

Reading Sample

Core data services (CDS) play a central role in SAP S/4HANA. With their technical capabilities for data modeling, they form the basis of any application. As it is one of the key technologies in the ABAP RESTful application programming model, we now want to take a detailed look at CDS and thus lay the foundation for the following chapters.



“Core Data Services: Data Modeling”



Contents



Index



The Author

Baumbusch, Jäger, Lensch

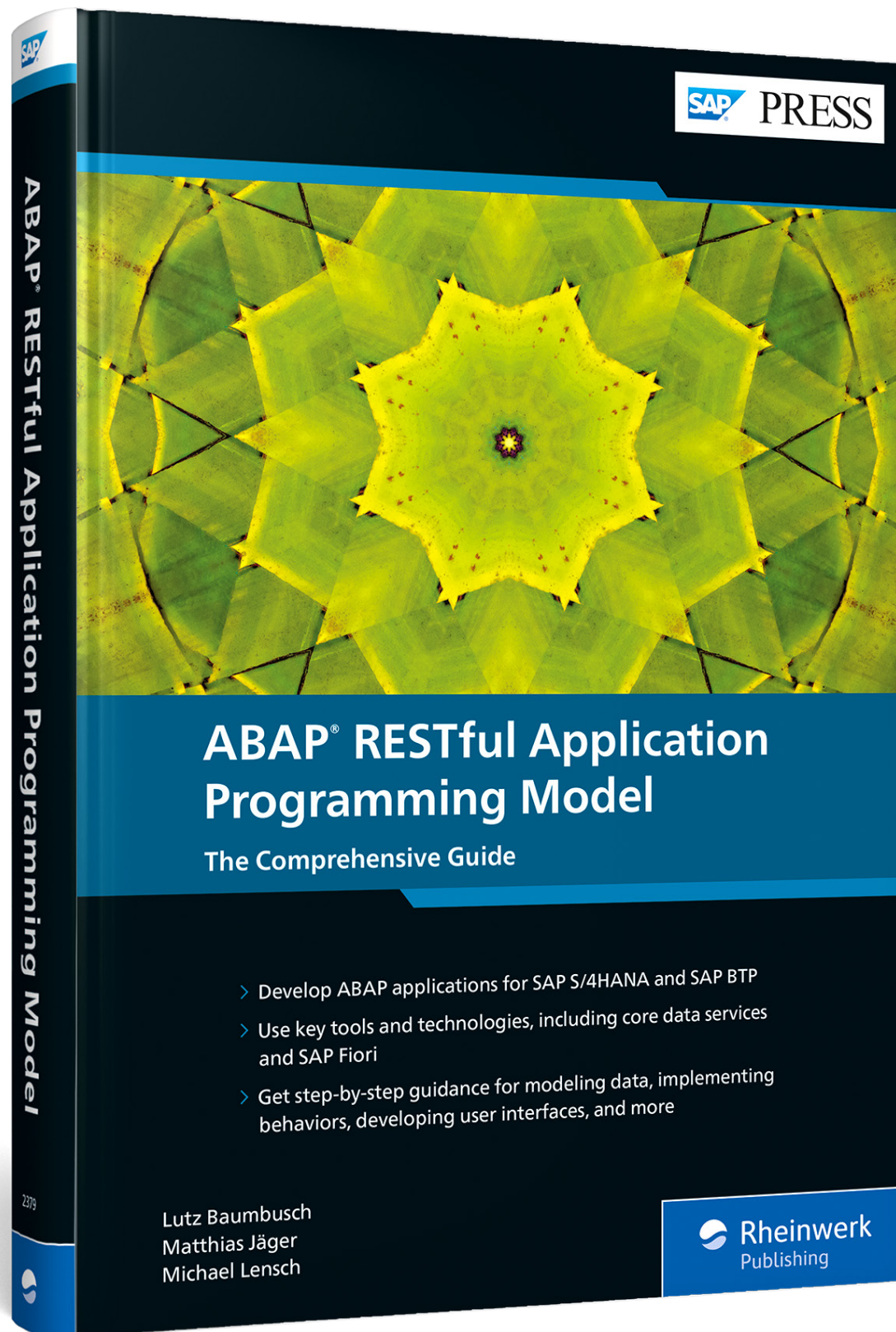
ABAP RESTful Application Programming Model: The Comprehensive Guide

508 pages, 2023, \$89.95

ISBN 978-1-4932-2379-4



www.sap-press.com/5647



Chapter 2

Core Data Services: Data Modeling

Core data services (CDS) play a central role in SAP S/4HANA. With their technical capabilities for data modeling, they form the basis of any application. As it is one of the key technologies in the ABAP RESTful application programming model, we now want to take a detailed look at CDS and thus lay the foundation for the following chapters.

ABAP core data services (ABAP CDS) have been available since ABAP 7.40 SPO5. They provide you with numerous options when you create applications. To be able to use them, it's necessary to work with ABAP development tools (ADT) in Eclipse. Editing CDS views using the classic ABAP Workbench is not supported.

Installing and Configuring ABAP Development Tools

For information on how to install Eclipse and ADT, and how to connect them to your ABAP backend system, go to <https://tools.hana.ondemand.com/>.



In Section 2.1, you'll become familiar with the basic concepts of CDS and why CDS plays a significant role in development for SAP S/4HANA. Based on the information in Section 2.2, you'll create your first CDS data model and gain initial experience with ADT. Section 2.3 will explain how to map relationships between CDS entities. In Section 2.4, we'll describe annotations, which are meta-information you can use to enrich your data model.

You can use access controls to restrict access to your CDS data model; this is described in Section 2.5. In Section 2.6, you'll learn how to enrich CDS views with additional information without any modification.

You won't always be able to map all requirements standard CDS functionalities. Section 2.7 describes how you can still achieve your goals in such a scenario. In Section 2.8, you'll learn about SAP's virtual data model (VDM), which provides you with another way to access the persisted data of an SAP S/4HANA system. Finally, in Section 2.9, we'll introduce some language elements that you can use when modeling business objects.

2.1 What are Core Data Services?

SAP defines CDS in SAP Help as an “infrastructure that developers can use to create semantically rich and persistent data models.” The resulting data models are built on top of existing database tables and are intended to be easier to understand and utilize. This is achieved by changing the perspective away from a purely technical view of the database tables to a view of data oriented toward the business object.

ABAP CDS versus
SAP HANA CDS

CDS views are used for flexible data retrieval and are the central tool for data modeling in the ABAP RESTful application programming model. There are two CDS implementations: ABAP and SAP HANA CDS. While *SAP HANA CDS* views are created at the database level, *ABAP CDS* views are maintained on the SAP application server or ABAP platform. One of the major advantages of developing ABAP CDS views over native SAP HANA CDS views created on the SAP HANA database is the connection to the SAP change and transport system (CTS). Working on the application server also allows you to continue working in your familiar ABAP environment while taking advantage of the capabilities of the SAP HANA database. In the following sections, we’ll only take a look at the concept of ABAP CDS views.

Syntax

The syntax of CDS is aligned with the SQL standard. CDS is therefore often referred to as an extension of SQL. In detail, CDS includes the following domain-specific languages:

- Data definition language (DDL) for data modeling
- Query language (QL) for querying data
- Data control language (DCL) for defining access restrictions

You’ll learn how to use these languages in the following sections.

Significance for
SAP S/4HANA

Since CDS plays a central role in SAP S/4HANA, it’s logical that it’s of great importance for the daily work of SAP S/4HANA development teams. CDS plays a crucial role in the implementation of the following basic concepts of the SAP S/4HANA architecture:

- Code-to-data paradigm
- Virtual data model (VDM)
- Programming models for SAP S/4HANA

Code-to-data
paradigm

The basic idea of the *code-to-data paradigm* is to move application logic from the application server to the database (code pushdown, see Figure 2.1). Calculations should be performed by the database system as much as possible. The results of these calculations are then delivered to the application server. This approach reduces the amount of data that must be transferred from the database to the application server. In addition to the extended

possibilities of ABAP SQL (formerly Open SQL), which have been available since ABAP 7.40 SPO5, and the ABAP-managed database procedures (AMDP), CDS views are excellently suited for implementing the code-to-data paradigm, for example, by defining calculations or access controls directly in the data model and thus shifting them to the database level. This enables you to harness the power of the SAP HANA database and let the database do the work for you.

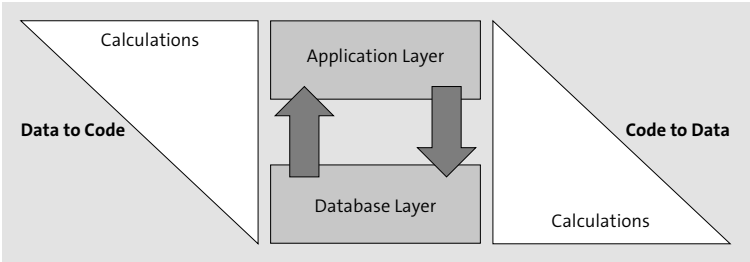


Figure 2.1 Code-to-Data Paradigm

CDS serve as a tool for data modeling. With the *virtual data model (VDM)*, SAP provides a data model based on CDS in the standard SAP S/4HANA system. VDM was developed by SAP according to extensive, detailed guidelines and standards regarding hierarchical structure and naming conventions. It maps the application data of SAP S/4HANA and is used by transactional and analytical applications and by the APIs provided for SAP S/4HANA. Figure 2.2 shows the hierarchical structure of the VDM using the example of the SalesOrder business object in ADT’s Dependency Analyzer. This tool is good for getting an initial overview of a view hierarchy.

Virtual data model

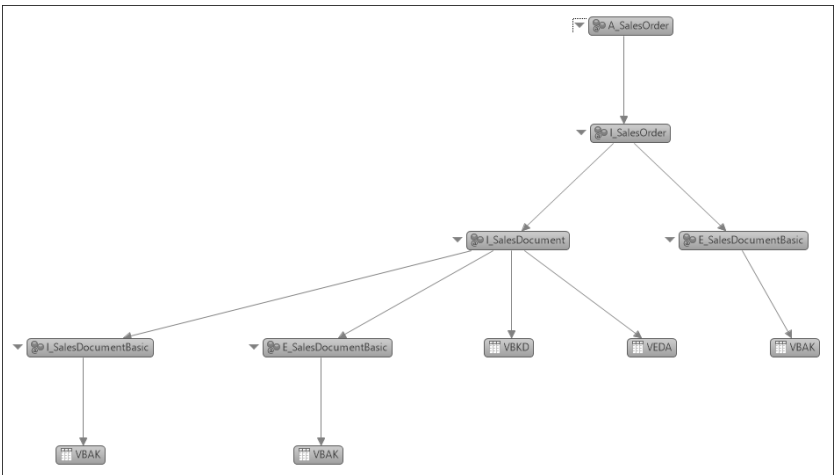


Figure 2.2 Excerpt from Virtual Data Model: API View of A_SalesOrder in the Dependency Analyzer of ABAP Development Tools

CDS as a component of programming models

The strict adherence to guidelines and standards results in a high degree of consistency and comprehensibility. Many of the CDS views in SAP S/4HANA are explicitly enabled for use. For SAP S/4HANA developers, VDM is another way to obtain data. It can be regarded as an additional data access layer. In Section 2.8, we'll go into more detail about the VDM.

ABAP programming model for SAP Fiori

However, classic ABAP applications developed on the basis of such data models are subject to certain limitations. You can only access the CDS view in read-only mode, and no transactional processing is possible. For this reason, and also because of the increasing importance of SAP Fiori as an SAP interface technology, SAP initially developed the ABAP programming model for SAP Fiori. Here, the transactional behavior is implemented by annotations that are used to automatically generate a Business Object Processing Framework (BOPF) model. The behavior implementation of the BOPF actions, derivations, and validations are carried out in ABAP.

ABAP RESTful application programming model

However, this approach introduces complexity and requires the parallel use of two modeling frameworks (CDS and BOPF). While type checking is already performed at design time in the typed CDS framework, errors often only occur at runtime in the very generic BOPF, which makes troubleshooting more difficult due to the complexity. The desire to reduce the complexity of the programming model eventually led to the introduction of the ABAP RESTful application programming model. The ABAP RESTful application programming model is also based on a semantic CDS data model. The transactional logic is expressed by the behavior definition and its implementation. The entities to be used by the consumer (e.g., an SAP Fiori app) are defined and published by a business service (see Figure 2.3).

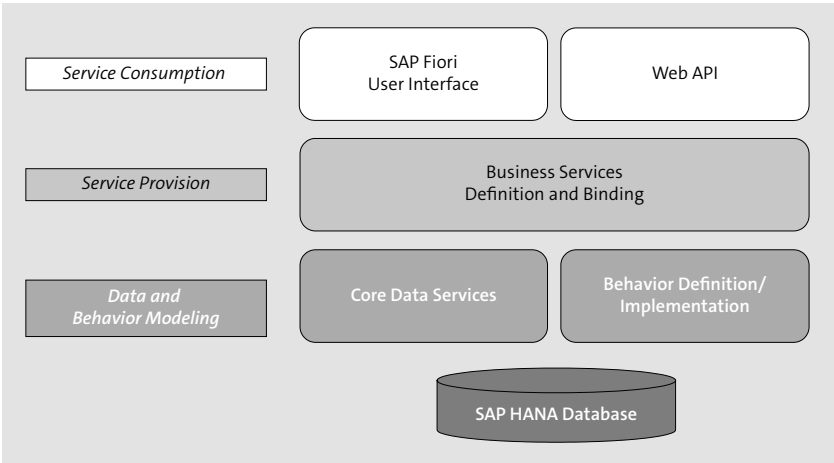


Figure 2.3 Layers of the ABAP RESTful Application Programming Model

2.2 Structure and Syntax of Core Data Services

In this section, we'll show you how to create your first simple CDS views. You'll learn how to proceed when developing CDS views, especially how to use the Eclipse development tool and ADT. You'll also get to know the basic syntax elements of CDS and some ways to obtain data. Here, we don't yet make a concrete reference to the ABAP RESTful application programming model. Rather, we intend to lay the foundation to understand the following chapters. If you already have experience with CDS views, you can skip Chapter 2. In Part II, we show you how you can apply essential CDS concepts in various scenarios.

The example in this section is based on the *ABAP flight reference scenario* provided by SAP. You'll create CDS views that read data from the tables in this reference scenario, and add simple logic to the CDS views you create. This way you'll develop an initial, simple CDS data model. In the process, you'll also learn about some functions you can use to analyze the created CDS views.

ABAP flight reference scenario

ABAP Flight Reference Scenario

The ABAP flight reference scenario provides sufficient sample data and logic to familiarize yourself with the ABAP RESTful application programming model. All information about this reference scenario, especially about the installation, can be found in a Git repository at <http://s-prs.de/v868501>.



The goal of this section is to merge and enrich information from the database table persistence model. Table 2.1 lists the database tables of the reference scenario we use.

Table	Meaning
/DMO/FLIGHT	Flights
/DMO/CARRIER	Airlines
/DMO/CONNECTION	Flight connections

Table 2.1 Tables Used from the Reference Scenario

As a complementary logic, for example, a simple statement and a currency conversion are to be performed.

2.2.1 Creating a Basic Interface View

Basic interface view The main task of a *basic interface view* is to read data from database tables and to convert the technical names of the database fields into more meaningful names (for this and other CDS view types, Section 2.8).

- Creating a CDS view** You can create a basic interface view as the first CDS view by following these steps:
1. Start Eclipse and open the ABAP perspective via the menu path **Window • Perspective • Open Perspective • Other...**
 2. In the popup window, select the **ABAP** perspective (see Figure 2.4).

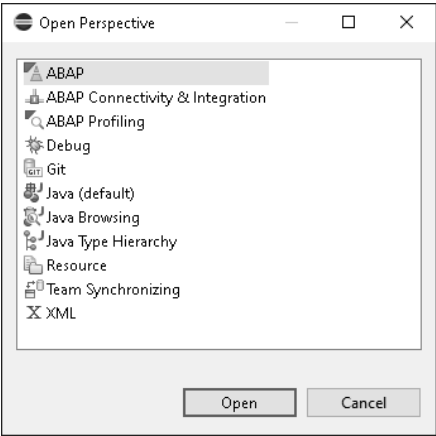


Figure 2.4 Selecting the ABAP Perspective

3. Select **File • New • Other** in the main menu.

4. In the next popup window, select the entry **ABAP • Core Data Services • Data Definition** and confirm the selection by clicking the **Next** button (see Figure 2.5).

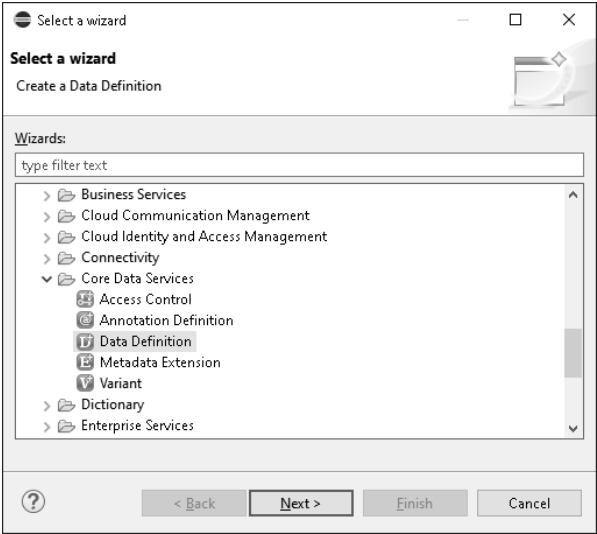


Figure 2.5 Selecting the Type of Development Object to Create

5. This calls a dialog to create a CDS view (see Figure 2.6). Enter the ABAP project in the **Project** field and your development package in the **Package** field. You should enter the name of the CDS transport object (data definition language source [DDLS] type) in the **Name** field (e.g., “ZI_Flight”) and a description in the **Description** field (e.g., “Flight”). Then, confirm your input by clicking the **Next** button.

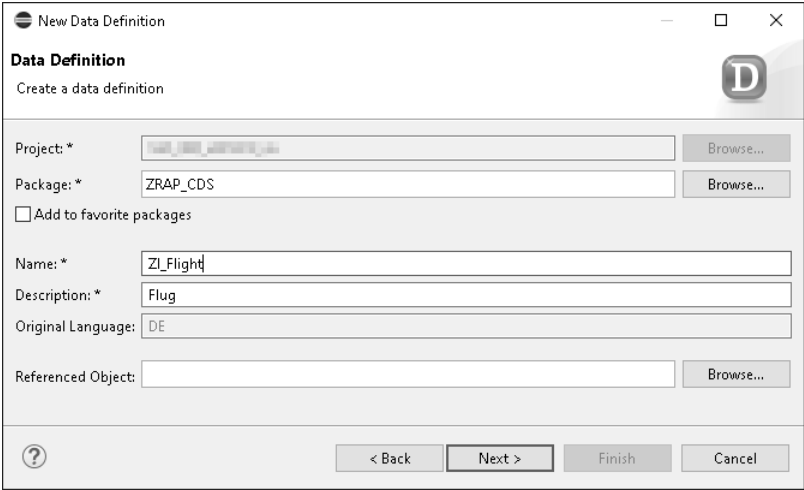


Figure 2.6 Properties of the CDS View

- 6. In the next screen, select a transport request (if you aren't working with local objects) and confirm again by clicking the **Next** button.
- 7. In the following screen, you can then select a template for the creation of your CDS view. The provided templates refer to commonly used CDS modeling options and simplify the creation of corresponding CDS views. If you want to work without a template, you must uncheck the **Use the selected template** box. To create a simple CDS view, select the **Define View** template and confirm your selection by clicking the **Finish** button (see Figure 2.7).

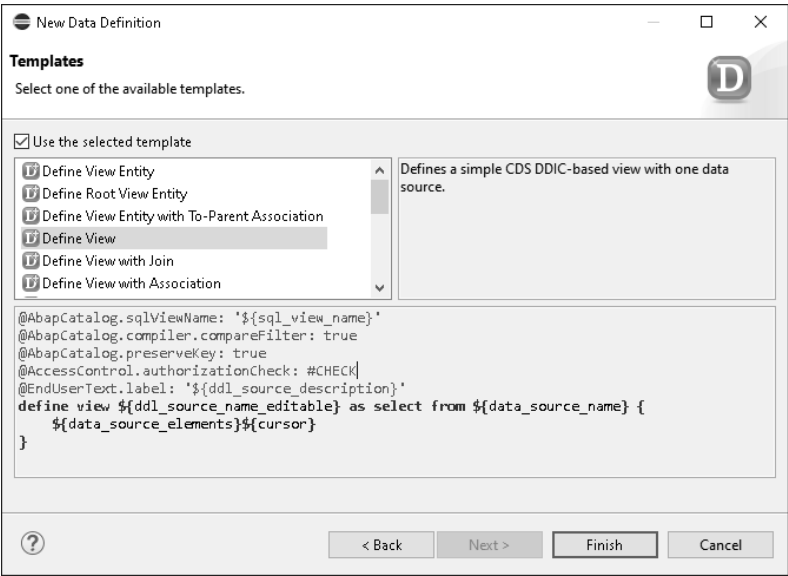


Figure 2.7 Selecting a Template

Editing the CDS view

You'll then be redirected to a source code-based editor where you can edit the details of your CDS view. Due to the selection of the template, some basic elements of the code will already be suggested (see Figure 2.8).

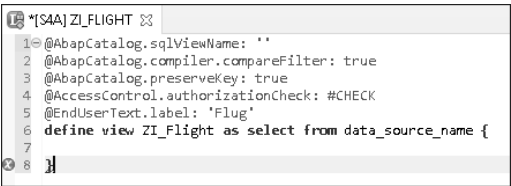


Figure 2.8 Coding Generated by the Template in the Source Code-Based Editor

SQL view name, data source, and field list

To complete your first CDS view, perform these steps:

- 1. You must assign a name for the *SQL* view in the ABAP dictionary, which is automatically generated when the CDS view gets activated.

- 2. You must also define the *data source* of your CDS view.
- 3. Also, you must define the *field list* (also referred to as an *element list* or *select list*).

Then, you assign the name of the ABAP dictionary representation of your CDS view using the following annotation:

```
@ABAPCatalog.sqlViewName: 'ZI_FLIGHTV'
```

Then, you must specify the data source after the language element as select from:

```
as select from /DMO/FLIGHT
```

The field list can be defined inside the curly brackets. The easiest way to do this is to place the cursor in this area and press the keyboard shortcut **Ctrl** + **Space**. Then, you can include individual elements in the field list, or select **Insert all elements (template)**, as shown in Figure 2.9.

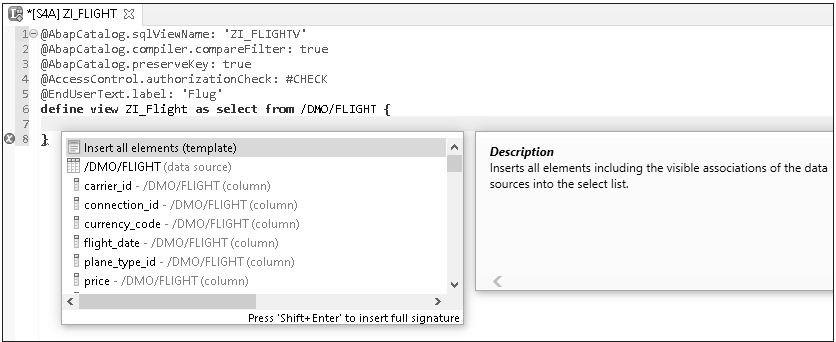


Figure 2.9 Defining the Field List

Now you must activate your CDS view using the keyboard shortcut **Ctrl** + **F3**. You have now created your first CDS view, activated it, and made it available for use. Listing 2.1 shows the complete coding.

Activating the CDS view

```
@AbapCatalog.sqlViewName: 'ZI_FLIGHTV'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Flight'
define view ZI_Flight
  as select from /dmo/flight
{
  key carrier_id      as CarrierId,
  key connection_id   as ConnectionId,
```

```
key flight_date    as FlightDate,
price             as Price,
currency_code     as CurrencyCode,
plane_type_id     as PlaneTypeId,
seats_max         as SeatsMax,
seats_occupied    as SeatsOccupied
}
```

Listing 2.1 CDS View ZI_Flight

2.2.2 Analyzing the Data Model

You should now take a closer look at your first CDS data model. When you activated it, you created three objects (see Figure 2.10).

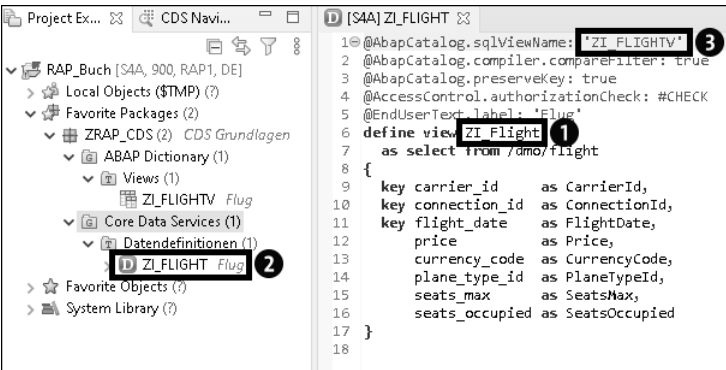


Figure 2.10 Objects of the CDS Data Model

- ❶ The CDS entity ZI_Flight represents the properties of your CDS view (that is, the SQL characteristic and additional metadata). You can reference this entity as a data source in an ABAP SQL statement, for example.
- ❷ The DDLS source code object ZI_FLIGHT is the transportable development object that you'll find in your transport request accordingly (see Figure 2.11).

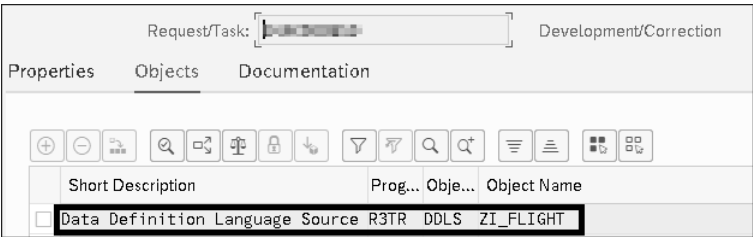


Figure 2.11 DDLS Transport Object

❸ When you activated the CDS view, you generated the SQL view ZI_FLIGHTV in the ABAP dictionary. You can display this in the dictionary maintenance Transaction SE11 (see Figure 2.12) and test it using Transaction SE16. Note that this is a SQL view based on CDS. Changes via Transaction SE11 are not possible for this reason. Also, not all CDS entity information can be represented in Transaction SE11. For example, the display and maintenance of meta information and the definition of calculations or type conversions are only possible in the Eclipse environment.

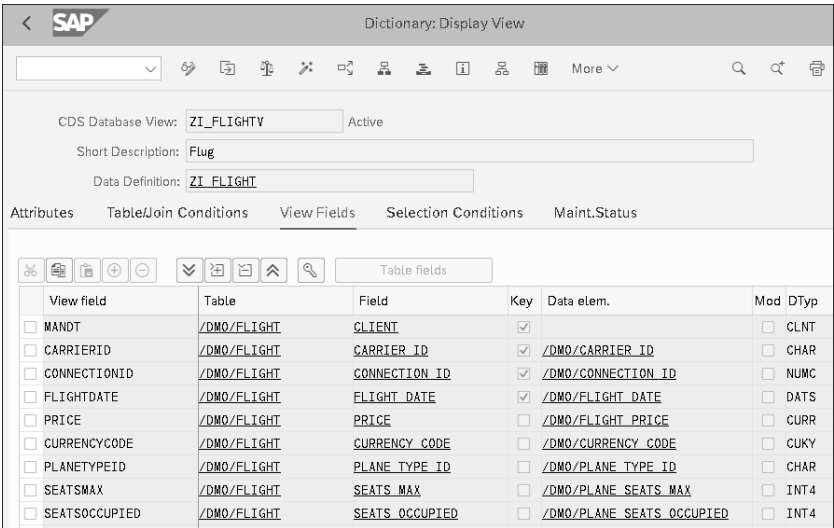


Figure 2.12 Representation of the CDS View in the ABAP Dictionary

«

Deleting CDS Views

Deleting only the SQL view isn't possible. CDS entity and SQL view can't be deleted independently. When the CDS entity gets deleted, the generated SQL view will also be deleted.

»

For testing and analyzing the CDS view result set, the ADT data preview function is useful. This is similar to the classic Transaction SE16 because you don't need to execute an SQL statement to test the CDS view. Simply right-click on the source code to open the context menu of the CDS editor and select the **Open With • Data Preview** path (see Figure 2.13).

Data preview

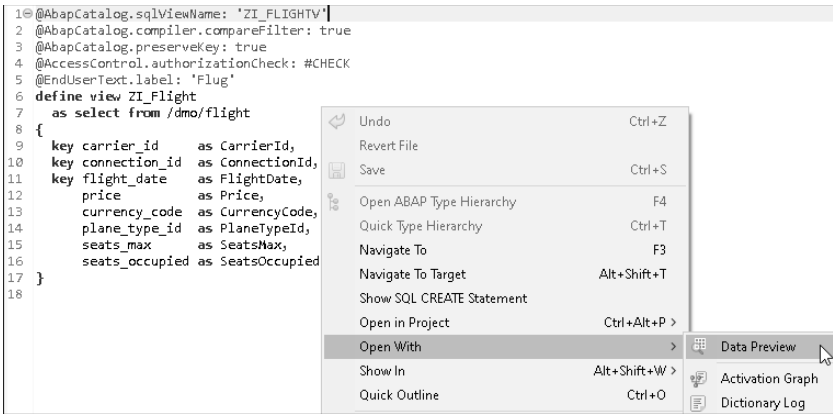


Figure 2.13 Data Preview in ABAP Development Tools

By default, the first 100 entries of the result set will be displayed (see Figure 2.14). When you click on the column headers, you can sort the result set. In addition, the following useful functions are available in the toolbar:

- **Add filter** restricts the result set.
- **Select Columns** selects the field list.
- **Number of Entries** displays the number of hits according to the chosen selection criteria.

CarrierId	ConnectionId	FlightDate	Price	CurrencyCode	PlaneTypeId	SeatsMax	SeatsOccupied
SQ	0001	2021-10-27	10818.00	SGD	767-200	260	223
SQ	0001	2020-12-31	5950.00	SGD	A340-600	330	168
SQ	0002	2021-10-28	11765.00	SGD	747-400	385	350
SQ	0002	2021-01-01	10953.00	SGD	747-400	385	334
SQ	0011	2021-10-28	2359.00	SGD	767-200	260	132
SQ	0011	2021-01-01	4880.00	SGD	A340-600	330	310
SQ	0012	2021-10-30	4665.00	SGD	767-200	260	236
SQ	0012	2021-01-03	2574.00	SGD	747-400	385	215
UA	0058	2021-10-25	6629.00	USD	767-200	260	200
UA	0058	2020-12-29	4996.00	USD	747-400	385	231

Figure 2.14 Result Set of the Data Preview

SQL console For a more detailed analysis of the data set, you can use the **SQL Console** function. It opens another view that displays the SQL statement underlying the selected records. This statement can also be edited. For example, you can add a WHERE clause to further restrict the result set. Selecting **Run** will re-run the query and display the updated result set (see Figure 2.15).

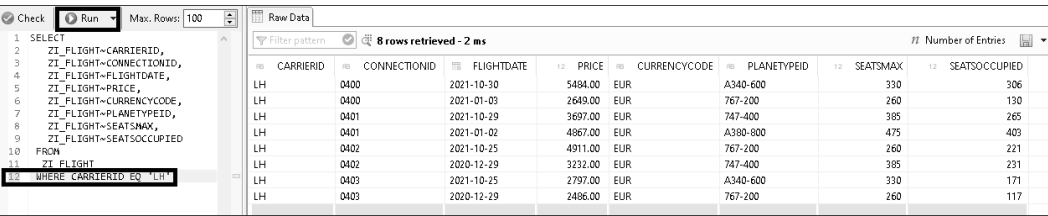


Figure 2.15 SQL Console Within the Data Preview

You can get information about the syntax and meaning of individual language elements by placing the cursor on the corresponding language element in the source code editor and pressing the **F1** key. The context-sensitive function then opens the ABAP CDS language reference in a separate view (see Figure 2.16).

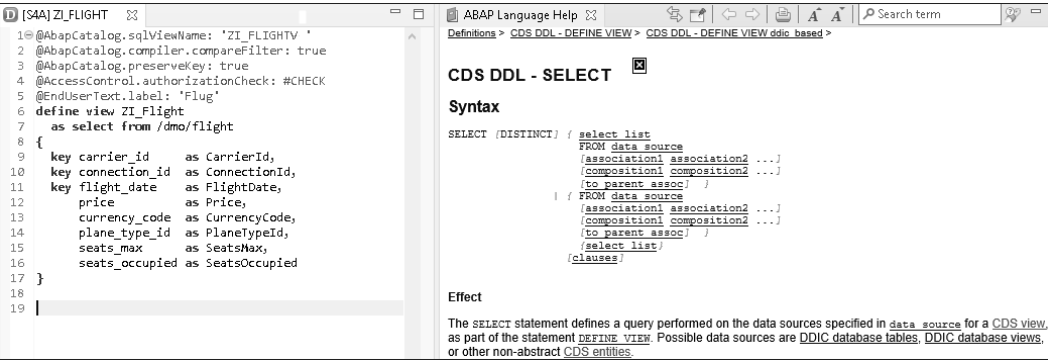


Figure 2.16 ABAP CDS Language Reference

On the other hand, if you need information about the elements used in the source editor, such as tables, views, or table fields, you can press the **F2** key. This opens a popup window with the corresponding information (see Figure 2.17).

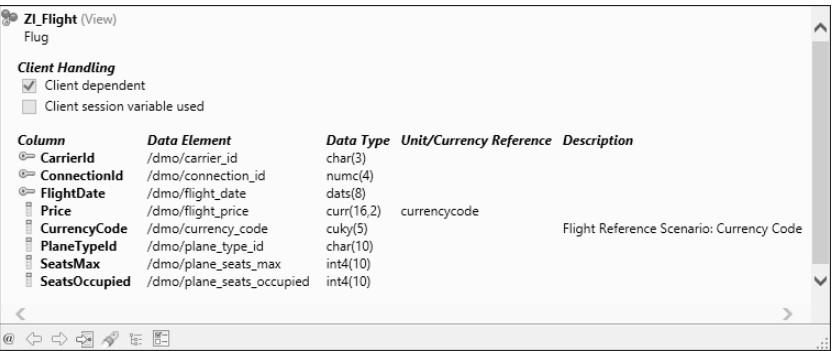


Figure 2.17 Information About the Selected Code Element

2.2.3 Using CDS Views

Call in ABAP You can use the CDS view within an ABAP program essentially in the same way as you are used to using database tables. For example, you can select data from the CDS view in a `SELECT` statement, as shown in Listing 2.2, or refer to the CDS view in the typing of variables. The use of uppercase and lowercase letters doesn't matter when using the CDS view, but it's recommended to use camel case notation for better readability. This notation will be retained when the development object is created, even if the technical name is converted to uppercase in the repository and is displayed that way in the project explorer in Eclipse. When generating OData services based on CDS views, this notation is preserved.

```
DATA flights TYPE STANDARD TABLE OF zi_flight.

SELECT carrierid, connectionid, flightdate, Price,
       CurrencyCode, PlaneTypeId, SeatsMax, SeatsOccupied
FROM   ZI_Flight
INTO TABLE @flights
WHERE  carrierid = 'LH'.
```

Listing 2.2 Using a CDS View in ABAP

Call in CDS A CDS view can also be called from another CDS view (see Listing 2.3). This enables a hierarchical structure of CDS-based data models, which is called a *view stack*.

```
@EndUserText.label: 'Flight detail'
define view ZI_FlightDetail
  as select from ZI_Flight
{
  key CarrierId,
```

Listing 2.3 Using a CDS View Within Another CDS View

2.2.4 Extending the Data Model

The created CDS view `ZI_Flight` accesses the database table `/DMO/FLIGHT` and makes its fields available for use. The technical field names of the database table have been replaced in the CDS view with meaningful labels in camel-case notation. This kind of naming is the first step from the more technical view of the database table to a more business-oriented view of the data and, therefore, one of the most important tasks of basic interface views.

We now want to extend the data model with additional information by providing data from additional data sources. We'll also add descriptive information (metadata). In addition, we'll use simple statements and calculations to move logic to the database level, which is in keeping with the code-to-data paradigm.

To do this, you must first create another CDS view `ZI_FlightDetail`, which is based on the CDS view `ZI_Flight`, by selecting data from `ZI_Flight`. This results in a hierarchical structure that's typical of CDS data models, which you can see in Listing 2.4.

```
01 @AbapCatalog.sqlViewName: 'ZI_FLIGHTDETAILLV'
02 @AbapCatalog.compiler.compareFilter: true
03 @AbapCatalog.preserveKey: true
04 @AccessControl.authorizationCheck: #NOT_REQUIRED
05 @EndUserText.label: 'Flight detail'
06 define view ZI_FlightDetail
07   with parameters
08     P_TargetCurrency :abap.cuky( 5 )
09   as select from ZI_Flight
10   association [1] to /DMO/I_Carrier as _Carrier
11     on $projection.CarrierId = _Carrier.AirlineID
12   association [1] to /DMO/I_Connection as _Connection
13     on $projection.ConnectionId = _Connection.ConnectionID
14     and $projection.CarrierId = _Connection.AirlineID
15 {
16   key CarrierId,
17   key ConnectionId,
18   key FlightDate,
19
20     @Semantics.amount.currencyCode: 'CurrencyCode'
21     Price,
22     @Semantics.currencyCode: true
23     CurrencyCode,
24
25     PlaneTypeId,
26     SeatsMax,
27     SeatsOccupied,
28
29     SeatsMax - SeatsOccupied as SeatsFree,
30
31   case SeatsOccupied
32     when SeatsMax
```

Other data sources
and metadata

CDS view `ZI_Flight-`
`Detail`

```
33     then 'X'
34     else ''
35     end as Flight0ccupied,
36
37     @Semantics.amount.currencyCode: 'TargetCurrency'
38     currency_conversion(
39       amount => Price,
40       source_currency => CurrencyCode,
41       round => 'X',
42       target_currency => :P_TargetCurrency,
43       exchange_rate_date => FlightDate
44     ) as PriceInTargetCurrency,
45
46     @Semantics.currencyCode: true
47     cast(:P_TargetCurrency
48       as vdm_v_target_currency
49       preserving type) as TargetCurrency,
50
51     _Carrier,
52     _Connection
53 }
54 where
55   FlightDate >= $session.system_date
```

Listing 2.4 CDS View ZI_FlightDetail

- Descriptive annotations

Descriptive properties are added in CDS by using annotations. For example, the annotation `@Semantics.currencyCode:true` in line 46 identifies the `CurrencyCode` field as a currency key. The annotation `@Semantics.amount.currencyCode:'TargetCurrency'` in line 37 identifies the field as an amount field for which the associated currency key is contained in the referenced field.
- Access control

While these two annotations refer to the respective elements of the CDS view, the scope of the annotation `@AccessControl.authorizationCheck:#NOT_REQUIRED` in line 4 comprises the view level. This annotation is used if no access control is currently provided for the CDS view, but appropriate roles can be defined at a later time.
- Calculations

With `SeatsMax - SeatsOccupied` as `SeatsFree` a simple calculation of the number of unoccupied seats of a flight is done in line 29. Arithmetic expressions (e.g., addition, subtraction, multiplication, division) can be used in CDS, as can aggregate functions (e.g., `MAX`, `MIN`, `AVG`, `SUM`, `COUNT`).

Built-in functions provide support for some common requirements. Built-in functions from a wide range of areas are available for this purpose: string functions, numeric functions, date and time conversion functions, and conversion functions. For example, the `currency_conversion` function in lines 38 to 44 performs a currency conversion for the value passed to the formal parameter `amount`.

The target currency is passed as a typed parameter in the example when the CDS view is called in lines 7 and 8: with parameters `P_TargetCurrency :abap.cuky(5)`. With `cast(:P_TargetCurrency as vdm_v_target_currency preserving type)` a type conversion to the semantic attributes of the data element `vdm_v_target_currency` is performed in line 47 to 49.

An example of a case distinction used in the `Select` statement can be found in the statement block `case SeatsOccupied when ...`

Case distinction

The access to the data of the CDS view `/DMO/I_Carrier` and `/DMO/I_Connection` happens via associations. Associations are used to map relationships between CDS entities. By expressing association [1] to `/DMO/I_Carrier` and explicitly releasing the associated data with `_Carrier` in line 10, a consumer of the CDS view gains access to these fields.

Association

Last but not least, the `Where` clause in lines 54 and 55 constrains the rows in the result set when accessing the CDS view. In the clause `where FlightDate >= $session.system_date` a *session variable* is used. Session variables contain global information about the current context, the contents of which correspond to the value of certain ABAP system fields when accessed. Direct access to the `SY-DATUM` system variable is not possible.

When you call the data preview now, a popup window for parameter input will display (see Figure 2.18). This behavior is determined by specifying the target currency as a typed parameter.

Parameter input

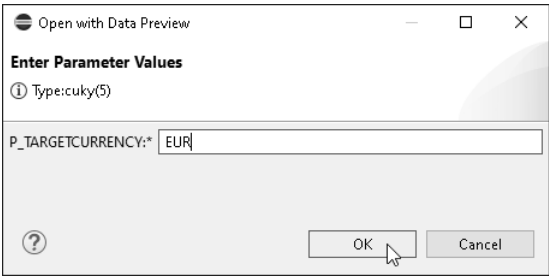


Figure 2.18 Popup Window for Parameter Input

The result set contains the calculated fields and has been filtered by flight date (see Figure 2.19).

ZI_FLIGHTDETAIL(...)

Raw Data

Filter pattern

20 rows retrieved - 24 ms

Parameter

SQL Console

Number of Entries

Select Columns

C..	C...	FlightDate	SeatsMax	SeatsOccupied	SeatsFree	FlightOccupied	Price	C..	PriceInTargetCurrency	TargetCurrency
UA	0058	2021-10-25	260	200	60		6629.00	USD	5354.86	EUR
LH	0402	2021-10-25	260	221	39		4911.00	EUR	4911.00	EUR
LH	0403	2021-10-25	330	171	159		2797.00	EUR	2797.00	EUR
UA	0059	2021-10-26	330	161	169		4131.00	USD	3337.00	EUR
SQ	0001	2021-10-27	260	223	37		10818.00	SGD	6654.69	EUR
AA	0017	2021-10-27	150	139	11		462.00	USD	373.20	EUR
SQ	0002	2021-10-28	305	350	35		11765.00	SGD	7237.24	EUR
SQ	0011	2021-10-28	260	132	128		2359.00	SGD	1451.14	EUR
AA	0018	2021-10-28	475	446	29		3781.00	USD	3054.27	EUR
UA	1537	2021-10-29	150	88	62		893.00	USD	721.36	EUR
AA	0015	2021-10-29	260	137	123		1911.00	USD	1543.69	EUR
LH	0401	2021-10-29	385	265	120		3697.00	EUR	3697.00	EUR
JL	0407	2021-10-29	385	254	131		5346.00	JPY	3974.13	EUR
AZ	0789	2021-10-29	475	441	34		8539.00	EUR	8539.00	EUR
SQ	0012	2021-10-30	260	236	24		4665.00	SGD	2869.67	EUR
AA	2678	2021-10-30	150	141	9		473.00	USD	382.09	EUR
LH	0400	2021-10-30	330	330	0	X	5484.00	EUR	5484.00	EUR
JL	0408	2021-10-30	475	432	43		8159.00	JPY	6065.27	EUR
AZ	0788	2021-10-30	260	221	39		7580.00	EUR	7580.00	EUR
AA	0322	2021-10-31	130	93	37		1103.00	USD	891.00	EUR

Figure 2.19 Data Preview of the Result Set of the ZI_FlightDetail View

Tracing associations

If you select the **Follow Association** entry in the context menu of the result set, as shown in Figure 2.20, you can display the result set of the selected navigation target.

ZI_FLIGHTDETAIL(...)												
Raw Data												
Filter pattern 20 rows retrieved - 24 ms												
	C..	C...	FlightDate	12	SeatsMax	12	SeatsOccupied	12	SeatsFree	12	Flight	
UA	0058		2021-10-25		260		200		60			
LH	0402		2021-10-25		260		221		39			
LH	0403		2021-10-25						159			
UA	0059		2021-10-26						169			
SQ	0001		2021-10-27						37			
AA	0017		2021-10-27						11			
SQ	0002		2021-10-28						35			
SQ	0011		2021-10-28						128			
AA	0018		2021-10-28						29			

Figure 2.20 Tracing Associations

In the next popup window, you can select one of the displayed associations (see Figure 2.21).

List of Associations												
_Carrier -> /DMO/_Carrier [0..1]												
_Connection -> /DMO/_Connection [0..1]												

Figure 2.21 Selecting an Association

Figure 2.22 shows the output of the result set of the associated entity.

ZI_FLIGHTDETAIL -> _Connection -> /DMO/_Connection												
Raw Data												
Filter pattern 1 rows retrieved - 0 ms												
	AirlineID	ConnectionID	DepartureAirport	DestinationAirport	DepartureTime	ArrivalTime	12	Distance				
LH		0402	FRA	EWR	01:30:00 PM	03:35:00 PM		6.217		KM		

Figure 2.22 Result Set for the _Connection Association

By creating this simple CDS data model, you have already become familiar with some of the language elements of the CDS data definition language (DDL). Due to the large number of available statements and functions, we can provide only a first overview here. Details on the syntax, which is based on SQL, can be found, for example, in the ABAP keyword documentation at <http://s-prs.de/v868502>.

Due to the use of associations and annotations, you have also already become familiar with two central concepts of CDS data modeling. Because of their importance, we'll discuss these concepts in greater detail in the following sections. We'll also dedicate a separate section to the implementation of access controls in CDS in Section 2.5.

2.3 Associations

Relationships are often represented as foreign key links at the database level. You can map relationships between CDS views using associations. Associations in CDS are thus an important tool for data modeling in the ABAP RESTful application programming model and a precursor for modeling business objects in the behavior definition.

An association links a CDS view (as the source data) to the target data source specified in the association definition by means of an `On` condition. Besides CDS views, database tables or SQL views can also be used as a target data source.

Linking CDS views

Warning: Other Target Data Sources

Using target data sources other than CDS views may result in limited functionality when using the CDS view in applications. For this reason, it's recommended to use only CDS views as the target data source.



From a technical point of view, associations are similar to a JOIN operation. However, they don't merely link multiple data sources; they are used to clarify the relationships between entities, and they contain additional semantic properties, such as cardinality information. Basically, an association is a description of a possible connection between entities. It's the conceptual view of the data that is the focus here.

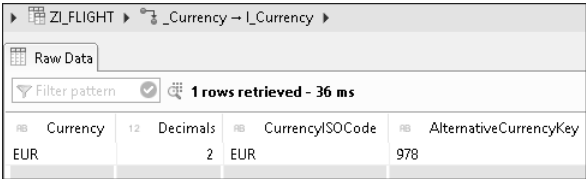
Join on demand A relationship represented by an association is therefore also referred to as a *functional relationship*. The association is necessary only if data from other entities is needed for a certain functionality. In this context, the term *join on demand* is often used.

Defining an association For clarity, we'll now extend the CDS view `ZI_Flight` in Listing 2.5 with an association to the standard CDS view `I_Currency`.

```
@AbapCatalog.sqlViewName: 'ZI_FLIGHTV'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Flight'
define view ZI_Flight
  as select from /dmo/flight
    association [0..1] to I_Currency as _Currency on
      $projection.CurrencyCode = _Currency.Currency
  {
    key carrier_id      as CarrierId,
    key connection_id   as ConnectionId,
    key flight_date     as FlightDate,
    price               as Price,
    currency_code       as CurrencyCode,
    plane_type_id       as PlaneTypeId,
    seats_max           as SeatsMax,
    seats_occupied      as SeatsOccupied,
    _Currency
  }
```

Listing 2.5 Association

By defining the association with `association [0..1] to I_Currency` and exposing it with `_Currency` as another element in the projection list, all components from the associated CDS view `I_Currency` are available for callers. The easiest way to test this again is to select the **Follow Association** context menu function in the data preview. Figure 2.23 shows the result.



Currency	Decimals	CurrencyISOCode	AlternativeCurrencyKey
EUR	2	EUR	978

Figure 2.23 Association in the Data Preview

At the database level, this exposure of the association alone doesn't define a join. You can check this by displaying the SQL statement `CREATE` via the context menu of the CDS editor. Select the **Show SQL CREATE Statement** entry here. The results are displayed in Figure 2.24. Alternatively, you can view the corresponding SQL view `ZI_FLIGHTV` in Transaction `SE11`. The actual join occurs only when the consumer accesses fields in the data source. Thus, no preselection of supposedly relevant fields of the data source is made during data modeling. This task is left to the user of the data model.

SQL statement
CREATE

```
CREATE OR REPLACE VIEW "ZI_FLIGHTV" AS SELECT
"/DMO/FLIGHT"."CLIENT" AS "MANDT",
"/DMO/FLIGHT"."CARRIER_ID" AS "CARRIERID",
"/DMO/FLIGHT"."CONNECTION_ID" AS "CONNECTIONID",
"/DMO/FLIGHT"."FLIGHT_DATE" AS "FLIGHTDATE",
"/DMO/FLIGHT"."PRICE" AS "PRICE",
"/DMO/FLIGHT"."CURRENCY_CODE" AS "CURRENCYCODE",
"/DMO/FLIGHT"."PLANE_TYPE_ID" AS "PLANETYPEID",
"/DMO/FLIGHT"."SEATS_MAX" AS "SEATSMAX",
"/DMO/FLIGHT"."SEATS_OCCUPIED" AS "SEATSOCCUPIED"
FROM "/DMO/FLIGHT" "/DMO/FLIGHT"
```

Figure 2.24 Display of the SQL Statement `CREATE` in the Popup Window with the Element Information

Access to associations of a CDS view via a higher-level CDS view built on top of it occurs in exactly the same way as access to the other components of that view. You can see an example in Listing 2.6. You can access individual components of the association or, if desired, re-expose the association under a different name, by assigning an appropriate alias.

Accessing
associations

```
define view ZI_FlightCurr as select from ZI_Flight
{
  key CarrierId,
  key ConnectionId,
  key FlightDate,
  CurrencyCode,
  _Currency.CurrencyISOCode,
  _Currency as _FlightCurrency
}
```

Listing 2.6 Using an Exposed Association in CDS

Accessing individual fields	Direct access to individual fields of an association (as is done in Listing 2.6, to the <code>CurrencyISOCode</code> field) is often referred to as an <i>ad-hoc association</i> . In this case, the activation of the CDS view creates a corresponding join at the database level, thus overriding the join on-demand principle.
Access from ABAP	It's also possible to access fields of exposed associations from ABAP SQL. This also applies, for example, to the use in a <code>WHERE</code> condition. <pre>SELECT Price, _Currency-CurrencyISOCode FROM zi_flight WHERE _Currency-Decimals = 0 INTO TABLE @DATA(currencies).</pre> Listing 2.7 Using an Exposed Association in ABAP SQL
Filter conditions	As an additional option, you can work with filter conditions when using associations in CDS-based data modeling, as in the following example: <pre>Currency[Currency = \$parameters.P_TargetCurrency] as _TargetCurrency</pre> This way you can further restrict the associated target records according to your requirements. When the path expression is converted to a join in the database, the filter becomes part of the <code>On</code> condition. Using filters in associations often improves readability and facilitates the interpretation of the data model. An additional <code>Where</code> clause would be less readable.
Cardinality	When defining associations, you can include the cardinality in square brackets. It's the cardinality of the target data source (that is, the possible number of related target records). This specification is optional. If you don't specify the cardinality, the default value <code>[0..1]</code> will be used. However, it's recommended to specify the cardinality as precisely as possible. It is primarily used to document the semantics of the data model and thus significantly improves the readability of your data model. Even if the cardinality isn't validated at runtime, it can lead to warnings and errors during syntax checking, as shown in the example in Figure 2.25. Incorrect cardinality specifications can also lead to incorrect data selections, such as duplicates or missing records in the result set.

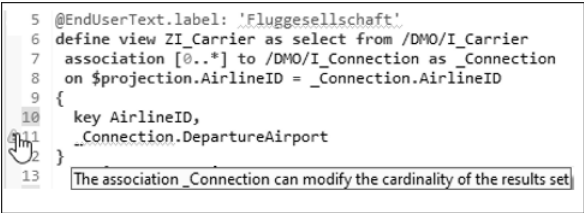


Figure 2.25 Syntax Warning About Cardinality

Just like specifying the cardinality, specifying a name for the association isn't mandatory. If a name isn't specified, the name of the target data source will be used implicitly. In any case, to ensure that the data model is readable, you should use the name assignment by means of the alias function. As a naming convention, SAP recommends an underscore (`_`) as the first character; this also helps distinguish associations from the other fields in the element list.

Alias for associations

Tip: Using Associations Instead of Joins
When defining a CDS data model, you should use associations rather than joins. If possible, you should also avoid the formation of implicit joins (ad-hoc associations). Instead, try to represent the relationships of the data using associations. In this way, you can provide the consumer with a fully comprehensive data image without having to generate an extensive join for it (and, thus, incurring performance losses).



2.4 Annotations

In the CDS views we've created over the course of this chapter, we've already used some CDS annotations. For example, in Listing 2.1, we specified the name of the SQL view created during activation by an annotation:

```
@AbapCatalog.sqlViewName: 'ZI_FLIGHTV'
```

Annotations enable you to add additional information (metadata) to your CDS data model. Annotations are therefore an essential part of data modeling. The information can be evaluated by the users of the data model. It can be used purely for documentation purposes, but also to activate certain functions.

According to their usage, the CDS annotations provided by SAP (*SAP annotations*) can be divided into

ABAP versus framework annotations

- ABAP annotations
- Framework-specific annotations

ABAP annotations are evaluated when the object defined in the CDS source code is activated or used. *Framework-specific annotations* are evaluated by the framework of another software component; for example, by the OData or analytics software components.



Warning: Use Only SAP Annotations

Currently, only annotations predefined by SAP are to be used in the CDS source codes. SAP partners and customers aren't yet allowed to define their own annotations.

From a technical point of view, CDS annotations (analogous to the CDS views that use them) are themselves CDS objects (object type DDLA) that are created and provided by an annotation definition.

Annotation definition

The definition of a CDS annotation is done using the following statement:

```
define annotation
```

The annotation definition specifies the unique name of the annotation and describes its technical properties:

```
Scope
Type
Allowed values (optional)
Default value (optional)
```

The structure of an annotation definition is illustrated in Listing 2.8. It contains an excerpt from the definition of the SAP annotation `Semantics`. The use of uppercase and lowercase letters is relevant here.

```
@Scope: [#ELEMENT, #PARAMETER]
define annotation Semantics
{
...

    eMail
    {
        type : array of String(10) enum
        {
            HOME;
            WORK;
            PREF;
            OTHER;
        }
    }
...
    systemDateTime
    {
        createdAt                : Boolean default true;
        lastChangedAt            : Boolean default true;
        localInstanceLastChangedAt : Boolean default true;
```

```
};
...
    @Scope:[#ELEMENT]
...
    currencyCode                : Boolean default true;

};
```

Listing 2.8 Sample Annotation Definition (Excerpt)

`Semantics` is the main annotation (*domain*) of this annotation definition. It's further structured by subannotations, such as `eMail` or `systemDateTime`. This results in a hierarchical structure. The domains are used to group thematically related subannotations. In Table 2.2, some domains and their field of application are listed (not complete).

Domain and subannotations

Domain	Area of Use
Analytics	Generation of evaluations using analytic manager, a system component that evaluates analytical annotations
OData	Generation of an OData service from the data obtained from a CDS entity.
EndUserText	Definition of translatable texts
UI	Display of data in user interfaces
VDM	Classification of CDS views for the purposes of structuring and interpretation
Semantics	Provision of information on the meaning and use of individual elements of a CDS View
Search	Marking CDS entities as searchable and defining search properties
ABAPCatalog	Determination of technical settings of the CDS entity
AccessControl	Access control management
ObjectModel	Definitions of structural and transactional aspects of the data model

Table 2.2 Important Domains for Annotations

Tip: Overview of All Annotations

`ABAP_DOCU_CDS_ANNOS_OVERVIEW` program lists all SAP annotations and their properties.



Scope of an annotation The specification `@Scope: [#ELEMENT, #PARAMETER]` enables you to define the *scope* of an annotation. It determines at which positions of a CDS source code the annotation may be specified. The specification of a scope is mandatory. Basically, a distinction is made between annotations with scope at the level of the entire CDS view and annotations that apply only to individual elements of a view.

If a scope can't be determined for an annotation or subannotation, a syntax error will occur. If a scope isn't specified in subannotations, those subannotations will adopt the scope of the parent annotation. So, in Listing 2.8, for the `currencyCode` subannotation, `@Scope:[#ELEMENT]` specifies a scope that differs from the `Semantics` annotation: `currencyCode` isn't permitted for use with parameters. Basically, the top level (CDS view, CDS view extension, CDS table function, CDS role, or CDS annotation definition) or its sublevels (elements, parameters, or associations) can be specified as the scope of annotations.



Example: Scope of the Scope Annotation

The `Scope` annotation is an *annotation definition annotation* (scope `#ANNOTATION`) that contains information about the annotation itself. More simply, it can also be called *meta-annotation*.

Structure of annotations

Annotations are typed either as single values, structures, or lists. For example, Listing 2.8 shows that you can specify the type of an email address as a list while the annotation `currencyCode` can be used to mark an element with a Boolean scalar value as a currency field.

Allowed values (corresponding to the typing) can be specified in the annotation definition. It's also possible to set a default value. This value is used when an annotation is specified without explicitly assigning a value to it as well. It doesn't mean that the default value applies if the annotation isn't used in the CDS data model.

The way to specify a CDS annotation in a CDS source code in detail is described by the *CDS annotation syntax*. That syntax is fixed and is supplemented by the CDS annotation definition with rules for using an annotation.

The specification of a CDS annotation for an element within the CDS source code always starts with an introductory `@`. This is followed by the specification of the domain (main annotation) and, separated by a dot in each case, the specification of the subannotations. The value of the annotation is then specified after a colon:

```
@Semantics.currencyCode: true
currency_code as CurrencyCode,
```

`Semantics` is the domain here, `currencyCode` is the subannotation, and `true` is the value.



Annotation After an Element to be Marked

Specifying annotations *after* the element to which the annotations are to be applied in the CDS source code is also possible, but it then starts with the characters `@<`:

```
currency_code as CurrencyCode
@<Semantics.currencyCode: true,
```

For readability, it's best to use the prefixed notation throughout.



Tip: Help Function in ABAP Development Tools

When specifying annotations, ADT supports you with source code coloring and code completion. To call the help for an ABAP annotation, you should position the cursor in the DDL source editor on the corresponding annotation and press the `F1` key.

When an object that is defined in CDS source code is activated, the annotations specified there with the CDS annotation syntax are stored in database tables of the ABAP dictionary, which can then be accessed for evaluation. The `DDHEADANNO` table contains the annotations that apply at the upper level (e.g., at the CDS view level). The tables `DDFIELDANNO` and `DDPARAMETERANNO` contain the annotations of the individual elements and parameters of a CDS view, respectively.

The class `CL_DD_DDL_ANNOTATION_SERVICE` is available for the evaluation of these database tables. When evaluating, it's essential to note that the actual value of an annotation isn't necessarily identical to the value specified in the CDS source code for the annotation. Possible sources of annotation values are:

- **Direct annotations**
Annotations specified directly in the DDL source code of the CDS entity.
- **Annotations from metadata extensions**
Metadata extensions add further annotations to a CDS entity or override existing annotations.
- **Indirect, inherited annotations**
When accessing other CDS entities, their direct and indirect annotations and the annotations from metadata extensions are inherited. This applies exclusively to annotations at the element level.

Saving in the ABAP Data Dictionary

■ Indirect, derived annotations

Annotations of the EndUserText domain are derived from the field identifiers of associated data elements.

Passing on annotations

The hierarchical structure of CDS data models is reflected in the determination of annotation values: Higher-level CDS views adopt annotations of lower-level CDS views. Annotations defined locally in a CDS view override these inherited annotations. These can be overwritten by annotations from metadata extensions. The result thus obtained is referred to as the *active annotations* of a CDS view.



Tip: Preventing the Annotations from Being Passed On

You can prevent annotations that are propagated in this way from being considered for a CDS view by specifying the annotation @Metadata.ignorePropagatedAnnotations: true with the scope ENTITY.

If you right-click on the source code in ADT and select **Open With • Active Annotations** in the context menu, the active annotations of the selected CDS entity will be displayed (see Figure 2.26).

Active Annotations for Entity ZI_Flight				
type filter text				
Annotated Elements	Annotation Value	Translated Text	Origin Data Source	Origin Data Element
▼ ZI_Flight				
▼ Entity annotations				
▼ @AbapCatalog				
sqlViewName	'ZI_FLIGHTV'			
▼ compiler				
compareFilter	true			
preserveKey	true			
▼ @AccessControl				
authorizationCheck	#CHECK			
▼ @EndUserText				
label	'Flug'			
▼ CarrierId				
▼ @EndUserText				
quickInfo			/DMO/FLIGHT (Database Table)	/DMO/CARRIER_ID
label			/DMO/FLIGHT (Database Table)	/DMO/CARRIER_ID
heading			/DMO/FLIGHT (Database Table)	/DMO/CARRIER_ID
> ConnectionId				
> FlightDate				
> Price				
▼ CurrencyCode				
▼ @Semantics				
currencyCode	true		ZI_Flight (CDS View)	
▼ @EndUserText				
quickInfo		Flight Reference S...	/DMO/FLIGHT (Database Table)	/DMO/CURRENCY_CODE
label		Currency Code	/DMO/FLIGHT (Database Table)	/DMO/CURRENCY_CODE
heading		Currency Code	/DMO/FLIGHT (Database Table)	/DMO/CURRENCY_CODE
▼ PlaneTypeId				
▼ @UI				
hidden	true		ZI_Flight (CDS View)	

Figure 2.26 Display of Active Annotations

We'll describe how to proceed when creating a metadata extension in Section 2.6.

2.5 Access Controls

Access controls ensure that only authorized persons are granted access to protected data.

Corresponding access restrictions can take place at different levels:

Access control levels

■ Application level

A user doesn't have the authorization to perform a specific function (e.g., a transaction).

■ Record level

Despite the authorization for a function, access to a subset of the data records is restricted.

■ Field level

A user is only granted access to individual elements of the authorized data records.

Protection against unauthorized data access is implemented in ABAP CDS by an authorization concept, which is based on a data control language (DCL). The `define role` statement allows you to define CDS roles for CDS entities. These roles contain access conditions that allow you to restrict the result of the data selections. These access conditions are implicitly evaluated each time the CDS entity for which the CDS role was defined is *directly* accessed (using ABAP SQL, service adaptation description language [SADL] query, or entity manipulation language [EML]). However, if this CDS entity only forms the data source for CDS views of a higher hierarchy level, the access control of the CDS entity remains without effect. Therefore, when modeling, you must be careful to create a separate CDS role for each CDS entity for which protection is required.

CDS roles

Warning: No Access Control for SQL Views

When you access the generated SQL view, the CDS access conditions won't be evaluated. For this reason, you should always use the CDS view in the ABAP SQL statements instead of the generated SQL view.



To create access control for a CDS view, you must select a package in the ADT project explorer and choose the path **New • Other ABAP Repository Object • Core Data Services • Access Control** from the context menu. Concerning the name of the CDS role, it's best to choose the same name as for the CDS view to which the access control is to apply. As usual, you'll be supported by templates (see Figure 2.27).

Creating access control

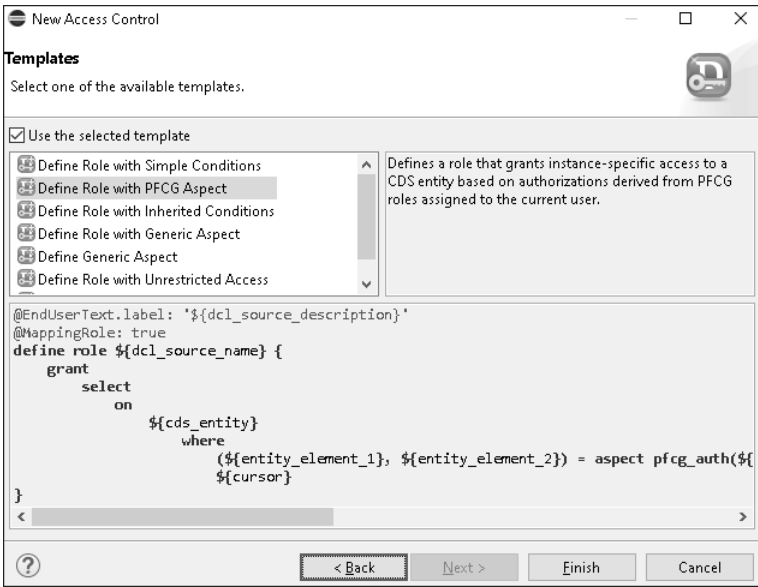


Figure 2.27 Templates Available When Creating a CDS Role

Characteristics of the CDS role

Once you’ve created the role using the wizard, you must define the characteristic of the CDS role in the source code editor. Listing 2.9 shows this using the example of the ZI_Flight role. When the CDS role is activated, a transportable ABAP development object of type DCLS (data control language source) is created.

```
@EndUserText.label: 'CDS role for ZI_Flight'
@MappingRole: true
define role ZI_Flight {
  grant select on ZI_Flight
  where ( CarrierId ) = aspect pfcg_auth( Z_DMO_CAR,
                                         CARRIER_ID,
                                         ACTVT = '03' )

  and ConnectionId like '04%';
}
```

Listing 2.9 CDS Role for CDS View ZI_Flight

User-dependent and user-independent check

In Listing 2.9, a user-dependent check for the CDS view ZI_Flight is combined with a user-independent check. For the user-dependent check, the classic SAP authorization concept is used, which is based on authorization objects and their assignment to authorization roles in Transaction PFCG. Authorizations are assigned to the user via these authorization roles. In addition, user-independent checks are made against the value of the ConnectionID element (linked by the logical AND operator).

A user who is assigned the PFCG authorization role Z_DMO_CARRIER from Figure 2.28 would, for example, receive the result from Figure 2.29 as the result of selecting the CDS view ZI_Flight (without further restriction of the selection conditions).

Role Z_DMO_CARRIER				
Maint. 0 unmain. org. levels, 0 open fields				
Status: Unchanged				
■ Status Edit ▾ ⏏ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ 🔍 Values				
Group/Object/Authorization/Field	Maintenance Sta...	A...	Value	Text
Object Class BC_A	Manual			Basis: Administration
Authorization Object Z_DMO_CAR	Manual			Z_DMO_CAR
Authorization T-SA81004200	Manual			Z_DMO_CAR
CARRIER_ID	Manual	66	AA, JL, LH, UA	Flight Reference Scenario: Carrier ID
ACTVT	Manual	66	Add or Create, Change, Display	Activity

Figure 2.28 Classic PFCG Authorization Role Z_DMO_CARRIER

ZI_FLIGHT									
Raw Data									
Filter pattern 12 rows retrieved - 7 ms SQL Console # Number of Entries Select Columns Add filter									
CarrierId	ConnectionId	FlightDate	Price	CurrencyCode	PlaneTypeId	SeatsMax	SeatsOccupied		
LH	0400	2021-10-30	5484.00	EUR	A340-600	330	330		
LH	0400	2021-01-03	2649.00	EUR	767-200	260	130		
LH	0401	2021-10-29	3697.00	EUR	747-400	385	265		
LH	0401	2021-01-02	4867.00	EUR	A380-800	475	403		
LH	0402	2021-10-25	4911.00	EUR	767-200	260	221		
LH	0402	2020-12-29	3232.00	EUR	747-400	385	231		
LH	0403	2021-10-25	2797.00	EUR	A340-600	330	171		
LH	0403	2020-12-29	2486.00	EUR	767-200	260	117		
JL	0407	2021-10-29	5346.00	JPY	747-400	385	254		
JL	0407	2021-01-02	4032.00	JPY	A340-600	330	165		
JL	0408	2021-10-30	8159.00	JPY	A380-800	475	432		
JL	0408	2021-01-03	6471.00	JPY	747-400	385	296		

Figure 2.29 Selection Result After Querying CDS View ZI_Flight

By implicitly evaluating the access condition when reading records from the database, the SQL statement is augmented with the WHERE condition derived from the access conditions, as you can see in the SQL trace of Transaction SACMSEL (the CDS access control runtime simulator) (see Figure 2.30).

CDS access control runtime simulator

```
SQL-Trace
SELECT
/* CDS access control applied */
"MANDT" , "CARRIERID" , "CONNECTIONID" , "FLIGHTDATE" , "PRICE" , "CURRENCYCODE" ,
"PLANETYPEID" , "SEATSMAX" , "SEATSOCCUPIED"
FROM
/* Entity name: ZI_FLIGHT CDS access controlled */ "ZI_FLIGHTV" "ZI_FLIGHT"
WHERE
"MANDT" = '000' AND "CARRIERID" IN ( 'AA' , 'JL' , 'LH' , 'UA' ) AND "CONNECTIONID"
LIKE '04%' LIMIT 200
```

Figure 2.30 SQL Statement When Reading Records from the ZI_Flight View with CDS Role ZI_Flight

In contrast, when indirectly accessing CDS view ZI_Flight, for example, via CDS view ZI_FlightDetail from Listing 2.4, which is based on it, the access conditions defined in CDS role ZI_Flight wouldn't be evaluated. Transaction SACMSEL allows you to test your CDS roles in detail. In addition to the executed SQL statement, you can see the result of the data selection and information about the underlying PFCG authorization roles.

Authorization logic
at database level

Unlike when you perform authorization checks in the classic ABAP programming model (where you first read the records to be checked from the database to the application server so that you can check them there at the individual record level using the `AUTHORITY-CHECK` statement), here you move the authorization logic to the database level by using CDS roles. This approach allows you to achieve significant performance gains.

AccessControl.
authorizationCheck

You can store information about documenting and controlling access using CDS roles in a CDS entity with the annotation `AccessControl.authorizationCheck`. In the following list, we'll describe the possible expressions of the annotation values:

- **#CHECK**
Access control is to be performed for the CDS entity via a CDS role (default value of the annotation). If no CDS role exists for the CDS entity, the syntax check generates a warning message.
- **#NOT_REQUIRED**
Access control isn't strictly required for the CDS entity. However, access control is performed if a CDS role is available. The syntax check warning message about a missing CDS role is omitted.
- **#NOT_ALLOWED**
There's no access control, even if a CDS role does exist.
- **#PRIVILEGED_ONLY**
This value is used in CDS entities for which direct access to the data should generally not be allowed (implemented by a condition in the CDS role that's never met). Access control for direct access to the CDS entity can be bypassed by specifying a special ABAP SQL statement (`WITH PRIVILEGED ACCESS`), but is otherwise performed.

To illustrate the use of the `#PRIVILEGED_ONLY` annotation value, let's look at the corresponding specification in the standard SAP CDS view `I_USER`:

```
@AccessControl.authorizationCheck: #PRIVILEGED_ONLY
define view I_User
  as select from usr21
```

The CDS role `I_USER` with the same name exists for this view (see Listing 2.10).

```
@MappingRole: true
//Deny ALL direct access to I_USER. Only privileged access allowed.
define role I_User {
  grant select on I_USER
  where UserID is null and UserID is not null;
}
```

Listing 2.10 CDS Role `I_USER` (Excerpt)

Because the condition `where UserID is null and UserID is not null` in the `Where` clause of the CDS role is never met, no records are selected in a direct call. A corresponding message is displayed in the ADT data preview (see Figure 2.31).

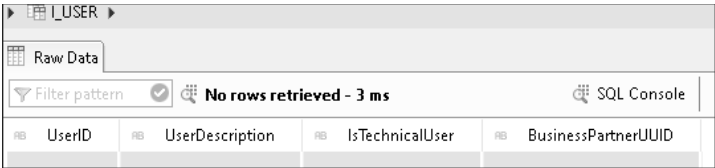


Figure 2.31 Data Preview for CDS View `I_USER`

In ABAP, this access control can be bypassed by adding `WITH PRIVILEGED ACCESS` (see the debugger excerpt in Figure 2.32). Access control is not executed and the records are selected from CDS view `I_USER` according to the selection in ABAP SQL.

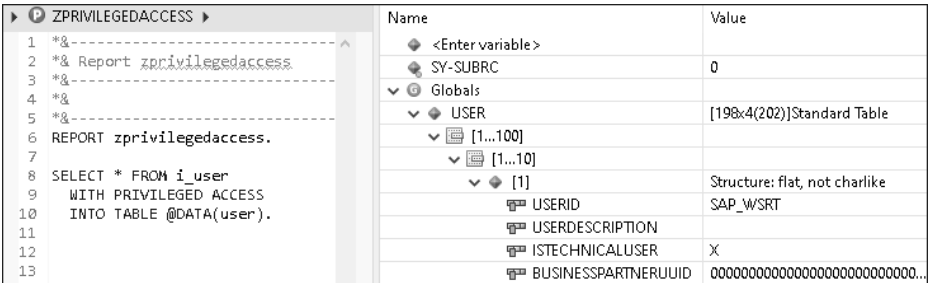


Figure 2.32 Debugger for the `WITH PRIVILEGED ACCESS` Statement

It's also possible to create multiple CDS roles for one CDS entity. These are then linked by an `OR` condition. The result of the selection can be extended by additional CDS roles, but not restricted. For clarity, however, you should create only one CDS role per CDS entity.

Multiple CDS roles

Contents

Foreword	15
Preface	17

PART I Basic Concepts and Technical Components

1	Introduction	23
1.1	What is the ABAP RESTful Application Programming Model? ...	24
1.1.1	The Purpose of the Programming Model	24
1.1.2	The REST Architectural Style	29
1.1.3	OData	34
1.1.4	Technological Innovations with SAP S/4HANA	36
1.1.5	Evolution of ABAP-Based Programming Models	38
1.2	Architecture and Concepts	42
1.2.1	Transaction Model	42
1.2.2	Implementation Types	43
1.2.3	Entity Manipulation Language	45
1.2.4	Technical Context of Applications and Runtime Environment	45
1.3	Development Objects	48
1.3.1	Data Modeling with Core Data Services	48
1.3.2	Behavior Definition	49
1.3.3	Behavior Implementation	50
1.3.4	Projection Layer	51
1.3.5	Business Services	51
1.3.6	Interaction of the Artifacts	52
1.4	ABAP Development Tools	53
1.5	Quality Attributes of the ABAP RESTful Application Programming Model	54
1.5.1	Evolution Capability	54
1.5.2	Development Efficiency	55
1.5.3	Testability	57
1.5.4	Separation between Business and Technology	57

1.6	Availability of the ABAP RESTful Application Programming Model	58
1.6.1	SAP BTP, ABAP Environment	58
1.6.2	ABAP Platform for SAP S/4HANA On-Premise	59
2	Core Data Services: Data Modeling	61
2.1	What are Core Data Services?	62
2.2	Structure and Syntax of Core Data Services	65
2.2.1	Creating a Basic Interface View	66
2.2.2	Analyzing the Data Model	70
2.2.3	Using CDS Views	74
2.2.4	Extending the Data Model	74
2.3	Associations	79
2.4	Annotations	83
2.5	Access Controls	89
2.6	Extensibility of CDS Entities	94
2.6.1	CDS View Extensions	94
2.6.2	CDS Metadata Extension	98
2.7	Additional CDS Functionality	101
2.7.1	Virtual Elements	101
2.7.2	CDS Custom Entities	105
2.8	Virtual Data Model	109
2.9	CDS Language Elements for Modeling Business Objects	113
3	Behavior Definition	117
3.1	What is a Behavior Definition?	117
3.1.1	Context and Structure of a Behavior Definition	118
3.1.2	Syntax of a Behavior Definition	121
3.1.3	Possible Behavior	122
3.2	Editing a Behavior Definition in ABAP Development Tools	129
3.2.1	Creating a Behavior Definition	129
3.2.2	Changing and Activating a Behavior Definition	132
3.2.3	Finding and Opening a Behavior Definition	133
3.2.4	Documenting Behavior Definitions and Relationships	134

3.3	Implementation Types	136
3.3.1	Managed Scenario	138
3.3.2	Unmanaged Scenario	140
3.4	Strict Mode	141
3.5	Entity Behavior Definition	142
3.6	Defining a Behavior Pool	143
3.6.1	Behavior Pool for Behavior Definition	143
3.6.2	Behavior Pool for the CDS Entity	144
3.6.3	Behavior Pool for the Implementation Group	144
3.7	Numbering	146
3.7.1	Early, External Numbering	147
3.7.2	Early, Internal Numbering	147
3.7.3	Late Numbering	149
3.8	Field Properties	150
3.8.1	Mandatory Fields	150
3.8.2	Protection Against Write Access	151
3.8.3	Combination: Mandatory Field in Case of Creation, Write Protection in Case of Updates	152
3.9	Field Mappings	153
3.10	Standard Operations for a CDS Entity	155
3.10.1	Create, Read, Update, and Delete	155
3.10.2	Create and Read Operation by Association	156
3.11	Specific Operations for a CDS Entity	159
3.11.1	Actions	159
3.11.2	Functions	166
3.12	Concurrency and Locking Behavior	168
3.12.1	Pessimistic Locking	169
3.12.2	Optimistic Locking	171
3.13	Internal Business Logic	173
3.13.1	Determinations	173
3.13.2	Validations	179
3.13.3	Calling Determinations via an Action	182
3.14	Authorization Checks	184
3.14.1	Authorization Master	185
3.14.2	Authorization-Dependent	187
3.14.3	Delegating Authorization Checks	188
3.15	Draft Handling	189
3.15.1	Enabling Draft Handling	190

3.15.2	Draft Handling in the Business Object Composition Tree	191
3.15.3	Draft Lifecycle and Draft Actions	193
3.16	Overarching Concepts	196
3.16.1	Dynamic Feature Control	196
3.16.2	Preliminary Checks of Operations	200
3.16.3	Internal Visibility of Operations	201
4	Entity Manipulation Language: Accessing Business Logic	205
4.1	Data Types	206
4.1.1	Derived Data Types	206
4.1.2	Implicit Return Parameters	209
4.2	EML Operations	210
4.2.1	READ ENTITIES	210
4.2.2	MODIFY ENTITIES	213
4.2.3	GET PERMISSIONS	216
4.2.4	SET LOCKS	217
4.2.5	COMMIT ENTITIES	218
4.2.6	ROLLBACK ENTITIES	219
4.3	Using EML Outside of Behavioral Implementations	220
4.3.1	Use in the Context of an ABAP Report	220
4.3.2	Implementation in the Context of a Test Class	222
5	Behavior Implementation	225
5.1	Business Object Provider API	225
5.2	Runtime Behavior of the ABAP RESTful Application Programming Model	226
5.2.1	Interaction Phase and Transaction Buffer	227
5.2.2	Save Sequence	228
5.3	Interfaces for the Interaction Handler and the Save Handler	229
5.4	Interaction Handler	230
5.4.1	FOR MODIFY	231
5.4.2	FOR INSTANCE AUTHORIZATION	234

5.4.3	FOR GLOBAL AUTHORIZATION	237
5.4.4	FOR FEATURES	239
5.4.5	FOR GLOBAL FEATURES	242
5.4.6	FOR LOCK	243
5.4.7	FOR READ	245
5.4.8	FOR READ BY ASSOCIATION	246
5.4.9	FOR DETERMINE	249
5.4.10	FOR VALIDATE	250
5.4.11	FOR NUMBERING	250
5.4.12	FOR PRECHECK	252
5.5	Save Handler	253
5.5.1	FINALIZE	254
5.5.2	CHECK_BEFORE_SAVE	256
5.5.3	ADJUST_NUMBERS	257
5.5.4	SAVE	258
5.5.5	CLEANUP	260
5.5.6	CLEANUP_FINALIZE	262
6	Business Services	263
6.1	Projection Layer	264
6.1.1	CDS Projection View	265
6.1.2	Projection Behavior Definition	266
6.2	Service Definition	267
6.3	Service Binding	268
6.4	Testing Business Services in the SAP Gateway Client	272
6.5	Testing UI Services with SAP Fiori Elements Preview	275
7	User Interfaces and SAP Fiori Elements	277
7.1	Development Tools	277
7.1.1	SAP Business Application Studio	278
7.1.2	Visual Studio Code	280
7.2	SAP Fiori Elements	281
7.2.1	Floorplans in SAP Fiori Elements	281
7.2.2	Selected UI Annotations	283

7.2.3	Defining UI Annotations in a CDS View	285
7.2.4	Generating Annotations via the Service Modeler	302

PART II Practical Application Development

8	Use Cases	315
8.1	Applications Types	315
8.2	Implementation Types	316
8.3	Selecting the Appropriate Implementation Type	318
9	Managed Scenario: Developing an Application with SAP Fiori Elements	321
9.1	Description of the Use Case	321
9.2	Building the Data Model	322
9.2.1	Database Tables	322
9.2.2	CDS Modeling	326
9.3	Creating Behavior Definitions	334
9.3.1	Creating Behavior Definitions for Certificate Management	334
9.3.2	Enabling Draft Handling	339
9.4	Defining a Business Service	340
9.4.1	Creating a Service Definition	341
9.4.2	Creating the Service Binding	342
9.5	Creating an SAP Fiori Elements User Interface	344
9.6	Enrichment with a Determination	352
9.7	Enrichment with a Validation	356
9.8	Enrichment with an Action	360
9.9	Generation and Deployment of the Application	362

10	Managed Scenario with Unmanaged Save: Integrating an Existing Application	371
10.1	Description of the Use Case	371
10.2	Building the Data Model	375
10.2.1	Overview of the Logical Data Model	375
10.2.2	Database Tables	377
10.2.3	CDS Modeling	379
10.3	Creating a Behavior Definition	385
10.4	Implementing the Create Purchase Order Function	387
10.4.1	Declaring Late Numbering	387
10.4.2	Setting Field Properties	388
10.4.3	Creating the Behavior Pool	390
10.4.4	Implementing Determinations	391
10.4.5	Save Sequence: Implementing the Creation via BAPI	398
10.4.6	Implementing Validations	403
10.5	Implementing the Delete Purchase Order Function	409
10.5.1	Save Sequence: Implementing the Deletion via BAPI	409
10.5.2	Implementing a Validation	414
10.6	Defining Business Services	416
10.6.1	Setting up the Projection Layer for the My Purchase Orders Application	416
10.6.2	Creating a Service Definition	418
10.6.3	Creating a Service Binding	419
10.7	Implementing Authorization Checks	419
10.7.1	Access Controls for Read Access	419
10.7.2	Access Controls for Write Access	421
10.8	Creating an SAP Fiori Elements User Interface	424
10.8.1	Creating a Metadata Extension	424
10.8.2	Generating and Deploying the Application	427
11	Unmanaged Scenario: Reusing Existing Source Code	429
11.1	Description of the Use Case	430
11.2	Description of the Existing Application	431
11.2.1	Database Tables	431

11.2.2	Source Code of the Existing Application	434
11.3	Extending the Data Model	437
11.4	Creating a Behavior Definition	443
11.5	Creating a Behavior Implementation	447
11.5.1	Implementing the Interaction Phase	450
11.5.2	Implementing the Save Sequence	458
11.6	Defining a Business Service	462

12 Specific Features for the SAP BTP,
ABAP Environment465

12.1	Technical Fundamentals	466
12.1.1	ABAP for Cloud Development	469
12.1.2	Technical Infrastructure Components	470
12.1.3	Migrating Legacy Code	472
12.2	Identity and Access Management	473
12.3	Deploying SAP Fiori Apps and Assigning Authorizations	476
12.3.1	Creating an IAM App and Business Catalog	477
12.3.2	Creating an IAM Business Role	479
12.3.3	Integration in SAP Fiori Launchpad	480
12.4	Consuming Business Services	484

13 Outlook491

13.1	Build	492
13.2	Extensibility	492
13.3	Integration and Reusability	494

Appendices497

A	Bibliography	497
B	Authors	499
	Index	501

Index

.env file	366	Access control	76, 89, 338, 419, 420
/IWFND/GW_CLIENT	272	Access, concurrent	168
/IWFND/MAINT_SERVICE	104, 272	Action	159
%CID	207, 453	<i>creating</i>	360
%CONTROL	207, 446	<i>defining</i>	298
%DATA	208	<i>executing</i>	216
%ELEMENT	208	<i>factory</i>	164
%FAIL	208	<i>function import</i>	274
%ISDRAFT	208	<i>input parameter</i>	160
%MSG	208	<i>instance-based</i>	159
%PARAM	208	<i>navigation</i>	476
%PID	207	<i>static</i>	159, 200
%TKY	207	Additional Save	140, 317
A		Ad-hoc association	82
<hr/>		ADJUST_NUMBERS	398
ABAP		ADT → ABAP development tools	
<i>classic application development</i>	38	Alias	142
<i>for cloud development</i>	469	Annotation	35, 64, 83, 283
<i>package</i>	323	ABAP	83
<i>programming models</i>	38	<i>AbapCatalog</i>	96
<i>report</i>	220	<i>AccessControl</i>	76, 92
ABAP annotation	83	<i>active</i>	88
ABAP development tools	53, 66, 129	<i>database tables</i>	87
ABAP Dictionary	39	<i>framework-specific</i>	83
ABAP Environment → SAP BTP,		<i>grouping</i>	289
ABAP Environment		<i>metadata extension</i>	98
ABAP Flight Reference Scenario	34	<i>ObjectModel</i>	102
ABAP PaaS → SAP BTP, ABAP Environment		<i>scope</i>	86
ABAP programming model for		<i>Semantics</i>	76
SAP Fiori	41	<i>syntax</i>	86
ABAP Repository	26	API	25
ABAP RESTful application programming		<i>direct call</i>	25
model	23	<i>framework-based</i>	25
API	493	<i>legacy application</i>	434
<i>availability</i>	58	<i>local</i>	316
<i>development objects</i>	48	<i>reusing</i>	434
<i>extensibility</i>	492	Application generator	363
<i>further development</i>	492	Application programming interface → API	
<i>history</i>	38	as projection on	115
<i>properties</i>	25	Association	64, 77, 79, 328
<i>quality features</i>	54	<i>accessing</i>	81
<i>request</i>	46	<i>cardinality</i>	82
<i>runtime behavior</i>	226	<i>draftable</i>	193
<i>transaction model</i>	42	<i>name</i>	83
<i>use cases</i>	315	<i>standard operation</i>	156
ABAP Unit	57, 222	Authorization check	184, 234, 419
abapGit	468	<i>delegating</i>	188
		<i>excluding</i>	189

Authorization check (Cont.)

global

instance-based

184, 237, 423

185, 421

Authorization concept

89

Authorization dependent

187

Authorization handler

237

Authorization master

185, 421

Authorization object

473

Authorization trace

422

Auto-completion

132

B

BAPI

373, 399

Basic interface view

66, 110

creating

326, 438

BDL

49

Before-image

234

Behavior

49

transactional

123

Behavior definition

49, 115, 117, 385

actions

299

activating

133

changing

132

creating

129

syntax

121

Behavior definition language

49, 118

syntax

121

Behavior implementation

50, 225, 230, 352, 390

Behavior Interface

494

Behavior pool

143, 225

creating

390, 448

Behavior projection

51

Binding type

268, 269, 343

BO consumer

45

BO framework

47

BO provider

43

BO runtime

225

BOPF

40, 495

BOPF managed

317

Brownfield approach

25, 318, 429

BSP

477

BSP application

369

Built-in function

77

Business application, architecture

26

Business logic, internal

125, 173

Business object

40, 48, 58

composition tree

120

field

125

interface

127

Business Object Framework

47

Business Object Processing Framework → BOPF

50, 225

Business Object Provider API

477

Business Server Pages

51, 115, 263

Business service

484

consuming

340, 416

defining

272

test

270

transporting

269

versioning

468

Business user

277

C

Calendar

296

Camel-case notation

74

Cardinality

82, 162

CDS

37, 61, 62

CDS behavior definition → Behavior definition

319, 487

CDS custom entity

161

CDS entity

105

abstract

94

custom

156

extending

156

instances

156

CDS metadata extension → metadata extension

51, 115

CDS projection view

330, 416, 439

creating

105

virtual element

89

CDS role

49

CDS root entity

285

CDS view

66

annotations

62

creating

110

SAP HANA

113

types

231

CDS view entity

228, 256

Change operation

229

CHECK_BEFORE_SAVE

229

CL_ABAP_BEHAVIOR_HANDLER

229

CL_ABAP_BEHAVIOR_SAVER

229

CL_ABAP_BEHV

260

CLEANUP

262

CLEANUP_FINALIZE

486

Client API

30

Client/server architecture

467

Cloud Connector

28

Cloud readiness

132

Code completion

Code pushdown

28, 37, 62

Code-to-data paradigm

122

Comment

218, 228

COMMIT ENTITIES

476, 485

Communication agreement

475, 485, 488

Communication scenario

30

Communication, stateless

111

Composite interface view

114, 214, 328

Composition

114

composition [] of

120

Composition tree

39, 393

Constructor expression

155

Consumer

51, 111

Consumption view

282

Content area

207

Content ID

30

Content type

40

Control flow

446

Control structure

219, 228

CONVERT KEY

61

Core Data Services R CDS

156

create

213

CREATE FIELDS

157, 214

Create-by-association

155

CRUD operation

98

Custom Field and Logic (App)

487

Custom query

297

Determination

173, 249

action

182

creating

352

declaring

391

implementing

392

infinite loop

177

method

249

side effects

302

times

174

determination

174

determine action

182

Dev space

278

Development efficiency

27, 55

Development flow

26

Development object

24

Displaying a button

300

Domain

85

Domain-specific language

27

Draft action

193, 195

Draft handling

26, 43, 189

enabling

190, 339

Draft instance

190

Draft lifecycle

193

Draft table

191, 339

DSL → Domain-specific language

126, 196

dynamic feature control

282

Dynamic page header

39

Dynpro

147

Early numbering

278

Eclipse Theia

101, 266, 332

Element, virtual

41

Embedded deployment

493

Embedded Steampunk

45, 47, 205, 206

EML

210

operations

169

Enqueue server

34, 48

Entity

120, 142

Entity behavior definition

35

Entity Manipulation Language → EML

Entity set

Entity tag → ETag

Error message

300, 403

ETag

dependent

172

field

192

handling

192

master

171

Data consistency

168

Data Control Language

89

Data Definition Language

27

Data model

34

extending

74

logical

375

Data source

69

Data type

39

derived

206, 446

Database table

322, 378

Date field

296

DDL

27

Default Authorization Value

474

Default filter

287

define behavior for

115, 120, 142

define service

115

define view

113

define view entity

113

delete

156, 409

DELETE (HTTP)

34

Deployment

362

Destination

484

Evolution capability 54

Exception class 358

Extension 281

Extension include view 96

F

Facet 290, 346

Factory action 164, 203

FAILED 209

Feature 122

Feature control 239

dynamic 196

global 196

instance-based 196

static 197

Field

business object 125

extension 97

mapping 126, 153, 401

name 112

property 150, 388

virtual 102

Filter bar 286

Filter condition 286, 417

FINALIZE 228, 254

Flight data model, new 34

Floorplan 281

Follow-up screen 297

Footer toolbar 282

FOR CREATE 452

FOR DETERMINE 249

FOR FEATURES 239

FOR GLOBAL FEATURES 242

FOR INSTANCE AUTHORIZATION 234

FOR LOCK 243, 409

FOR MODIFY 231

FOR NUMBERING 250

FOR PRECHECK 252, 423

FOR READ 245, 450

FOR READ by association 246

FOR UPDATE 454

FOR VALIDATE 250

Framework-specific annotation 83

Function 166

Function group 434

Function import 274, 299

Function Module 101

Function module 430

Function, built-in 77

Functional relationship 80

G

GET 33

GET PERMISSIONS 216

Git client 278

Global feature control 242

Greenfield approach 25, 318

Guided development 312

H

Handler class 448

HATEOS 32

Header toolbar 282

Help view 381

HTTP 29, 31

client 29

endpoint 35

method 29

request 29

response 29

server 29

service 468

Hub deployment 41

Hyperlink 30

Hypertext Transfer Protocol → HTTP

I

IAM → Identity and Access Management

Identity and Access Management 473

app 474, 477

business catalog 475, 478

business role 473, 475, 479

Implementation class 352, 448

Implementation group 144

implementation in class 143

Implementation type 43, 49, 120, 136, 316

abstract 141

BOPF managed 317

choosing 386

managed 317

selecting 318

unmanaged 429

IN SIMULATION MODE 219

InA 269

Inbound navigation 476

Infinite loop 177

Information Access 269

Infrastructure component 470

inheriting conditions from 420

Inner join 382

Input help 295, 440

Input parameter

action 160

data type 160

Instance

active 190

draft 190

Integration 494

Interaction handler 228, 230, 392, 450

Interaction phase 42, 227, 386, 450

Interface view layer 111

internal 201

J

Join 81

Join on Demand 80

K

Key field 385

Key user extension 98

Key value

permanent 146

temporary 146

Key, semantic 228

Knowledge Transfer Document 134

KTD 134

L

Late numbering 149, 229, 388

Launchpad page 480

Launchpad space 480

Layering, systems 31

Legacy code 430

Link 30

List report 276, 281

annotations 285

creating 304, 363

Local API 316

Lock

exclusive 192

explicit 217

optimistic 171

pessimistic 169

Lock dependency 243

Lock master 169, 243, 409

Lock object 169

Lock table 169

Lock-dependent 170

Locking behavior 168, 409

Logical Unit of Work 43

M

Maintain Business Roles (App) 479

Managed 120, 138, 316

Managed BO provider 44, 138, 261

Managed query 44, 105, 316

Managed scenario 43, 138

Managed scenario → Managed scenario

Message class 357, 403

Meta-annotation 86

Metadata 75, 346

Metadata extension 87, 98, 344, 424, 440

MODIFY ENTITIES 213

Modularization 39

N

Name, external 142

Navigation 297

Number range

interval 149

management 471

Numbering 146

early 146, 147, 250

external 146, 147

internal 146, 147

late 146, 149, 229, 373, 387

managed 148

O

Object page 276, 282

annotations 289

section 289

OData 34, 37, 47

query 35

 V2 268

 V4 268

vocabulary 35

OData service 34

ABAP Environment 468

metadata 274

publication 270

testing 272

on modify 174

on save 174, 178, 179

Open Data Protocol → OData

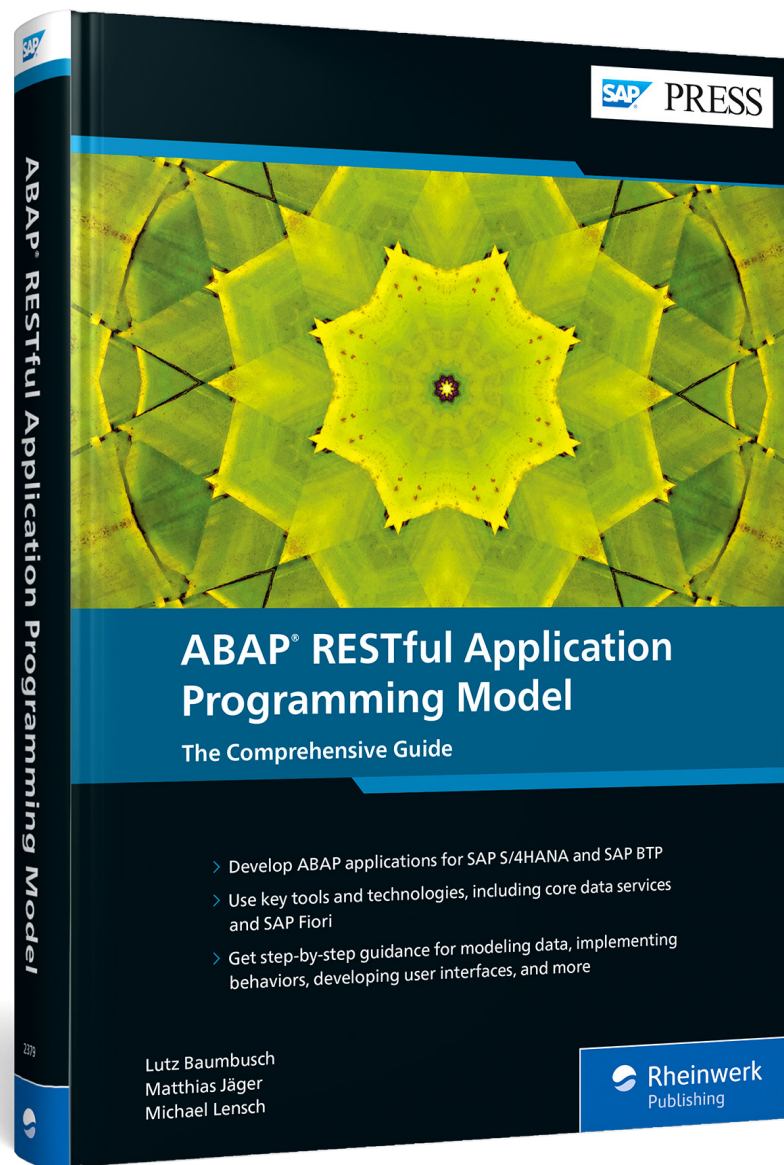
Operation

specific 125

Operation (Cont.)	
<i>standard</i>	50, 124, 155
<i>visibility</i>	201
<i>writing</i>	124
Optimistic lock	171
Orchestration framework	47, 266
Outbound service	485
Output parameter → return parameter	
P	
PATCH	34
Persistent table	138, 153
Pessimistic locking	169
Point of no return	254
POST	33
Precheck	200, 252
Pretty Printer	380
Private view	112
Programming language, domain-specific	27
Programming model	24, 38
Projection behavior definition	119, 266, 418
Projection layer	51, 416
Projection view → CDS projection view	
Property, transactional	123
Proxy	31
PUT	34
Q	
Query	44, 316
Quick fix	339
R	
RAP → ABAP RESTful application programming model	
READ ENTITIES	210
Read-by-association	157, 212
Reference parameter	233
Relation Explorer	135
Relationship, functional	80
Release contract C1	469
Remote API	271
Remote API view	112
Remote Function Call	468
REPORTED	300
Repository object, released	469
Representation	30
Required entry field	150
Required field	150
Resource	29, 30
REST	29, 30
<i>architecture principles</i>	25, 30
<i>compliant software architecture</i>	32
RESTful API	33
result	162
Return parameter	162
<i>implicit</i>	209
Reusability	494
RFC	468
Roadmap	491
role	89
ROLLBACK ENTITIES	219
root	114
Root entity	118, 382
Root URL	270
Runtime component	46
S	
SACMSEL	91
SADL framework	47, 266
SAP Annotation	83
SAP API Business Hub	36
SAP BTP Cockpit	279
SAP BTP, ABAP environment	58, 465
<i>architecture</i>	466
<i>use cases</i>	466
SAP Business Application Studio	278
SAP Business Technology Platform	278
SAP Destination Service	467, 484
SAP Fiori	37, 277
SAP Fiori app	315
SAP Fiori design guidelines	277
SAP Fiori elements	28, 277, 344, 424
<i>draft handling</i>	190
<i>floorplan</i>	281
<i>preview</i>	275, 344
SAP Fiori launchpad	37, 467, 480
<i>ABAP environment</i>	480
<i>page</i>	480
<i>space</i>	480
SAP Fiori Tools	281, 363
SAP Fiori user experience	28
SAP Gateway	37, 47
SAP Gateway Client	104, 272
SAP HANA	28, 37
SAP HANA CDS view	62
SAP NetWeaver Application Server	
ABAP	38
SAP S/4HANA	36

SAPUI5	37, 277
SAVE	259
Save handler	229, 253, 458
Save option	
<i>CDS entity</i>	140
<i>managed scenario</i>	138
Save sequence	43, 228, 398
<i>determinations</i>	178
<i>implementation</i>	458
SAVE_MODIFIED	258
Saver class	448, 458
SCM	486
Search help	295, 425
Search property	287
selective	164, 168
Semantic object	476
Separation of Concerns	57, 100
Server	30
Service binding	52, 263, 268
<i>binding type</i>	268, 343
<i>creating</i>	419
<i>editor</i>	270
<i>publishing</i>	270, 343
<i>versioning</i>	269
Service consumer	341
Service Consumption Model	486
Service definition	51, 263, 267, 341, 418
Service document	35
Service endpoint, local	270
Service metadata	34, 274
Service Modeler	302
Service provider	341
Service URL	35, 272
Session variable	77
SET LOCKS	217
Side effect	182, 302
Side-by-side extension	466
Software architecture,	
REST-compliant	32
SQL	269
SQL Console	72
SQL view	68
Standard operation	50, 124, 155
Standardization	26
Stateless communication	30
STAUTHTRACE	422
Steampunk → SAP BTP, ABAP environment	
Strict mode	141
Syntax	62
Syntax check	141, 469
T	
Table	
<i>customer-specific</i>	431
Table expression	39
Target data source	79
Test class	222
Testability	57
Text, translation	300
Time	253
Time stamp	173
Total ETag	192
<i>process</i>	173
Transaction	
<i>/IWFND/GW_CLIENT</i>	272
<i>/IWFND/MAINT_SERVICE</i>	104, 272
<i>SACMSEL</i>	91
<i>STAUTHTRACE</i>	422
Transaction buffer	42, 226, 260
Transaction control	374
Transaction model	42
Transaction owner	228
Transactional behavior	123
Transactional key	208
Translation	300
Trigger condition	173
<i>combination</i>	176, 181
<i>determination</i>	174
<i>field</i>	175, 180
<i>standard operation</i>	174, 179
<i>validation</i>	179
Trigger → trigger condition	
Type conversion	77
U	
UI annotation	41, 283, 309
UI service	269, 275
UI.facet	284
UI.fieldGroup	284
UI.headerInfo	284
UI.identification	284
UI.item	284
UI.selectionField	284
Unified Resource Identifier → URI	
Uniform Resource Locator	29
Uniqueness check	147
Unmanaged	316
Unmanaged query	105, 316
Unmanaged Save	139, 317, 371
Unmanaged scenario	44, 140, 429
update	156

URI	29	Visibility	201
URI option	274	Visual Studio Code	278, 280, 281
URL	29		
use	266	W	
Use case	315		
User experience	37	Web API	269, 316
UUID	148	where-used list	420
UX	37	Whitelisted API	469
		With additional save	140
V		With unmanaged save	139
		World Wide Web	29
Validation	179, 250, 403	Write lock → lock	
<i>creating</i>	356	Write protection	151
<i>defining</i>	300, 403		
<i>implementing</i>	404, 414	X	
<i>notification</i>	300		
Variant control	282	XCO library	470
VDM	63, 109	XML file	308
View stack	74, 96		
View-Browser	111	Z	
Virtual data model	63, 109		
Virtual element	101, 266, 332	Z table	431



Baumbusch, Jäger, Lensch

ABAP RESTful Application Programming Model: The Comprehensive Guide

508 pages, 2023, \$89.95
ISBN 978-1-4932-2379-4

 www.sap-press.com/5647



Lutz Baumbusch has been working as an SAP developer since 2000 and has been responsible for international SAP projects in various roles and areas. At All for One Group SE, he prepares current developer topics for internal and external training in the SAP S/4HANA development team. He studied at the Karlsruhe Institute of Technology.



Matthias Jäger is a freelance SAP developer of software products based on the ABAP platform. In addition to his work as a developer, he works as an architect and coach, designs cross-product, technical aspects, and leads training courses. In 2004 he started his career as an SAP developer and as a developer of Java-based software products. Since then, he has worked as a developer, development manager, and trainer in different SAP implementation projects in the logistics sector. He has been a regular guest lecturer at DHBW Stuttgart, where he teaches ABAP programming and the use of system analysis methods. He studied business informatics in Heidenheim as part of a dual degree program.



Michael Lensch leads a team of SAP developers at All for One Group SE. As development manager, he is responsible for development in SAP S/4HANA implementation projects in Germany and abroad. Since 2014, he has also led a technical team that deals with development for SAP HANA and SAP S/4HANA. He studied computer science at Trier University of Applied Sciences.

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.