For more information, see Related Information.
Target URL	Navigation target URL, including the protocol. For example, http://help.sap.com.

Related Information

[Intent-Based Navigation \[page 676\]](#)

[Navigation Target and Target Mapping \[page 677\]](#)

[Intent-Based Navigation in App Launcher Tiles \[page 678\]](#)

[OData Structure for Dynamic App Launchers \[page 679\]](#)

10.4.1.3.1.1 Intent-Based Navigation

The intent-based navigation mechanism in Fiori Launchpad allows users to launch applications in different views or modes depending on the runtime parameters.

This is achieved by defining application navigation targets using abstract intents, which at runtime are resolved into actual URLs by the Fiori Launchpad target resolution service.

Intent-based navigation is helpful in the following use cases:

- Enabling the user to make a selection from multiple navigation targets.
- When extending and customizing Fiori scenarios, you need to be able to change a target without modifying the application code.
- Enabling communication between Fiori apps that have different life cycles and might not be available in the same productive environment.

Syntax

An intent is a combination of the following elements:

Semantic object	Represents a business entity, such as a customer, a sales order, or a product. Enables you to refer to such entities in an abstract implementation-independent way.
Action	Defines an operation, such as display or approve purchase order. This operation is intended to be performed on a semantic object, such as a purchase order or a product.
Parameters	Optional. Parameters that define an instance of the semantic object, for example, employee ID.

Intents have the following syntax: `#<semantic object>-<action>?<semantic object parameter>=<value1>`.

For example, the intent `#SalesOrder-displayFactSheet?SalesOrder=27` specifies a fact sheet for sales order number 27. At runtime, this intent is resolved into the actual URL `https://<server>:<port>/sap/hana/uis/clients/ushell-app/shells/fiori/FioriLaunchpad.html#SalesOrder-displayFactSheet?SalesOrder=27`.

Workflow

To configure intent-based navigation for an application, the administrator should perform the following tasks:

1. In a navigation target tile, configure the actual application navigation URL.
2. Configure a target mapping tile to map an intent (a combination of a semantic object and an action) to the same navigation target.
3. Configure the navigation in an app launcher tile to the same intent.

Related Information

[Navigation Target and Target Mapping \[page 677\]](#)

[Intent-Based Navigation in App Launcher Tiles \[page 678\]](#)

10.4.1.3.1.2 Navigation Target and Target Mapping

Navigation target and target mapping are auxiliary tiles used for configuring intent-based navigation in app launcher tiles.

These tiles are maintained in tile catalogs, but cannot be added to Fiori Launchpad sites.

In a navigation target tile, you configure the actual application navigation target, whereas in a target mapping tile you map an intent to this navigation target. At runtime, this mapping is resolved to the actual target URL.

The following sections describe the properties that you need to configure in each of the tiles.

General

General properties of both tiles.

Property	Applies to	Description
<i>Title</i>	Navigation Target	Title of the tile
<i>Description</i>	Navigation Target	Description displayed below the title
<i>Information</i>	Target Mapping	Optional additional information

Target

Properties that define the application navigation target. Values of the properties that apply to both tiles need to be identical.

Property	Applies to	Description
<i>Namespace Level 1</i>	Both	Comprises the application namespace in combination with <i>Namespace Level 2</i> , for example, test
<i>Namespace Level 2</i>	Both	Comprises the application namespace in combination with <i>Namespace Level 1</i> , for example, Comp
<i>Application Alias</i>	Both	Alias of the application, for example, compAlias
<i>Application ID</i>	Target Mapping	Leave empty

Property	Applies to	Description
<i>URI</i>	Navigation Target	URI of the navigation target, for example, /content/ui5/TestComponent
<i>Type</i>	Navigation Target	Type of the navigation target: URL
<i>View Name</i>	Navigation Target	Name of the SAPUI5 component to display this target; enter in the format SAPUI5.Component=<name>
<i>Application Parameters</i>	Navigation Target	Optional. An &-separated list of parameters to pass to the target application.

Intent

Properties of the intent that you map to the navigation target.

Property	Description
<i>Semantic Object</i>	Semantic object on which to perform an action. Enter the technical name of the semantic object, for example, SalesOrder .
<i>Action</i>	Action to perform on the semantic object when the user clicks on the tile, for example, display .

Related Information

[Intent-Based Navigation \[page 676\]](#)

[Intent-Based Navigation in App Launcher Tiles \[page 678\]](#)

10.4.1.3.1.3 Intent-Based Navigation in App Launcher Tiles

You can enable intent-based navigation in app launcher tiles by configuring the navigation properties.

In the [Navigation](#) section of an app launcher tile configuration page, set the properties described in the table below.

Note

The property values should be equal to the corresponding values in the target mapping and navigation target tiles that are configured for this app launcher tile.

Property	Description
Use Semantic Object Navigation	Select this checkbox and configure the following properties as described below.
Semantic Object	Semantic object on which to perform an action. Enter the technical name of the semantic object, for example, SalesOrder .
Action	Action to perform on the semantic object when the user clicks on the tile, for example display .
Parameters	Key-value pairs defining parameters for the semantic object, for example orderID=4711 . If you enter multiple parameters, separate them with an ampersand (&), for example orderID=10000&custID=c82200 .
Target URL	Not required if you chose to use semantic object navigation.

Related Information

[App Launcher Tiles \[page 673\]](#)

[Navigation Target and Target Mapping \[page 677\]](#)

[Intent-Based Navigation \[page 676\]](#)

10.4.1.3.1.4 OData Structure for Dynamic App Launchers

You can use OData services to retrieve data to display on a dynamic app launcher tile.

In order to feed an app launcher with dynamic content, you have to create an OData service that returns the configuration properties as in the following example structure:

```
{
    "d": {
        "icon": "sap-icon://travel-expense",
        "info": "Quarter Ends!",
        "infoState": "Critical",
        "number": 43.333,
        "numberDigits": 1,
        "numberFactor": "k",
        "numberState": "Positive",
        "numberUnit": "EUR",
        "stateArrow": "Up",
        "subtitle": "Quarterly overview",
        "title": "Travel Expenses"
    }
}
```


Properties

Property	Description
<code>icon</code>	Enter an <code>sap-icon://</code> URL, for example <code>sap-icon://cart</code>. You can look up the names of the available icons in the app launcher tile configuration.
<code>info</code>	Text to be displayed at the bottom of the tile.
<code>infoState</code>	The color of the tile is adapted according to the value of this property. The precise color depends on the theme that you have selected in UI theme designer. Allowed values: Negative, Neutral, Positive, Critical
<code>number</code>	Number to be displayed in the top right corner of the tile.
<code>numberDigits</code>	Number of digits to be displayed following the decimal separator (decimal point or decimal comma, depending on the language settings).
<code>numberFactor</code>	A factor for scaling numbers when displaying large numbers, for example, 1.000.000 (-> <code>number = 1</code> and <code>numberFactor = "M"</code>) or for percentages (<code>number = 22.2</code> and <code>numberFactor = "%" </code>). Note The scaling is not performed by the front end; it has to be provided by the application developer.
<code>numberState</code>	The color of the number is adapted according to the value of this property. The precise color depends on the theme that you have selected in UI theme designer. Allowed values: Negative, Neutral, Positive, Critical
<code>numberUnit</code>	Unit to be displayed below the number, for example, USD .
<code>stateArrow</code>	Displays an arrow indicating a trend. Allowed values: None, Up, Down
<code>subtitle</code>	Subtitle to be displayed below the tile title.

Property	Description
<code>targetParams</code>	List of key-value-pairs separated by ampersands. When the application is launched (by clicking on it), these parameters are passed to the application as business parameters (if semantic object-based navigation is used) or as URL parameters (if URL-based navigation is used). If any parameters have been entered in the Parameters field in the tile configuration, the parameters passed by the OData service are appended to the list of parameters to be passed to the application.
<code>title</code>	Title to be displayed in the tile.

If the service returns an entity collection (rather than a single entity), all values from the **number** elements are accumulated.

10.4.1.3.2 News Tile

News tiles can be configured to display news feeds.

In the configuration page of a News tile, set the following parameters:

Configuration Parameters	Description
Tile Default Image	A URL that sets the default image for the News tile. You can set this parameter to select an alternate default image to display on the News tile. By default, the News application provides 12 default images and cycles through these default images in sequence.
Always Use Default Image	When this checkbox is selected, the News tile ignores any image that accompanies an RSS feed article. Depending on the checkbox selection, the following order of precedence is used: - When not selected: - Image from the RSS Article (if present) - Image from the RSS Channel (if present) - Image from Tile Default Image (if set) - Image from one of the twelve (12) default images - When selected: - Image from Tile Default Image (if set) - Image from one of the twelve (12) default images
Article Cycle Interval (secs)	An integer (the minimum and default value is 5). This parameter controls the rate at which the articles cycle through the News tile.
Article Refresh Interval	Select a value from the dropdown box. The default value is 15 minutes.

Configuration Parameters	Description
	This parameter controls the rate at which the News tile requests new articles from the value defined in the Article Feeds parameter.
Article Feeds	You can configure up to 10 RSS feeds. The News tile monitors the RSS feeds and retrieves new articles based on the value defined in the Article Refresh Interval parameter. Note If the URL references an external feed, the feed must be CORS-compliant. If the URL references an internal feed, the feed must originate from the same server and port as the launchpad.
Feed #1 – Feed #10	The URL of the RSS feed.
Inclusion Filters	You can configure up to 5 inclusion filters. The News tile filters the feeds and includes any articles that contain the same text in the title of the article.
Filter #1 – Filter #5	Filter text that is compared to the title of the article. If the text is found, the article is included in the list of articles.
Exclusion Filters	You can configure up to 5 exclusion filters. The News tile filters the feeds and excludes any articles that contain the same text in the title of the article.
Filter #1 – Filter #5	Filter text that is compared to the title of the article. If the text is found, the article is excluded from the list of articles.

Note

Consider the following limitations for the News tile parameters:

- Feed URLs are limited to the following sources:
 - Internal sources (same URL and port as the Suite Page Builder application)
 - Any external CORS-compliant source
- URL format should follow the `http://[server]:[port]/[path]` pattern. URLs that use `feed://` as the transport are not supported.
- UI5 URL validation requires the tilde '~' character to be replaced by the sequence `~`. For example, in the path `...filterID=content~tag`, the tilde should be replaced by `...filterID=content~tag`.
- Bookmarking and direct navigation to the list of feed articles is not supported.

10.4.2 SAP Fiori Launchpad Sites

A SAP Fiori launchpad site serves as an entry point to SAP Fiori applications, which are developed and run on the SAP HANA platform.

SAP Fiori launchpad provides the apps with services such as navigation, embedded support, and application configuration.

The following section provides information about creating, designing, and administering SAP Fiori launchpad sites.

Related Information

[Creating a Launchpad Site \[page 683\]](#)

[Designing a Launchpad Site \[page 684\]](#)

[Creating Content for Application Sites \[page 670\]](#)

[Configuring Access to Launchpad Content \[page 689\]](#)

[SAP Fiori Launchpad in User Interface Add-On](#)

10.4.2.1 Creating a Launchpad Site

You create a SAP Fiori launchpad site in the SAP HANA studio.

Procedure

1. In the project's context menu in *Project Explorer*, choose ► *New* ► *Other* ►.
2. In the *New* dialog box, choose ► *SAP HANA* ► *Application Development* ► *UIS Application Site* ►, and then choose *Next*.
3. In the *New Application Site* dialog box, select a parent folder, and enter the site properties: *File Name*, and optionally *Title* and *Description*.
4. Choose the site type, *Fiori Launchpad*.
5. Choose *Finish*.
The newly created site opens for design in the embedded browser window.

Related Information

[Designing a Launchpad Site \[page 684\]](#)

10.4.2.2 Designing a Launchpad Site

You can visually design and manage SAP Fiori launchpad sites in the browser-based design environment that is embedded in SAP HANA studio.

Procedure

1. To open a launchpad site for editing in the embedded browser, in *Project Explorer*, double-click the site's `.xsappsite` file.

Note

If you open the site from its context menu, make sure that you choose the default *SAP HANA Application Site Editor*. Choosing another editor is not recommended.

You can perform the following tasks:

Task	Instructions
Create and edit groups of tiles	Create and Edit Groups [page 685]
Manage tiles in groups	Add and Organize Tiles in Groups [page 686]
Select a theme for the site	Select a Site Theme [page 687]
Access the runtime version of the site	Click  (Options), and choose <i>Runtime Version</i> from the dropdown menu
Assign site content to roles	Configuring Access to Launchpad Content [page 689]
Enable end users to personalize the site	Select or deselect the <i>Enable personalization</i> checkbox
At runtime, display the groups and tiles in this site and in catalogs in the user's language	Select or deselect the <i>Display in User's Language</i> checkbox

2. To save your changes, from the main menu, choose **File** > **Save**.
3. To make the site available to end users, activate it by choosing **Team** > **Activate** from the context menu of the site's `.xsappsite` file.

10.4.2.2.1 Create and Edit Groups

In a Fiori Launchpad site, you can create and edit groups of tiles.

Context

In a Fiori Launchpad site, the list of groups appear in the [Groups](#) panel, whereas the tiles included in the currently selected group appear in the content area of the page.

Site designers can create groups as predefined content for end users. End users can personalize their sites by modifying existing groups and creating their own groups.

Note

When planning groups, consider that access to a group is assigned to a role and applies to all the tiles within the group. Therefore, do not place tiles that require different permission levels in the same group, for example, tiles for managers and for employees.

Procedure

1. To create a group, click the  (Add group) icon at the bottom of the [Groups](#) panel.
2. In the [Create Group](#) dialog box that opens, enter the group's properties and choose [Save](#). The new group is added to the [Groups](#) panel.

You can perform the following tasks:

Task	Instructions
Edit the group's title	Click  (Edit title). In the dialog box that opens, edit the title and choose Save .
Add, delete or move tiles in the group	Add and Organize Tiles in Groups [page 686]
Delete the group	Drag the group to the trash can image in the lower-left corner of the panel.

Related Information

[Add and Organize Tiles in Groups \[page 686\]](#)

10.4.2.2.2 Add and Organize Tiles in Groups

In a Fiori Launchpad site, you can add and organize tiles in a group.

Context

To add a tile to a group, perform the following steps:

Procedure

1. Open a group and click  (Add tile) in the content area. The *Add Tile to Group <Name of Selected Group>* page opens.
2. Open the catalog dropdown list. The *Catalogs* dialog box opens.
3. Clear the default selection, type the required catalog name or part of it, and choose one of the displayed suggestions. The chosen catalog is displayed below the dropdown list.
4. Click the  (Add tile) icon of the tile that you want to add. The icon changes to  (Tile added), indicating that the tile has been added to the group.
5. Repeat steps 3-4 to add more tiles from the same or other catalogs.
6. Click  (Back) to return to the group.

You can perform the following tasks:

Task	Instructions
Move a tile in the group	Drag a tile to a required position within the group.
Delete a tile from the group	Drag a tile to the trash can image in the lower-right corner of the page.

Related Information

[Create and Edit Groups \[page 685\]](#)

10.4.2.2.3 Select a Site Theme

You can select a theme for a Fiori Launchpad site.

Prerequisites

Site themes are available in your SAP HANA system. For information about creating and importing custom themes, see [Related Information](#).

Procedure

1. Click  (Options) and choose [Select Theme](#). The [Select Theme](#) dialog box opens.
2. Select a theme from the list of available themes and choose [OK](#).

Results

After activation of the site, the selected theme is applied to the runtime version of the site.

Related Information

[Create and Import Custom Themes \[page 687\]](#)

10.4.2.2.3.1 Create and Import Custom Themes

You can create custom themes using the UI theme designer and import these themes into your SAP HANA system.

Context

UI theme designer is a browser-based tool that allows you to develop custom themes by modifying theme templates provided by SAP. For information about this tool, see *UI Theme Designer* under [Related Information](#).

Procedure

1. In the UI theme designer tool, create and export a custom SAPUI5 theme that you want to use for your sites.
A .zip file containing the exported theme is saved on your local machine.
2. Import the exported theme into your project in SAP HANA Studio:
 - a. Copy the contents of the .zip file into your project.
 - b. To register the theme in the THEME database table using the table import mechanism, create the following files in your project:

- myTheme.hdbti

```
import = [  
{  
    hdbtable = "sap.hana.uis.db::THEMES";  
    file = "<package of your project>:myTheme.csv";  
    delimField = ";";  
    header = true; // Mandatory for preventing upgrade errors if the  
    structure of .hdbtable changes  
    keys = ["ID" : "<unique ID from myTheme.csv>"];  
}  
];
```

- myTheme.csv

```
// Mandatory header  
ID;NAME;ROOT_PATH  
// <unique ID>;<name of the theme>;<location of the theme on the SAP  
HANA server>  
// For example: 1;new_sap_bluecrystal;/tests/themes/myTheme/UI5/
```

For information about table import, see *Data Provisioning Using Table Import* under Related Information.

3. Activate the .hdbti and .csv files by choosing **Team** > **Activate** from each file's context menu.
4. Repeat the above steps for any other custom theme that you want to use.

⚠ Caution

A mismatch between the SAPUI5 versions of the UI theme designer and your SAP HANA system might cause unexpected behavior of custom themes.

Related Information

[UI Theme Designer](#)

[Data Provisioning Using Table Import \[page 267\]](#)

10.4.2.3 Configuring Access to Launchpad Content

Configure role-based access to content in SAP Fiori launchpad sites.

Prerequisites

End users are assigned to SAP HANA roles, as well as to the predefined `sap.hana.uis.db::SITE_USER` role. For more information about managing and authorizing users, see *User Provisioning* in the *SAP HANA Administration Guide*.

Context

To enable user access to the content in a launchpad site, you need to assign user roles to content items, such as tile catalogs and groups of tiles.

Note

The predefined designer `sap.hana.uis.db::SITE_DESIGNER` role has access to all content, so there is no need to assign content items to this role.

Procedure

1. To open the site for editing in the embedded browser, in *Project Explorer*, double-click the site's `.xsappsite` file.
2. Click  (Options), and choose *Role Assignment* from the dropdown menu.
The *Role Assignment* page opens. The side panel displays a list of roles defined in your SAP HANA system, and the content area displays assignments of the currently selected role, organized by tabs.
3. In the side panel, select a role to which you want to assign content, and click the relevant content type (*Catalogs* or *Groups*).
4. Perform the following tasks:

Task	Instructions
Search for roles	Enter text in the search box of the side panel, and click  (Search). The role list is filtered by the search text.
Search for content items	Enter text in the search box of the content area, and click  (Search). The list of content items is filtered by the search text.

Task	Instructions
Assign content items to the role	Click  (Assign) to open the assignment dialog box. If needed, search for, and then select one or more content items that you want to assign to the role, and click OK . Selected items are added to the list.
Unassign content items from the role	Select one or more content items that you want to unassign and click  (Unassign). Selected items are removed from the list.

Related Information

[Setting Up Roles and Privileges \[page 692\]](#)

10.4.3 Creating a Standard Site

Context

Note

Standard sites and related features are deprecated as of SAP HANA SPS 09, and are replaced by SAP Fiori launchpad sites.

10.4.4 Configuring the SAP HANA Home Page

You can configure a supplied Fiori Launchpad site or any other application site to serve as the home page for an SAP HANA system.

Context

The supplied Fiori Launchpad site displays a collection of tiles that provide access to various SAP HANA resources and documentation. To configure this or any other site as the home page, you need to configure it as the root page of a HANA system.

Procedure

1. In the *Systems* view of the SAP HANA studio, double-click a system instance.
2. In the system administration page that opens, select the *Configuration* tab.
3. Expand the *xsengine.ini* > *httpserver* nodes.
4. If the *root_page* parameter under *httpserver* does not exist, from the context menu of *httpserver*, choose *Add Parameter*:
 - a. In the *Add Parameter Wizard*, in the *Scope Selection* step, make sure that *System* is selected, and choose *Next*.
 - b. In the *Key Value Pairs* step, in the *Key* field, enter **root_page**.
 - c. In the *Value* field, enter the following URL for a Fiori Launchpad site that you created **/sap/hana/uis/clients/ushell-app/shells/fiori/FioriLaunchpad.html?siteId=<site ID>**.
 - d. Choose *Finish*.
5. If the *root_page* parameter already exists, from its context menu, choose *Change*:
 - a. In the *Change Configuration Wizard*, in the *New Value* field, enter the URL as described in the previous step.
 - b. Choose *Save*.

Results

After logging on to an SAP HANA system machine at `http://<host>:<port>`, the configured home page opens.

11 Setting Up Roles and Privileges

Every user who wants to work directly with the SAP HANA database must have a database user with the necessary privileges. Although privileges can be granted to users directly, roles are the standard way to authorize users. A role is a collection of privileges.

Overview of Roles and Privileges

After users have successfully logged on to SAP HANA, they can do only those things they are authorized to do. This is determined by the privileges that they have been granted. Several privilege types exist in the SAP HANA database, for example system privileges, object privileges, and application privileges.

Privileges can be granted to users directly or indirectly through roles. Roles are the standard mechanism of granting privileges as they allow you to implement complex, reusable authorization concepts that can be modeled on business roles. It is possible to create roles as pure runtime objects that follow classic SQL principles (catalog roles) or as design-time objects in the repository of the SAP HANA database (repository roles). In general, repository roles are recommended as they offer more flexibility. For example, they can be transported between systems. For more information, see *Catalog Roles and Repository Roles Compared* in the *SAP HANA Security Guide*.

Note

Part of the logon process is user authentication. SAP HANA supports several authentication mechanisms, including user name/password authentication and external authentication services such as SAML and Kerberos. For more information, see *SAP HANA Authentication and Single Sign-On* in the *SAP HANA Security Guide*.

Application Authorization

Application developers define application descriptors to specify how users accessing their applications are authenticated and what authorization is required. For more information, see *Creating the Application Descriptors*.

User Management

User administrators are responsible for creating database users and granting them the required roles and privileges. For more information about creating and authorizing users, as well as other user provisioning tasks, see *User Provisioning* in the *SAP HANA Administration Guide*.

Related Information

[SAP HANA Authentication and Single Sign-On](#)
[Creating the Application Descriptors \[page 81\]](#)

11.1 Create a Design-Time Role

You use the role editor of the SAP HANA studio to create a role in the SAP HANA repository.

Prerequisites

- A shared project exists with a suitable package for storing roles.
- You have the system, object, and privileges required for creating and activating objects in the repository. For more information, see *Roles and Permissions*.

Caution

Theoretically, a user with authorization to create and activate repository objects can change a role that he has been granted. Once the role is activated, the user has the new privileges that he or she just added. Therefore, it is important that roles in production systems are imported from a test or development system and that changes to imported objects are not allowed. This danger is however not specific to roles but also applies to other repository objects, for example, modeled views.

- You have granted privileges on any catalog-only objects that you plan to grant in the new role to the technical user `_SYS_REPO`. For more information, see *Roles as Repository Objects*.

Context

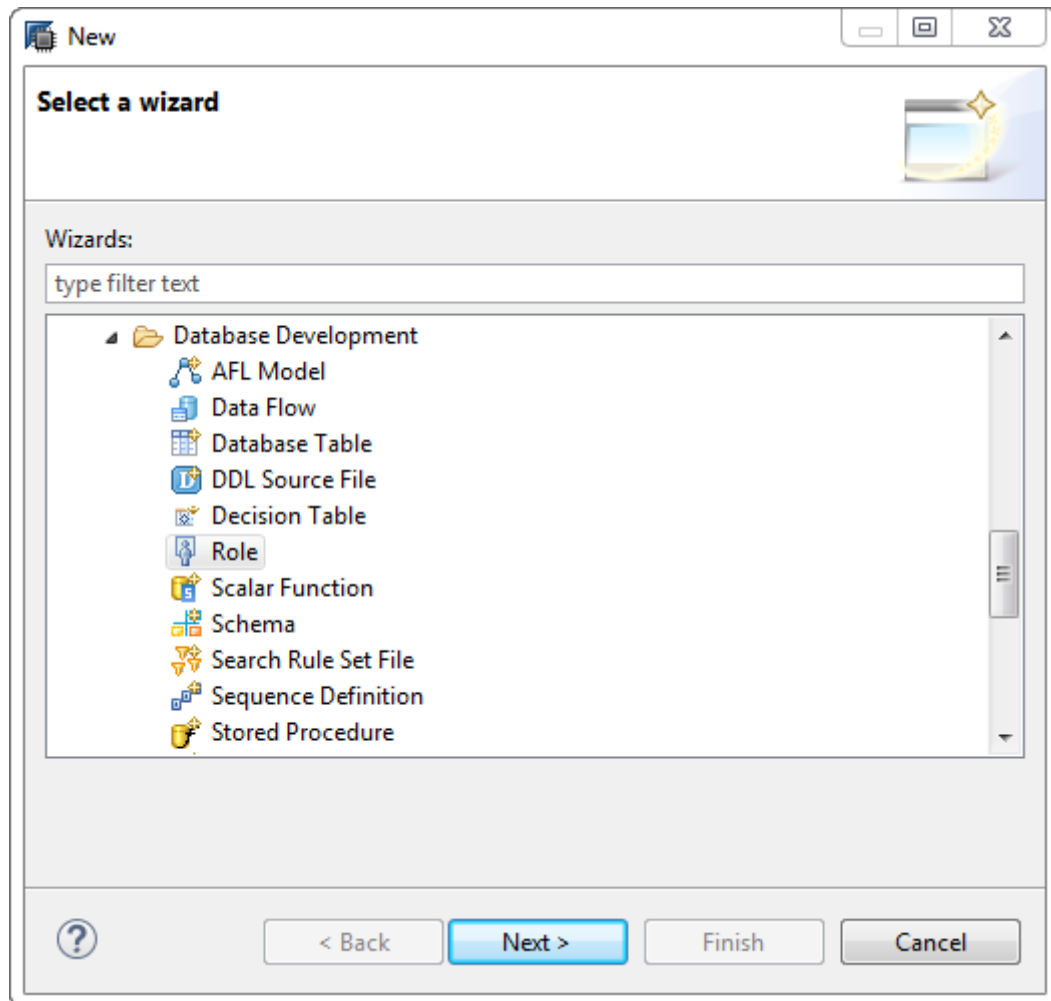
The design-time definition of a role is specified in a text file with the extension `.hdbrole`. In the SAP HANA studio, you create and define a role in a role-specific text editor using the role domain-specific language (DSL) (see *Role Domain-Specific Language Syntax*).

Procedure

1. Create the role:
 - a. From the main menu in the SAP HANA studio, choose **File** > **New** > **Other** > **SAP HANA** > **Database Development** > **Role**.

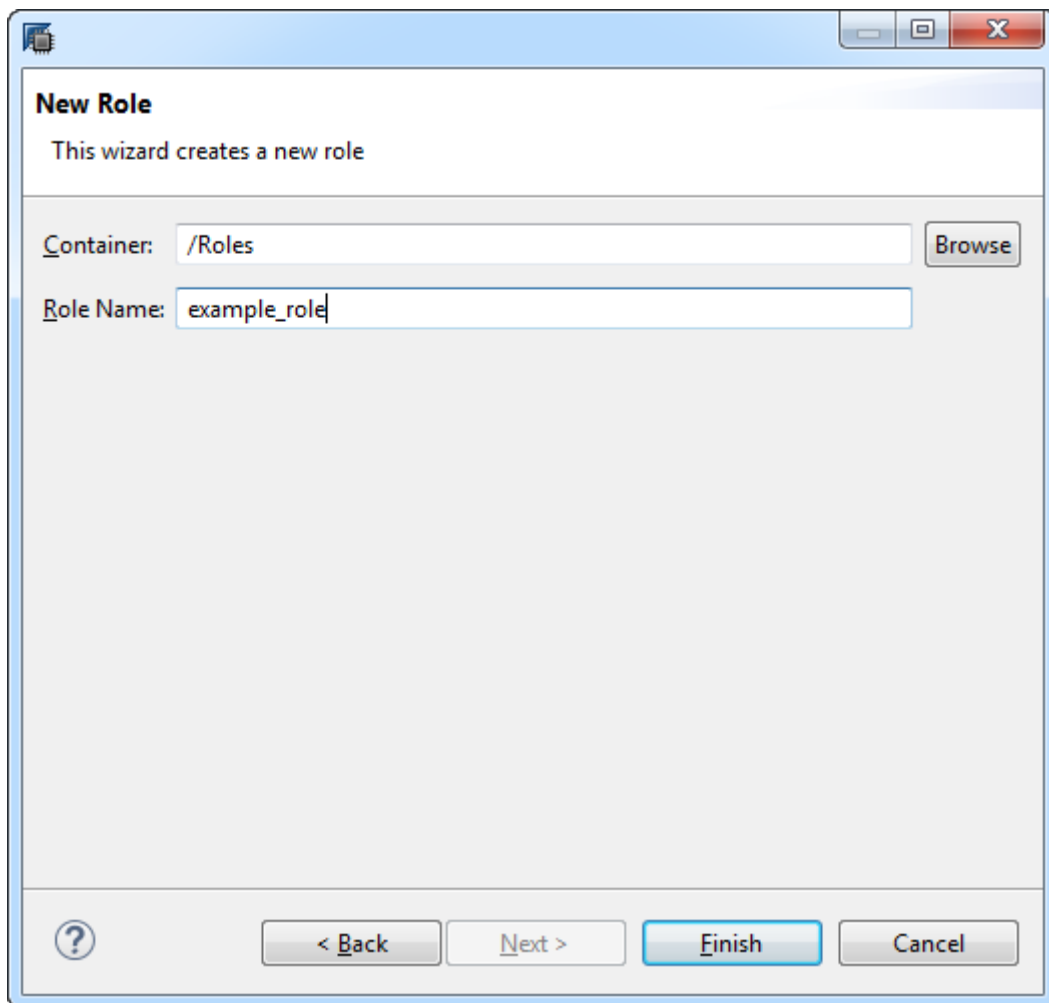
The *New Role* dialog box appears.

Create New Role



- b. In the *Container* field, enter the path to the package where you want to create the role and in the *Role name* field, enter the name of the new role.

Package and Role Name



c. Choose *Finish*.

The new role appears in the *Project Explorer* view and opens in the role text editor as follows:

```
role Roles::example_role
{
}
```

The role is now ready to be defined.

2. Specify the role(s) that you want to grant to the new role.

You can specify both catalog roles and repository roles.

```
Roles::example_role
  extends role sap.example::role1
  extends catalog role "CATROLE2"
{
}
```

3. Grant the required privileges.

Note

Unlike when you create a role using SQL, it is not possible to grant **ALL PRIVILEGES** when you create a role in the repository. You must grant every privilege individually.

- System privileges:

```
{
    // multiple privileges in one line are OK
    system privilege: BACKUP ADMIN, USER ADMIN;
    // you can also split lists into multiple entries
    system privilege: LICENSE ADMIN;
}
```

- Object privileges on design-time objects, that is tables, views, procedures, and sequences:

```
{
    sql object sap.example:MY_VIEW.attributeview: DROP;
    // object privileges can be split across lines
    sql object sap.example:MY_PROCEDURE.hdbprocedure: DROP;
    // a single privilege can be given on multiple objects in a single line
    sql object
    sap.example:MY_VIEW.attributeview, sap.example:MY_OTHER_VIEW.analyticview,
    sap.example:MY_THIRD_VIEW.analyticview: SELECT;
}
```

→ Tip

Many object types can be created in the SAP HANA repository. To verify that you have the correct extension, refer to the object file in the relevant package in the [Project Explorer](#) view.

- Object privileges on catalog objects:

```
{
    // catalog objects must always be qualified with the schema name
    catalog sql object "MY_SCHEMA"."MY_TABLE": SELECT;
}
```

📌 Note

You must always qualify catalog objects with the schema name. You must also reference catalog objects within double quotes, unlike design-time objects.

⚠ Caution

Do not grant object privileges on a catalog object if it was created in design time. If you do, the next time the design-time object is activated (which results in the creation of a new version of the catalog object), the privilege on the original catalog object will be removed from the role. Always grant privileges on design-time objects.

- Privileges on design-time schemas:

```
{
    catalog schema "MY_SCHEMA": SELECT;
    schema sap.example:MY_OTHER_SCHEMA.hdbschema: SELECT;
}
```

📌 Note

You must still use the deprecated extension `. schema` if you are referring to a repository schema that uses this extension.

- Privileges on catalog schemas:

```
{
  catalog schema "MY_SCHEMA": SELECT;
}
```

- Package privileges:

```
{
  package sap.example: REPO.READ;
}
```

- Analytic privileges:

```
{
  analytic privilege: sap.example:sp1.analyticprivilege,
  sap.example:AP2.analyticprivilege;
  catalog analytic privilege: "sp3";
}
```

- Catalog analytic privileges:

```
{
  catalog analytic privilege: "sp3";
}
```

- Application privileges:

```
{
  application privilege: sap.example::Execute;
}
```

Note

Application privileges are implemented using the application-privileges file (`.xsprivileges`).

- Save the role by choosing **File > Save**.
The role is saved as an `.hdbrole` file. After it has been saved, the file is committed to the repository.
- Activate the role by right-clicking it in the *Project Explorer* view and choosing **Team > Activate**.

Caution

Any changes made to a previously activated version of the role in runtime will be reverted on activation. This is to ensure that the design-time version of a role in the repository and its activated runtime version contain the same privileges. It is therefore important that the activated runtime version of a role is not changed in runtime.

Results

The activated role is now visible in the *Systems* view under **Security > Roles** following the naming convention `<package> : <role_name>` and can be granted to users as part of user provisioning. For more information, see *Grant Privileges to Users in the SAP HANA Studio*.

Example: Complete Role Definition Example

```
role Roles::example_role

  extends role sap.example::role1
  extends catalog role "CATROLE1", "CATROLE2"
{
  // system privileges
  system privilege: BACKUP ADMIN, USER ADMIN;
  // schema privileges
  catalog schema "SYSTEM": SELECT;
  schema sap.example:app1.hdbschema: INSERT, UPDATE, DELETE;
  // sql object privileges
  // privileges on the same object may be split up in several lines
  catalog sql object "SYSTEM"."TABLE2": SELECT;
  catalog sql object "SYSTEM"."TABLE2": INSERT, UPDATE, DELETE;
  // or a list of objects may get a list of privileges (object = table, view,
  procedure, sequence)
  // SELECT, DROP for all objects in list
  sql object sap.example:VIEW1.attributeview, sap.example:PROC1.hdbprocedure,
  sap.example:SEQ1.sequence: SELECT, DROP;
  // additional INSERT, UPDATE for TABLE1
  sql object sap.example:MY_VIEW.attributeview: DROP;
  // analytic privileges
  analytic privilege: sap.example:sp1.analyticprivilege,
  sap.example:AP2.analyticprivilege;
  catalog analytic privilege: "sp3";
  // design time privileges
  package sap.example: REPO.EDIT_NATIVE_OBJECTS;
  package sap.example, sap.co: REPO.READ;
  application privilege: sap.example::Execute, sap.example::Save;
}
```

Related Information

[Role Domain-Specific Language Syntax \[page 703\]](#)

[Repository Roles \[page 700\]](#)

[Role Domain-Specific Language Syntax \[page 703\]](#)

[System Privileges \[page 713\]](#)

[Object Privileges \[page 720\]](#)

[Analytic Privileges \[page 728\]](#)

[Package Privileges \[page 730\]](#)

[Application Privileges \[page 731\]](#)

11.1.1 Database Roles

A database role is a collection of privileges that can be granted to either a database user or another role in runtime.

A role typically contains the privileges required for a particular function or task, for example:

- Business end users reading reports using client tools such as Microsoft Excel
- Modelers creating models and reports
- Database administrators operating and maintaining the database and its users

Privileges can be granted directly to users of the SAP HANA database. However, roles are the standard mechanism of granting privileges as they allow you to implement complex, reusable authorization concepts that can be modeled on business roles.

Creation of Roles

Roles in the SAP HANA database can exist as runtime objects only (catalog roles), or as design-time objects that become catalog objects on deployment (database artifact with file suffix `.hdbrole`).

In an SAP HANA XS classic environment, database roles are created in the built-in repository of the SAP HANA database using either the SAP HANA Web Workbench or the SAP HANA studio. These are also referred to as repository roles. In an SAP HANA XS advanced environment, design-time roles are created using the SAP Web IDE and deployed using SAP HANA deployment infrastructure (SAP HANA DI, or HDI).

Design-time roles are created using the SAP Web IDE and deployed using SAP HANA deployment infrastructure (SAP HANA DI, or HDI).

Note

Due to the container-based model of HDI where each container corresponds to a database schema, HDI roles, once deployed, are schema specific.

SAP HANA XS advanced has the additional concept of application roles and role collections. These are independent of database roles in SAP HANA itself. In the XS advanced context, SAP HANA database roles are used only to control access to database objects (for example, tables, views, and procedures) for XS advanced applications. For more information about the authorization concept of XS advanced, see the *SAP HANA Security Guide*.

Role Structure

A role can contain any number of the following privileges:

- **System privileges** for general system authorization, in particular administration activities
- **Object privileges** (for example, SELECT, INSERT, UPDATE) on database objects (for example, schemas, tables, views, procedures, and sequences)
- **Analytic privileges** on SAP HANA information models

- **Package privileges** on repository packages (for example, `REPO.READ`, `REPO.EDIT_NATIVE_OBJECTS`, `REPO.ACTIVATE_NATIVE_OBJECTS`)
- **Application privileges** for enabling access to SAP HANA-based applications developed in an SAP HANA XS classic environment

Note

There are no HDI or XS advanced equivalents in the SAP HANA authorization concept for package privileges on repository packages and applications privileges on SAP HANA XS classic applications. For more information about the authorization concept of XS advanced, see the *SAP HANA Security Guide*.

A role can also contain other roles.

Roles Best Practices

For best performance of role operations, in particular, granting and revoking, keep the following basic rules in mind:

- Create roles with the smallest possible set of privileges for the smallest possible group of users who can share a role (principle of least privilege).
- Avoid granting object privileges at the schema level to a role if only a few objects in the schema are relevant for intended users.
- Avoid creating and maintaining all roles as a single user. Use several role administrator users instead.

Related Information

[Create a Database Role](#)

[Roles \(.hdbrole and .hdbroleconfig\)](#)

[Authorization in SAP HANA XS Advanced](#)

[Create a Design-Time Role \[page 693\]](#)

[Change a Design-Time Role](#)

11.1.1.1 Repository Roles

In an SAP HANA XS classic environment, role developers create database roles as design-time objects in the built-in repository of the SAP HANA database using either the SAP HANA Web Workbench or the SAP HANA studio.

Note

SAP HANA XS classic and the SAP HANA repository are deprecated as of SAP HANA 2.0 SPS 02. For more information, see SAP Note 2465027.

Roles created in the repository differ from roles created directly as runtime objects using SQL in several ways.

What authorization does a user need to grant privileges to a role?

According to the authorization concept of the SAP HANA database, a user can only grant a privilege to a user directly or indirectly in a role if the following prerequisites are met:

- The user has the privilege him- or herself
- The user is authorized to grant the privilege to others (**WITH ADMIN OPTION** or **WITH GRANT OPTION**)

A user is also authorized to grant object privileges on objects that he or she owns.

The technical user `_SYS_REPO` is the owner of all objects in the repository, as well as the runtime objects that are created on activation. This means that when you create a role as a repository object, you can grant the following privileges:

- Privileges that have been granted to the technical user `_SYS_REPO` and that `_SYS_REPO` can grant further. This is automatically the case for system privileges, package privileges, analytic privileges, and application privileges. Therefore, all system privileges, package privileges, analytic privileges, and application privileges can always be granted in design-time roles.
- Privileges on objects that `_SYS_REPO` owns. `_SYS_REPO` owns all activated objects. Object privileges on non-activated runtime objects must be explicitly granted to `_SYS_REPO`.

Note

This is true even for objects belonging to schema `SYS`.

It is recommended that you use a technical user to do this to ensure that privileges are not dropped when the granting user is dropped (for example, because the person leaves the company).

The following table summarizes the situation described above:

Privilege	Action Necessary to Grant in Repository Role
System privilege	None
Package privilege	None
Analytic privilege	None
Application privilege	None
SQL object on activated object (for example, attribute view, analytic view)	None
SQL object privilege on runtime object (for example, replicated table)	Grant privilege to user <code>_SYS_REPO</code> with WITH GRANT OPTION

Note

Technically speaking, only the user `_SYS_REPO` needs the privileges being granted in a role, not the database user who creates the role. However, users creating roles in the SAP HANA Web-based Development Workbench must at least be able to **select** the privileges they want to grant to the role. For this, they need either the system privilege `CATALOG READ` or the actual privilege to be granted.

What about the WITH ADMIN OPTION and WITH GRANT OPTION parameters?

When you create a role using SQL (that is, as a runtime object), you can grant privileges with the additional parameters `WITH ADMIN OPTION` or `WITH GRANT OPTION`. This allows a user who is granted the role to grant the privileges contained within the role to other users and roles. However, if you are implementing your authorization concept with privileges encapsulated within roles created in design time, then you do not **want** users to grant privileges using SQL statements. For this reason, it is not possible to pass the parameters `WITH ADMIN OPTION` or `WITH GRANT OPTION` with privileges when you model roles as repository objects.

Similarly, when you grant an activated role to a user, it is not possible to allow the user to grant the role further (`WITH ADMIN OPTION` is not available).

How are repository roles granted and revoked?

It is not possible to grant and revoke activated design-time roles using the `GRANT` and `REVOKE` SQL statements. Instead, roles are granted and revoked through the execution of the procedures `GRANT_ACTIVATED_ROLE` and `REVOKE_ACTIVATED_ROLE`. Therefore, to be able to grant or revoke a role, a user must have the object privilege `EXECUTE` on these procedures.

How are repository roles dropped?

It is not possible to drop the runtime version of a role created in the repository using the SQL statement `DROP ROLE`. To drop a repository role, you must delete it in the repository and activate the change. The activation process deletes the runtime version of the role.

Can changes to repository roles be audited?

The auditing feature of the SAP HANA database allows you to monitor and record selected actions performed in your database system. One action that is typically audited is changes to user authorization. If you are using roles created in the repository to grant privileges to users, then you audit the creation of runtime roles through activation with the audit action `ACTIVATE REPOSITORY CONTENT`.

Related Information

[SAP Note 2465027](#) 

11.1.1.2 Role Domain-Specific Language Syntax

The design-time definition of a role is specified in a text file with the extension `.hdbrole`. Roles are defined using a domain-specific language (DSL).

❖ Example

```
role Roles::example_role

  extends role sap.example::role1
  extends catalog role "CATROLE1", "CATROLE2"
{
  system privilege: BACKUP ADMIN, USER ADMIN;

  catalog sql object "SYSTEM"."TABLE2": SELECT;
  catalog sql object "SYSTEM"."TABLE2": INSERT, UPDATE, DELETE;
  sql object sap.example:VIEW1.attributeview,
sap.example:PROC1.hdbprocedure, sap.example:SEQ1.sequence: SELECT, DROP;
  sql object sap.example:MY_VIEW.attributeview: DROP;
  catalog schema "SYSTEM": SELECT;
  schema [page 705] sap.example:app1.hdbschema: INSERT, UPDATE, DELETE;

  analytic privilege: sap.example:sp1.analyticprivilege,
sap.example:AP2.analyticprivilege;
  catalog analytic privilege: "sp3";

  package sap.example: REPO.EDIT_NATIVE_OBJECTS;
  package sap.example, sap.co: REPO.READ;
  application privilege: sap.example::Execute, sap.example::Save;
}
```

A role definition specifies the following information:

- The package in which the role is created
- The role name
- Other roles granted to the role
- The privileges granted to the role

The package and role name are specified as follows:

```
role <package_name>::<role_name>
```

The keywords listed below are used to specify which roles and privileges are granted to the role. Keywords other than those listed here are not allowed in a design-time role-definition file (`.hdbrole`).

📌 Note

The following general conventions apply when modeling a role definition using the role DSL:

- Comments start with a double-slash (`//`) or double-dash (`--`) and run to the end of the line.
- When specifying a reference to a design-time object, you must always specify the package name as follows:
 - `<package>::<object>` if you are referencing a design-time role
 - `<package>.<object>.<extension>` if you are referencing any other design-time object
- When specifying multiple privileges on the same object or the same privilege on multiple objects, you can do so individually line by line, or you can group them on a single line. Separate multiple objects and/or multiple privileges using a comma.

extends role

```
extends role <package>.<package>::<role>
```

The keyword `extends role` allows you to include another design-time role in the role. If role A extends role B, role B is granted to role A. This means that effectively A has all privileges that B has.

extends catalog role

```
extends catalog role "<role>"
```

The keyword `extends catalog role` allows you to include a catalog role in the role. If role A extends role B, role B is granted to role A. This means that effectively A has all privileges that B has.

system privilege

```
{  
    system privilege: BACKUP ADMIN, USER ADMIN;  
    system privilege: LICENSE ADMIN;  
}
```

The `system privilege` keyword allows you to grant a system privilege to the role.

For more information about all available system privileges, see *System Privileges (Reference)*.

sql object

```
{  
    sql object sap.example:MY_VIEW.attributeview: DROP;  
    sql object sap.example:MY_PROCEDURE.hdbprocedure: DROP;  
    sql object sap.example:MY_VIEW.attributeview,  
    sap.example:MY_OTHER_VIEW.analyticview, sap.example:MY_THIRD_VIEW.analyticview:  
    SELECT;  
}
```

The `sql object` keyword allows you to grant an object privilege on a design-time object (table, view, procedure, sequence) to the role.

→ Tip

Many object types can be created in the repository. To verify the correct extension, refer to the object file in the relevant package in the *Project Explorer* view (SAP HANA studio) or the file explorer (SAP HANA Web-based Developer Workbench).

For more information about all available object privileges and to which object types they apply, see *Object Privileges (Reference)*.

catalog sql object

```
{  
    catalog sql object "MY_SCHEMA"."MY_TABLE": SELECT;  
}
```

The `catalog sql object` keyword allows you to grant an object privilege on a catalog object (table, view, procedure, sequence) to the role.

Note

Catalog objects must always be qualified with the schema name. Catalog objects must also be referenced within double quotes, unlike design-time objects.

Caution

Do not grant object privileges on a catalog object if it was created in design time. If you do, the next time the design-time object is activated (which results in the creation of a new version of the catalog object), the privilege on the original catalog object will be removed from the role. Therefore, grant privileges on design-time objects.

For more information about all available object privileges and to which object types they apply, see *Object Privileges (Reference)*.

catalog schema

```
{  
    catalog schema "MY_SCHEMA": SELECT;  
}
```

The `catalog schema` keyword allows you to grant a catalog schema to the role.

For more information about the object privileges that apply to schemas, see *Object Privileges (Reference)*.

schema

```
{  
    schema sap.example:MY_OTHER_SCHEMA.hdbschema: SELECT;  
}
```

The `schema` keyword allows you to grant a design-time schema to the role.

Note

You must still use the deprecated extension `.schema` if you are referring to a repository schema that uses this extension.

For more information about the object privileges that apply to schemas, see *Object Privileges (Reference)*.

package

```
{  
    package sap.example: REPO.READ;  
}
```

The `package` keyword allows you to grant a repository package to the role.

For more information about all available package privileges, see *Package Privileges*.

analytic privilege

```
{  
    analytic privilege: sap.example:sp1.analyticprivilege,  
    sap.example:AP2.analyticprivilege;  
}
```

The `analytic privilege` keyword allows you to grant a design-time analytic privilege to the role.

For more information, see *Analytic Privileges*.

catalog analytic privilege

```
{  
    catalog analytic privilege: "sp3";  
}
```

The `catalog analytic privilege` keyword allows you to grant an activated analytic privilege to the role.

For more information, see *Analytic Privileges*.

application privilege

```
{  
    application privilege: sap.example::Execute;  
}
```

The `application privilege` keyword allows you to grant an application privilege to the role.

Note

Application privileges are implemented using the application-privileges file (`.xsprivileges`).

For more information, see *Application Privileges*.

Related Information

[System Privileges \(Reference\) \[page 714\]](#)

[Object Privileges \(Reference\) \[page 721\]](#)

[Package Privileges \[page 730\]](#)

[Analytic Privileges \[page 728\]](#)

[Application Privileges \[page 731\]](#)

[The Application-Privileges File](#)

11.1.1.3 Custom Role for Developers

Create a custom role for developers so that you can to grant developers all required privileges quickly and efficiently.

A role enables you to assign various types of privileges to a user, for example: SQL privileges, analytic privileges, system privileges, as well as application and package privileges. You can also restrict the type of privilege, for example, to `SELECT`, `INSERT` or `UPDATE` statements (or any combination of desired statements). You can use an existing role as the basis for a new, extended, custom role. The privileges granted by an extended role include all the privileges specified in all the roles that are used as the basis of the extended role plus any additional privileges defined in the new extended role itself.

Note

It is not possible to restrict the privileges granted by the existing role that you are extending. For example, if role A extends role B, role A will always include all the privileges specified in role B.

The following example shows how to create a `DEVELOPMENT` role as a design-time object. Note that a role-definition file must have the suffix `.hdbrole`, for example, `MyRoleDefinition.hdbrole`.

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.

After activating the design-time role definition, you can grant the resulting runtime role object to application developers, for example, by executing the `_SYS_REPO` procedure `GRANT_ACTIVATED_ROLE`. The call requires

the parameters: **ROLENAME** (the name of the runtime role object you want to assign) and **USERNAME** (the name of the user to whom you want to assign the new runtime role).

```
call "_SYS_REPO".  
"GRANT_ACTIVATED_ROLE"('acme.com.data:MyUserRole', 'GranteeUserName');
```

The example role illustrated in this topic defines the following privileges for the SAP HANA application developer:

- Schema privileges:
 - **_SYS_BIC**
SELECT and EXECUTE for all tables
- Object privileges:
 - Schema **_SYS_BI**
 - SELECT privilege for all **BIMC_*** tables
 - UPDATE, INSERT, and DELETE privilege for **M_*** tables
 - Catalog object **REPOSITORY_REST (SYS)**
EXECUTE privilege
- Analytic privileges:
 - **_SYS_BI_CP_ALL**
SELECT for the data preview on the views
- Design-time privileges:
 - Package privileges:
 - For the root package
REPO.MAINTAIN_NATIVE_PACKAGES
 - For packages containing application content
REPO.EDIT_NATIVE_OBJECTS
REPO.ACTIVATE_NATIVE_OBJECTS
 - Application privileges:
Application privileges are used to authorize user (and client) access to an application, for example, to start the application or perform administrative actions in the application. When creating a role for developers, make sure that the developers have (at least) the following application privileges:
 - **Execute** and **Save** privileges for the applications the developers are assigned to work on. The application privileges can be defined in a **.xsprivileges** file, which must reside in application package to which the defined privileges apply.
 - The privileges granted with the debugger role that is included with SAP HANA XS.

Note

It is also possible to grant application privileges in SAP HANA studio, for example, using the list of privileges displayed in the [Application Privileges](#) tab in the ► [Security](#) ► [\[Users | Roles\]](#) runtime area. To grant (or revoke) application privileges, the granting (or revoking) user must also have the object privilege **Execute** for the **GRANT_APPLICATION_PRIVILEGE** or **REVOKE_APPLICATION_PRIVILEGE** procedure respectively.

- Additional privileges
User **_SYS_REPO** requires the SELECT privilege on **<schema_where_tables_reside>** to enable the activation and data preview of information views.

Example: Application-Development Role-Definition Example

```
role <package_name>::DEVELOPMENT
// extends role com.acme::role1
// extends catalog role "CATROLE1", "CATROLE2"
{
// system privileges
// system privilege: BACKUP ADMIN, USER ADMIN;
// schema privileges
catalog schema "_SYS_BIC": SELECT, EXECUTE;
// sql object privileges
// privileges on the same object may be split up in several lines
catalog sql object "SYS"."REPOSITORY_REST": EXECUTE;
catalog sql object "_SYS_BI"."BIMC_ALL_CUBES": SELECT;
catalog sql object "_SYS_BI"."BIMC_CONFIGURATION": SELECT;
catalog sql object "_SYS_BI"."BIMC_DIMENSIONS": SELECT;
catalog sql object "_SYS_BI"."BIMC_PROPERTIES": SELECT;
catalog sql object "_SYS_BI"."BIMC_VARIABLE": SELECT;
catalog sql object "_SYS_BI"."BIMC_VARIABLE_ASSIGNMENT": SELECT;
catalog sql object "_SYS_BI"."BIMC_VARIABLE_VALUE": SELECT;
catalog sql object "_SYS_BI"."M_CONTENT_MAPPING": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_FISCAL_CALENDAR": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_IMPORT_SERVER_CONFIG": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_REPLICATION_EXCEPTIONS": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_SCHEMA_MAPPING": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_TIME_DIMENSION": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_TIME_DIMENSION_MONTH": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_TIME_DIMENSION_WEEK": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_TIME_DIMENSION_YEAR": UPDATE, INSERT, DELETE;
catalog sql object "_SYS_BI"."M_USER_PERSONALIZATION": UPDATE, INSERT, DELETE;
// analytic privileges
catalog analytic privilege: "_SYS_BI_CP_ALL";
// design time privileges
package com.acme: REPO.MAINTAIN_NATIVE_PACKAGES;
package com.acme.myapps: REPO.EDIT_NATIVE_OBJECTS;
package com.acme.myapps: REPO.ACTIVATE_NATIVE_OBJECTS;
application privilege: com.acme.myapps.app1::Execute, com.acme.xs.app1::Save;
application privilege: com.acme.myapps.debugger::Execute;
}
```

Related Information

[Repository Roles \[page 700\]](#)

[Create a Design-Time Role](#)

[Create a Design-Time Role \[page 693\]](#)

[The Application-Privileges File](#)

[Privileges \[page 710\]](#)

11.1.2 Privileges

Several privilege types are used in SAP HANA (system, object, analytic, package, and application).

Privilege Type	Applicable To	Target User	Description
System privilege	System, database	Administrators, developers	System privileges control general system activities. They are mainly used for administrative purposes, such as creating schemas, creating and changing users and roles, monitoring and tracing. System privileges are also used to authorize basic repository operations. System privileges granted to users in a particular tenant database authorize operations in that database only. The only exception is the system privileges DATABASE ADMIN, DATABASE STOP, DATABASE START, and DATABASE AUDIT ADMIN. These system privileges can only be granted to users of the system database. They authorize the execution of operations on individual tenant databases. For example, a user with DATABASE ADMIN can create and drop tenant databases, change the database-specific properties in configuration (*.ini) files, and perform database-specific backups.

Privilege Type	Applicable To	Target User	Description
Object privilege	Database objects (schemas, tables, views, procedures and so on)	End users, technical users	Object privileges are used to allow access to and modification of database objects, such as tables and views. Depending on the object type, different actions can be authorized (for example, SELECT, CREATE ANY, ALTER, DROP). Schema privileges are object privileges that are used to allow access to and modification of schemas and the objects that they contain. Source privileges are object privileges that are used to restrict access to and modification of remote data sources, which are connected through SAP HANA smart data access. Object privileges granted to users in a particular database authorize access to and modification of database objects in that database only. That is, unless cross-database access has been enabled for the user. This is made possible through the association of the requesting user with a remote identity on the remote database. For more information, see <i>Cross-Database Authorization in Tenant Databases</i> in the <i>SAP HANA Security Guide</i>.
Analytic privilege	Analytic views	End users	Analytic privileges are used to allow read access to data in SAP HANA information models (that is, analytic views, attribute views, and calculation views) depending on certain values or combinations of values. Analytic privileges are evaluated during query processing. Analytic privileges granted to users in a particular database authorize access to information models in that database only.

Privilege Type	Applicable To	Target User	Description
Package privilege	Packages in the classic repository of the SAP HANA database	Application and content developers working in the classic SAP HANA repository	Package privileges are used to allow access to and the ability to work in packages in the classic repository of the SAP HANA database. Packages contain design time versions of various objects, such as analytic views, attribute views, calculation views, and analytic privileges. Package privileges granted to users in a particular database authorize access to and the ability to work in packages in the repository of that database only. Note With SAP HANA XS advanced, source code and web content are not versioned and stored in the SAP HANA database, so package privileges are not used in this context. For more information, see <i>Authorization in SAP HANA XS Advanced</i>.
Package privilege	Packages in the classic repository of the SAP HANA database	Application and content developers working in the classic SAP HANA repository	Package privileges are not relevant in the SAP HANA service for SAP BTP context as the SAP HANA repository is not supported.
Application privilege	SAP HANA XS classic applications	Application end users, technical users (for SQL connection configurations)	Developers of SAP HANA XS classic applications can create application privileges to authorize user and client access to their application. They apply in addition to other privileges, for example, object privileges on tables. Application privileges can be granted directly to users or roles in runtime in the SAP HANA studio. However, it is recommended that you grant application privileges to roles created in the repository in design time. Note With SAP HANA XS advanced, application privileges are not used. Application-level authorization is implemented using OAuth and authorization scopes and attributes. For more information, see <i>Authorization in SAP HANA XS Advanced</i>.
Application privilege	SAP HANA XS classic applications	Application end users, technical users (for SQL connection configurations)	Application privileges are not relevant in the SAP HANA service for SAP BTP context as SAP HANA XS classic is not supported.

Note

There are no HDI or XS advanced equivalents in the SAP HANA authorization concept for package privileges on repository packages and applications privileges on SAP HANA XS classic applications. For more information about the authorization concept of XS advanced, see the *SAP HANA Security Guide*.

Privileges on Users

An additional privilege type, privileges on users, can be granted to users. Privileges on users are SQL privileges that users can grant on their user. ATTACH DEBUGGER is the only privilege that can be granted on a user.

For example, User A can grant User B the privilege ATTACH DEBUGGER to allow User B debug SQLScript code in User A's session. User A is only user who can grant this privilege. Note that User B also needs the object privilege DEBUG on the relevant SQLScript procedure.

For more information, see the section on debugging procedures in the *SAP HANA Developer Guide*.

Related Information

GRANT

[Cross-Database Authorization in Tenant Databases](#)

[Authorization in SAP HANA XS Advanced](#)

[Debug an External Session \[page 418\]](#)

[Recommendations for Database Users, Roles, and Privileges](#)

11.1.2.1 System Privileges

System privileges control general system activities.

System privileges are mainly used to authorize users to perform administrative actions, including:

- Creating and deleting schemas
- Managing users and roles
- Performing data backups
- Monitoring and tracing
- Managing licenses

System privileges are also used to authorize basic repository operations, for example:

- Importing and exporting content
- Maintaining delivery units (DU)

System privileges granted to users in a particular database authorize operations in that database only. The only exception is the system privileges DATABASE ADMIN, DATABASE STOP, and DATABASE START, DATABASE AUDIT ADMIN. These system privileges can only be granted to users of the system database. They authorize the execution of operations on individual tenant databases. For example, a user with DATABASE ADMIN can create and drop tenant databases, change the database-specific properties in configuration (*.ini) files, and perform database-specific or full-system data backups.

Note

System privileges should always be assigned with caution.

Related Information

[System Privileges \(Reference\) \[page 714\]](#)

[SAP Note 2950209](#)

11.1.2.1.1 System Privileges (Reference)

System privileges control general system activities.

General System Privileges

System privileges restrict administrative tasks. The following table describes the supported system privileges in an SAP HANA database.

System Privilege	Description
ADAPTER ADMIN	Controls the execution of the following adapter-related statements: CREATE ADAPTER, DROP ADAPTER, and ALTER ADAPTER. It also allows access to the ADAPTERS and ADAPTER_LOCATIONS system views.
AGENT ADMIN	Controls the execution of the following agent-related statements: CREATE AGENT, DROP AGENT, and ALTER AGENT. It also allows access to the AGENTS and ADAPTER_LOCATIONS system views.
ALTER CLIENTSIDE ENCRYPTION KEYPAIR	Authorizes a user to add a new version of a client-side encryption key pair (CKP), or to drop all older versions of the CKP.
AUDIT ADMIN	Controls the execution of the following auditing-related statements: CREATE AUDIT POLICY, DROP AUDIT POLICY, and ALTER AUDIT POLICY, as well as changes to the auditing configuration. It also allows access to the AUDIT_LOG, XSA_AUDIT_LOG, and ALL_AUDIT_LOG system views.

System Privilege	Description
AUDIT OPERATOR	Authorizes the execution of the following statement: ALTER SYSTEM CLEAR AUDIT LOG. It also allows access to the AUDIT_LOG system view.
AUDIT READ	Authorizes read-only access to the rows of the AUDIT_LOG, XSA_AUDIT_LOG, and ALL_AUDIT_LOG system views.
BACKUP ADMIN	Authorizes BACKUP and RECOVERY statements for defining and initiating backup and recovery procedures. It also authorizes changing system configuration options with respect to backup and recovery.
BACKUP OPERATOR	Authorizes the BACKUP statement to initiate a backup.
CATALOG READ	Authorizes unfiltered access to the data in the system views that a user has already been granted the SELECT privilege on. Normally, the content of these views is filtered based on the privileges of the user. CATALOG READ does not allow a user to view system views on which they have not been granted the SELECT privilege.
CERTIFICATE ADMIN	Authorizes the changing of certificates and certificate collections that are stored in the database.
CLIENT PARAMETER ADMIN	Authorizes a user to override the value of the CLIENT parameter for a database connection or to overwrite the value of the \$\$client\$\$ parameter in an SQL query.
CREATE CLIENTSIDE ENCRYPTION KEYPAIR	Authorizes a user to create client-side encryption key pairs.
CREATE REMOTE SOURCE	Authorizes the creation of remote data sources by using the CREATE REMOTE SOURCE statement.
CREATE SCENARIO	Controls the creation of calculation scenarios and cubes (calculation database).
CREATE SCHEMA	Authorizes the creation of database schemas using the CREATE SCHEMA statement.
CREATE STRUCTURED PRIVILEGE	Authorizes the creation of structured (analytic privileges). Only the owner of the privilege can further grant or revoke that privilege to other users or roles.
CREDENTIAL ADMIN	Authorizes the use of the statements CREATE CREDENTIAL, ALTER CREDENTIAL, and DROP CREDENTIAL.

System Privilege	Description
DATA ADMIN	Authorizes reading all data in the system views. It also enables execution of Data Definition Language (DDL) statements in the SAP HANA database. A user with this privilege cannot select or change data in stored tables for which they do not have access privileges, but they can drop tables or modify table definitions.
DATABASE ADMIN*	Authorizes all statements related to tenant databases, such as CREATE, DROP, ALTER, RENAME, BACKUP, and RECOVERY.
DATABASE AUDIT ADMIN*	Authorizes a user to create and manage audit policies for a tenant database.
DATABASE BACKUP ADMIN*	Authorizes all statements related to view, create, schedule, and cancel database backups except for recovering or copying a database.
DATABASE BACKUP OPERATOR*	Authorizes a user to view, create, schedule, and cancel database backups.
DATABASE RECOVERY OPERATOR*	Authorizes a user to recover or copy a database.
DATABASE START*	Authorizes a user to start any database in the system and to select from the M_DATABASES view.
DATABASE STOP*	Authorizes a user to stop any database in the system and to select from the M_DATABASES view.
DROP CLIENTSIDE ENCRYPTION KEYPAIR	Authorizes a user to drop other users' client-side encryption key pairs.
ENCRYPTION ROOT KEY ADMIN	Authorizes all statements related to management of root keys: Allows access to the system views pertaining to encryption (for example, ENCRYPTION_ROOT_KEYS, M_ENCRYPTION_OVERVIEW, M_PERSISTENCE_ENCRYPTION_STATUS, M_PERSISTENCE_ENCRYPTION_KEYS, and so on).
EXPORT	Authorizes EXPORT to a file on the SAP HANA server. The user must also have the SELECT privilege on the source tables to be exported.
EXTENDED STORAGE ADMIN	Authorizes the management of SAP HANA dynamic tiering and the creation of extended storage.

System Privilege	Description
IMPORT	Authorizes the import activity in the database using the IMPORT statements. Additional privileges may also be required to be able to execute an IMPORT. See the IMPORT statement for more information.
INIFILE ADMIN	Authorizes making changes to system settings.
LDAP ADMIN	Authorizes the use of the CREATE ALTER DROP VALIDATE LDAP PROVIDER statements.
LICENSE ADMIN	Authorizes the use of the SET SYSTEM LICENSE statement to install a new license.
LOG ADMIN	Authorizes the use of the ALTER SYSTEM LOGGING [ON OFF] statements to enable or disable the log flush mechanism.
MONITOR ADMIN	Authorizes the use of the ALTER SYSTEM statements for events.
OPTIMIZER ADMIN	Authorizes the use of the ALTER SYSTEM statements concerning SQL PLAN CACHE and ALTER SYSTEM UPDATE STATISTICS statements, which influence the behavior of the query optimizer.
PARTITION ADMIN	Authorizes the use of all non-destructive partitioning operations when altering a table.
RESOURCE ADMIN	Authorizes statements concerning system resources (for example, the ALTER SYSTEM RECLAIM DATAVOLUME and ALTER SYSTEM RESET MONITORING VIEW statements). It also authorizes use of the Kernel Profiler statements, and many of the statements available in the Management Console.
ROLE ADMIN	Authorizes the creation and deletion of roles by using the CREATE ROLE and DROP ROLE statements. It also authorizes the granting and revoking of roles by using the GRANT and REVOKE statements. Activated repository roles, meaning roles whose creator is the predefined user _SYS_REPO, can neither be granted to other roles or users nor dropped directly. Not even users with the ROLE ADMIN privilege can do so. Check the documentation concerning activated objects.
SAVEPOINT ADMIN	Authorizes the execution of a savepoint using the ALTER SYSTEM SAVEPOINT statement.

System Privilege	Description
SCENARIO ADMIN	Authorizes all calculation scenario-related activities (including creation).
SERVICE ADMIN	Authorizes the ALTER SYSTEM [START CANCEL RECONFIGURE] statements for administering system services of the database.
SESSION ADMIN	Authorizes the ALTER SYSTEM commands concerning sessions to stop or disconnect a user session or to change session variables.
SSL ADMIN	Authorizes the use of the SET...PURPOSE SSL statement. It also allows access to the PSES system view.
STRUCTUREDPRIVILEGE ADMIN	Authorizes the creation, reactivation, and dropping of structured (analytic) privileges.
SYSTEM REPLICATION ADMIN	Authorizes the use of ALTER SYSTEM statements related to system replication.
TABLE ADMIN	Authorizes the LOAD, UNLOAD and MERGE DELTA statements for tables and table partitions, as well as the ALTER TABLE statement for those clauses that do change the structure of the table and do not allow access to table data either explicitly or implicitly, for example: LOB REORGANIZE, CLEAR COLUMN JOIN DATA STATISTICS, and PRELOAD.
TRACE ADMIN	Authorizes the use of the ALTER SYSTEM statements related to database tracing (including the Kernel Profiler feature) and the changing of trace system settings.
TRUST ADMIN	Authorizes the use of statements to update the trust store.
USER ADMIN	Authorizes the creation and modification of users by using the CREATE ALTER DROP USER statements.
VERSION ADMIN	Authorizes the use of the ALTER SYSTEM RECLAIM VERSION SPACE statement of the multi-version concurrency control (MVCC) feature.
WORKLOAD ADMIN	Authorizes execution of the workload class and mapping statements (for example, CREATE ALTER DROP WORKLOAD CLASS, and CREATE ALTER DROP WORKLOAD MAPPING).
WORKLOAD ANALYZE ADMIN	Used by the Analyze Workload, Capture Workload, and Replay Workload applications when performing workload analysis.

System Privilege	Description
WORKLOAD CAPTURE ADMIN	Authorizes access to the monitoring view M_WORKLOAD_CAPTURES to see the current status of capturing and captured workloads, as well of execution of actions with the WORKLOAD_CAPTURE procedure.
WORKLOAD REPLAY ADMIN	Authorizes access to the monitoring views M_WORKLOAD_REPLAY_PREPROCESSES and M_WORKLOAD_REPLAYS to see current status of preprocessing, preprocessed, replaying, and replayed workloads, as well as the execution of actions with the WORKLOAD_REPLAY procedure.
<identifier>.<identifier>	Components of the SAP HANA database can create new system privileges. These privileges use the component-name as the first identifier of the system privilege and the component-privilege-name as the second identifier.

Repository System Privileges

Note

The following privileges authorize actions on individual packages in the SAP HANA repository, used in the SAP HANA Extended Services (SAP HANA XS) classic development model. With SAP HANA XS advanced, source code and web content are no longer versioned and stored in the repository of the SAP HANA database.

System Privilege	Description
REPO.EXPORT	Authorizes the export of delivery units for example
REPO.IMPORT	Authorizes the import of transport archives
REPO.MAINTAIN_DELIVERY_UNITS	Authorizes the maintenance of delivery units (DU, DU vendor and system vendor must be the same)
REPO.WORK_IN_FOREIGN_WORKSPACE	Authorizes work in a foreign inactive workspace
REPO.CONFIGURE	Authorize work with SAP HANA Change Recording, which is part of SAP HANA Application Lifecycle Management
REPO.MODIFY_CHANGE	
REPO.MODIFY_OWN_CONTRIBUTION	
REPO.MODIFY_FOREIGN_CONTRIBUTION	

Related Information

[GRANT](#)

[Developer Authorization in the Repository](#)

11.1.2.2 Object Privileges

Object privileges are SQL privileges that are used to allow access to and modification of database objects.

For each SQL statement type (for example, SELECT, UPDATE, or CALL), a corresponding object privilege exists. If a user wants to execute a particular statement on a simple database object (for example, a table), he or she must have the corresponding object privilege for either the actual object itself, or the schema in which the object is located. This is because the schema is an object type that contains other objects. A user who has object privileges for a schema automatically has the same privileges for all objects currently in the schema and any objects created there in the future.

Object privileges are not only grantable for database catalog objects such as tables, views and procedures. Object privileges can also be granted for non-catalog objects such as development objects in the repository of the SAP HANA database.

Initially, the owner of an object and the owner of the schema in which the object is located are the only users who can access the object and grant object privileges on it to other users.

An object can therefore be accessed only by the following users:

- The owner of the object
- The owner of the schema in which the object is located
- Users to whom the owner of the object has granted privileges
- Users to whom the owner of the parent schema has granted privileges

Caution

The database owner concept stipulates that when a database user is deleted, all objects created by that user and privileges granted to others by that user are also deleted. If the owner of a schema is deleted, all objects in the schema are also deleted even if they are owned by a different user. All privileges on these objects are also deleted.

Note

The owner of a table can change its ownership with the `ALTER TABLE` SQL statement. In this case, the new owner becomes the grantor of all privileges on the table granted by the original owner. The original owner is also automatically granted all privileges for the table with the new owner as grantor. This ensures that the original owner can continue to work with the table as before.

Authorization Check on Objects with Dependencies

The authorization check for objects defined on other objects (that is, stored procedures and views) is more complex. In order to be able to access an object with dependencies, both of the following conditions must be met:

- The user trying to access the object must have the relevant object privilege on the object as described above.
- The user who created the object must have the required privilege on all underlying objects **and** be authorized to grant this privilege to others.

If this second condition is not met, only the owner of the object can access it. He cannot grant privileges on it to any other user. This cannot be circumvented by granting privileges on the parent schema instead. Even if a user has privileges on the schema, he will still not be able to access the object.

Note

This applies to procedures created in DEFINER mode only. This means that the authorization check is run against the privileges of the user who created the object, not the user accessing the object. For procedures created in INVOKER mode, the authorization check is run against the privileges of the accessing user. In this case, the user must have privileges not only on the object itself but on all objects that it uses.

→ Tip

The SAP HANA studio provides a graphical feature, the authorization dependency viewer, to help troubleshoot authorization errors for object types that typically have complex dependency structures: stored procedures and calculation views.

Related Information

[GRANT Statement \(Access Control\)](#)
[ALTER TABLE Statement \(Data Definition\)](#)
[Resolve Errors Using the Authorization Dependency Viewer](#)
[Object Privileges \(Reference\) \[page 721\]](#)
[Cross-Database Authorization in Tenant Databases](#)

11.1.2.2.1 Object Privileges (Reference)

Object privileges are used to allow access to and modification of database objects, such as tables and views.

The following table describes the supported object privileges in an SAP HANA database.

Object Privilege	Command Types	Applies to	Privilege Description
ALL PRIVILEGES	DDL & DML	- Schemas - Tables - Views	This privilege is a collection of all Data Definition Language (DDL) and Data Manipulation Language (DML) privileges that the grantor currently possesses and is allowed to grant further. The privilege it grants is specific to the particular object being acted upon. This privilege collection is dynamically evaluated for the given grantor and object.
ALTER	DDL	- Schemas - Tables - Views - Functions/procedures	Authorizes the ALTER statement for the object.
AGENT MESSAGING	DDL	- Schemas - Tables - Views - Functions/procedures	Authorizes the user with which the agent communicates with the data provisioning server using HTTP protocol.
ATTACH DEBUGGER	DDL	User	Authorizes debugging across different user sessions. For example, userA can grant ATTACH DEBUGGER to userB to allow userB to debug a procedure in userA's session (userB still needs DEBUG privilege on the procedure, however).
CREATE ANY	DDL	- Schemas - Tables - Views - Sequences - Functions/procedures - Remote sources - Graph workspaces - Triggers	Authorizes all CREATE statements for the object.

Object Privilege	Command Types	Applies to	Privilege Description
CREATE OBJECT STRUCTURED PRIVILEGE	DDL	- Schemas - Views	Authorizes creation of structured privileges on the object even if the user does not have the CREATE STRUCTURED PRIVILEGE.
CREATE REMOTE SUBSCRIPTION	DDL	Remote Source	Authorizes the creation of remote subscriptions executed on this source entry.
CREATE VIRTUAL FUNCTION	DDL	Remote sources	Authorizes creation of virtual functions (the REFERENCES privilege is also required).
CREATE VIRTUAL PROCEDURE	DDL	Remote sources	Authorizes creation of virtual procedure to create and run procedures on a remote source.
CREATE VIRTUAL PACKAGE	DDL	Schemas	Authorizes creation of virtual packages that can be run on remote sources.
CREATE VIRTUAL TABLE	DDL	Remote sources	Authorizes the creation of proxy tables pointing to remote tables from the source entry.
CREATE TEMPORARY TABLE	DDL	Schemas	Authorizes the creation of a temporary local table, which can be used as input for procedures, even if the user does not have the CREATE ANY privilege for the schema.
DEBUG	DML	- Schemas - Calculation Views - Functions/procedures	Authorizes debug functionality for the procedure or calculation view or for the procedures and calculation views of a schema.
DEBUG MODIFY	DDL	Functions/procedures	For internal use only.

Object Privilege	Command Types	Applies to	Privilege Description
DELETE	DML	- Schemas - Tables - Views - Functions/procedures	Authorizes the DELETE and TRUNCATE statements for the object. While DELETE applies to views, it only applies to updatable views (that is, views that do not use a join, do not contain a UNION, and do not use aggregation).
DROP	DDL	- Schemas - Tables - Views - Sequences - Functions/procedures - Remote sources - Graph workspaces	Authorizes the DROP statements for the object.
EXECUTE	DML	- Schemas - Functions/procedures	Authorizes the execution of a SQLScript function or a database procedure by using the CALLS or CALL statement respectively. It also allows a user to execute a virtual function.
INDEX	DDL	- Schemas - Tables	Authorizes the creation, modification, or dropping of indexes for the object.
INSERT	DML	- Schemas - Tables - Views	Authorizes the INSERT statement for the object. The INSERT and UPDATE privilege are both required on the object to allow the REPLACE and UPSERT statements to be used. While INSERT applies to views, it only applies to updatable views (views that do not use a join, do not contain a UNION, and do not use aggregation).

Object Privilege	Command Types	Applies to	Privilege Description
PROCESS REMOTE SUBSCRIPTION EXCEPTION	DDL	- Remote Source - Remote Subscription	Authorizes processing exceptions on this source entry.
LINKED DATABASE	DDL	Remote source	Authorizes execution of linked database queries.
REFERENCES	DDL	- Schemas - Tables	Authorizes the usage of all tables in this schema or this table in a foreign key definition, or the usage of a personal security environment (PSE). It also allows a user to reference a virtual function package.
REMOTE TABLE ADMIN	DDL	Remote sources	Authorizes the creation of tables on a remote source object.
SELECT	DML	- Schemas - Tables - Views - Sequences - Graph workspaces	Authorizes the SELECT statement for the object or the usage of a sequence. When selection from system-versioned tables, users must have SELECT on both the table and its associated history table.
SELECT CDS METADATA	DML	- Schemas - Tables	Authorizes access to CDS metadata from the catalog.
SELECT METADATA	DML	- Schemas - Tables	Authorizes access to the complete metadata of all objects in a schema (including procedure and view definitions), including objects that may be located in other schemas.
SQLSCRIPT LOGGING	DML	- Function - Library - Procedure - Schema	Authorizes the collection of logs for a SQLScript object.

Object Privilege	Command Types	Applies to	Privilege Description
TRIGGER	DDL	- Schemas - Tables	Authorizes the CREATE/ALTER/DROP/ENABLE and DISABLE TRIGGER statements for the specified table or the tables in the specified schema.
UNMASKED	DML	- Schemas - Views - Tables	Authorizes access to masked data in user-defined views and tables. This privilege is required to view the original data in views and tables that are defined by using the WITH MASK clause.
UPDATE	DML	- Schemas - Tables - Views	While UPDATE applies to views, it only applies to updatable views (views that do not use a join, do not contain a UNION, and do not use aggregation).
USAGE	DDL	Client Side Encryption Column Key	Authorizes client side encryption keys.

Object Privilege	Command Types	Applies to	Privilege Description
USERGROUP OPERATOR	DML	User groups	Authorizes a user to change the settings for a user group, and to add and remove users to/from a user group. Users with the USERGROUP OPERATOR privilege can also create and drop users, but only within the user group they have the USERGROUP OPERATOR privilege on (CREATE USER <i><user_name></i> SET USERGROUP <i><usergroup_name></i>). A user can have the USERGROUP OPERATOR privilege on more than one user group, and a user group can have more than one user with the USERGROUP OPERATOR privilege on it. When granting USERGROUP OPERATOR to a user group, you must include the keyword USERGROUP before the name of the user group (for example: GRANT USERGROUP OPERATOR ON USERGROUP <i><usergroup></i> TO <i><grantee></i>). This is slightly different syntax than granting USERGROUP OPERATOR to a user.
<i><identifier>.<identifier></i>	DDL		Components of the SAP HANA database can create new object privileges. These privileges use the component-name as first identifier of the system privilege and the component-privilege-name as the second identifier.

Related Information

[GRANT](#)

11.1.2.3 Analytic Privileges

Analytic privileges grant different users access to different portions of data in the same view based on their business role. Within the definition of an analytic privilege, the conditions that control which data users see is either contained in an XML document or defined using SQL.

Row-Level Access Control

Standard object privileges (SELECT, ALTER, DROP, and so on) implement coarse-grained authorization at object level only. Users either have access to an object, such as a table, view or procedure, or they don't. While this is often sufficient, there are cases when access to data in an object depends on certain values or combinations of values. Analytic privileges are used in the SAP HANA database to provide such fine-grained control at row level of which data individual users can see within the same view.

❖ Example

Sales data for all regions are contained within one analytic view. However, regional sales managers should only see the data for their region. In this case, an analytic privilege could be modeled so that they can all query the view, but only the data that each user is authorized to see is returned.

Creating Analytic Privileges

Although analytic privileges can be created directly as catalog objects in runtime, we recommend creating them as design-time objects that become catalog objects on deployment (database artifact with file suffix `.hdbanalyticprivilege`).

In an SAP HANA XS classic environment, analytic privileges are created in the built-in repository of the SAP HANA database using either the SAP HANA Web Workbench or the SAP HANA studio. In an SAP HANA XS advanced environment, they are created using the SAP Web IDE and deployed using the SAP HANA deployment infrastructure (SAP HANA DI).

📘 Note

HDI supports only analytic privileges deployed using SQL (see below). Furthermore, due to the container-based model of HDI, where each container corresponds to a database schema, analytic privileges created in HDI are schema specific.

XML- Versus SQL-Based Analytic Privileges

Before you implement row-level authorization using analytic privileges, you need to decide which type of analytic privilege is suitable for your scenario. In general, SQL-based analytic privileges allow you to more easily formulate complex filter conditions using sub-queries that might be cumbersome to model using XML-based analytic privileges.

→ Recommendation

SAP recommends the use of SQL-based analytic privileges. Using the [SAP HANA Modeler](#) perspective of the SAP HANA studio, you can migrate XML-based analytic privileges to SQL-based analytic privileges. For more information, see the *SAP HANA Modeling Guide for SAP HANA Studio*.

📘 Note

As objects created in the repository, XML-based analytic privileges are deprecated as of SAP HANA 2.0 SPS 02. For more information, see SAP Note 2465027.

The following are the main differences between XML-based and SQL-based analytic privileges:

Feature	SQL-Based Analytic Privileges	XML-Based Analytic Privileges
Control of read-only access to SAP HANA information models: • Attribute views • Analytic views • Calculation views	Yes	Yes
Control of read-only access to SQL views	Yes	No
Control of read-only access to database tables	No	No
Design-time modeling using the SAP HANA Web-based Workbench or the SAP HANA Modeler perspective of the SAP HANA studio	Yes	Yes
📘 Note This corresponds to development in an SAP HANA XS classic environment using the SAP HANA repository.		
Design-time modeling using the SAP Web IDE for SAP HANA	Yes	No
📘 Note This corresponds to development in an SAP HANA XS advanced environment using HDI.		
Transportable	Yes	Yes
HDI support	Yes	No
Complex filtering	Yes	No

Enabling an Authorization Check Based on Analytic Privileges

All column views modeled and activated in the SAP HANA modeler and the SAP HANA Web-based Development Workbench automatically enforce an authorization check based on analytic privileges. XML-based analytic privileges are selected by default, but you can switch to SQL-based analytic privileges.

Column views created using SQL must be explicitly registered for such a check by passing the relevant parameter:

- `REGISTerviewFORAPCHECK` for a check based on XML-based analytic privileges
- `STRUCTURED PRIVILEGE CHECK` for a check based on SQL-based analytic privileges

SQL views must always be explicitly registered for an authorization check based on analytic privileges by passing the `STRUCTURED PRIVILEGE CHECK` parameter.

Note

It is not possible to enforce an authorization check on the same view using both XML-based and SQL-based analytic privileges. However, it is possible to build views with different authorization checks on each other.

Related Information

[Create Static SQL Analytic Privileges \(SAP Web IDE for SAP HANA\)](#)

[Create Dynamic SQL Analytic Privileges \(SAP Web IDE for SAP HANA\)](#)

[Create Analytic Privileges Using SQL Expressions \(SAP Web IDE for SAP HANA\)](#)

[Create Classical XML-Based Analytic Privileges \(SAP HANA Web Workbench\)](#)

[Create Static SQL Analytic Privileges \(SAP HANA Web Workbench\)](#)

[Create Classical XML-based Analytic Privileges \(SAP HANA Studio\) \[page 734\]](#)

[Create SQL Analytic Privileges \(SAP HANA Studio\) \[page 737\]](#)

[Convert Classical XML-based Analytic Privileges to SQL-based Analytic Privileges \(SAP HANA Studio\)](#)

[SAP Note 2465027](#) 

11.1.2.4 Package Privileges

Package privileges authorize actions on individual packages in the SAP HANA repository.

Privileges granted on a repository package are implicitly assigned to the design-time objects in the package, as well as to all sub-packages. Users are only allowed to maintain objects in a repository package if they have the necessary privileges for the package in which they want to perform an operation, for example to read or write to an object in that package. To be able perform operations in all packages, a user must have privileges on the root package `.REPO_PACKAGE_ROOT`.

If the user authorization check establishes that a user does not have the necessary privileges to perform the requested operation in a specific package, the authorization check is repeated on the parent package and recursively up the package hierarchy to the root level of the repository. If the user does not have the necessary privileges for any of the packages in the hierarchy chain, the authorization check fails and the user is not permitted to perform the requested operation.

In the context of repository package authorizations, there is a distinction between native packages and imported packages.

- **Native package**
A package that is created in the current system and expected to be edited in the current system. Changes to packages or to objects the packages contain must be performed in the original development system where they were created and transported into subsequent systems. The content of native packages are regularly edited by developers.
- **Imported package**
A package that is created in a remote system and imported into the current system. Imported packages should not usually be modified, except when replaced by new imports during an update. Otherwise, imported packages or their contents should only be modified in exceptional cases, for example, to carry out emergency repairs.

Note

The SAP HANA administrator can grant the following package privileges to an SAP HANA user: edit, activate, and maintain.

Related Information

[Package Privilege Options \[page 77\]](#)

11.1.2.5 Application Privileges

In SAP HANA Extended Application Services (SAP HANA XS), application privileges define the authorization level required for access to an SAP HANA XS application, for example, to start the application or view particular functions and screens.

Application privileges can be assigned to an individual user or to a group of users, for example, in a user **role**. The user role can also be used to assign system, object, package, and analytic privileges, as illustrated in the following graphic. You can use application privileges to provide different levels of access to the same application, for example, to provide advanced maintenance functions for administrators and view-only capabilities to normal users.

X4D (SYSTEM) xsdev.sap.corp 00	
acme.com.hana.xs.appl.roles::Display	
Granted Roles Part of Roles System Privileges Object Privileges Analytic Privileges Package Privileges Application Privileges	
+ x	
Application Privilege	Grantor
acme.com.hana.xs.appl::Display	_SYS_REPO
acme.com.hana.xs.appl::View	_SYS_REPO

Application Privileges for Users and User Roles

If you want to define application-specific privileges, you need to understand and maintain the relevant sections in the following design-time artifacts:

- Application-privileges file (.xsprivileges)
- Application-access file (.xsaccess)
- Role-definition file (<RoleName>.hdbrole)

Application privileges can be assigned to users individually or by means of a user **role**, for example, with the “*application privilege*” keyword in a role-definition file (<RoleName>.hdbrole) as illustrated in the following code. You store the roles as design-time artifacts within the application package structure they are intended for, for example, acme.com.hana.xs.appl.roles.

```
role acme.com.hana.xs.appl.roles::Display
{
  application privilege: acme.com.hana.xs.appl::Display;
  application privilege: acme.com.hana.xs.appl::View;
  catalog schema "ACME_XS_APP1": SELECT;
  package acme.com.hana.xs.appl: REPO.READ;
  package ".REPO_PACKAGE_ROOT" : REPO.READ;
  catalog sql object "_SYS_REPO"."PRODUCTS": SELECT;
  catalog sql object "_SYS_REPO"."PRODUCT_INSTANCES": SELECT;
  catalog sql object "_SYS_REPO"."DELIVERY_UNITS": SELECT;
  catalog sql object "_SYS_REPO"."PACKAGE_CATALOG": SELECT;
  catalog sql object "ACME_XS_APPL"."acme.com.hana.xs.appl.db::SYSTEM_STATE":
  SELECT, INSERT, UPDATE, DELETE;
}
```

The application privileges referenced in the role definition (for example, Display and View) are actually defined in an application-specific .xsprivileges file, as illustrated in the following example, which also contains entries for additional privileges that are not explained here.

Note

The .xsprivileges file must reside in the package of the application to which the privileges apply.

The package where the .xsprivileges resides defines the scope of the application privileges; the privileges specified in the .xsprivileges file can only be used in the package where the .xsprivileges resides (or any sub-packages). This is checked during activation of the .xsaccess file and at runtime in the by the XS JavaScript API \$.session.(has|assert)AppPrivilege().

```
{
  "privileges" : [
    { "name" : "View", "description" : "View Product Details" },
```

```

{ "name" : "Configure", "description" : "Configure Product Details" },
{ "name" : "Display", "description" : "View Transport Details" },
{ "name" : "Administrator", "description" : "Configure/Run Everything" },
{ "name" : "ExecuteTransport", "description" : "Run Transports"},
{ "name" : "Transport", "description" : "Transports"}
]
}

```

The privileges are **authorized** for use with an application by inserting the [authorization](#) keyword into the corresponding .xsaccess file, as illustrated in the following example. Like the .xsprivileges file, the .xsaccess file must reside either in the root package of the application to which the privilege authorizations apply or the specific subpackage which requires the specified authorizations.

Note

If a privilege is inserted into the .xsaccess file as an authorization requirement, a user must have this privilege to access the application package where the .xsaccess file resides. If there is more than one privilege, the user must have at least one of these privileges to access the content of the package.

```

{
  "prevent_xsrf": true,
  "exposed": true,
  "authentication": {
    "method": "Form"
  },
  "authorization": [
    "acme.com.hana.xs.appl:Display",
    "acme.com.hana.xs.appl:Transport"
  ]
}

```

Related Information

[Custom Role for Developers \[page 707\]](#)

[Creating the Application Descriptors \[page 81\]](#)

11.2 Creating Analytic Privileges

You can create analytic privileges based on either an XML document (the "classic" variation) or an SQL definition.

Related Information

[Analytic Privileges \[page 728\]](#)

[Create Classical XML-Based Analytic Privileges](#)

[Create Static SQL Analytic Privileges](#)

[Create Classical XML-based Analytic Privileges \[page 734\]](#)

[Create SQL Analytic Privileges \[page 737\]](#)

11.2.1 Create Classical XML-based Analytic Privileges

Create analytic privileges for information views and assign them to different users to provide selective access that are based on certain combinations of data.

Prerequisites

If you want to use a classical XML-based analytic privilege to apply data access restrictions on information views, set the [Apply Privileges](#) property for the information view to [Classical Analytic Privileges](#).

1. Open the information view in the view editor.
2. Select the [Semantics](#) node.
3. Choose the [View Properties](#) tab.
4. In the [Apply Privileges](#) dropdown list, select [Classical Analytic Privileges](#).

Context

Analytic privileges help restrict data access to information views based on attributes or procedures. You can create and apply analytic privileges for a selected group of models or apply them to all models across packages.

After you create analytic privileges, assign it to users. This restricts users to access data only for certain combinations of dimension attributes.

Procedure

1. Launch SAP HANA studio.
2. In the [SAP HANA Systems](#) view, expand the content node.
3. In the navigator pane, select a package where you want to create the new analytic privilege.
4. In the context menu of the package, select  [New](#)  [Analytic Privilege](#) .
5. Provide a name and description.
6. In the [Type](#) dropdown list, select [Classical Analytic Privilege](#).
7. Choose [Finish](#).
8. Define validity for the analytic privilege.
In the [Privilege Validity](#) section, specify the time period for which the analytic privilege is valid.

- a. Choose [Add](#).
- b. Select a required operator.
- c. Provide the validity period based on the selected operator.

9. Define scope of the analytic privilege.

In the [Secured Models](#) section, select the models for which the analytic privileges restrictions are applicable.

- a. Choose [Add](#).
- b. If you want to create an analytic privilege and apply the data access restrictions for selected list of models, in the [Select Information Models](#) dialog, select the models for which you want to apply the analytic privilege restrictions.
- c. Choose [OK](#).
- d. If you want to create an analytic privilege and apply the data access restrictions for all models, then in the [General](#) section, select the [Apply to all information models](#) checkbox.

10. Select attributes.

Use attributes from the secured models to define data access restrictions.

- a. In the [Associated Attributes Restrictions](#) section, choose [Add](#).
- b. In the [Select Objects](#) dialog, select the attributes.

Note

Select a model if you want to use all attributes from the model to define restrictions.

- c. Choose [OK](#).

11. Define attribute restrictions

Modeler uses the restrictions defined on the attributes to restrict data access. Each attribute restriction is associated with only one attribute, but can contain multiple value filters. You can create more than one attribute restriction.

- a. In the [Assign Restrictions](#) section, choose [Add](#).
- b. In the [Type](#) dropdown list, select a restriction type.
- c. Select the required operator and provide a value using the value help.
- d. For catalog procedure or repository procedure, you can also provide values using the syntax <schema name>::<procedure name> or <package name>::<procedure name> respectively.

12. Define Attribute Restrictions Using Hierarchy Node Column

If you have enabled SQL access for calculation views, modeler generates a node column. You can use the node column to filter and perform SQL group by operation. For analytic privileges, you can maintain a filter expression using this node column.

- a. Select the [Hierarchy](#) tab.
- b. In the [Hierarchy](#) dropdown list, select a parent-child hierarchy.
- c. In the [Value](#) field, select a node value.

For example, if the node column is SalesRepHierarchyNode for a parent-child hierarchy, then you can create a hierarchical analytic privilege for a calculation view that filters the subtree of the node at runtime. "SalesRepHierarchyNode" = MAJESTIX

Note

You can create hierarchical analytic privileges only if all secured models are shared dimensions used in star join calculation views and if the view property of the calculation views is enabled for SQL access.

13. Activate analytic privileges.

- a. If you want to activate the analytic privilege, then in the toolbar choose [Save and Activate](#).
- b. If you want to activate the analytic privilege along with all objects, then in the toolbar choose [Save and Activate All](#).

Note

Activate the analytic privilege only if you have defined at least one restriction on attributes in the [Associated Attributes Restrictions](#) section.

14. Assign privileges to a user.

If you want to assign privileges to an authorization role, then in your SAP HANA studio, execute the following steps:

- a. In the [SAP HANA Systems](#) view, go to ► [Security](#) ► [Authorizations](#) ► [Users](#) ►.
- b. Select a user.
- c. In the context menu, choose [Open](#).
- d. In the [Analytic Privileges](#) tab page, add the privilege.
- e. In the editor toolbar, choose [Deploy](#).

Related Information

[Analytic Privileges](#)

[Structure of Analytic Privileges \[page 740\]](#)

[Example: Using Analytic Privileges](#)

[Example: Create an XML-Based Analytic Privilege with Dynamic Value Filter \[page 747\]](#)

[Supported Restriction Types in Analytic Privileges](#)

11.2.2 Create SQL Analytic Privileges

SQL based analytic privileges provides you the flexibility to create analytic privileges within the familiar SQL environment. You can create and apply SQL analytic privileges for a selected group of models or apply them to all models across packages.

Prerequisites

If you want to use a SQL analytic privilege to apply data access restrictions on information views, set the [Apply Privileges](#) property for the information view to [SQL Analytic Privileges](#).

1. Open the information view in the view editor.
2. Select the [Semantics](#) node.
3. Choose the [View Properties](#) tab.
4. In the [Apply Privileges](#) dropdown list, select [SQL Analytic Privileges](#).

Context

SAP HANA modeler support types SQL analytic privileges, the static SQL analytic privileges with predefined static filter conditions, and dynamic SQL analytic privileges with filter conditions determined dynamically at runtime using a database procedure.

Procedure

1. Launch SAP HANA studio.
2. In the [SAP HANA Systems](#) view, expand the [Content](#) node.
3. In the navigator pane, select a package where you want to create the new analytic privilege.
4. In the context menu of the package, select ► [New](#) ► [Analytic Privilege](#) ►.
5. Provide a name and description.
6. In the [Type](#) dropdown list, select [SQL Analytic Privilege](#).
7. Choose [Finish](#).
8. In the header region, select [SQL Editor](#).

Note

You can also use the attribute editor to create the analytic privilege using the attribute restrictions and then switch to the SQL editor to deploy the same privilege as SQL analytic privilege.

9. Select information models.

If you want to create an analytic privilege and apply the data access restrictions for selected list of models, in the [Secured Models](#) section,

- a. Choose [Add](#).
- b. In the [Select Information Models](#) dialog, select the models for which you want to apply the analytic privilege restrictions.
- c. Choose [OK](#).

10. Defining static SQL analytic privileges.

If you want to define static SQL analytic privileges, then

- a. In the SQL editor, provide the attribute restrictions and its validity.

For example,

```
((("REGION" = 'EAST') OR ("REGION" = 'NORTH')) AND (("CUSTOMER_ID" =
'SAP')) AND ((CURRENT_DATE BETWEEN 2015-05-15 00:00:00.000 AND 2015-05-15
23:59:59.999)))
```

Note

If you have enabled SQL access for calculation views (of type dimensions used in a star join calculation view), modeler generates a node column. For analytic privileges, you can maintain a filter expression using this node column.. For example, if SalesRepHierarchyNode is the node column that modeler generates for a parent-child hierarchy, then "SalesRepHierarchyNode" = "MAJESTIX" is a possible filter expression.

11. Defining dynamic SQL analytic privileges.

Dynamic SQL analytic privileges determine the filter condition string at runtime. If you want to define dynamic SQL analytic privileges,

- a. In the SQL editor, specify the procedure within the CONDITION PROVIDER clause.

For example, CONDITION PROVIDER schema_name.procedure_name.

12. Activate analytic privileges.

- a. If you want to activate the analytic privilege, then in the toolbar choose [Save and Activate](#).
- b. If you want to activate the analytic privilege along with all objects, then in the toolbar choose [Save and Activate All](#).

Note

Activate the analytic privilege only if you have defined at least one restriction on attributes in the [Associated Attributes Restrictions](#) section.

13. Assign privileges to a user.

If you want to assign privileges to an authorization role, then in your SAP HANA studio, execute the following steps:

- a. In the [SAP HANA Systems](#) view, go to ► [Security](#) ► [Authorizations](#) ► [Users](#) ►.
- b. Select a user.
- c. In the context menu, choose [Open](#).
- d. In the [Analytic Privileges](#) tab page, add the privilege.
- e. In the editor toolbar, choose [Deploy](#).

11.2.3 Analytic Privileges

Analytic privileges grant different users access to different portions of data in the same view based on their business role. Within the definition of an analytic privilege, the conditions that control which data users see is either contained in an XML document or defined using SQL.

Standard object privileges (SELECT, ALTER, DROP, and so on) implement coarse-grained authorization at object level only. Users either have access to an object, such as a table, view or procedure, or they don't. While this is often sufficient, there are cases when access to data in an object depends on certain values or combinations of values. Analytic privileges are used in the SAP HANA database to provide such fine-grained control at row level of which data individual users can see within the same view.

❖ Example

Sales data for all regions are contained within one analytic view. However, regional sales managers should only see the data for their region. In this case, an analytic privilege could be modeled so that they can all query the view, but only the data that each user is authorized to see is returned.

XML- Versus SQL-Based Analytic Privileges

Before you implement row-level authorization using analytic privileges, you need to decide which type of analytic privilege is suitable for your scenario. In general, SQL-based analytic privileges allow you to more easily formulate complex filter conditions that might be cumbersome to model using XML-based analytic privileges.

The following are the main differences between XML-based and SQL-based analytic privileges:

Feature	SQL-Based Analytic Privileges	XML-Based Analytic Privileges
Control of read-only access to SAP HANA information models: • Attribute views • Analytic views • Calculation views	Yes	Yes
Control of read-only access to SQL views	Yes	No
Control of read-only access to database tables	No	No
Design-time modeling in the <i>Editor</i> tool of the SAP HANA Web Workbench	Yes	Yes
Design-time modeling in the <i>SAP HANA Modeler</i> perspective of the SAP HANA studio	Yes	Yes
Transportable	Yes	Yes
Complex filtering	Yes	No

Enabling an Authorization Check Based on Analytic Privileges

All column views modeled and activated in the SAP HANA modeler and the SAP HANA Web-based Development Workbench automatically enforce an authorization check based on analytic privileges. XML-based analytic privileges are selected by default, but you can switch to SQL-based analytic privileges.

Column views created using SQL must be explicitly registered for such a check by passing the relevant parameter:

- `REGISTerviewFORAPCHECK` for a check based on XML-based analytic privileges
- `STRUCTURED PRIVILEGE CHECK` for a check based on SQL-based analytic privileges

SQL views must always be explicitly registered for an authorization check based analytic privileges by passing the `STRUCTURED PRIVILEGE CHECK` parameter.

Note

It is not possible to enforce an authorization check on the same view using both XML-based and SQL-based analytic privileges. However, it is possible to build views with different authorization checks on each other.

11.2.3.1 Structure of XML-Based Analytic Privileges

An analytic privilege consists of a set of restrictions against which user access to a particular attribute view, analytic view, or calculation view is verified. In an XML-based analytic privilege, these restrictions are specified in an XML document that conforms to a defined XML schema definition (XSD).

Note

As objects created in the repository, XML-based analytic privileges are deprecated as of SAP HANA SPS 02. For more information, see SAP Note 2465027.

Each restriction in an XML-based analytic privilege controls the authorization check on the restricted view using a set of value filters. A value filter defines a check condition that verifies whether or not the values of the view (or view columns) qualify for user access.

The following restriction types can be used to restrict data access:

- View
- Activity
- Validity
- Attribute

The following operators can be used to define value filters in the restrictions.

Note

The activity and validity restrictions support only a subset of these operators.

- `IN <list of scalar values>`
- `CP <pattern with *>`

- EQ (=), LE (<=), LT (<), GT (>), GE (>=) <scalar value>
- BT <scalar value as lower limit><scalar value as upper limit>
- IS_NULL
- NOT_NULL

All of the above operators, except IS_NULL and NOT_NULL, accept empty strings (" ") as filter operands. IS_NULL and NOT_NULL do not allow any input value.

The following are examples of how empty strings can be used with the filter operators:

- For the IN operator: IN ("", "A", "B") to filter on these exact values
- As a lower limit in comparison operators, such as:
 - BT ("", "XYZ"), which is equivalent to NOT_NULL AND LE "XYZ""GT "", which is equivalent to NOT_NULL
 - LE "", which is equivalent to EQ ""
 - LT "", which will always return false
 - CP "", which is equivalent to EQ ""

The filter conditions CP "*" will also return rows with empty-string as values in the corresponding attribute.

View Restriction

This restriction specifies to which column views the analytic privilege applies. It can be a single view, a list of views, or all views. An analytic privilege must have exactly one cube restriction.

❖ Example

```
IN ("Cube1", "Cube2")
```

📘 Note

When an analytic view is created in the SAP HANA modeler, automatically generated views are included automatically in the cube restriction.

📘 Note

The SAP HANA modeler uses a special syntax to specify the cube names in the view restriction:
 _SYS_BIC:<package_hierarchy>/<view_name>

For example:

```
<cubes>
  <cube name="_SYS_BIC:test.sales/AN_SALES" />
  <cube name="_SYS_BIC:test.sales/AN_SALES/olap" />
</cubes>
```

Activity Restriction

This restriction specifies the activities that the user is allowed to perform on the restricted views, for example, read data. An analytic privilege must have exactly one activity restriction.

❖ Example

EQ "read", or EQ "edit"

📘 Note

Currently, all analytic privileges created in the SAP HANA modeler are automatically configured to restrict access to READ activity only. This corresponds to SQL SELECT queries. This is due to the fact that the attribute, analytic, and calculation views are read-only views. This restriction is therefore not configurable.

Validity Restriction

This restriction specifies the validity period of the analytic privilege. An analytic privilege must have exactly one validity restriction.

❖ Example

GT 2010/10/01 01:01:00.000

Attribute Restriction

This restriction specifies the value range that the user is permitted to access. Attribute restrictions are applied to the actual attributes of a view. Each attribute restriction is relevant for one attribute, which can contain multiple value filters. Each value filter represents a logical filter condition.

📘 Note

The SAP HANA modeler uses different ways to specify attribute names in the attribute restriction depending on the type of view providing the attribute. In particular, attributes from attribute views are specified using the syntax "`<package_hierarchy>/<view_name>$<attribute_name>`", while local attributes of analytic views and calculation views are specified using their attribute name only. For example:

```
<dimensionAttribute name="test.sales/AT_PRODUCT$PRODUCT_NAME">
  <restrictions>
    <valueFilter operator="IN">
      <value value="Car" />
      <value value="Bike" />
    </valueFilter>
  </restrictions>
</dimensionAttribute>
```

Value filters for attribute restrictions can be static or dynamic.

- A **static** value filter consists of an operator and either a list of values as the filter operands or a single value as the filter operand. All data types are supported except those for LOB data types (CLOB, BLOB, and NCLOB).

For example, a value filter (EQ 2006) can be defined for an attribute YEAR in a dimension restriction to filter accessible data using the condition YEAR=2006 for potential users.

Note

Only attributes, not aggregatable facts (for example, measures or key figures) can be used in dimension restrictions for analytic views.

- A **dynamic** value filter consists of an operator and a stored procedure call that determines the operand value at runtime.

For example, a value filter (IN (GET_MATERIAL_NUMBER_FOR_CURRENT_USER())) is defined for the attribute MATERIAL_NUMBER. This filter indicates that a user with this analytic privilege is only allowed to access material data with the numbers returned by the procedure GET_MATERIAL_NUMBER_FOR_CURRENT_USER.

It is possible to combine static and dynamic value filters as shown in the following example.

Example

```
<dimensionAttribute name=" test.sales/AT_PRODUCT$PRODUCT_NAME ">
  <restrictions>
    <valueFilter operator="CP"> <value value="ELECTRO*" /> </
valueFilter>
    <valueFilter operator="IN"> <procedureCall
schema="PROCEDURE_OWNER" procedure="DETERMINE_AUTHORIZED_PRODUCT_FOR_USER" />
</valueFilter >
  </restrictions>
</dimensionAttribute>
<dimensionAttribute name=" test.sales/AT_TIME$YEAR ">
  <restrictions>
    <valueFilter operator="EQ"> <value value="2012" /> </valueFilter>
    <valueFilter operator="IN"> <procedureCall
schema="PROCEDURE_OWNER" procedure="DETERMINE_AUTHORIZED_YEAR_FOR_USER" /> </
valueFilter >
  </restrictions>
```

An analytic privilege can have multiple attribute restrictions, but it must have at least one attribute restriction. An attribute restriction must have at least one value filter. Therefore, if you want to permit access to the whole content of a restricted view, then the attribute restriction must specify all attributes.

Similarly, if you want to permit access to the whole content of the view with the corresponding attribute, then the value filter must specify all values.

The SAP HANA modeler automatically implements these two cases if you do not select either an attribute restriction or a value filter.

Example

Specification of all attributes:

```
<dimensionAttributes>
  <allDimensionAttributes/ >
</dimensionAttributes>
```


❖ Example

Specification of all values of an attribute:

```
<dimensionAttributes>
  <dimensionAttribute name="PRODUCT">
    <all />
  </dimensionAttribute>
</dimensionAttributes>
```

Logical Combination of Restrictions and Filter Conditions

The result of user queries on restricted views is filtered according to the conditions specified by the analytic privileges granted to the user as follows:

- Multiple analytic privileges are combined with the logical operator OR.
- Within one analytic privilege, all attribute restrictions are combined with the logical operator AND.
- Within one attribute restriction, all value filters on the attribute are combined with the logical operator OR.

Example

You create two analytic privileges AP1 and AP2. AP1 has the following attribute restrictions:

- Restriction R11 restricting the attribute Year with the value filters (EQ 2006) and (BT 2008, 2010)
- Restriction R12 restricting the attribute Country with the value filter (IN ("USA", "Germany"))

Given that multiple value filters are combined with the logical operator OR and multiple attribute restrictions are combined with the logical operator AND, AP1 generates the condition:

```
((Year = 2006) OR (Year BT 2008 and 2010)) AND (Country IN ("USA", "Germany"))
```

AP2 has the following restriction:

Restriction R21 restricting the attribute Country with the value filter (EQ "France")

AP2 generates the condition:

```
(Country = "France")
```

Any query of a user who has been granted both AP1 and AP2 will therefore be appended with the following WHERE clause:

```
((Year = 2006) OR (Year BT 2008 and 2010)) AND (Country IN ("USA", "Germany")) OR  
(Country = "France")
```

Related Information

[SAP Note 2465027](#) 

11.2.3.1.1 Dynamic Value Filters in the Attribute Restriction of XML-Based Analytic Privileges

The attribute restriction of an XML-based analytic privilege specifies the value range that the user is permitted to access using value filters. In addition to static scalar values, stored procedures can be used to define filters.

By using storing procedures to define filters, you can have user-specific filter conditions be determined dynamically in runtime, for example, by querying specified tables or views. As a result, the same analytic privilege can be applied to many users, while the filter values for authorization can be updated and changed independently in the relevant database tables. In addition, application developers have full control not only to design and manage such filter conditions, but also to design the logic for obtaining the relevant filter values for the individual user at runtime.

Procedures used to define filter conditions must have the following properties:

- They must have the security mode DEFINER.
- They must be read-only procedures.
- A procedure with a predefined signature must be used. The following conditions apply:
 - No input parameter
 - Only 1 output parameter as table type with one single column for the IN operator
 - Only 1 output parameter of a scalar type for all unary operators, such as EQUAL
 - Only 2 output parameters of a scalar type for the binary operator BETWEEN
- Only the following data types are supported as the scalar types and the data type of the column in the table type:
 - Date/Time types DATE, TIME, SECONDDATE, and TIMESTAMP
 - Numeric types TINYINT, SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, and DOUBLE
 - Character string types VARCHAR and NVARCHAR
 - Binary type VARBINARY

NULL as Operand for Filter Operators

In static value filters, it is not possible to specify NULL as the operand of the operator. The operators IS_NULL or NOT_NULL must be used instead. In dynamic value filters where a procedure is used to determine a filter condition, NULL or valid values may be returned. The following behavior applies in the evaluation of such cases during the authorization check of a user query:

Filter conditions of operators with NULL as the operand are disregarded, in particular the following:

- EQ NULL, GT NULL, LT NULL, LE NULL, and CP NULL
- BT NULL and NULL

If no valid filter conditions remain (that is, they have all been disregarded because they contain the NULL operand), the user query is rejected with a “Not authorized” error.

❖ Example

Dynamic analytic privilege 1 generates the filter condition (Year >= NULL) and dynamic analytic privilege 2 generates the condition (Country EQ NULL). The query of a user assigned these analytic privileges (combined with the logical operator OR) will return a “Not authorized” error.

❖ Example

Dynamic analytic privilege 1 generates the filter condition (Year >= NULL) and dynamic analytic privilege 2 generates the condition (Country EQ NULL AND Currency = "USD"). The query of a user assigned these analytic privileges (combined with the logical operator OR) will be filtered with the filter Currency = 'USD'.

In addition, a user query is not authorized in the following cases even if further applicable analytic privileges have been granted to the user.

- The BT operator has as input operands a valid scalar value and NULL, for example, BT 2002 and NULL or BT NULL and 2002
- The IN operator has as input operand NULL among the value list, for example, IN (12, 13, NULL)

Permitting Access to All Values

If you want to allow the user to see all the values of a particular attribute, instead of filtering for certain values, the procedure must return "*" and "" (empty string) as the operand for the CP and GT operators respectively. These are the only operators that support the specification of all values.

Implementation Considerations

When the procedure is executed as part of the authorization check in runtime, note the following:

- The user who must be authorized is the database user who executes the query accessing a secured view. This is the session user. The database table or view used in the procedure must therefore contain a column to store the user name of the session user. The procedure can then filter by this column using the SQL function SESSION_USER. This table or view should only be accessible to the procedure owner.

⚠ Caution

Do not map the executing user to the application user. The application user is unreliable because it is controlled by the client application. For example, it may set the application user to a technical user or it may not set it at all. In addition, the trustworthiness of the client application cannot be guaranteed.

- The user executing the procedure is the _SYS_REPO user. In the case of procedures activated in the SAP HANA modeler, _SYS_REPO is the owner of the procedures. For procedures created in SQL, the EXECUTE privilege on the procedure must be granted to the _SYS_REPO user.
- If the procedure fails to execute, the user's query stops processing and a "Not authorized" error is returned. The root cause can be investigated in the error trace file of the indexserver, `indexserver_alert_<host>.trc`.

When designing and implementing procedures as filter for dynamic analytic privileges, bear the following in mind:

- To avoid a recursive analytic privilege check, the procedures should only select from database tables or views that are not subject to an authorization check based on analytic privileges. In particular, views activated in the SAP HANA modeler are to be avoided completely as they are automatically registered for the analytic privilege check.

- The execution of procedures in analytic privileges slows down query processing compared to analytic privileges containing only static filters. Therefore, procedures used in analytic privileges must be designed carefully.

11.2.3.1.2 Example: Create an XML-Based Analytic Privilege with Dynamic Value Filter

Use the CREATE STRUCTURED PRIVILEGE statement to create an XML-based analytic privilege that contains a dynamic procedure-based value filter and a fixed value filter in the attribute restriction.

Context

Note

The analytic privilege in this example is created using the CREATE STRUCTURED PRIVILEGE statement. Under normal circumstances, you create XML-based analytic privileges using the SAP HANA modeler or the SAP HANA Web-based Development Workbench. Analytic privileges created using CREATE STRUCTURED PRIVILEGE are not owned by the user _SYS_REPO. They can be granted and revoked only by the actual database user who creates them.

Assume you want to restrict access to product data in secured views as follows:

- Users should only see products beginning with ELECTRO, or
- Users should only see products for which they are specifically authorized. This information is contained in the database table PRODUCT_AUTHORIZATION_TABLE in the schema AUTHORIZATION.

To be able to implement the second filter condition, you need to create a procedure that will determine which products a user is authorized to see by querying the table PRODUCT_AUTHORIZATION_TABLE.

Procedure

1. Create the table type for the output parameter of the procedure:

```
CREATE TYPE "AUTHORIZATION"."PRODUCT_OUTPUT" AS TABLE("PRODUCT" int);
```

2. Create the table that the procedure will use to check authorization:

```
CREATE TABLE "AUTHORIZATION"."PRODUCT_AUTHORIZATION_TABLE" ("USER_NAME"
NVARCHAR(128), "PRODUCT" int);
```

3. Create the procedure that will determine which products the database user executing the query is authorized to see based on information contained in the product authorization table:

```
CREATE PROCEDURE "AUTHORIZATION"."DETERMINE_AUTHORIZED_PRODUCT_FOR_USER" (OUT
VAL "AUTHORIZATION"."PRODUCT_OUTPUT")
LANGUAGE SQLSCRIPT SQL SECURITY DEFINER READS SQL DATA AS
```

```
BEGIN
    VAL = SELECT PRODUCT FROM "AUTHORIZATION"."PRODUCT_AUTHORIZATION_TABLE"
WHERE USER_NAME = SESSION_USER;
END;
```

Note

The session user is the database user who is executing the query to access a secured view. This is therefore the user whose privileges must be checked. For this reason, the table or view used in the procedure should contain a column to store the user name so that the procedure can filter on this column using the SQL function SESSION_USER.

Caution

Do not map the executing user to the application user. The application user is unreliable because it is controlled by the client application. For example, it may set the application user to a technical user or it may not set it at all. In addition, the trustworthiness of the client application cannot be guaranteed.

4. Create the analytic privilege:

```
CREATE STRUCTURED PRIVILEGE '<?xml version="1.0" encoding="utf-8"?>
<analyticPrivilegeSchema version="1">
  <analyticPrivilege name="AP2">
    <cubes>
      <allCubes />
    </cubes>
    <validity>
      <anyTime/>
    </validity>
    <activities>
      <activity activity="read" />
    </activities>
    <dimensionAttributes>
      <dimensionAttribute name="PRODUCT">
        <restrictions>
          <valueFilter operator="CP"> <value value="ELECTRO*" /> </
valueFilter>
          <valueFilter operator="IN"> <procedureCall schema="AUTHORIZATION"
procedure="DETERMINE_AUTHORIZED_PRODUCT_FOR_USER"/> </valueFilter>
        </restrictions>
      </dimensionAttribute>
    </dimensionAttributes>
  </analyticPrivilege>
</analyticPrivilegeSchema>';
```

Results

Now when a database user requests access to a secured view containing product information, the data returned will be filtered according to the following condition:

```
(product LIKE "ELECTRO*" OR product IN
(AUTHORIZATION.DETERMINE_AUTHORIZED_PRODUCT_FOR_USER( ) )
```

11.2.3.2 Structure of SQL-Based Analytic Privileges

An analytic privilege consists of a set of restrictions against which user access to a particular attribute view, analytic view, calculation view, or SQL view is verified. In an SQL-based analytic privilege, these restrictions are specified as filter conditions that are fully SQL based.

SQL-based analytic privileges are created using the `CREATE STRUCTURED PRIVILEGE` statement:

```
CREATE STRUCTURED PRIVILEGE <privilege_name> FOR <action> ON <view_name>  
<filter_condition>;
```

The `FOR` clause is used to restrict the type of access (only the `SELECT` action is supported). The `ON` clause is used to restrict access to one or more views with the same filter attributes.

The `<filter condition>` parameter is used to restrict the data visible to individual users. The following methods of specifying filter conditions are possible:

- Fixed filter (`WHERE`) clause
- Dynamically generated filter (`CONDITION PROVIDER`) clause

Fixed Filter Clauses

A **fixed filter clause** consists of an `WHERE` clause that is specified in the definition of the analytic privilege itself.

You can express fixed filter conditions freely using SQL, including subqueries.

By incorporating SQL functions into the subqueries, in particular `SESSION_USER`, you can define an even more flexible filter condition.

❖ Example

```
country IN (SELECT a.country FROM authorizationtable a WHERE SESSION_USER=  
a.user_name)
```

📌 Note

A **calculation view** cannot be secured using an SQL-based analytic privilege that contains a complex filter condition if the view is defined on top of analytic and/or attributes views that themselves are secured with an SQL-based analytic privilege with a complex filter condition.

→ Remember

If you use a subquery, you (the creating user) must have the required privileges on the database objects (tables and views) involved in the subquery.

Comparative conditions can be nested and combined using `AND` and `OR` (with corresponding brackets).

→ Tip

To create an analytic privilege that allows either access to all data or no data in a view, set a fixed filter condition such as `1=1` or `1!=1`.

Dynamically Generated Filter Clauses

With a dynamically generated filter clause, the **WHERE** clause that specifies the filter condition is generated every time the analytic privilege is evaluated. This is useful in an environment in which the filter clause changes very dynamically. The filter condition is determined by a procedure specified in the **CONDITION PROVIDER** clause, for example:

Sample Code

```
CREATE STRUCTURED PRIVILEGE dynamic_ap FOR SELECT ON schema1.v1 CONDITION
PROVIDER schema2.procedure1;
```

Procedures in the **CONDITION PROVIDER** clause must have the following properties:

- They must have the security mode **DEFINER**.
- They must be read-only procedures.
- They must have a predefined signature. Here, the following conditions apply:
 - No input parameter
 - Only one output parameter for the filter condition string of string type **NVARCHAR**, **VARCHAR**, **CLOB**, or **NCLOB**
While **VARCHAR** and **NVARCHAR** have length limitations of 5000 characters, **CLOB** and **NCLOB** can be used to accommodate longer filter strings.
- The procedure may only return conditions expressed with the following operators:
 - **=**, **<=**, **<**, **>**, **>=**
 - **LIKE**
 - **BETWEEN**
 - **IN**
 - **NOT (...)**
 - **!=**

A complex filter condition, that is a subquery, may not be returned.

→ Tip

A procedure that returns the filter condition **1=1** or **1>1** can be used to create an analytic privilege that allows access to all data or no data in a view.

- The procedure must be executable by either the user **_SYS_REPO** (default) or the owner of the analytic privilege. This means that:
 - **_SYS_REPO** or the owner of the analytic privilege must be the owner of the procedure, or
 - The owner of the procedure has all privileges on the underlying tables/views with **WITH GRANT OPTION** and has granted the **EXECUTE** privilege on the procedure to **_SYS_REPO** or the owner of the analytic privilege

The parameter **[authorization] execute_dynamic_analytic_privilege_by_sys_repo_user** in the **indexserver.ini** configuration file determines which user must be able to execute the procedure.

⚠ Caution

Changing the value of this parameter while there are views already secured using analytic privileges with a dynamically generated filter may result in the procedure no longer being executable.

- The procedure must return a valid filter string. In particular, the filter string must not be empty and must represent a valid WHERE condition for the view.

If errors occur in procedure execution or an invalid filter string (empty or not applicable) is returned, the user receives a `Not authorized` error, even if he has the analytic privileges that would grant access.

Related Information

[CREATE STRUCTURED PRIVILEGE Statement \(Access Control\)](#)

11.2.3.2.1 Examples: Securing Views Using SQL-Based Analytic Privileges

Use the CREATE STRUCTURED PRIVILEGE statement to create SQL-based analytic privileges for different scenarios.

Context

The examples provided here take you through the following scenarios:

- [Example 1: Securing a column view using an SQL-based analytic privilege with a fixed filter clause \[page 751\]](#)
- [Example 2: Securing an SQL view using an SQL-based analytic privilege with a complex filter clause \(subquery\) \[page 753\]](#)
- [Example 3: Securing a column view using an SQL-based analytic privilege with a dynamically generated filter clause \[page 755\]](#)

Note

The analytic privileges in these examples are created using the CREATE STRUCTURED PRIVILEGE statement. Under normal circumstances, you create SQL-based analytic privileges using the SAP HANA Web IDE. They can be granted and revoked only by the actual database user who creates them.

Example 1: Secure a Column View Using an SQL-Based Analytic Privilege with a Fixed Filter Clause

Prerequisites

The database user TABLEOWNER has set up a calculation scenario based on the table SALES_TABLE, which contains the data to be protected.

Context

All sales data is contained in a single view. You want to restrict user access so that sales managers can see only information about the product "car" in the sales region UK and Germany. You want to do this by creating an analytic privilege with a fixed filter clause.

A fixed filter clause consists of an SQL WHERE clause that is specified in the definition of the analytic privilege itself.

→ Tip

In the following procedure, you might find it easier to use the graphical editors to create the calculation view and analytic privilege.

Procedure

1. Create the view containing the sales data:

```
CREATE COLUMN VIEW "TABLEOWNER"."VIEW_SALES" TYPE CALCULATION WITH PARAMETERS
( 'PARENTCALCINDEXSCHEMA'='TABLEOWNER',
  'PARENTCALCINDEX'='CALCSCEN_SALES',
  'PARENTCALCNODE'='SALES_TABLE',
  'REGISTerviewFORAPCHECK'='0') STRUCTURED PRIVILEGE CHECK
;
```

📌 Note

You can see above that the authorization check using XML-based analytic privileges is disabled with 'REGISTerviewFORAPCHECK'='0', while the authorization check using SQL-based analytic privileges is enabled with STRUCTURED PRIVILEGE CHECK. Both checks cannot be enabled at the same time.

2. Create the analytic privilege:

```
CREATE STRUCTURED PRIVILEGE AP_SALES_1 FOR SELECT
ON TABLEOWNER.VIEW_SALES
WHERE REGION IN ('DE', 'UK')
AND PRODUCT = 'CAR'
;
```

→ Remember

When specifying filters, remember the following:

- You can specify only the SELECT action in the FOR clause.
- You can specify one or more views with the same filter attributes in the ON clause
- You can specify comparative conditions between attributes and constant values using only the following operators:
 - =, <=, <, >, >=
 - LIKE
 - BETWEEN

- IN
- You can create complex filter conditions by including SQL statements as subqueries inside the WHERE clause. Example 2 illustrates how you do this. But remember: A **calculation view** cannot be secured using an SQL-based analytic privilege that contains a complex filter condition if the view is defined on top of analytic and/or attributes views that themselves are secured with an SQL-based analytic privilege with a complex filter condition.
Also remember that if you use a subquery, you must have the required privileges on the database objects (tables and views) involved in the subquery.

3. Grant the SELECT privilege on the view TABLEOWNER.VIEW_SALES to the relevant users/roles:

```
GRANT SELECT on TABLEOWNER.VIEW_SALES to <SALES_MANAGERS>;
```

→ Remember

Only the view owner or a user who has the SELECT privilege WITH GRANT OPTION on the view can perform the grant.

4. Grant the analytic privilege to the relevant users/roles:

```
GRANT STRUCTURED PRIVILEGE AP_SALES_1 TO <SALES_MANAGERS>;
```

→ Remember

Owner of the analytic privilege or any user who was granted the analytic privilege with GRANT OPTION can grant it.

Example 2: Secure an SQL View Using an SQL-Based Analytic Privilege with a Complex Filter Clause (Subquery)

Prerequisites

The database user TABLEOWNER has created a table TABLEOWNER.SALES, which contains the data to be protected.

Context

All sales data is contained in a single view. You want to restrict access of user MILLER so that he can see only product information from the year 2008. You want to do this by creating an analytic privilege with a complex filter clause.

With a complex filter clause, the SQL WHERE clause that specifies the filter condition includes an SQL statement, or a subquery. This allows you to create complex filter conditions to control which data individual users see.

→ Tip

In the following procedure, you might find it easier to use the graphical editors to create the calculation view and analytic privilege.

Procedure

1. Create the view containing the sales data which needs to be secured:

```
CREATE VIEW "VIEWOWNER"."ROW_VIEW_SALES_ON_SALES" AS SELECT
    * FROM "TABLEOWNER"."SALES" WITH STRUCTURED PRIVILEGE CHECK
;
```

→ Remember

The user creating the view must have the SELECT privilege WITH GRANT OPTION on the table TABLEOWNER.SALES.

2. Create the table containing user-specific authorization data:

```
CREATE COLUMN TABLE "VIEWOWNER"."AUTHORIZATION_VALUES" ("VALUE" VARCHAR(256),
    "USER_NAME" VARCHAR(20));
```

3. Insert authorization information for user MILLER:

```
INSERT INTO "VIEWOWNER"."AUTHORIZATION_VALUES" VALUES('2008', 'MILLER');
```

4. Create the analytic privilege using a subquery as the condition provider:

```
CREATE STRUCTURED PRIVILEGE AP_ROW_VIEW_SALES_ON_SALES FOR SELECT
    ON "VIEWOWNER"."ROW_VIEW_SALES_ON_SALES"
WHERE (CURRENT_DATE BETWEEN 2015-01-01 AND 2015-01-11) AND YEAR IN (SELECT
    VALUE FROM VIEWOWNER.AUTHORIZATION_VALUES WHERE USER_NAME = SESSION_USER)
;
```

→ Remember

- Subqueries allow you to create complex filter conditions, but remember: A **calculation view** cannot be secured using an SQL-based analytic privilege that contains a complex filter condition if the view is defined on top of analytic and/or attributes views that themselves are secured with an SQL-based analytic privilege with a complex filter condition.
- The user creating the analytic privilege must have the SELECT privilege on the objects involved in the subquery, in this case table VIEWOWNER.AUTHORIZATION_VALUES.
- The session user is the database user who is executing the query to access a secured view. This is therefore the user whose privileges must be checked. For this reason, the table containing the authorization information needs a column to store the user name so that the subquery can filter on this column using the SQL function SESSION_USER.

⚠ Caution

While application user can also be used, it is not recommended. The application user is unreliable because it is controlled by the client application. For example, it may set the application user to a

technical user or it may not set it at all. In addition, the trustworthiness of the client application cannot be guaranteed.

5. Grant the SELECT privilege on the view VIEWOWNER.ROW_VIEW_SALES_ON_SALES to user MILLER.

```
GRANT SELECT ON "VIEWOWNER"."ROW_VIEW_SALES_ON_SALES" TO MILLER;
```

→ Remember

Only the view owner or a user who has the SELECT privilege WITH GRANT OPTION on the view can perform the grant.

6. Grant the analytic privilege to user MILLER.

```
GRANT STRUCTURED PRIVILEGE AP_ROW_VIEW_SALES_ON_SALES TO MILLER;
```

→ Remember

Owner of the analytic privilege or any user who was granted the analytic privilege with GRANT OPTION can grant it.

Example 3: Secure a Column View Using an SQL-Based Analytic Privilege with a Dynamically Generated Filter Clause

Prerequisites

The database user TABLEOWNER has set up a calculation scenario based on the table SALES_TABLE, which contains the data to be protected.

Context

All sales data is contained in a single view. You want to restrict access of user ADAMS so that he can see only information about cars bought by customer Company A or bikes sold in 2006. You want to do this by creating an analytic privilege with a dynamically generated filter clause.

With a dynamically generated filter clause, the SQL WHERE clause that specifies the filter condition is generated every time the analytic privilege is evaluated. This is useful in an environment in which the filter clause changes very dynamically.

→ Tip

In the following procedure, you might find it easier to use the graphical editors to create the calculation view and analytic privilege.

Procedure

1. Create the view containing the sales data:

```
CREATE COLUMN VIEW "TABLEOWNER"."VIEW_SALES" TYPE CALCULATION WITH PARAMETERS
('PARENTCALCINDEXSCHEMA'='TABLEOWNER',
 'PARENTCALCINDEX'='CALCSCEN_SALES',
 'PARENTCALCNODE'='SALES_TABLE',
 'REGISTERVIEWFORAPCHECK'='0') STRUCTURED PRIVILEGE CHECK
;
```

2. Create a table containing user-specific filter strings:

```
CREATE COLUMN TABLE "AUTHORIZATION"."AUTHORIZATION_FILTERS" ("FILTER"
VARCHAR(256),
 "USER_NAME" VARCHAR(20))
;
```

3. Create an authorization filter for user ADAMS:

```
INSERT
INTO "AUTHORIZATION"."AUTHORIZATION_FILTERS" VALUES(' (CUSTOMER=' 'Company A' '
AND PRODUCT=' 'Car' ' ) OR ( YEAR=' '2006' ' AND PRODUCT=' 'Bike' ' ) ',
 'ADAMS' )
;
```

→ Remember

Filters containing comparative conditions must be defined as specified in example 1.

4. Create the database procedure that provides the filter clause for the analytic privilege and grant it to object owner of the project:

```
CREATE PROCEDURE "PROCOWNER"."GET_FILTER_FOR_USER"(OUT OUT_FILTER
VARCHAR(5000))
LANGUAGE SQLSCRIPT SQL SECURITY DEFINER READS SQL DATA AS
    v_Filter VARCHAR(5000);
    CURSOR v_Cursor FOR SELECT "FILTER" FROM
"PROCOWNER"."AUTHORIZATION_FILTERS" WHERE "USER_NAME" = SESSION_USER;
BEGIN
    OPEN v_Cursor;
    FETCH v_Cursor INTO v_Filter;
    OUT_FILTER := v_Filter;
    CLOSE v_Cursor;
END;
GRANT EXECUTE ON "PROCOWNER"."GET_FILTER_FOR_USER";
```

→ Remember

When using procedures as the condition provider in an SQL-based analytic privilege, remember the following:

- Procedures must have the following properties:
 - They must have the security mode DEFINER.
 - They must be read-only procedures.
 - A procedure with a predefined signature must be used. The following conditions apply:
 - No input parameter
 - Only one output parameter for the filter condition string of string type NVARCHAR, VARCHAR, CLOB, or NCLOB

While VARCHAR and NVARCHAR have length limitations of 5000 characters, CLOB and NCLOB can be used to accommodate longer filter strings.

- The procedure may **not** return a complex filter condition, that is a subquery.
- The procedure must be executable by object owner of the project, that is, either object owner of the project must be the owner of the procedure or the owner of the procedure has all privileges on the underlying tables/views with GRANT OPTION and has granted the EXECUTE privilege on the procedure to the object owner of the project.
- The session user is the database user who is executing the query to access a secured view. This is therefore the user whose privileges must be checked. For this reason, the table or view used in the procedure should contain a column to store the user name so that the procedure can filter on this column using the SQL function SESSION_USER.
- If the filter procedure returns an empty or inapplicable filter string in the output parameter, the user also receives a **Not authorized** error.

5. Create the analytic privilege using the procedure as condition provider:

```
CREATE STRUCTURED PRIVILEGE AP_SALES_2 FOR  
SELECT ON "TABLEOWNER"."VIEW_SALES" CONDITION PROVIDER  
"AUTHORIZATION"."GET_FILTER_FOR_USER";
```

On evaluation of the analytic privilege for user ADAMS, the WHERE clause (CUSTOMER='Company A' AND PRODUCT='Car') OR (YEAR='2006' AND PRODUCT='Bike'), as provided by the procedure GET_FILTER_FOR_USER, will be used.

6. Grant the SELECT privilege on the view TABLEOWNER.VIEW_SALES to user ADAMS:

```
GRANT SELECT on TABLEOWNER.VIEW_SALES to ADAMS;
```

→ Remember

Only the view owner or a user who has the SELECT privilege WITH GRANT OPTION on the view can perform the grant.

7. Grant the analytic privilege to user ADAMS:

```
GRANT STRUCTURED PRIVILEGE AP_SALES_2 TO ADAMS;
```

→ Remember

Owner of the analytic privilege or any user who was granted the analytic privilege with GRANT OPTION can grant it.

11.2.3.3 Runtime Authorization Check of Analytic Privileges

When a user requests access to data stored in an attribute, analytic, calculation, or SQL views, an authorization check based on analytic privileges is performed and the data returned to the user is filtered accordingly. The EFFECTIVE_STRUCTURED_PRIVILEGES system view can help you to troubleshoot authorization problems.

Access to a view and the way in which results are filtered depend on whether the view is independent or associated with other views (dependent views).

Independent Views

The authorization check for a view that is not defined on another column view is as follows:

1. The user's authorization to access the view is checked.

A user can access the view if **both** of the following prerequisites are met:

- The user has been granted the **SELECT** privilege on the view or the schema in which it is located.

Note

The user does not require **SELECT** on the underlying base tables or views of the view.

- The user has been granted an analytic privilege that is applicable to the view.
Applicable analytic privileges are those that meet **all** of the following criteria:

XML-Based Analytic Privilege	SQL-Based Analytic Privilege
A view restriction that includes the accessed view	An ON clause that includes the accessed view.
A validity restriction that applies now	If the filter condition specifies a validity period (for example, WHERE (CURRENT_TIME BETWEEN ... AND ...) AND <actual filter>), it must accommodate the time when the user executes the query.
An action in the activity restriction that covers the action requested by the query	An action in the FOR clause that covers the action requested by the query
Note All analytic privileges created and activated in the SAP HANA modeler and SAP HANA Web-based Development Workbench fulfill this condition. The only action supported is read access (SELECT).	Note All analytic privileges created and activated in the SAP HANA Web-based Development Workbench fulfill this condition. The only action supported is read access (SELECT).
An attribute restriction that includes some of the view's attributes	A filter condition that applies to the view
	Note When the analytic privilege is created, the filter is checked immediately to ensure that it applies to the view. If it doesn't, creation will fail. However, if the view definition subsequently changes, or if a dynamically generated filter condition returns a filter string that is not executable with the view, the authorization check will fail and access is rejected.

If the user has the **SELECT** privilege on the view but no applicable analytic privileges, the user's request is rejected with a **Not authorized** error. The same is true if the user has an applicable analytic privilege but doesn't have the **SELECT** privilege on the view.

2. The value filters specified in the dimension restrictions (XML-based) or filter condition (SQL-based) are evaluated and the appropriate data is returned to the user. Multiple analytic privileges are combined with the logical operator **OR**.

For more information about how multiple attribute restrictions and/or multiple value filters in XML-based analytic privileges are combined, see *XML-Based Analytic Privileges*.

Dependent Views

The authorization check for a view that is defined on other column views is more complex. Note the following behavior.

Calculation and SQL views

- Individual views in the hierarchy are filtered according to their respective analytic privileges, which use the logical OR combination.
- The filtered result of the calculation view is derived from the filtered result of its underlying views. This corresponds to a logic AND combination of the filters generated by the analytic privileges for the individual views.

Result filtering on the view is then performed as follows:

- The user has been granted the **SELECT** privilege on the view or the schema that contains the view.
- The user has been granted analytic privileges that apply to the view itself **and** all the other column views in the hierarchy that are registered for a structured privilege check.

Calculation views and SQL views can be defined by selecting data from other views. This can lead to a complex view hierarchy that requires careful design of row-level authorization.

Analytic views

An analytic view can also be defined on attribute views, but this does **not** represent a view dependency or hierarchy with respect to authorization check and result filtering. If you reference an attribute view in an analytic view, analytic privileges defined on the attribute view are not applied.

This represents a view hierarchy for which the prerequisites described above for calculation views also apply.

- Currency or unit conversions
- Calculated attributes
- Calculated measures that use attributes, calculated attributes, or input parameters in their formulas

If an analytic view designed in the SAP HANA modeler contains one of the elements listed below, it will automatically be activated with a calculation view on top. The name of this calculation view is the name of the analytic view with the suffix `/olap`.

Troubleshooting Failed Authorization

Using the `EFFECTIVE_STRUCTURED_PRIVILEGES` system view, you can quickly see:

- Which analytic privileges apply to a particular view, including the dynamic filter conditions that apply (if relevant)
- Which filter is being applied to which view in the view hierarchy (for views with dependencies)
- Whether or not a particular user is authorized to access the view

Query `EFFECTIVE_STRUCTURED_PRIVILEGES` as follows:

```
SELECT * from "PUBLIC"."EFFECTIVE_STRUCTURED_PRIVILEGES" where ROOT_SCHEMA_NAME = '<schema>' AND ROOT_OBJECT_NAME = '<OBJECT>' AND USER_NAME = '<USER>';
```


Related Information

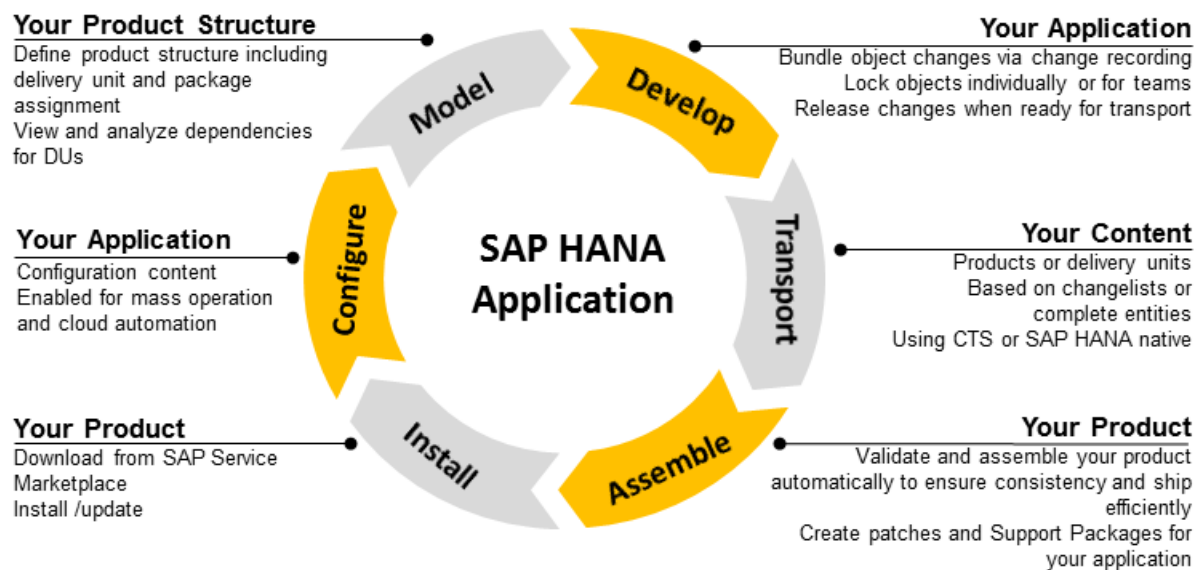
[Structure of XML-Based Analytic Privileges \[page 740\]](#)

[EFFECTIVE_STRUCTURED_PRIVILEGES System View](#)

12 SAP HANA Application Lifecycle Management in XS Classic

SAP HANA application lifecycle management supports you in all phases of an SAP HANA application lifecycle, from modeling your product structure, through application development, transport, assembly, and installation.

The following graphic illustrates the phases in a product lifecycle of an SAP HANA application:



Phases of SAP HANA Application Lifecycle Management

- **Model**
You define your product structure to provide a framework for efficient software development. This includes creating delivery units and assigning packages to delivery units. The delivery units are then bundled in products.
- **Develop**
You perform software development in repository packages. SAP HANA application lifecycle management supports you with change tracking functions.
- **Transport**
You can transport your developed content in different ways according to your needs. You can choose between transporting products or delivery units, based on change lists or complete entities. The transport type can be native SAP HANA transport or transport using Change and Transport System (CTS). You can also export delivery units, and import them into another system.
- **Assemble**

The developed software plus the metadata defined when modeling your product structure as well as possible translation delivery units are the basis for assembling your add-on product. You can also build Support Packages and patches for your product.

- **Install**

You can install SAP HANA products that you downloaded from SAP Support Portal or that you assembled yourself.

- **Configure**

If the SAP HANA product delivers configuration content, you can use the process engine of SAP HANA application lifecycle management to automate configuration tasks.

All phases of SAP HANA application lifecycle management are documented in the *SAP HANA Application Lifecycle Management Guide*. The tasks related to the **Install** and **Configure** phases of SAP HANA application lifecycle management are relevant for system administrators and are therefore also documented in the *SAP HANA Administration Guide*. The tasks related to software development are documented in the *SAP HANA Developer Guide (for SAP HANA Studio)*.

Availability of SAP HANA Application Lifecycle Management

SAP HANA application lifecycle management for XS classic is installed with SAP HANA as automated content. You can access the SAP HANA application lifecycle management functions in different ways:

- Using the SAP HANA Application Lifecycle Management user interface, which is available in the following locations:
 - On the SAP HANA XS Web server at the following URL: `http://<WebServerHost>:80<SAPHANAinstance>/sap/hana/xs/lm`.
 - Using a link in SAP HANA Web-based Development Workbench.
For example, to open the home screen, choose ► [Navigation Links](#) ► [Lifecycle Management](#) ► in the SAP HANA Web-based Development Workbench Editor tool.
 - Using the context menu in SAP HANA studio.
For example, to open the home screen from, choose ► [Lifecycle Management](#) ► [Application Lifecycle Management](#) ► [Home Screen](#) ► from the context menu for a particular system in the *SAP HANA Administration Console* perspective in SAP HANA studio.
- Using the command line tool `hdba lm`.
The file is shipped with the SAP client installation. If you leave the default installation options unchanged, `hdba lm` is located in the `.. \sap\hdbclient` directory.

You cannot perform all application lifecycle management tasks with one tool. For example, assembling products and software components can only be done using the `hdba lm` tool, whereas the full set of transport functions is available only in the XS user interface. Whenever a function is available in the XS user interface it is documented there. When used in SAP HANA studio, the functions are the same as in the XS user interface. Therefore, these options are not separately documented.

❗ Note

For information about the SAP HANA platform lifecycle management tools, see the *SAP HANA Server Installation and Update Guide* and *SAP HANA Platform Lifecycle Management* in the *SAP HANA Administration Guide*.

Related Information

[SAP HANA Application Lifecycle Management Guide](#)
[SAP HANA Administration Guide](#)

13 SAP HANA Database Client Interfaces

SAP HANA provides a selection of client interfaces for connecting applications to retrieve and update data.

SAP HANA exposes data with client and web-based interfaces.

SAP HANA supports many common database application programming interfaces (APIs). For example, a spreadsheet application can use ODBO to consume analytic views and enable users to create pivot tables, or a Web application can use an OData interface to access and display data.

- Client interfaces are available as long as the SAP HANA client is installed. The following APIs are supported:

API	Description
JDBC	The JDBC driver
ODBC	The ODBC driver
SQLDBC	The SQLDBC API (for internal use only)
ODBO/MDX	The ODBO API driver that executes MDX statements (most commonly used with Microsoft Excel)
📌 Note If you are using the SAP HANA Client version ≥ 2.16, and you want to use the ODBO interface, you need to use the ODBO driver provided in the SAP HANA Client for EXCEL 2.0, which is available for download from the SAP Support Portal. For more information, see <i>Related Information</i> below.	
Python DB API	The Python DB API (hdbcli.dbapi)
ADO.NET	The data provider for Microsoft ADO.NET
Node.js	The Node.js driver for JavaScript on Joyent's Node.js software platform
Go	The Go driver

- Web-based interfaces must be defined by the application developer, who determines what data to expose and to whom. The following web-based interfaces are supported:
 - OData
 - Server-Side JavaScript

Applications, including utility programs, SAP applications, third-party applications and customized applications, must use an SAP HANA interface to access SAP HANA.

Note

For more information about using the client interfaces to connect to the SAP HANA database, details of which version of the individual interface is supported, as well as a collection of tips and programming tricks, see the *SAP HANA Client Interface Programming Reference* [Related Information](#) below.

Related Information

[SAP HANA Client Interface Programming Reference \(Node.js Application Programming\)](#)

[SAP Support Portal \(SAP HANA Client for EXCEL 2.0\)](#) 

[SAP Note 3537501](#) 

14 Migrating XS Classic Applications to XS Advanced

Migrate XS classic applications to XS advanced with on-premise or Cloud-based migration tools.

SAP HANA XS advanced model includes the *XS Advanced Migration Assistant*, which automates large parts of the process of migrating XS classic application artifacts to XS advanced, for example, the conversion of many design-time artifacts from the XS classic file format and syntax to the syntax, format, and file extension required to build and deploy the migrated application in the XS advanced run-time environment. The *XS Advanced Migration Assistant* is available for download from the SAP Software Download Center, as described in *The XS Advanced Migration Assistant in Related Information* below.

Related Information

[The XS Advanced Migration Assistant \[page 770\]](#)

[Migrating XS Classic Applications to XS Advanced Model with the XS Advanced Migration Assistant \[page 766\]](#)

[Migrate an XS Classic Application to SAP CAP and SAP HANA Cloud \(SAP HANA Cloud Migration Guide\)](#)

14.1 Migrating XS Classic Applications to XS Advanced Model with the XS Advanced Migration Assistant

SAP HANA provides tools to help you migrate an XS classic application to the XS advanced run-time environment.

From HANA 1.0 SPS 11, SAP HANA includes an additional run-time environment for application development: SAP HANA extended application services (XS), **advanced model**. SAP HANA XS advanced model represents an evolution of the application server architecture within SAP HANA by building upon the strengths (and expanding the scope) of SAP HANA extended application services (XS), classic model. If you are developing new applications, it is recommended to use SAP HANA XS advanced model.

→ Tip

If you want to migrate existing XS classic applications to run in the new XS advanced run-time environment, SAP recommends that you first check the features available with the installed version of XS advanced. If the XS advanced features match the requirements of the XS classic application you want to migrate, then you can start the migration process, for example, using the *XS Advanced Migration Assistant*, which is an on-premise tool. Migration tools are designed to automate many of the steps in the migration process and are described in more detail in this section.

Before starting the migration process, it is recommended to review the design of your XS classic application and compare it with the design required for applications running in the XS advanced-model run-time

environment. It is important to bear in mind the fundamental differences in design between the XS classic and XS advanced run-time environments.

For example, the security concept underpinning XS advanced is different to the one used in XS classic. In XS classic, business users are also database users, to whom privileges can be assigned which grant or restrict access to data. In XS advanced, business users are not linked to database access privileges, and it is no longer possible to differentiate between users by assigning different privileges to database-related objects or schemas. Instead, authorization scopes are defined in the application layer.

You can also optimize some of the components used in the XS classic application to make an automatic migration easier and faster. Alternatively, you can rewrite the application, to take advantage of the new features XS advanced provides.

SAP HANA Extended Application Services, Classic Model

In XS classic applications, data-intensive and model-based calculations must be close to the data and, therefore, need to be executed in the index server, for example, using SQLScript or the code of the specialized functional libraries. The presentation (view) logic runs on the client – for example, as an HTML5 application in a Web browser or on a mobile device. Native application-specific code, supported by SAP HANA Extended Application Services, classic model, can be used to provide a thin layer between the clients on one side, and the views, tables and procedures in the index server on the other side. A typical XS classic application contains control-flow logic based on request parameter; invokes views and stored procedures in the index server; and transforms the results to the response format expected by the client.

SAP HANA Extended Application Services, Advanced Model

SAP HANA Extended Application Services advanced model (XS advanced) adds an application platform to the SAP HANA in-memory database. In the Cloud, this platform is provided by Cloud Foundry. An SAP-developed run-time environment is bundled with SAP HANA on-premise which provides a compatible platform that enables applications to be deployed to both worlds: the Cloud and on-premise. XS advanced is optimized for simple deployment and the operation of business applications that need to be deployed in both worlds. For this reason, the XS advanced programming model fully embraces the Cloud Foundry model and leverages its concepts and technologies. In areas where Cloud Foundry as an intentionally generic platform for distributed Web applications does not address relevant topics or offers choice, the XS advanced programming model provides guidance that is in line with the general Cloud programming model.

In simple terms, XS advanced is basically the Cloud Foundry open-source Platform-as-a-Service (PaaS) with a number of tweaks and extensions provided by SAP. These SAP enhancements include amongst other things: an integration with the SAP HANA database, OData support, compatibility with XS classic model, and some additional features designed to improve application security. XS advanced also provides support for business applications that are composed of multiple micro-services, which are implemented as separate Cloud Foundry applications, which combined are also known as Multi-Target Applications (MTA). A multi-target application includes multiple so-called “modules” which are the equivalent of Cloud Foundry applications.

Related Information

[The XS Advanced Application-Migration Process \[page 768\]](#)

[The XS Advanced Migration Assistant \[page 770\]](#)

[The XS Application Migration Report \[page 771\]](#)

[SAP HANA XS Advanced Migration Guide](#)

14.1.1 The XS Advanced Application-Migration Process

The high-level steps required to migrate an XS classic application to the XS advanced run-time environment.

The XS Advanced Migration Assistant enables you to automate most of the tasks required to migrate an XS classic application to the XS advanced run-time environment. The following table lists the main steps required to complete the migration process:

→ Tip

For more information about the migration assistant and a step-by-step guide to the migration process, see the *SAP HANA XS Advanced Migration Guide*.

Application Migration Steps

Steps	Action	Comments
1	Prepare for the migration operation	Some XS classic artifacts have been deprecated or are no longer used in XS advanced applications, for example: • Application Function Library (AFL) models (*.aflpmm1) • Decision Tables (*.decrule) All deprecated object types used in the migrated XS classic application must be migrated manually, for example, using the view-migration tool included in SAP HANA Studio's <i>Modeler</i> perspective. For more information about this tool and the migration process, see the <i>SAP HANA XS Advanced Migration Guide</i>.
2	Prepare the SAP HANA source systems and required users	Since the application-migration process requires access to the SAP HANA XS classic Repository, it is necessary to configure the source system that hosts the Repository where the legacy XS classic application artifacts are located. ⓘ Note If your source system is older than SAP HANA SPS 11, you also need an “external parse system”. Your target XS advanced system can be configured to play the role of external parser.

Steps	Action	Comments
3	Migrate the XS classic application	Install and run the <i>XS Advanced Migration Assistant</i> and then start the migration process. ▲ Restriction The target XS advanced system where you plan to deploy the migrated application must be running HANA 2.0. You cannot use the <i>XS Advanced Migration Assistant</i> to help you migrate your XS classic application to an SAP HANA 1.0 system, for example, SPS 11 or SPS 12.
4	Locate and read the report generated by the <i>XS Advanced Migration Assistant</i> .	The report generated by the <i>XS Advanced Migration Assistant</i> lists the problems found during the migration process; the problems must be fixed before running the migration again. Typically, the problems concern the following artifacts: • Deprecated attribute views • Incorrectly formatted calculation views • Security artifacts (<code>.hdbroles</code> and <code>.xsprivileges</code>) • Database artifacts (<code>.hdbtable</code>, <code>.hdbview</code>, etc.) • XS JavaScript artifacts (<code>.xsjs/</code> <code>.xsjslib</code>)
5	Deploy the migrated application to XS advanced	Upload the successfully migrated design-time application artifacts to XS advanced. Build the new XS advanced application using the migrated artifacts. Set up the target system (including any required user-provided services). Deploy the migrated and built application to the XS advanced run time.

Related Information

[The XS Advanced Migration Assistant \[page 770\]](#)

[The XS Application Migration Report \[page 771\]](#)

[Legacy Object Types not Supported in XS Advanced \[page 773\]](#)

[SAP HANA XS Advanced Migration Guide](#)

14.1.2 The XS Advanced Migration Assistant

The migration assistant helps you to migrate an XS classic application to the XS advanced model run time environment.

SAP HANA XS advanced model includes the XS Advanced Migration Assistant - a tool that is designed to help you migrate an XS classic application to the XS advanced run-time environment. After downloading and installing the XS Advanced Migration Assistant, you can use it to automate large parts of the migration process, for example, the conversion of many design-time artifacts from the XS classic file format and syntax to the syntax, format, and file extension required to build and deploy the migrated application in the XS advanced run-time environment.

→ Tip

The *XS Advanced Migration Assistant* is available for download from the SAP Software Download Center listed in *Related Information* below. Search for the component *XSAC MIGRATION 1* in *SUPPORT PACKAGES AND PATCHES*.

The XS Advanced Migration Assistant scans the XS classic application and generates a report that indicates which design-time artifacts can be automatically migrated to the format required in XS advanced and which artifacts require attention before the migration process can be started. For example, scripted (text-based) calculation views, attribute views, and analytic views and privileges can all now be converted automatically into graphical calculation views using tools included in the Migration Assistant; some deprecated artifacts, for example, decision tables, must be manually converted using the XS classic admin tool SAP HANA Studio.

When all the issues listed in the migration report have been addressed and fixed, the XS Advanced Migration Assistant must be run again to perform the migration. During the migration process, the migration assistant converts the XS classic application's design-time artifacts to the appropriate format for XS advanced and places the migrated artifacts in the application-project containers required to be able to build and deploy the migrated application in the XS advanced run-time environment.

During the application-migration process, SAP HANA XS classic Repository objects are migrated to their equivalent counterparts in XS advanced and placed in the appropriate project container, for example:

- *web/*
All user-interface objects and any static Web content
- *xsjs/*
Application code, XS jobs and schedules, and HTTP destinations
- *db/*
Database objects such as tables, views, procedures, and so on

The migration assistant puts objects whose destination cannot be automatically established into the *migration* container. For these objects, it is necessary to decide how each file must be processed. Objects in the *migration* container must be copied manually by a developer into the appropriate target container in the XS advanced application project, for example, *web*, *xsjs*, or *db*. This includes (but is not limited to) the following file types:

ⓘ Note

For more details about which design-time artifacts are copied to which container during the migration process, see the *SAP HANA XS Advanced Migration Guide* in *Related Information*.

- *.properties*
In XS classic applications, the *.properties* file typically contains translation text. However, the XS Advanced Migration Assistant is not able to establish if a *.properties* file really is a translation-

text file. The migration assistant also does not know to which target application container to copy the `.properties` file, for example, the `web`, `xsjs`, or `db` container.

- `.xml`
The XS Advanced Migration Assistant checks for known structures in `.xml` files and, depending on what it finds and recognizes, copies the file to the appropriate application project container (`web`, `xsjs`, or `db`). If the migration assistant is not able to determine the type of XML file to be migrated, the `.xml` file is copied to the `migration` container from where it must be moved manually to the appropriate application-project container.
- `.json`
The XS Advanced Migration Assistant checks for known structures in `.json` files and copies the file to the appropriate XS advanced application project container. If it is not possible to determine the type of file to be migrated, the migration assistant copies the file to the `migration` container from where it must be moved manually to the appropriate XS advanced application-project container.

Related Information

[The XS Application Migration Report \[page 771\]](#)

[SAP HANA XS Advanced Migration Guide](#)

[SAP Software Download Center \(Logon required\)](#) 

14.1.3 The XS Application Migration Report

The XS Advanced Migration Assistant generates a report, which lists details of the migration operation including any problems that need to be fixed.

The report generated by the *XS Advanced Migration Assistant* is in the form of an HTML file that is split into convenient categories, for example, “Security”, “Database”, or “XS JavaScript”. The information generated for each category includes a summary of the migration as well as a comprehensive list of any problems that occurred during the migration. The location of the migration report is specified in the options you can define before you run the *XS Advanced Migration Assistant*.

The source objects from the provided XS classic application's delivery units (DU) are exported from the source XS classic system, analyzed, migrated, and then written into the folder structure required for an XS advanced application.

→ Tip

For more detailed information about the steps required to migrate XS classic applications to XS advanced, see the SAP HANA XS Advanced Migration Guide in *Related Information*.

The format of the information in the report matches the order and scope of the phases the migration process goes through, for example, preparation, security, database, etc. The following table lists the sections included

in migration report generated by the *XS Advanced Migration Assistant* and provides a brief description of each section's contents:

XS Advanced Migration Report Elements

Report Section	Content
Preparation	A list of the XS classic artifacts that require manual migration before you start the migration of the XS classic application. For example, using the tools provided in SAP HANA Studio, you need to manually migrate decision tables and files generated by the Application Function Modeler (*.af1pmm1) artifacts. After the manual migration of the listed artifacts is complete, it is necessary to run the XS Advanced Migration Assistant again.
Security	Since the security concept used in XS advanced is incompatible with the one used in XS Classic, manual migration steps are required in order to complete the migration of an XS classic application's security artifacts to XS Advanced. The <i>Warnings</i> section includes a list of the roles and access privileges that must be checked using the XS JavaScript command <code>getConnection()</code> in the XS JS \$.hdb API. ⚠ Restriction Due to the change in security concepts between XS classic and XS advanced, not all the features configured in the XS classic application's access-configuration file (*.xsaccess) can be migrated to the XS advanced application router's configuration file (xs-app.json).
Unsupported Features	A list of the XS classic objects that are no longer supported or have been discontinued. These are objects which the migration assistant cannot migrate automatically. Any problems listed in this section must be fixed by hand.
Database Artifacts	Information about design-time database artifacts which created warnings during the migration process. 📌 Note The technical HDI user in XS advanced must be granted permission to access the SAP HANA database in order to be able to read and retrieve the XS classic database artifacts, for example, tables, views, procedures, and so on.
XS JavaScript Migration	A list of the statements in XS JavaScript code (for example, in the application's code *.xsjs and libraries *.xsjslib which cannot be migrated automatically; you must migrate this XS JavaScript code manually.
Translation	A list of the XS classic application's *.hdbtextbundle files, which contain UI texts and possibly existing translation texts. The contents of the .hdbtextbundle files are migrated to .properties files using the rules concerning references to (and from) .hdbtextbundle files.
Unknown File Types or Content	A list of the components that the <i>XS Advanced Migration Assistant</i> was not able to identify. For each unidentified component the file type is specified. You must decide whether these objects are relevant, and copy them into the appropriate target container directory if needed.

Report Section	Content
Objects not Migrated	Details of the objects which the <i>XS Advanced Migration Assistant</i> was not able to migrate, for example, because the objects are either not relevant (do not belong to the application project) or have been successfully migrated to another object type.

Related Information

[SAP HANA XS Advanced Migration Guide](#)

14.1.4 Legacy Object Types not Supported in XS Advanced

Some XS classic object types cannot be migrated to XS advanced.

The files listed in the following table cannot be migrated to XS advanced and are not moved to the XS advanced application project that is generated by the migration process.

Note

Strictly speaking, not all of the object types listed in the following table are XS classic objects, for example: Eclipse-related files such as *.project and *.classpath or the Windows *.bat file, and Python *.pyc files. The one thing in common that **all** the files in this table share is that they are no longer relevant for XS advanced applications, and are not migrated by the XS Advanced Migration Assistant.

Non-Migratable Object Types

Object Type	Description
*.xsapp	An application-specific file in an XS classic repository package that defines the root folder of a native SAP HANA application. All files in that package (and any subpackages) are available to be called via URL.
*.project	An Eclipse project descriptor used in SAP HANA studio for . The *.project file is a design-time artifact that is stored in the SAP HANA XS classic repository.
*.pyc	Compiled Python files
*.regignore	A file containing ignore or exclude patterns for XS Classic repository used by HANA Studio Repository Team Provider
*.DS_Store	MacOS Desktop Services Store used for custom file attributes, icons, etc. Should not be committed to any project repository.
*.bat	MS DOS Batch files

Object Type	Description
*.classpath	Eclipse JDT project Classpath configuration
*.jdtscope	Eclipse JDT project configuration
*.gitignore	A file containing ignore (exclude) patterns for Git client
*.db	A database file used by Sybase SQL Anywhere, SQLite, and others
*.xsprivileges	A file that defines a privilege that can be assigned to an SAP HANA Extended Application Services application, for example, the right to start or administer the application.
*.xssecurestore	The design-time file that creates an application-specific secure store; the store is used by the application to store data safely and securely in name-value form.
*.xssqlcc	A file that enables execution of SQL statements from inside server-side JavaScript code with credentials that are different to those of the requesting user

Related Information

[SAP HANA XS Advanced Migration Guide](#)

15 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 an 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.

© 2025 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see <https://www.sap.com/about/legal/trademark.html> for additional trademark information and notices.