



PUBLIC
2020-05-15

SAP - ABAP RESTful Application Programming Model

Content

1	ABAP RESTful Application Programming Model	6
2	Before You Start	9
2.1	Prerequisites	9
2.2	Constraints	9
3	Getting Started	13
3.1	Downloading the ABAP Flight Reference Scenario	13
3.2	Developing an OData Service for Simple List Reporting	14
	Defining the Data Model with CDS	16
	Creating an OData Service	24
	Designing the User Interface for a Fiori Elements App	35
4	Concepts	43
4.1	Data Modeling and Behavior	46
	Query	47
	Business Object	55
4.2	Business Service	98
	Business Object Projection	100
	Service Definition	108
	Service Binding	111
4.3	Service Consumption	113
4.4	Runtime Frameworks	115
4.5	Entity Manipulation Language (EML)	116
5	Develop	121
5.1	Developing Read-Only List Reporting Apps	123
	Determining the Data Model for the Read-Only Scenario	125
	Implementing Associations for Existing CDS Views	131
	Changing UI Field Labels and Descriptions	133
	Displaying Text for Unreadable Elements	135
	Providing Value Help for the Selection Fields	138
	Adding Search Capabilities	141
5.2	Developing Managed Transactional Apps	144
	Reference Business Scenario	148
	Developing a Ready-to-Run Business Object	153
	Developing Business Logic	175
	Developing a Projection Layer for Flexible Service Consumption	237

	<i>Cloud</i> Defining Business Services Based on Projections	262
5.3	Developing Unmanaged Transactional Apps.	263
	Reference Business Scenario	266
	Providing CDS Data Model with Business Object Structure	269
	Defining and Implementing Behavior of the Business Object.	288
	Defining Business Service for Fiori UI.	350
	Adding Another Layer to the Transactional Data Model	354
5.4	Developing a Web API.	365
	Publishing a Web API.	367
5.5	<i>Cloud</i> Developing a UI Service with Access to a Remote Service.	369
	<i>Cloud</i> Scenario Description.	371
	<i>Cloud</i> Preparing Access to the Remote OData Service.	374
	<i>Cloud</i> Creating a Database Table for the Persistent Fields.	378
	<i>Cloud</i> Using a CDS Custom Entity for Data Modeling.	378
	<i>Cloud</i> Consuming the Remote OData Service	382
	<i>Cloud</i> Defining an OData Service.	417
6	Extend.	420
7	Common Tasks.	421
7.1	Defining Text Elements	422
	Providing Text by Text Elements in the Same Entity	422
	Getting Text Through Text Associations.	424
	Getting Language-Dependent Text in Projection Views.	426
7.2	Providing Value Help.	428
	Simple Value Help.	430
	Value Help with Additional Binding.	435
7.3	Enabling Text and Fuzzy Searches in SAP Fiori Apps	437
7.4	Using Virtual Elements in CDS Projection Views.	440
	Modeling Virtual Elements.	442
7.5	Using Aggregate Data in SAP Fiori Apps.	446
	Annotating Aggregate Functions in CDS.	447
	OData Interpretation of Aggregation Annotations.	451
7.6	<i>Cloud</i> Automatically Drawing Primary Key Values in Managed BOs	455
7.7	Adding Feature Control.	459
	Static Feature Control.	460
	Dynamic Feature Control.	463
	Defining Static and Dynamic Feature Control	466
	Implementing Dynamic Feature Control.	468
7.8	Consuming Business Objects with EML.	469
7.9	Using Type and Control Mapping	477

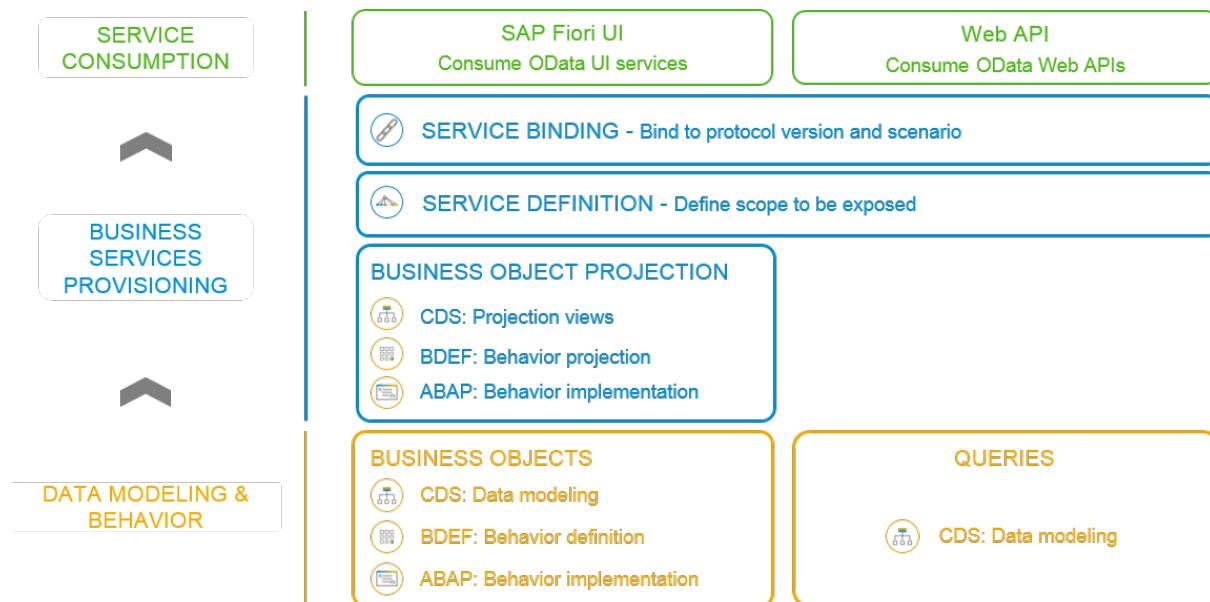
7.10	Using Groups in Large Development Projects	481
7.11	Adding Authorization Control to Managed Business Objects	487
	Modeling Authorization Control	488
	Implementing Authorization Control	490
7.12	Implementing an Unmanaged Query	494
	Implementation	496
	Returning Requested Entity in an Unmanaged Query	499
	Requesting and Setting Data or Count in an Unmanaged Query	500
	Implementing Filtering in an Unmanaged Query	501
	Using Parameters in an Unmanaged Query	503
	Implementing Search in an Unmanaged Query	505
	Implementing Paging in an Unmanaged Query	507
	Implementing Sorting in an Unmanaged Query	509
	Considering Requested Elements in an Unmanaged Query	510
	Implementing Aggregations in an Unmanaged Query	511
7.13	Adding Field Labels and Descriptions	513
7.14	Defining CDS Annotations for Metadata-Driven UIs	515
	Tables and Lists	516
	Detail Pages	521
	Field Groups	525
	Annotations Similar to dataField	526
	Charts	530
	Data Points	532
	Contact Data	543
	Navigation	544
	Actions	550
	Field Manipulation	552
8	Reference	558
8.1	CDS Annotations	558
	Aggregation Annotations	559
	AccessControl Annotations	561
	Consumption Annotations	562
	ObjectModel Annotations	565
	OData Annotations	568
	Search Annotations	602
	Semantics Annotations	606
	UI Annotations	610
8.2	API Documentation	717
	Unmanaged BO Contract	718
	Unmanaged Query API	741

API for Virtual Elements.	756
8.3 Tool Reference.	758
Exploring Business Objects.	759
Working with Behavior Definitions.	760
Working with Business Services.	767
Creating Projection Views.	773
8.4 ABAP Flight Reference Scenario.	774
8.5 Naming Conventions for Development Objects	780
9 What's New.	783
9.1 Cloud Version 2005.	783
9.2 Cloud Version 2002.	786
9.3 Cloud Version 1911.	788
9.4 Cloud Version 1908.	791
9.5 Cloud Version 1905.	796
9.6 Cloud Version 1902.	799
9.7 Cloud Version 1811.	802
10 Glossary.	803

1 ABAP RESTful Application Programming Model

The ABAP RESTful Application Programming Model (in short RAP) defines the architecture for efficient end-to-end development of intrinsically SAP HANA-optimized OData services (such as Fiori apps) in *SAP Cloud Platform ABAP Environment* [page 818] or *Application Server ABAP*. It supports the development of all types of Fiori applications as well as publishing Web APIs. It is based on technologies and frameworks such as Core Data Services (CDS) for defining semantically rich data models and a service model infrastructure for creating OData [page 816] services with bindings to an OData protocol and ABAP-based application services for custom logic and SAPUI5-based user interfaces – as shown in the figure below.

Architecture Overview



- Design Time [page 43]

Target Audience

ABAP developers who want to provide (OData) services within the scope of ABAP RESTful application programming model.

Validity of Documentation

This documentation refers to the range of functions that have been shipped as part of delivery of the application server for

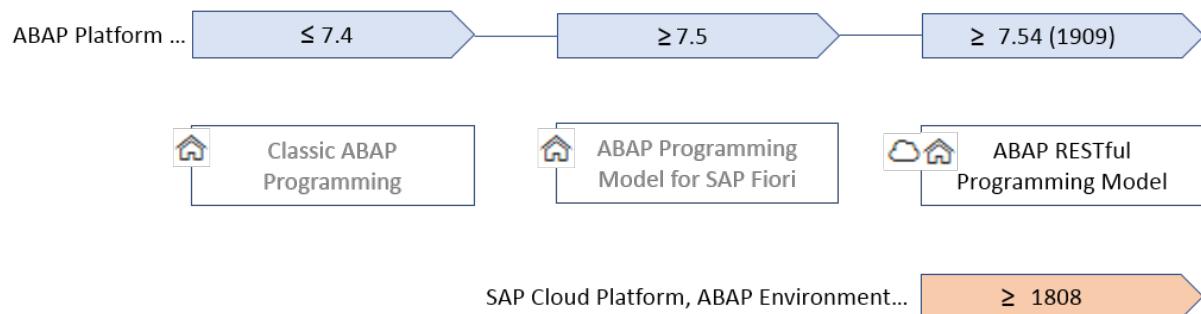
- SAP Cloud Platform ABAP Environment

Note

To highlight the specifics for SAP Cloud Platform releases, the icon is used.

Classification of ABAP RESTful Application Programming Model within the Evolution of ABAP Programming Model

This image is interactive. Hover over each area for a description.



- [ABAP RESTful Application Programming Model \[page 6\]](#)
- [ABAP RESTful Application Programming Model \[page 6\]](#)

The *ABAP RESTful Application Programming Model* is the evolutionary successor of the *ABAP Programming Model for SAP Fiori*. It is generally available to customers and partners within **SAP Cloud Platform ABAP Environment** starting with release 1808 and within **ABAP Platform** starting with release 7.54 SP00 (1909 FPS00).

For more information about the evolution of the ABAP programming model, read this [blog](#) on the community portal.

Contents

[Before You Start... \[page 9\]](#)

[Getting Started \[page 13\]](#)

[Concepts \[page 43\]](#)

[Develop \[page 121\]](#)

[Extend \[page 420\]](#)

[Common Tasks \[page 421\]](#)

[CDS Annotations \[page 558\] \(Reference\)](#)

[API Documentation \[page 717\] \(Reference\)](#)

[Tool Reference \[page 758\] \(Reference\)](#)

[Glossary \[page 803\]](#)

2 Before You Start...

... check the [Prerequisites \[page 9\]](#) and [Constraints \[page 9\]](#)

2.1 Prerequisites

SAP Cloud Platform ABAP Environment

You have access to and an account for *SAP Cloud Platform, ABAP environment*.

More on this: [Getting Global Account](#)

Development Environment (IDE)

- You have installed ABAP Development Tools (ADT).
SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.
- You have access to and a user account for the relevant space in *SAP Cloud Platform Cloud Foundry*.

More on this:



Authorizations

To create development artifacts described in this guide, you need the developer authorization profile for the *ABAP Environment*.

Knowledge

Basic knowledge of

- ABAP Core Data Services (CDS)
- ABAP Objects.

2.2 Constraints

The current version of the ABAP RESTful programming model still has some constraints when ...

Defining Custom Entities

Associations with custom entity as source or target support only attribute bindings ($A1 = A2$ and $B1 = B2$), but no

- OR, NOT
- Other operators than '='
- Using something else than CDS elements as operands (e.g. no literals or variables)

Using Abstract Entities

You cannot expose an OData service that includes abstract entities. Whereas abstract entities are allowed to be used in a service definition, the publishing of a service via a service binding causes a dump error.

Using Parameters in CDS Views

- The usage of simple data types for parameters is not supported.
- CDS Views with parameters do not work properly in Fiori Elements UI services. The filter value is not sent to the backend.

Where Clause in CDS Projection Views

In transactional operations, including the transactional READ, the where clause in projection views is not respected. Applications must ensure that the instances that are created, updated, or read via the projection conform to the where clause.

Developing Services with Implementation Type Managed

When working with the managed implementation type the following constraints are given:

- **Creating Child Entity Instances:** Instances of child entities can only be created by a `create-by-association`.
- **CDS View Fields:** In CDS views, you have to use the same name for the fields as in the underlying database table. No aliasing is supported. You can alias the CDS elements in the CDS projection view.
- **Determinations:**
Changes caused by determinations cannot trigger other determinations.
Determinations that are triggered at entire entity level can only be defined for the `create` operation.
- **Validations:** Validations can only be triggered at field level.
- **Authority:** Only instance-based authorizations are available. That means, static authorizations are not available. Therefore, you cannot apply authorization checks to create operations.
- **Numbering:** Late numbering is not supported.
- **Actions:** Factory actions are not supported.
- **Functions:** Functions are not supported.

Using the Fiori Elements App Preview

The Fiori Elements App preview does not support the navigation to the object page of nested subentities, which means to more than one child entity. This also affects the create functionality for nested subentities. The create button is only displayed for the root and one child entity of a business object when accessing the preview via the root.

To test UI features or the create functionality for nested subentities, you can test the OData service with the Web IDE and configure the application to enable navigation to any number of child entities.

Alternatively, you can access the preview via the parent entity of the nested subentity that you want to test.

• Example

You want to test the create functionality of the subentity `BookingSupplement` in the `Travel` business object, which is the second child entity in the hierarchy. Instead of starting the preview in the service binding via the root entity `Travel` or the composition `Travel-to_Booking`, access the preview via the child entity `Booking` or the composition `Booking-to_BookSupplement` to see the complete feature scope of the nested subentity `BookingSupplement`.

The screenshot shows a hierarchical tree of entity sets and associations. The root node is 'Entity Set and Association'. Under it, there are several nodes: 'TravelAgency', 'BookingSupplement', 'Booking' (which is expanded to show 'to_BookSupplement', 'to_Carrier', 'to_Connection', 'to_Customer', and 'to_Travel'), 'Airline', 'FlightConnection', 'CurrencyText', 'Passenger', 'Flight', 'Supplement', 'SupplementText', 'Travel' (expanded to show 'to_Agency', 'to_Booking', 'to_Currency', and 'to_Customer'), 'Country', and 'Currency'. A mouse cursor is hovering over the 'to_BookSupplement' node under 'Booking'. A tooltip box appears with the text 'Double click or use context menu on the nodes to launch Preview for Fiori Elements App.'.

Fiori Elements Preview Testing for Nested Entities

Using the Service Consumption Model

- Even though you can delete all generated artifacts except for the generated service definition, it is recommended not to do so as it corrupts the service consumption model. If you edit or delete a generated artifact, then the form editor for the service consumption model does not open and an error is displayed. Also, if you delete any of the generated artifacts, you cannot delete a service consumption model object. You need to recreate the deleted artifact for the form editor and object deletion to work.
- For a service entity set, the remote OData service may have support only for one CRUD operation, for example, READ. Currently, code snippets are displayed for all the operations even if the support is provided only for one operation.

Using the Data Preview

Data Preview does not render properly if you use the dark theme.

Cloud Exposing Actions for OData

As of cloud release 1811, the OData names of function imports for actions that are declared in a behavior definition do not carry the name of their related entity set anymore. Instead, OData only adopts the name of the action that is defined in the behavior definition.

The following table illustrates the changes in action names as displayed in the metadata of an OData service.

Before ABAP Environment 1811

```
<FunctionImport  
Name="<entity\_ref><action\_name>" >
```

After ABAP Environment 1811

```
<FunctionImport Name="<action\_name>" >
```

3 Getting Started

This **Getting Started** section provides you with the fundamental basics of development with the ABAP RESTful Programming Model.

For demonstration and learning purposes we provide the **ABAP Flight Reference Scenario** which simulates an application used by a travel agency for booking flights. The first thing to do is therefore to import the ABAP Flight Reference Scenario in your ADT to get sample data: [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

The getting started guide helps you to create a complete application based on the existing data model from the ABAP Flight Reference Scenario with the most basic features: [Developing an OData Service for Simple List Reporting \[page 14\]](#).

3.1 Downloading the ABAP Flight Reference Scenario

The ABAP Flight Scenario contains demo content that you can import into your development environment.

The ABAP Flight Reference Scenario helps you to get started with development in the context of the ABAP RESTful Application Programming Model. It contains demo content that you can play around with and use to build your own sample applications.

Sample Data

First of all, the reference scenario contains data. You can use database tables that are filled with travel data including master data items, such as customer, flights, airports, or booking supplements. The structure of the complete data model allows you to build simple but also more complex services. In this way, it is easy to follow the steps in the development guides while building your own application based on the same database tables as in the given examples.

For an overview of the available database tables, see [ABAP Flight Reference Scenario \[page 774\]](#). They are available in the package `/DMO/FLIGHT_LEGACY`. This package also includes a data generator with which you can fill the database tables.

Sample Services

The development guides for the ABAP RESTful Application Programming model are based on the sample data from the ABAP Flight Reference Scenario. That means that you can compare the documentation with the productive code that was used to build the documentation scenario. In addition, the ABAP Flight Reference Scenario also includes a demo package with the development objects that are created during the course of the development guides. That means, the whole demo scenario can be downloaded and tested. You obtain full demo services with code built by following conventions and best practices and you can use and reuse the delivered objects for your development.

Apart from one exception (service binding), the development object of the development guides in the [Develop \[page 121\]](#) section can be downloaded into your cloud system.

The following demo scenarios are available for you:

- Developing Read-Only List Reporting Apps [page 123] in the package /DMO/FLIGHT_READONLY
- Developing Unmanaged Transactional Apps [page 263] in the package /DMO/FLIGHT_UNMANAGED
- Developing Managed Transactional Apps [page 144] in the package /DMO/FLIGHT_MANAGED

Legacy Coding

The reference scenario also includes legacy coding. This legacy coding is based on function modules and exemplifies legacy applications that you can include in your new ABAP code. Above all, the legacy coding is relevant for the development guide, that explains how to build a new service on the basis of an existing application. It illustrates how you build an application with the unmanaged implementation type. The legacy coding that is used in this scenario is available in the package /DMO/FLIGHT_LEGACY.

Downloading the ABAP Flight Reference Scenario from GitHub

You can download the complete ABAP Flight Reference Scenario for the ABAP RESTful Application Programming Model from GitHub.

<https://github.com/SAP-samples/abap-platform-refscen-flight/tree/Cloud-Platform> ↗ .

The steps to include the development objects in your ADT are described in the [README.md](#) file.

→ Remember

The namespace /DMO/ is reserved for the demo content. Apart from the downloaded ABAP Flight Scenario, do not use the namespace /DMO/ and do not create any development objects in the downloaded packages. You can access the development objects in /DMO/ from your own namespace.

3.2 Developing an OData Service for Simple List Reporting

The following guide describes the basic development tasks to create a simple list reporting app based on a query.

Introduction

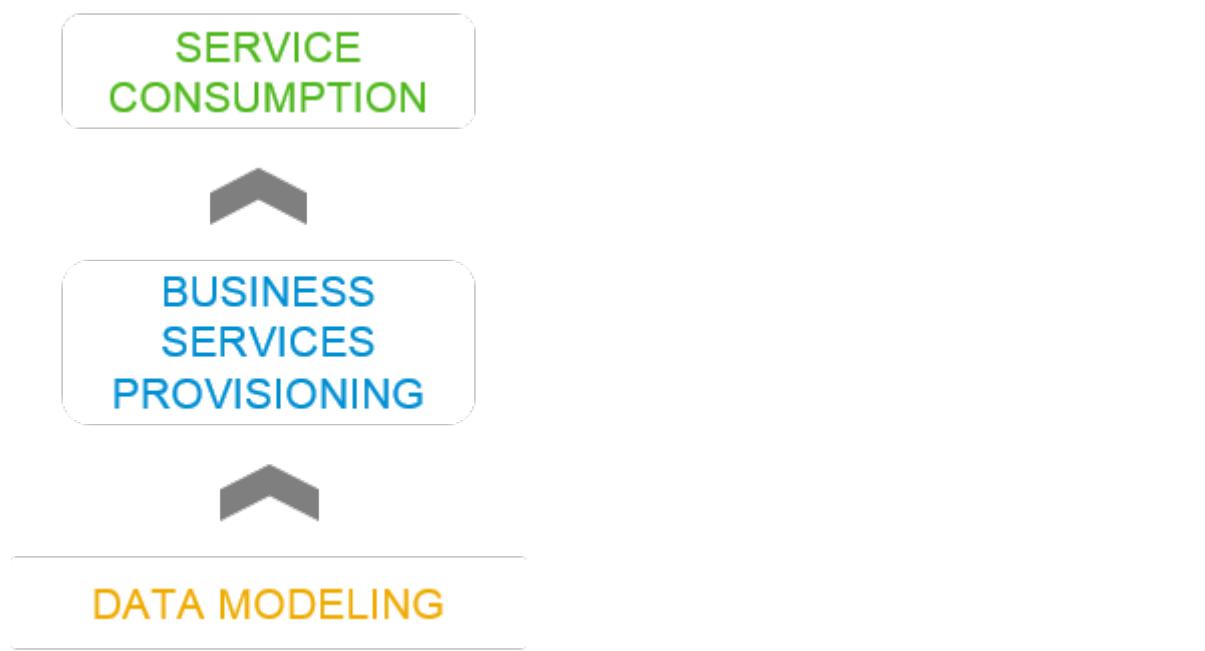
The following sections serve as an introductory guide for the development of an OData service based on the ABAP RESTful Application Programming Model . It forms the basic building block for more elaborate scenarios with extended read-only features or transactional processing.

An OData service makes it possible to create and consume queryable and interoperable RESTful APIs. A SAP Fiori Elements application consumes OData services like this, but it also possible for other Web clients to make use of an OData service that is created with the ABAP RESTful Application Programming Model .

This programming model provides a framework that facilitates your application development. All included technologies, such as Core Data (CDS) or business services, are usable and accessible with ABAP Development Tools (ADT), providing easy access to the necessary infrastructure.

The following guide starts from a data model assuming that database tables already exist. It uses the ABAP Flight Reference Scenario (in short Flight Scenario), which provides example data comprising travel information with flight data. For a detailed description of the database tables that are used in this scenario, refer to [ABAP Flight Reference Scenario \[page 774\]](#)

You are guided step-by-step through the new application model in three consecutive building blocks:



- [Defining the Data Model with CDS \[page 16\]](#)
- [Creating an OData Service \[page 24\]](#)
- [Designing the User Interface for a Fiori Elements App \[page 35\]](#)

You start by implementing a CDS view as a new data model layer using a data source that is already provided. You also use basic CDS annotations to manifest semantics for the data model. The next step is to create an OData service by defining and binding a service based on the corresponding CDS view. As soon as the OData service is published in the local system repository, it is ready to be consumed using an OData client, such as a SAP Fiori app. Finally, you learn how to use UI annotations as a UI technology independent semantic description of the user interface layout.

The result of this Getting Started guide is a consumable OData Service, which can be easily used to set up a Fiori Elements travel booking app, from which you can derive information about flight connections. Navigation properties are added to this application to receive more information about bookings, customers, and agencies in the other scenarios in the [Develop \[page 121\]](#) section. These other development guides also cover extended read-only and transactional features, whereas the Getting Started guide only deals with the most basic read-only features for setting up an OData Service. The scenarios in the Develop section assume that you understood the steps that are described in the following guide.

i Note

Via ABAPGit you can import the service including the related development objects into your development environment for comparison and reuse. You find the service in the package /DMO/FLIGHT_READONLY. The suffix for development objects in this development guide is _R. Be aware that the development objects might contain more than explained in the Getting Started guide. This is because the Getting Started scenario is enhanced in the first development guide [Developing Read-Only List Reporting Apps \[page 123\]](#) which uses this same demo objects.

For information about downloading the ABAP Flight Reference Scenario, see [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

Prerequisites

Developing the scenario that is described in the subsequent chapters requires the following:

-  You have access to and an account for **SAP Cloud Platform, ABAP environment**.
- You have installed ABAP Development Tools (ADT).
SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.
- To recreate the demo scenario, the *ABAP Flight Reference Scenario* must be available in your ABAP system. You can download the complete reference scenario from [GitHub: Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

Objectives

By the end of this Getting Started section, you will be able to:

- Create a data definition and define a CDS view
- Implement an ABAP CDS view based on an existing database table
- Define an OData service and expose a CDS view for this service
- Bind the OData service against a protocol and publish it locally
- Use semantics annotations in CDS
- Understand some basic UI annotations in CDS

3.2.1 Defining the Data Model with CDS

The data model for an OData service must be defined in CDS.

This introductory programming guide uses example data from the Flight Reference Scenario. The Getting Started scenario uses the database table /dmo/connection. It provides information about airline and connection numbers, flight times, and data related to planes.

In the CDS layer we use and manipulate data that is persisted in the database. To make data available in the ABAP application server, CDS views use SQL queries to project persisted data to the ABAP layer. This is necessary to create an OData service to make the data ready to be consumed. More information about CDS: .

To define a data model based on the ABAP CDS view concept, you first need to create a data definition as the relevant ABAP Repository object using a wizard in [ABAP Development Tools](#).

Task 1: Creating a Data Definition for a CDS View [page 17]

In the second step, you implement an elementary CDS view from scratch by defining a simple query for flights based on a single data source from the ABAP Flight Reference Scenario.

Task 2: Implementing the CDS View as a Data Model [page 19]

In the final task of this section, you have the option of using the test environment to verify the output (a results set) of the CDS view you have just implemented.

Task 3: Verifying the Results Set in the Data Preview Tool [page 22]

3.2.1.1 Creating a Data Definition for a CDS View

Use the data definition wizard to create the relevant development object for a CDS view.

Context

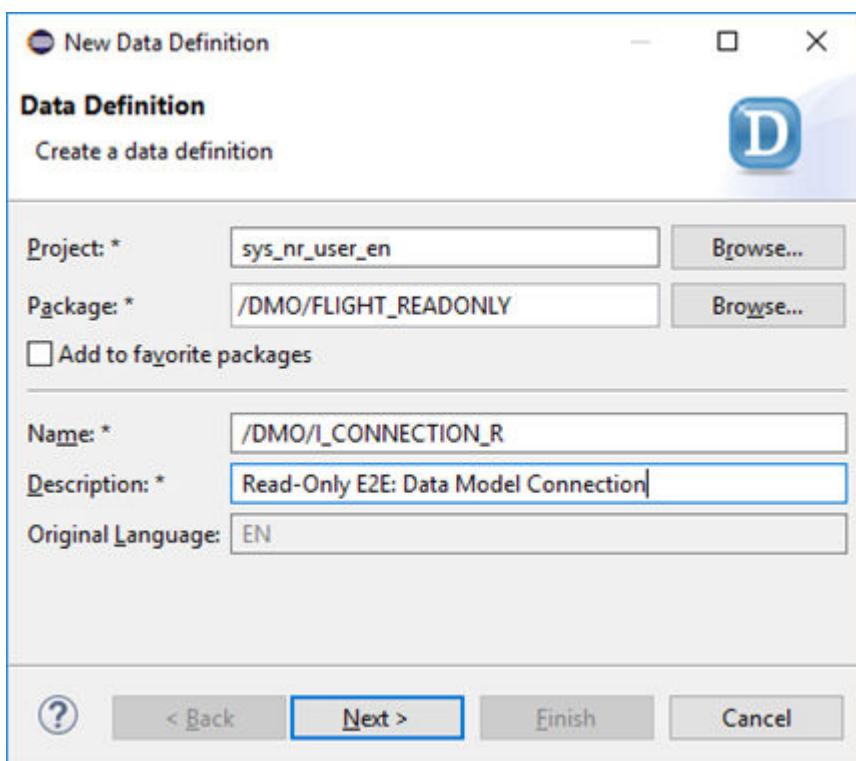
For our simple read-only scenario, we want to define data that is exposed by an OData service to make it available for an OData client. For this purpose, you create a development object to define an ABAP CDS entity (for example, a CDS view). The data definition provides you with the appropriate development object for the CDS view, which is included in ABAP development tools and directly accesses the standard ABAP functions.

Procedure

1. Launch the [ABAP Development Tools](#).
2. In your ABAP project, select the package node in which you want to store the development objects for the Getting Started scenario.
3. Open the context menu and choose [New](#) [Other ABAP Repository Object](#) [Core Data Services](#) [Data Definition](#) to launch the creation wizard for a data definition.
4. In addition to the [Project](#) and [Package](#), which are already specified depending on the package you selected, enter the [Name](#) (while respecting your namespace) and a [Description](#) for the data definition you want to create.

i Note

The maximum length for the name of a data definition is 30 characters.



First wizard page when creating a data definition

5. Choose *Next*.
6. Assign a transport request.
7. Choose *Finish* or choose next to select a template for the data definition.

Choosing finish directly provides you with the correct template.

Results

In the selected package, the ABAP back-end system creates an inactive version of a data definition and stores it in the ABAP Repository. As a result, the data definition editor is opened. The generated source code already provides you with the necessary view annotations and adds placeholders for the names of the database view and for the data source for query definition. The name for the actual CDS view is predefined on the basis of the name for the data definition, but can be changed in the data definition editor.

```

@AbapCatalog.sqlViewName: 'sql_view_name'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Read-Only E2E: Data Model Connection'
define view /DMO/I_Connection_R as select from data_source_name {
}
  
```

The generated template code in the data definition editor

Next Steps

Now that you have created a data definition, you can implement the CDS view as a data model for your OData service.

3.2.1.2 Implementing the CDS View as a Data Model

Use a predefined database table as the data source for a CDS view.

Prerequisites

- You have created the data definition artifact in ABAP Development Tools.
- The database table `/dmo/connection` is available for you.

Context

In this step, you implement an interface view as a new data model using a predefined data source.

Procedure

1. If you have not yet already done so, open the new data definition in the editor.
2. Specify the names of the following:
 - a. Database view to be generated in the ABAP Dictionary: `/DMO/ICONNECT_R`

i Note

We use a shortened name due to character limitations in database views of 16 characters.

 - b. Actual CDS view: `/DMO/I_Connection_R`

The data definition editor already provides a suggestion for the name using the name that you specified for the data definition in the creation wizard. However, these names do not have to be the same. You can overwrite it in the `define` statement. Note that the names of the database view and the CDS view must not be the same.
3. In the `SELECT` statement, enter the predefined database table `/dmo/connection` as a data source and define an optional alias name for the data source.

An alias is useful especially when you use multiple data sources or whenever the name of the data source is not descriptive or too long.

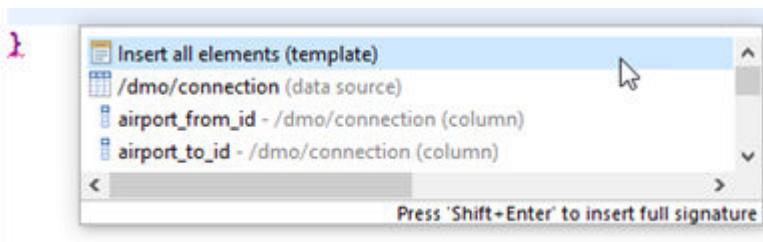
```
... select from /dmo/connection as Connection{
```

4. Add the fields of `/dmo/connection` to the SELECT list and assign alias names to each item field as follows:

```
{
    Connection.carrier_id      as AirlineID,
    Connection.connection_id   as ConnectionID,
    Connection.airport_from_id as DepartureAirport,
    Connection.airport_to_id   as DestinationAirport,
    Connection.departure_time  as DepartureTime,
    Connection.arrival_time   as ArrivalTime,
    Connection.distance        as Distance,
    Connection.distance_unit   as DistanceUnit
}
```

→ Tip

Whenever you insert table fields or view elements in the SELECT list, you can make use of the content assist function in the data definition editor (`CTRL` + `SPACE`).



Inserting fields using semantic auto-completion

5. To document the key semantics of the new data model, define the `AirlineID` and `ConnectionID` elements as `KEY` elements in the current CDS view:

```
key connection.carrier_id      as AirlineID,
key connection.connection_id   as ConnectionID,
```

6. Click the activation button or use the shortcut `Ctrl` + `F3` to activate the data definition.

To check the syntax before activation, click or use the shortcut `Ctrl` + `F2`.

Results

The resulting source code for the CDS view is the following:

```
@AbapCatalog.sqlViewName: '/DMO/ICONNECT_R'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E: Data Model Connection'
define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
{
  key Connection.carrier_id      as AirlineID,
  key Connection.connection_id   as ConnectionID,
  Connection.airport_from_id    as DepartureAirport,
  Connection.airport_to_id      as DestinationAirport,
  Connection.departure_time     as DepartureTime,
  Connection.arrival_time       as ArrivalTime,
```

```
        Connection.distance      as Distance,  
        Connection.distance_unit as DistanceUnit  
    }
```

The source code above is used to define a quite simple CDS view named `/DMO/I_Connection_R`. This view is implemented using a query that performs a `SELECT` statement, where the database table `/dmo/connection` is used as the data source. The select list includes a set of fields that are relevant for the scenario. The `KEY` elements in the selection list are used to define the key field semantics of the CDS view.

When the data definition source is activated, the following objects are created in ABAP Dictionary:

- The actual entity of the CDS view `/DMO/I_Connection_R`
- An SQL view `/DMO/ICONNECT_R`

Next Steps

Mark the elements `Distance` and `DistanceUnit` as semantically related.

3.2.1.2.1 Relating Semantically Dependent Elements

Use the `@Semantics` annotation to relate the quantity element to its unit of measure element.

Context

The CDS view `/DMO/I_Connection_R` that you created contains elements that are heavily dependent on each other **semantically**, namely `Distance` and `DistanceUnit`. In CDS, you can use semantic annotations to standardize semantics that have an impact on the consumer side for these elements. In general, elements that need to be marked as having semantic content to guarantee that they are handled correctly are elements that contain the following:

- Amounts of money
 - These elements need a reference to the currency related to this element.
- Amounts of measures
 - These elements need a reference to the unit of measure related to this element.

If you create annotations that define a link to the unit for the amounts, the amounts and their units are always handled as being dependent on each other in the OData service. On UIs in particular, amounts are displayed with the correct decimals with regard to their unit.

In the CDS view `/DMO/I_Connection_R`, you therefore need to proceed as described in the following to always display the distance together with the distance unit.

Procedure

1. Open the CDS view [`/DMO/I_Connection_R`](#).
2. Mark the element `DistanceUnit` as a unit of measure with the annotation
`@Semantics.unitOfMeasure: true.`
3. Define the relationship between amount and unit of measure with the annotation
`@Semantics.quantity.unitOfMeasure: '<ElementRef>' on Distance and reference the element DistanceUnit.`

```
@Semantics.quantity.unitOfMeasure: 'DistanceUnit'  
Connection.distance      as Distance,  
@Semantics.unitOfMeasure: true  
Connection.distance_unit as DistanceUnit
```

4. Activate the CDS view.

Results

If you expose the CDS view to an OData service, the elements are always handled as being semantically related to each other. This means that they are given the OData annotation `sap:unit` and `sap:semantics` in the OData metadata document. On UIs in particular, the elements are always displayed as being attached to each other.

Related Information

[Semantics Annotations \[page 606\]](#)

3.2.1.3 Verifying the Results Set in the Data Preview Tool

Use the data preview tool to check the elements in the CDS view.

Prerequisites

The data definition has correct syntax and has been activated.

Context

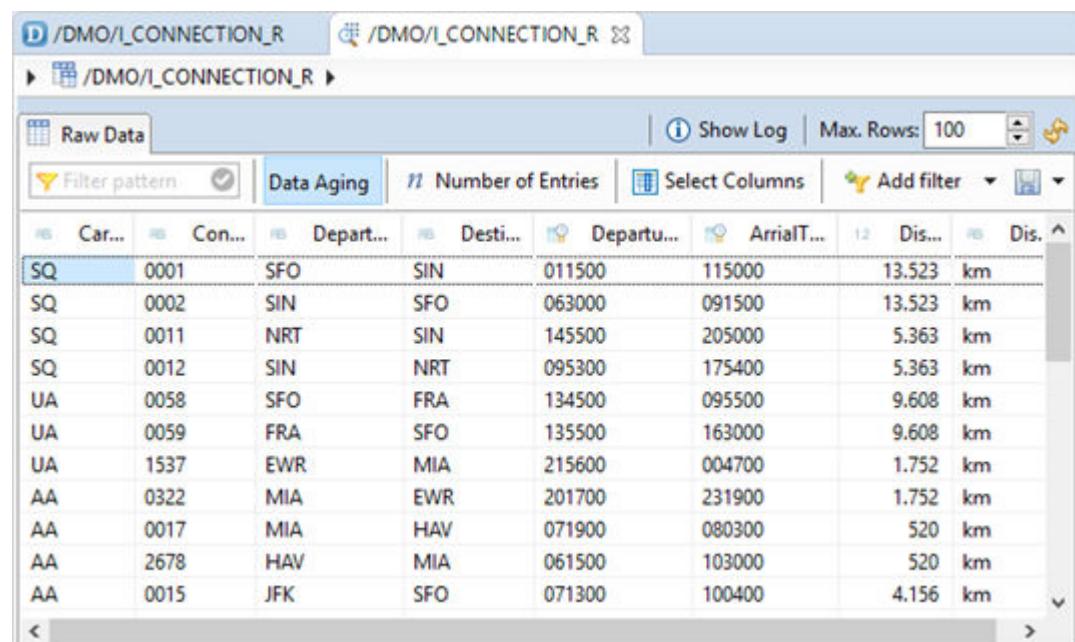
You have created a data definition and implemented a CDS view with data from the database table [/dmo/connection](#). Now you have the option of launching the test environment (in the data preview tool), which enables you to verify that the persisted data from the database is now displayed in the CDS view.

Procedure

In the data definition editor, position the cursor somewhere in the CDS source code. Open the context menu and choose **Open With > Data Preview** or use the shortcut **F8**.

Results

The CDS view does not require any parameters, which means the data preview displays the results set of the data selection query directly.



The screenshot shows the SAP Data Preview tool interface. The title bar displays the URL [/DMO/I_CONNECTION_R](#). The main area contains a table with the following data:

Car...	Con...	Depart...	Desti...	Departu...	ArrialT...	Dis...	Dis...
SQ	0001	SFO	SIN	011500	115000	13.523	km
SQ	0002	SIN	SFO	063000	091500	13.523	km
SQ	0011	NRT	SIN	145500	205000	5.363	km
SQ	0012	SIN	NRT	095300	175400	5.363	km
UA	0058	SFO	FRA	134500	095500	9.608	km
UA	0059	FRA	SFO	135500	163000	9.608	km
UA	1537	EWR	MIA	215600	004700	1.752	km
AA	0322	MIA	EWR	201700	231900	1.752	km
AA	0017	MIA	HAV	071900	080300	520	km
AA	2678	HAV	MIA	061500	103000	520	km
AA	0015	JFK	SFO	071300	100400	4.156	km

Results sets in the data preview tool

i Note

You can sort the entries by element by clicking the column header.

3.2.2 Creating an OData Service

Business service artifacts enable the publishing of an OData service using ABAP Development Tools.

In the previous step, you defined a data model based on the persisted data source `/dmo/connection` in the data definition `/DMO/I_Connection_R`. You can now use this data model and expose it for an OData service. The OData service makes it possible for UI technologies to query data and consume it. The following steps are necessary to include the CDS view in an OData service.

To define a service, you first need to create a service definition as the relevant ABAP Repository object using a wizard.

Task 1: [Creating a Service Definition \[page 24\]](#)

The next step is to define the scope of the OData service by exposing the relevant CDS views (including their metadata and their behavior).

Task 2: [Exposing a CDS View for an OData Service \[page 26\]](#)

To define the type and category of the OData service, you need to create a service binding as the relevant ABAP Repository object. There is also a wizard available for this.

Task 3: [Creating a Service Binding \[page 27\]](#)

In the next step, you use the form-based editor of the service binding to publish the service locally.

Task 4: [Publishing the OData Service Locally \[page 29\]](#)

You have the option of checking the resulting OData service by viewing its metadata. The service binding offers a simple solution for this.

Task 5: [Verifying the OData Metadata \[page 31\]](#)

You can also take a look at how the UI of a Fiori Elements of the OData service looks like with the preview tool of the service binding.

Task 6: [Previewing the Resulting UI Service \[page 33\]](#)

3.2.2.1 Creating a Service Definition

Use the service definition wizard to create the relevant development object that defines the scope of the OData service

Context

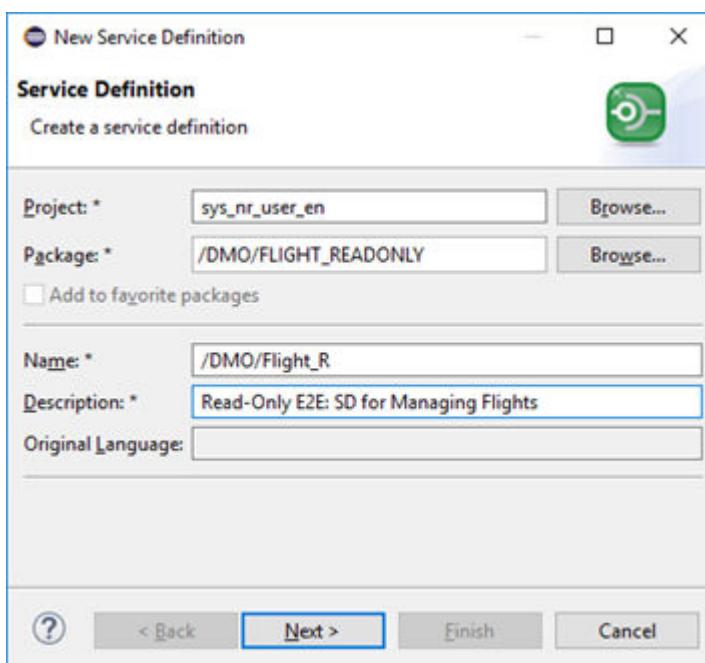
The service definition is a projection of the models and related behavior that you want to expose. In a service definition, you define the OData service to determine which CDS entities are part of the service. This service is then exposed either as a UI service or a Web API by a service binding artifact. A service definition can be integrated in various protocols without any reimplementation.

Procedure

1. In your ABAP project, select the relevant package node in the *Project Explorer*.
2. Open the context menu and choose ► **New** ► **Other ABAP Repository Object** ► **Business Services** ► **Service Definition** to launch the creation wizard.
3. In addition to the *Project* and *Package*, which are already specified depending on the package you selected, enter the *Name* and a *Description* for the service definition you want to create.

i Note

The maximum length for the name of a service definition is 30 characters.



Wizard page when creating a service definition

4. Choose *Next*.
5. Assign a transport request.
6. Choose *Finish*.

Results

The ABAP back-end system creates an inactive version of a service definition and stores it in the ABAP Repository.

In the *Project Explorer*, the new service definition is added to the *Business Services* folder of the corresponding package node. As a result, the service definition editor is opened:

Next Steps

Now that you have created a service definition, you can choose one or more CDS entities to be exposed in the service.

Related Information

[Creating Service Definitions \[page 767\]](#)

3.2.2.2 Exposing a CDS View for an OData Service

Assign the scope of the OData service.

Prerequisites

You have created the service definition artifact in ABAP Development Tools.

Context

In the service definition editor, you determine the CDS entities that you want to expose in an OData service.

Procedure

1. If you have not yet already done so, open the new service definition in the editor.
The name of the service is already specified in accordance with the name you gave in the service definition wizard. It cannot be changed to a different name.
2. Specify the name of each CDS entity that you want to expose for the service. For the getting started read-only scenario, there is only one CDS view to be exposed: [/DMO/I_Connection_R](#)
3. Optionally, you can assign an alias for the CDS view.
An alias is useful, especially when you use multiple CDS views or whenever the name of the CDS view is not descriptive or too long.

4. Click the activation button  or use the shortcut `Ctrl` + `F3` to activate the service definition.

To check the syntax before activation, click  or use the shortcut `Ctrl` + `F2`.

Results

The resulting source code for the service definition is as follows:

```
@EndUserText.label: 'Read-Only E2E: SD for Managing Flights'  
define service /DMO/FLIGHT_R {  
    expose /DMO/I_Connection_R as Connection;  
}
```

The source code above is used to define a service definition named `/DMO/FLIGHT_R`. It exposes the CDS view `/DMO/I_Connection_R` to be included in the service.

Next Steps

Now that the service exists, you can determine the binding type and category for the service using a service binding.

3.2.2.3 Creating a Service Binding

Use the service binding wizard to create the relevant development object to bind the service to a protocol and, if necessary, to an OData client.

Prerequisites

You have defined a service and exposed CDS entities that are included in the service.

Context

A service binding implements the protocol that is used for the OData service. It uses a service definition that projects the data models and their related behaviors to the service.

Procedure

1. In your ABAP project, select the relevant package node in the *Project Explorer*.
2. Open the context menu and choose to launch the creation wizard.
3. In addition to the *Project* and *Package*, which are already specified depending on the package you selected, enter the *Name* and a *Description* for the service binding you want to create.

i Note

The maximum length for the name of a service binding is 26 characters.

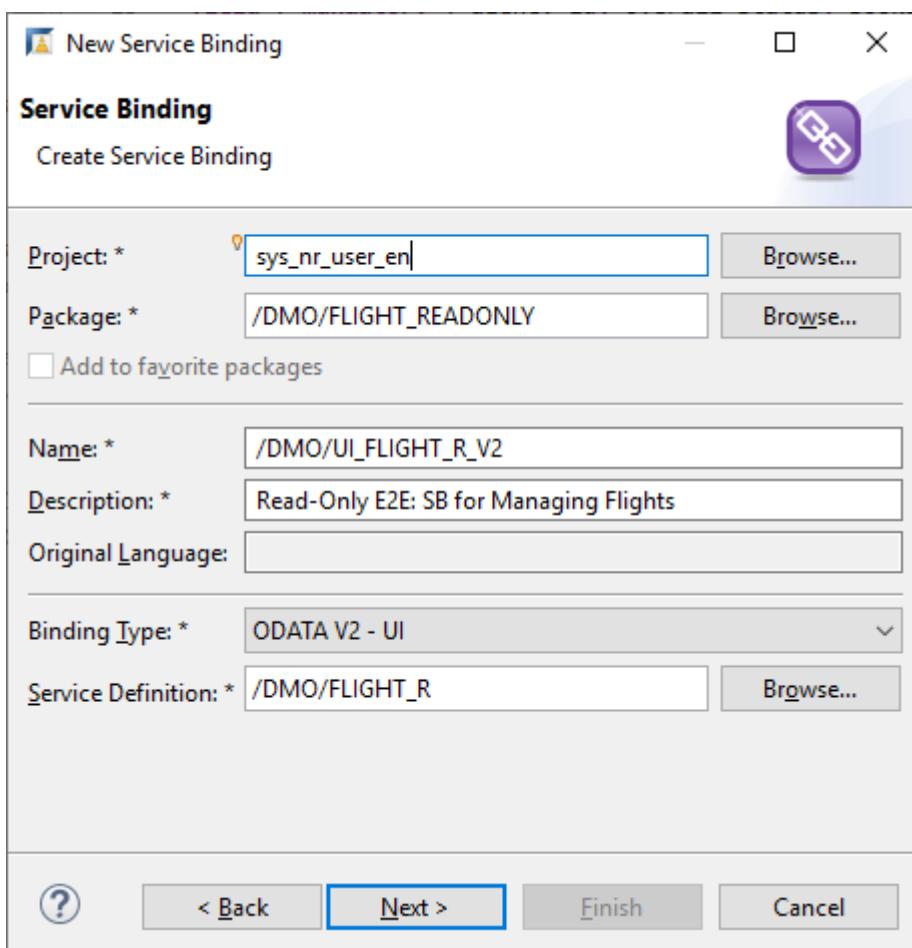
4. Select the *Binding Type ODATA V2 - UI*.

i Note

The available categories are UI and Web API. A UI-based OData service can be consumed by any SAP UI5 application. An OData service with Web API binding is exposed as an API.

This scenario is aimed at creating a UI service with OData V2.

5. Search for the *Service Definition* that you want to use as a base for your service binding: [/DMO/FLIGHT_R](#).



Wizard page when creating a service binding

6. Choose *Next*.
7. Assign a transport request.
8. Choose *Finish*.

Results

The ABAP back end creates a service binding and stores it in the ABAP Repository.

In the *Project Explorer*, the new service binding is added to the *Business Services* folder of the corresponding package node. As a result, the service binding form editor is opened and you can verify the information you have entered.

The screenshot shows the SAP ABAP Service Binding Artifact Form Editor. The title bar says "Service Binding: /DMO/UI_FLIGHT_R_V2". The main area is divided into several sections:

- General Information:** Describes general information about the service binding.
- Binding Type:** Set to "ODATA V2 - UI".
- Service Versions:** A table listing service versions associated with the service binding. It includes a "type filter text" input field and buttons for "Add..." and "Remove". The table shows one entry: Version 0001, Service Definition /DMO/FLIGHT_R.
- Service Version Details:** Shows the selected service version's details. The "Default Authorization Values" are listed as 2636A6A0B4137B4EE416075493521CHT. The "Local Service Endpoint" status is "Inactive" with an "Activate" button.
- Local Service Endpoint:** A note stating that the local service endpoint is not active and provides a link to activate it.

At the bottom center of the editor is the label "Service Binding Artifact Form Editor".

As soon as you have created the service binding for a service, the service is registered in your local system. It is not yet active.

Next Steps

Activate the service binding to make it ready for consumption.

Related Information

[Creating Service Binding \[page 769\]](#)

3.2.2.4 Publishing the OData Service Locally

To make the service ready for consumption, use the activation button in the service binding form editor.

Prerequisites

You have created the service binding and specified the binding type and category.

Context

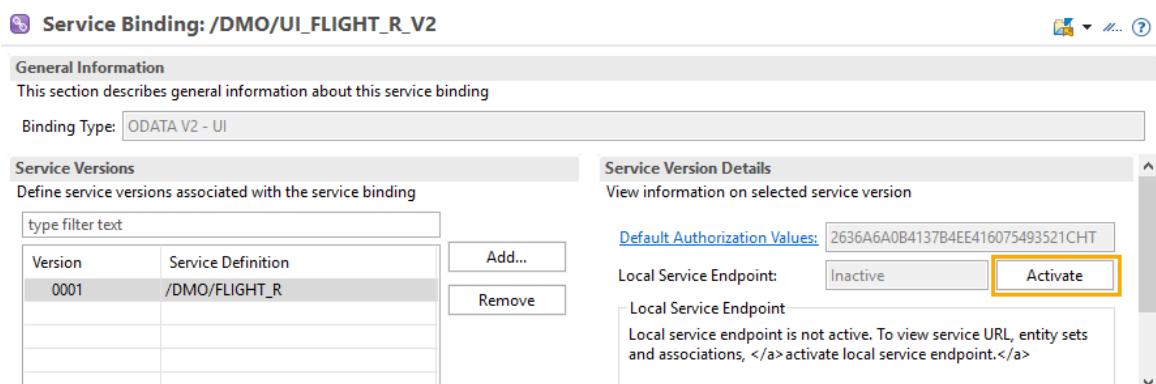
To make the service available and consumable by an OData client you have to activate the service.

Procedure

1. If you have not already done so, open the new service binding in the form editor.

The binding type and category are already defined and cannot be changed once the service binding is created. You can verify the type and category in the general information section in the form editor. As soon as you have specified the binding for the service, it is ready for publishing. The service is then available for consumption.

2. Choose the *Activate* button in the form editor.



The *Activate* button in the tool bar of ADT is not applicable to publish the service. You have to use the *Activate* button in the form editor.

Results

The OData service `/DMO/UI_FLIGHT_R_V2` is published locally, which means that it is activated in SAP Gateway. The service is bound to the protocol OData V2 for the category UI. This means it can now be consumed by a SAPUI5 application.

The screenshot shows the SAP Service Binding form editor for a service named /DMO/UI_FLIGHT_R_V2. On the left, under 'Service Versions', there is a table with one entry: Version 0001 and Service Definition /DMO/FLIGHT_R. On the right, under 'Service Version Details', there are fields for 'Default Authorization Values' (containing a long GUID), 'Local Service Endpoint' (set to 'Active' with a 'Deactivate' button), 'Service URL' (set to /sap/opu/odata/DMO/UI_FLIGHT_R_V2), and 'Entity Set and Association' (listing 'Connection').

The binding type and service information is displayed in the service binding form editor

On the left side of the form editor, the service list with the version and the service definition is filled. The right side of the form editor shows the service details. It provides a URL to view the metadata of the service and lists the entity sets that are exposed for the service. The service contains the entities that you have exposed in the service definition. The service binding editor shows the names that you assigned as alias.

Related Information

[Using Service Binding Editor \[page 771\]](#)

3.2.2.5 Verifying the OData Metadata

Use the URI in the service binding form editor to check the metadata document of the published OData service.

Prerequisites

You have published an OData service using a service binding.

Context

In the previous steps we defined an OData service and published it. It is now ready for to be consumed by an HTTP protocol. To verify the data that the OData service exposes, the service offers a metadata document in which all relevant service elements are listed.

Procedure

1. If you have not already done so, open the service binding for the relevant service.
2. To open the service document of the OData service, choose the link to the service URL ([/sap/opu/odata/sap/DMO/UI_FLIGHT_R_V2](#)) that is provided in the form editor for the relevant line in the service details section.

A browser opens that displays the service document.

3. Add `/$metadata` to the URI to view the metadata of the OData service.

`.../sap/opu/odata/DMO/UI_FLIGHT_R_V2/$metadata`

The metadata document displays the relevant information that the OData service provides for an OData client in a CSDL (Common Schema Definition Language).

i Note

As labels are language dependent, they are only displayed if the language of the browser and the maintained data elements are in the same language, or if a fallback language matches the browser configurations.

```
<?xml version="1.0" encoding="UTF-8"?>
<edmx:Edmx xmlns:sap="http://www.sap.com/Protocols/SAPData" xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
  xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx" Version="1.0">
  + <edmx:Reference xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Uri="https://HOST/sap/opu/odata/IWFND/CATALOGSERVICE;v=2/Vocabularies
    (TechnicalName="%2FIVBEP%2FVOC_COMMON",Version="0001",SAP_Origin="")/$value">
  + <edmx:Reference xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Uri="https://HOST/sap/opu/odata/IWFND/CATALOGSERVICE;v=2/Vocabularies
    (TechnicalName="%2FIVBEP%2FVOC_CAPABILITIES",Version="0001",SAP_Origin="")/$value">
  + <edmx:Reference xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Uri="https://HOST/sap/opu/odata/IWFND/CATALOGSERVICE;v=2/Vocabularies
    (TechnicalName="%2FIVBEP%2FVOC_COMMUNICATION",Version="0001",SAP_Origin="")/$value">
  - <edmx:DataServices m:DataServiceVersion="2.0">
    - <Schema xml:lang="en" Namespace="cds_xdmoflight_r_sd" xmlns="http://schemas.microsoft.com/ado/2008/09/edm" sap:schema-version="1">
      - <EntityType sap:content-version="1" sap:label="Read-Only E2E: Data Model Connection" Name="ConnectionsType">
        - <Key>
          <PropertyRef Name="CarrierID"/>
          <PropertyRef Name="ConnectionID"/>
        </Key>
        <Property sap:label="Airline ID" Name="CarrierID" sap:quickinfo="Flight Reference Scenario: Carrier ID" sap:display-format="UpperCase" MaxLength="3"
          Nullable="false" Type="Edm.String"/>
        <Property sap:label="Flight Number" Name="ConnectionID" sap:quickinfo="Flight Reference Scenario: Connection ID" sap:display-format="NonNegative"
          MaxLength="4" Nullable="false" Type="Edm.String"/>
        <Property sap:label="Departure Airport" Name="DepartureAirport" sap:quickinfo="Flight Reference Scenario: From Airport" sap:display-format="UpperCase"
          MaxLength="3" Nullable="false" Type="Edm.String"/>
        <Property sap:label="Destination Airport" Name="DestinationAirport" sap:quickinfo="Flight Reference Scenario: To Airport" sap:display-format="UpperCase"
          MaxLength="3" Nullable="false" Type="Edm.String"/>
        <Property sap:label="Departure" Name="DepartureTime" sap:quickinfo="Flight Reference Scenario: Departure Time" Type="Edm.Time" Precision="0"/>
        <Property sap:label="Arrival" Name="ArrivalTime" sap:quickinfo="Flight Reference Scenario: Arrival Time" Type="Edm.Time" Precision="0"/>
        <Property sap:label="Flight Distance" Name="Distance" sap:quickinfo="Flight Reference Scenario: Flight Distance" Type="Edm.Int32"
          sap:unit="DistanceUnit"/>
        <Property sap:label="Internal UoM" Name="DistanceUnit" sap:quickinfo="Unit of Measurement" MaxLength="3" Type="Edm.String" sap:semantics="unit-of-
          measure"/>
      </EntityType>
      + <EntityContainer Name="cds_xdmoflight_r_sd_Entities" sap:supported-formats="atom json xlsx" m:IsDefaultEntityContainer="true">
        <atom:link xmlns:atom="http://www.w3.org/2005/Atom" href="https://HOST/sap/opu/odata/DMO/FLIGHT_R_SB_UI_V2/$metadata" rel="self"/>
        <atom:link xmlns:atom="http://www.w3.org/2005/Atom" href="https://HOST/sap/opu/odata/DMO/FLIGHT_R_SB_UI_V2/$metadata" rel="latest-
          version"/>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

OData metadata

i Note

Depending on your browser and the xml format you choose, the layout of the metadata might differ.

For the described scenario, the following OData annotations are relevant:

- **EntityType:** Introduces a CDS entity that is exposed for the service.
sap:label: Provides a semantic description for the entity type. It retrieves the description that was entered in the wizard for the data definition as no other label is defined.
Name: Specifies the name of the OData entity. It uses the name of the CDS entity and attaches *Type*. If an alias is used in the service definition, it uses the alias.
- **Key:** Introduces the OData properties that are specified as keys for the OData entities. If the service is based on CDS entities, it uses the keys of the CDS entities.

Property: Introduces an OData property that is exposed in the service. If the service is based on a CDS entity, it uses the elements of the CDS view as properties.

sap:label: Provides a more informative description than just the name of the property. It retrieves the field label text of the data element if the CDS element is not labeled differently.

Name: Specifies the name of the OData property. The service uses the name of the CDS elements. It retrieves the alias if there is one.

sap:quickinfo: Provides a semantic description for the property. It retrieves the description of the data element that is used in the database table /dmo/connection if no other description is defined.

sap:unit: Specifies that the respective OData property describes an amount whose unit is provided with the referenced property. In this case, as we have defined it in CDS with semantics annotations, the property DistanceUnit provides the unit for the Distance.

sap:semantics DistanceUnit only contains currency codes. This information is taken from the data element that is used for the database table /dmo/connection, which is stored in ABAP Dictionary.

i Note

The information that is taken from the data elements can be checked in the data definition. Click a CDS element in the data definition and press **F2**. A pop-up opens and you can navigate to all the underlying elements, displaying the semantic information for the respective OData property.

Next Steps

To check the output of a *SAP Fiori UI* you can preview the app with the previewing functionality of the service binding.

3.2.2.6 Previewing the Resulting UI Service

Use the preview function in the service binding to check how the UI of a Fiori application looks like.

Prerequisites

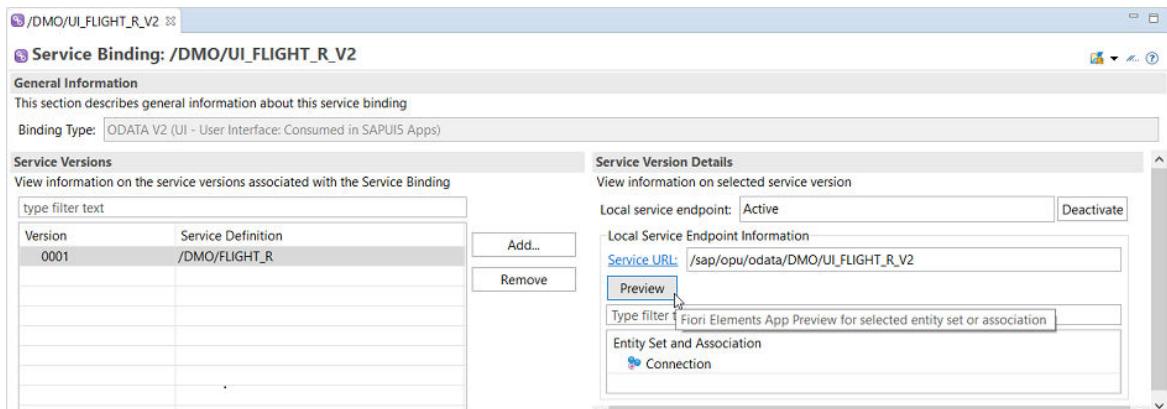
You have published an OData service using a service binding.

Context

The published OData service is ready to be consumed by an HTTP protocol. You can set up a Fiori application based on this service. The service binding artifact offers a tool which you can use to preview a simple list reporting Fiori application.

Procedure

1. If you have not yet already done so, open the service binding for the relevant service.
2. To open the Fiori Elements app preview in the service information section, select the relevant entity set (*Connection*) and choose the **Preview** button.



3. Choose **Open Fiori Elements App Preview**.
Your internet browser opens a new tab and requests authorization for the system.
4. Enter your user name and password for the relevant system.
You now have access to the system and the Fiori Elements app UI is displayed. The columns of the elements that you have in the CDS views appear. The app does not show any data yet.
5. To display data in the list report, first select the items that you want to display by clicking the configuration button and choosing the elements from the column section.
You need to select at least one element, otherwise you get an error message when retrieving the data.
6. Choose **Go** to display the data of the items you selected.

Results

The Fiori Elements App preview opens in your browser. You see the connection data that you implemented in the CDS view. The following image displays the list report when selected all available fields.

The screenshot shows a Fiori application interface with the SAP logo at the top. The title bar includes 'Standard *' and a dropdown arrow. On the right, there are buttons for 'Adapt Filters', 'Go', and a search icon. Below the header is a toolbar with navigation icons (back, forward, search) and a gear icon for settings. The main area displays a table of flight data with the following columns: Airline ID, Arrival, Departure, Departure Airport, Destination Airport, Flight Distance, Flight Number, and Internal UoM. The data rows are:

Airline ID	Arrival	Departure	Departure Airport	Destination Airport	Flight Distance	Flight Number	Internal UoM
AA	10:04:00 AM	7:13:00 AM	JFK	SFO	4,156	15	km
AA	8:03:00 AM	7:19:00 AM	MIA	HAV	520	17	km
AA	3:06:00 PM	6:40:00 AM	SFO	JFK	4,156	18	km
AA	11:19:00 PM	8:17:00 PM	MIA	EWR	1,752	322	km
AA	10:30:00 AM	6:15:00 AM	HAV	MIA	520	2678	km
AZ	10:13:00 AM	1:25:00 PM	VCE	NRT	9,595	788	km

3.2.3 Designing the User Interface for a Fiori Elements App

UI annotations can be used in CDS to configure the look of the user interface of a Fiori App.

To define annotations that concern the UI, we use CDS annotations. CDS offers the option of defining a universal setup for the presentation and order of items in the CDS layer. This is independent of the UI technology or application device, which benefits the reuse of one OData service for multiple applications. The application developer does not have to configure the setting for every application, but can reuse the settings that were defined in the back end.

You are introduced to necessary and useful UI annotations that define the presentation of your data in a UI service.

Task: [Defining UI Annotations \[page 36\]](#).

The task addresses different components of the user interface separately.

In the section [List Items \[page 36\]](#) CDS offers the option of defining a universal setup for the presentation and order of, you will learn how to order and present the columns of your list report.

The second section [List Report Header \[page 38\]](#) deals with the items in the list report header.

The section [Object Page \[page 39\]](#) describes the configuration of an object page and its items.

→ Tip

You can always check the influence of the UI annotations on the UI with the preview option in the service binding form editor.

3.2.3.1 Defining UI Annotations

The presentation and order of the CDS elements in a SAP Fiori Elements user interface is configured in CDS with annotations.

Context

You have created an OData service and published it locally. The UI can now be set up with UI annotations in the CDS layer to define a UI layout independent from the application or the user device. You can always check the influence of UI annotations by using the preview function in the service binding artifact.

List Items

Context

Using the following annotations, you specify which of the elements appear in the list report when starting the app. In addition to their order, you can also rename them if you want to display them with a name other than the name specified in the CDS entity. The columns that are shown in the UI are then predefined and you can retrieve data by choosing [GO](#) without determining the columns to be displayed.

Procedure

1. Open the CDS view for which you want to determine the list report. In our case: [/DMO/I_Connection_R](#).
2. For the headline of the list, use the annotation `@UI.headerInfo:typeNamePlural: 'name'`.
This annotation is an entity annotation because it concerns the whole entity rather than a specific element.

```
@AbapCatalog.sqlViewName: '/DMO/ICONNECT_R'  
@AbapCatalog.compiler.compareFilter: true  
@AccessControl.authorizationCheck: #NOT_REQUIRED  
@EndUserText.label: 'Read-Only E2E: Data Model Connection'  
@UI.headerInfo.typeNamePlural: 'Connections'  
define view /DMO/I_Connection_R  
...
```

3. Specify a position for each element that you want to show in the list report with the annotation `@UI.lineItem: [{ position:decfloat }]`.

i Note

The value's number does not represent an absolute measure and works as a relative value to the positions of the other elements instead. Hence, the elements are arranged in ascending order with regard to the annotation value.

...

```

define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
{
  @UI.lineItem: [ { position: 10 } ]
  key   Connection.carrier_id      as AirlineID,
  @UI.lineItem: [ { position: 20 } ]
  key   Connection.connection_id   as ConnectionID,
  @UI.lineItem: [ { position: 30 } ]
    Connection.airport_from_id as DepartureAirport,
  @UI.lineItem: [ { position: 40 } ]
    Connection.airport_to_id   as DestinationAirport,
  @UI.lineItem: [ { position: 50 } ]
    Connection.departure_time  as DepartureTime,
  @UI.lineItem: [ { position: 60 } ]
    Connection.arrival_time   as ArrivalTime,
  @Semantics.quantity.unitOfMeasure: 'DistanceUnit'
    Connection.distance        as Distance,    /** secondary
information, not to be displayed on list report entry page
  @Semantics.unitOfMeasure: true
    Connection.distance_unit  as DistanceUnit    /** secondary
information, not to be displayed on list report entry page
}

```

4. You can display the elements with a name other than the name specified in CDS by labeling them with the annotation `@UI.lineItem.label: label`. In particular, you can label element with names containing spaces. The label is displayed in the column header of the list report.

```

...
define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
{
  @UI.lineItem: [ { position: 10, label: 'Airline' } ]
  key   Connection.carrier_id      as AirlineID,
  @UI.lineItem: [ { position: 20, label:'Connection Number' } ]
  key   Connection.connection_id   as ConnectionID,
  @UI.lineItem: [ { position: 30 , label: 'Departure Airport Code' } ]
    Connection.airport_from_id as DepartureAirport,
  @UI.lineItem: [ { position: 40 , label: 'Destination Airport Code' } ]
    Connection.airport_to_id   as DestinationAirport,
  @UI.lineItem: [ { position: 50 , label: 'Departure Time' } ]
    Connection.departure_time  as DepartureTime,
  @UI.lineItem: [ { position: 60 , label: 'Arrival Time' } ]
    Connection.arrival_time   as ArrivalTime,
  @Semantics.quantity.unitOfMeasure: 'DistanceUnit'
    Connection.distance        as Distance,    /** secondary
information, not to be displayed on list report entry page
  @Semantics.unitOfMeasure: true
    Connection.distance_unit  as DistanceUnit    /** secondary
information, not to be displayed on list report entry page
}

```

Results

The source code specifies which of the elements of the CDS view `/DMO/I_Connection_R` are displayed in the list report and in which order. In addition, the list report is given the title *Connections*. When starting the app, you do not have to select columns in the settings since they are already displayed. Press the *GO* button to retrieve data.

The screenshot shows a SAP Fiori application interface. At the top, there's a header bar with the SAP logo and a search icon. Below the header, the title 'Connections' is displayed with a dropdown arrow. On the left, there's a 'Standard' dropdown and some navigation icons. On the right, there are buttons for 'Adapt Filters', 'Go', and a gear icon. The main area is a table with the following columns: Airline, Connection Number, Departure Airport Code, Destination Airport Code, Departure Time, and Arrival Time. The table contains three rows of data:

Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time
AA	17	JFK	SFO	11:00:00 AM	2:01:00 PM >
AZ	555	FCO	FRA	7:00:00 PM	9:05:00 PM >
AZ	789	TYO	FCO	11:45:00 AM	7:25:00 PM >

Below the table, a message says 'List report after UI configuration in the data definition'.

List Report Header

Context

The following annotations specify the items that are shown in the list report header.

You can define a header for the list report or you can implement selection fields on top of the list report to filter for a specific item. One selection field always refers to one element, but you can have more than one selection field in a single list report header.

Procedure

To include selection fields for the key elements in the header, use the annotation

`@UI.selectionField.position: decfloat` on the respective elements.

i Note

The value's number does not represent an absolute measure and works as a relative value to the positions of the other selection fields instead. Hence, the selection fields are arranged in ascending order with regard to the annotation value.

```
...
    @UI.lineItem: [ { position: 30 , label: 'Departure Airport Code'} ]
    @UI.selectionField: [ { position: 10 } ]
        Connection.airport_from_id as DepartureAirport,
    @UI.lineItem: [ { position: 40 , label: 'Destination Airport Code'} ]
    @UI.selectionField: [ { position: 20 } ]
        Connection.airport_to_id as DestinationAirport,
...

```

Results

The selection field annotation is used on the key elements of the CDS view to create a selection field in the header on the list report. Using these selection fields, you can filter for specific list items.

Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time
AA	17	JFK	SFO	11:00:00 AM	2:01:00 PM >
SQ	2	SIN	SFO	5:00:00 PM	7:25:00 PM >
UA	941	FRA	SFO	2:30:00 PM	5:06:00 PM >

UI with selection fields filtered for connections to a specific destination airport

Object Page

Context

Whereas the list report gives a general overview of the list items, the object page shows more detailed information about a single list item. You navigate to the object page by clicking the item in the list report.

Procedure

1. `/DMO/I_Connection_R` using the annotation `@UI.headerInfo.typeName: 'name'`.

```
@AbapCatalog.sqlViewName: '/DMO/ICONNECT_R'  
@AbapCatalog.compiler.compareFilter: true  
@AccessControl.authorizationCheck: #NOT_REQUIRED  
@EndUserText.label: 'Read-Only E2E: Data Model Flight'  
@UI.headerInfo.typeName: 'Connection'  
define view /DMO/I_Connection_R  
...
```

2. Create a standard facet for the object page with the annotation `@UI.facet.purpose: #STANDARD`. This annotation must be in the element section.

A facet is a type of section in the object page. It can contain diagrams or other information in a discrete part of the user interface.

```
define view /DMO/I_Connection_R
```

```

        as select from /dmo/connection as Connection
{
@UI.facet: [ { purpose: #STANDARD } ]
...

```

3. Specify the type of the facet. In our case, the object page displays the detailed information of one list item. Use the annotation @UI.facet.type: #IDENTIFICATION_REFERENCE.

```

define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection{
    @UI.facet: [ {
      purpose: #STANDARD,
      type:   #IDENTIFICATION_REFERENCE } ]
...

```

4. Specify a name for the object page facet header. Use the annotation @UI.facet.label: 'name'.

```

define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
{
  @UI.facet: [ {
    purpose: #STANDARD,
    type:   #IDENTIFICATION_REFERENCE,
    label:  'Connection' } ]
...

```

5. To define the position of the facet, use the annotation @UI.facet.position: decfloat.

```

define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection{
    @UI.facet: [ {
      purpose: #STANDARD,
      type:   #IDENTIFICATION_REFERENCE,
      label:  'Connection',
      position: 10 } ]
...

```

An object page of a type identification reference is created. You can now define the elements that are displayed in the object page.

6. Specify the position and the label for each element that you want to show in the object page. Use the annotations @UI.identification.position: 'decfloat' and @UI.identification.label: 'name' on each element.

```

{ ...
  @UI: { identification:[ { position: 10, label: 'Airline' } ] }
key   Connection.carrier_id      as AirlineID,
  @UI: { identification:[ { position: 20, label: 'Connection Number' } ] }
key   Connection.connection_id   as ConnectionID,
  @UI: { identification:[ { position: 30, label: 'Departure Airport
Code' } ] }
  @UI.selectionField: [ { position: 10 } ]
    Connection.airport_from_id as DepartureAirport,
  @UI: { identification:[ { position: 40, label: 'Destination Airport
Code' } ] }
  @UI.selectionField: [ { position: 20 } ]
    Connection.airport_to_id   as DestinationAirport,
  @UI: { identification:[ { position: 50, label: 'Departure Time' } ] }
    Connection.departure_time  as DepartureTime,
  @UI: { identification:[ { position: 60, label: 'Arrival Time' } ] }
    Connection.arrival_time    as ArrivalTime,
  @Semantics.quantity.unitOfMeasure: 'DistanceUnit'
  @UI: { identification:[ { position: 70, label: 'Distance' } ] }

```

```

        Connection.distance      as Distance,      /** secondary
information, not to be displayed on list report entry page
@Semantics.unitOfMeasure: true
        Connection.distance_unit as DistanceUnit /** secondary
information, not to be displayed on list report entry page
}

```

The following image displays the object page after clicking the connection item [JL 407](#).

General Information

General Information

Airline:
JL

Connection Number:
407

Departure Airport Code:
NRT

Destination Airport Code:
FRA

Departure Time:
1:30:00 PM

Arrival Time:
5:35:00 PM

Distance:
9,100 km

Object page with identification reference

7. Activate the CDS view.

Results

The resulting source code, including all annotations that are relevant for the UI in the data definition, is as follows:

```

@AbapCatalog.sqlViewName: '/DMO/ICONNECT_R'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E: Data Model Connection'
@UI.headerInfo: { typeName: 'Connection',
                  typeNamePlural: 'Connections' }
define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
{
  @UI.facet: [
    { id: 'Connection',
      purpose: '#STANDARD, }

```

```

type:      #IDENTIFICATION_REFERENCE,
label:    'Connection' ]
@UI.lineItem: [ { position: 10, label: 'Airline' } ]
@UI: { identification:[ { position: 10, label: 'Airline' } ] }
key Connection.carrier_id      as AirlineID,
@UI.lineItem: [ { position: 20, label:'Connection Number' } ]
@UI: { identification:[ { position: 20, label: 'Connection Number' } ] }
key Connection.connection_id   as ConnectionID,
@UI: { identification:[ { position: 30, label: 'Departure Airport
Code' } ]
@UI.lineItem: [ { position: 40 , label: 'Destination Airport Code'} ]
@UI.selectionField: [ { position: 10 } ]
Connection.airport_from_id as DepartureAirport,
@UI.lineItem: [ { position: 40 , label: 'Destination Airport Code'} ]
@UI: { identification:[ { position: 40, label: 'Destination Airport
Code' } ]
@UI.selectionField: [ { position: 20 } ] ]
Connection.airport_to_id   as DestinationAirport,
@UI.lineItem: [ { position: 50 , label: 'Departure Time' } ]
@UI: { identification:[ { position: 50, label: 'Departure Time' } ] }
Connection.departure_time  as DepartureTime,
@UI.lineItem: [ { position: 60 , label: 'Arrival Time' } ]
@UI: { identification:[ { position: 60, label: 'Arrival Time' } ] }
Connection.arrival_time    as ArrivalTime,
@Semantics.quantity.unitOfMeasure: 'DistanceUnit'
@UI: { identification:[ { position: 70, label: 'Distance' } ] }
Connection.distance        as Distance,           /** secondary
information, not to be displayed on list report entry page
@Semantics.unitOfMeasure: true
Connection.distance_unit   as DistanceUnit      /** information is
given with element Distance via semantic connection
}

```

Related Information

[UI Annotations \[page 610\]](#)

4 Concepts

The content in **Concepts** provides background information about the ABAP RESTful Programming Model and helps you to understand the concepts behind it.

The ABAP RESTful Programming Model has unified the development of [OData services \[page 816\]](#) with ABAP. It is based on three pillars that facilitate your development.

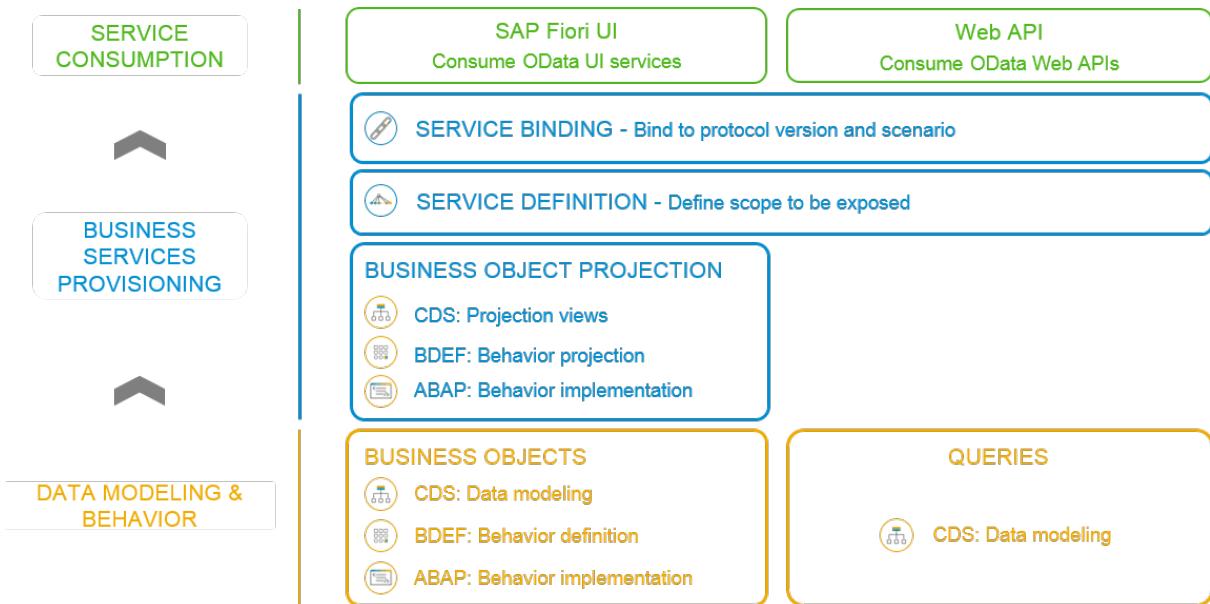
- **Tools:** The approach to integrate all implementation tasks in one development environment optimizes the development flow and offers an end-to-end experience in one tool environment. New development artifacts support the application developer to develop in a standardized way.
- **Language:** The ABAP language has been aligned and extended to support the development with the ABAP RESTful Programming Model, together with CDS. The application developer uses typed APIs for standard implementation tasks and benefits from auto-completion, element information, and static code checks.
- **Frameworks:** Powerful frameworks represent another important pillar of the ABAP RESTful Programming Model. They assume standard implementation tasks with options for the application developer to use dedicated code exits for application-specific business logic.

Learn how these pillars are incorporated into the architecture of the ABAP RESTful Programming Model in the following topics.

Design Time

The following diagram structures the development of an OData service from a design time perspective. In other words, it displays the major development artifacts that you have to deal with during the creation of an OData service with the ABAP RESTful Programming Model. The diagram takes a bottom-up approach that resembles the development flow. The main development tasks can be categorized in three layers, data modeling and behavior, business services provisioning and service consumption.

Hover over the building blocks and get more information and click to find out detailed information about the components.

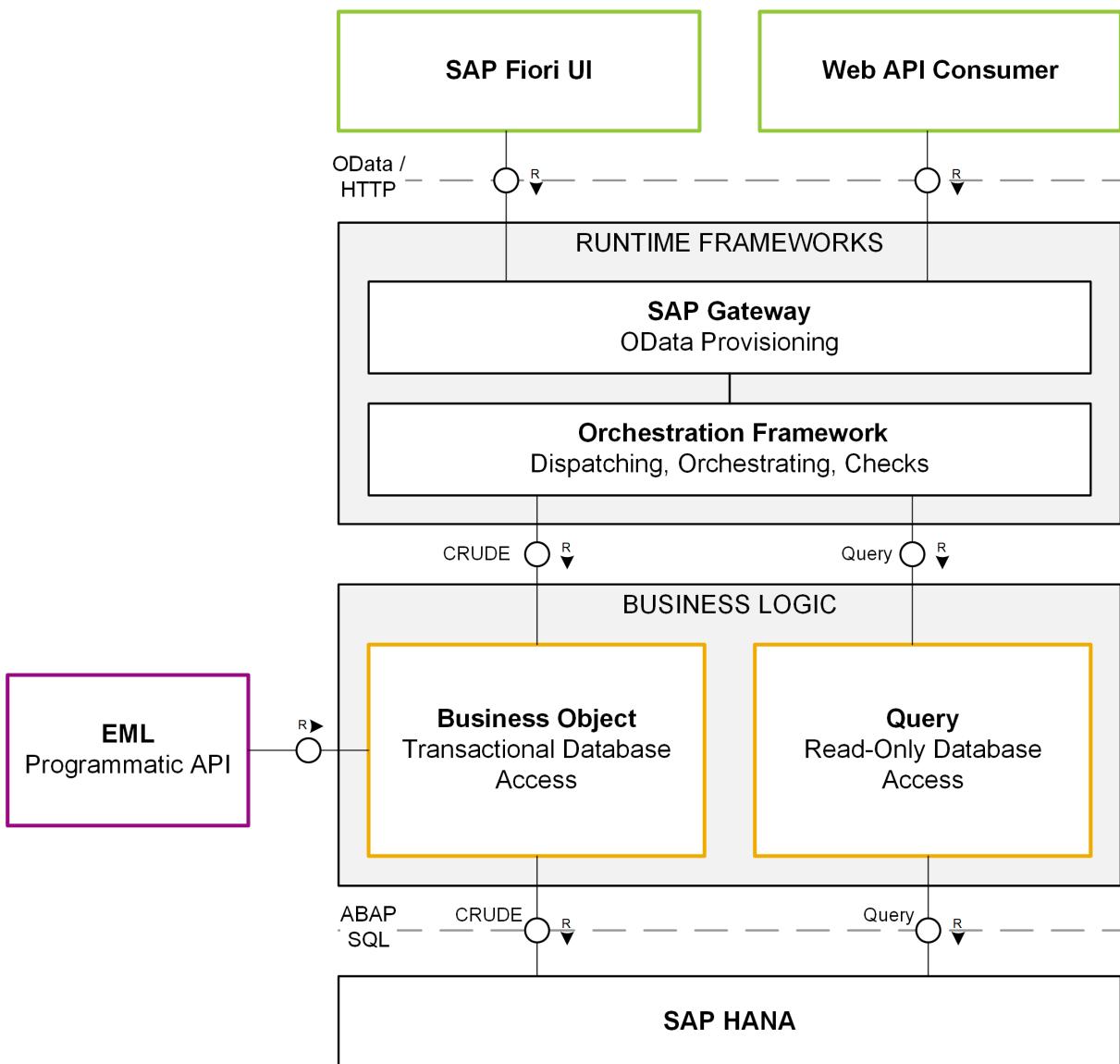


- Data Modeling and Behavior [page 46]
- Business Object [page 55]
- Query [page 47]
- Business Service [page 98]
- Service Definition [page 108]
- Service Binding [page 111]
- Service Consumption [page 113]
- Web API [page 115]
- UI service [page 114]
- Business Object Projection [page 100]

Runtime

The following diagram provides a runtime perspective of the ABAP RESTful Programming Model. Runtime objects are necessary components to run an application. This runtime stack is illustrated in a top-down approach. An OData client sends a request, which is then passed to the generic runtime frameworks. These frameworks prepare a consumable request for ABAP code and dispatch it to the relevant business logic component. The request is executed by the [business object \[page 805\]](#) (BO) when data is modified or by the [query \[page 817\]](#) if data is only read from the data source.

Hover over the building blocks to get more information and click to navigate to more detailed information about the components.



- [UI service \[page 114\]](#)
- [Web API \[page 115\]](#)
- [Runtime Frameworks \[page 115\]](#)
- [Business Object \[page 55\]](#)
- [Query \[page 47\]](#)
- [Entity Manipulation Language \(EML\) \[page 116\]](#)

A more detailed description is available for the following concepts:

- [Data Modeling and Behavior \[page 46\]](#)
 - [Business Object \[page 55\]](#)
 - [Business Object Projection \[page 100\]](#)
 - [Query \[page 47\]](#)

- [Business Service \[page 98\]](#)
 - [Service Definition \[page 108\]](#)
 - [Service Binding \[page 111\]](#)
- [Service Consumption \[page 113\]](#)
- [Runtime Frameworks \[page 115\]](#)
- [Entity Manipulation Language \(EML\) \[page 116\]](#)

4.1 Data Modeling and Behavior

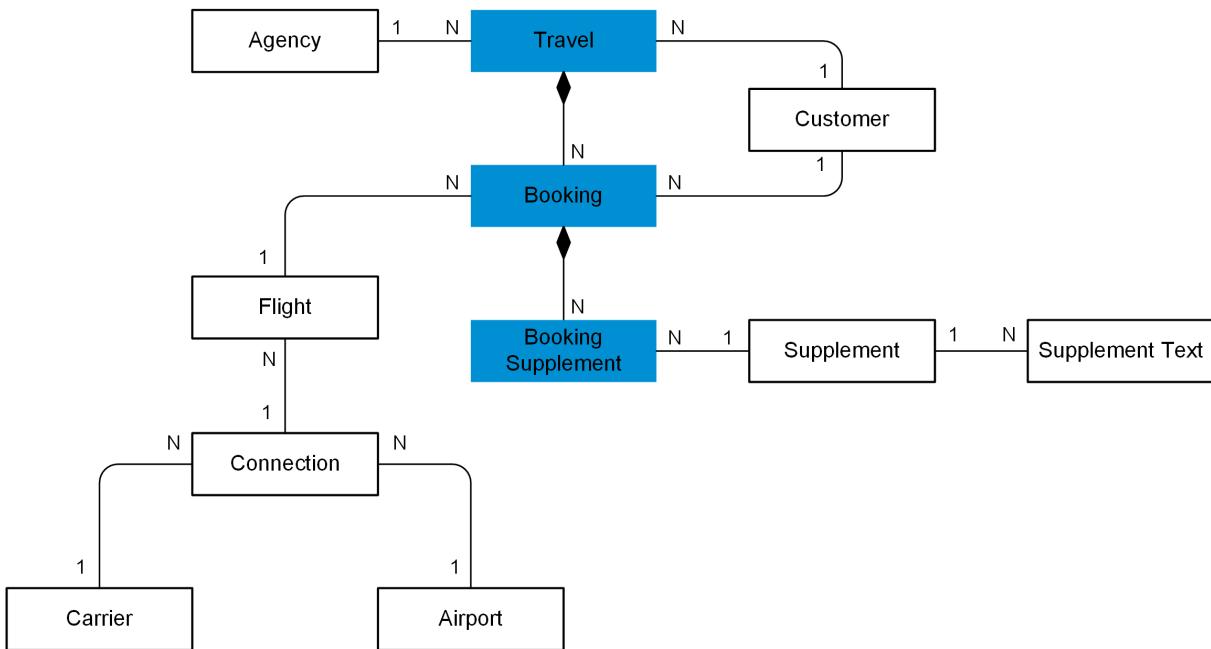
The layer of data modeling and behavior deals with data and the corresponding business logic.

Data Model

The [data model \[page 811\]](#) comprises the description of the different entities involved in a business scenario, for example `travel` and `booking`, and their relationships, for example the parent-child relationship between `travel` and `booking`. The ABAP RESTful Programming Model uses [CDS \[page 808\]](#) to define and organize the data model. CDS provides a framework for defining and consuming semantic data models. These data models have a physical representation in the database in the form of database views which are automatically created based on a CDS data model. Every real-world entity is represented by one [CDS entity \[page 809\]](#). View building capabilities allow you to define application-specific characteristics in the data model. That means, CDS entities are the fundamental building blocks for your application. When using the CDS entity for a data selection, the data access is executed by the SQL-view, which is defined in the CDS entity.

Depending on the use case, data models support transactional access or query access to the database. Thus, data models are used in [business objects \[page 805\]](#) or [queries \[page 817\]](#) respectively.

The following diagram gives you an overview of the data model that is used in the development guides of this documentation. Every block refers to one database table view and the respective CDS entity. The blue boxes represent a `Travel` business object, with its child entities `Booking` and `Booking Supplement`. The white boxes represent the entities that are not part of the business object, but support with value helps or text associations. For read-only access to the database, that is simple data retrieval, the data model is used for the query.



Data Model Used in the Development Guides of this Documentation

Behavior

The behavior describes what can be done with the data model, for example if the data can be updated.

In transactional scenarios, the business object behavior defines which operations and what characteristics belong to a business object. For read-only scenarios, the behavior of the data model is defined by the query capabilities, for example if the data is filterable.

Learn more about the business object and the query in the following topics.

[Business Object \[page 55\]](#)

[Query \[page 47\]](#)

4.1.1 Query

A query is the connecting interface for read-only access to the database in [OData services \[page 816\]](#). It is used for list reports or analytical reports to process data.

As the non-transactional counterpart of a [business object \[page 805\]](#), it consists of a [data model \[page 811\]](#), generic and modeled query capabilities and a runtime. This threefold division is known from the BO concept. However, a query provides only read access to the database. Its runtime never modifies data, but only executes structured data retrieval, for example for filtering.

Data Model

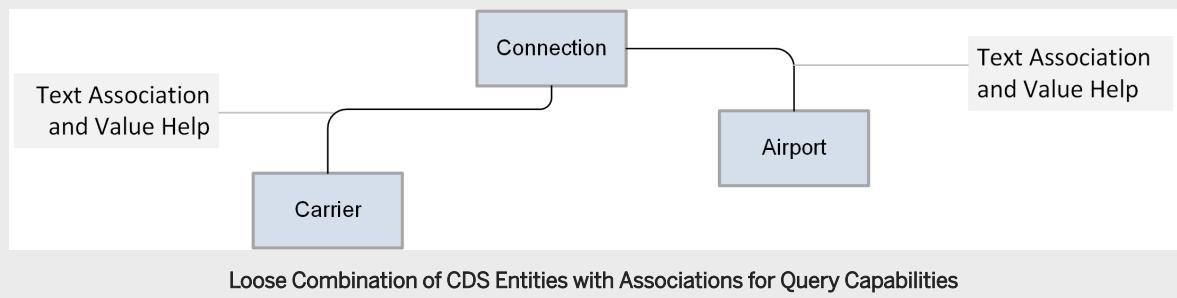
The data model for a query is provided with [CDS entities \[page 808\]](#). They structure and group database fields to execute query capabilities on them. The SQL select to retrieve data from the database is generically integrated in the CDS view.

A query operates on a loose combination of CDS entities. Each entity represents a real-world artifact and contains the relevant information about it. For example, the information of [Flight Connections](#) or [Airports](#) is manifested in CDS entities. The entities are not strictly structured. Their connections, which are modeled with [associations \[page 804\]](#), only provide a functional relationship. In other words, only if data from other entities is needed for a certain functionality is the association necessary. In contrast to BO [compositions \[page 808\]](#), there is no existential relationship for such associations.

• Example

When providing text for ID elements, you need an association to a text providing CDS entity to get the text from there. The association is only relevant to get information from the text provider. There is no other structural relationship.

In case of [Flight Connections](#), an association is created to get the information about the long text of the [airport ID](#) in the [Airport](#) entity and the full name of the airline in the [Carrier](#) entity.



Query Capabilities

Query capabilities provide read access to the database and process data to structure them for a certain output. In contrast to [BO behavior \[page 806\]](#), the capabilities do not need to be defined in a separate artifact. Some of the query capabilities which result from OData query options are generically available and applicable. The query framework provides the SQL statement to retrieve the structured data for these capabilities, for example in filtering.

Other capabilities are explicitly modeled by the developer in the source code of the CDS entity. These capabilities depend on associated CDS entities. The application developer has to define this dependency in the CDS entity. In this case, [CDS annotations \[page 809\]](#) indicate which CDS entity or element is involved, as it is the case for text or value help provisioning. Most of the explicitly modeled capabilities are based on the query of associated CDS entities.

The following table lists the available query capabilities.

Generally Applicable Capabilities	Explicitly Modeled Capabilities
paging	search
sorting	value help
filtering	aggregation
counting	text provisioning
column selections	

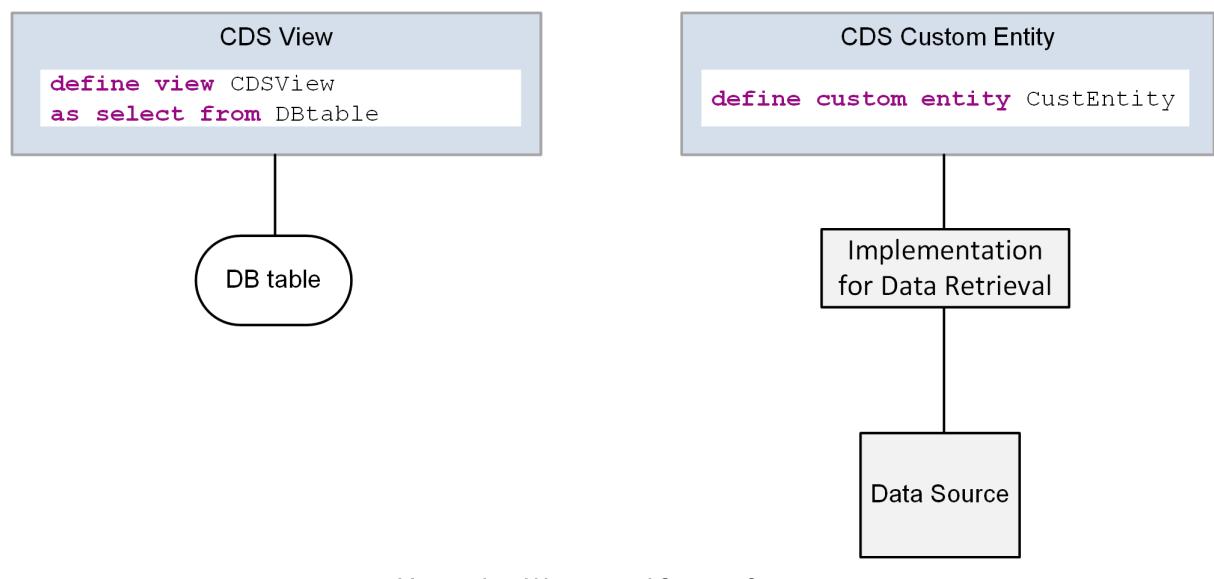
All of these features do not modify data on the database but process data to structure them for a certain output.

Query Runtime

The runtime of a query is usually managed by the query framework ([SADL \[page 818\]](#)). The framework takes into account all query capabilities that are mentioned previously. The application developer does not have to deal with the construction of the SQL statement to retrieve data from a database table. The data model for the managed runtime is provided in [CDS entity \[page 809\]](#).

There is also the option to handle the query manually. We speak of an [unmanaged query \[page 821\]](#) in this case. An unmanaged query can be used, for example, if the data source of a query is not a database table. That means, the framework cannot provide the SQL statement to access the database. Instead, the application developer needs to implement every query capability to retrieve the data matching the OData request. For the unmanaged implementation type, the data model is manifested in a CDS custom entity. In contrast to CDS views, [CDS custom entities \[page 809\]](#) do not provide an SQL SELECT for the data retrieval from the database. A [query implementation class \[page 818\]](#) must be implemented to execute the data retrieval.

The following diagram exemplifies the runtime of a managed and an unmanaged query.



Managed and Unmanaged Query in Contrast

For more information about the query runtime, see [Query Runtime Implementation \[page 50\]](#).

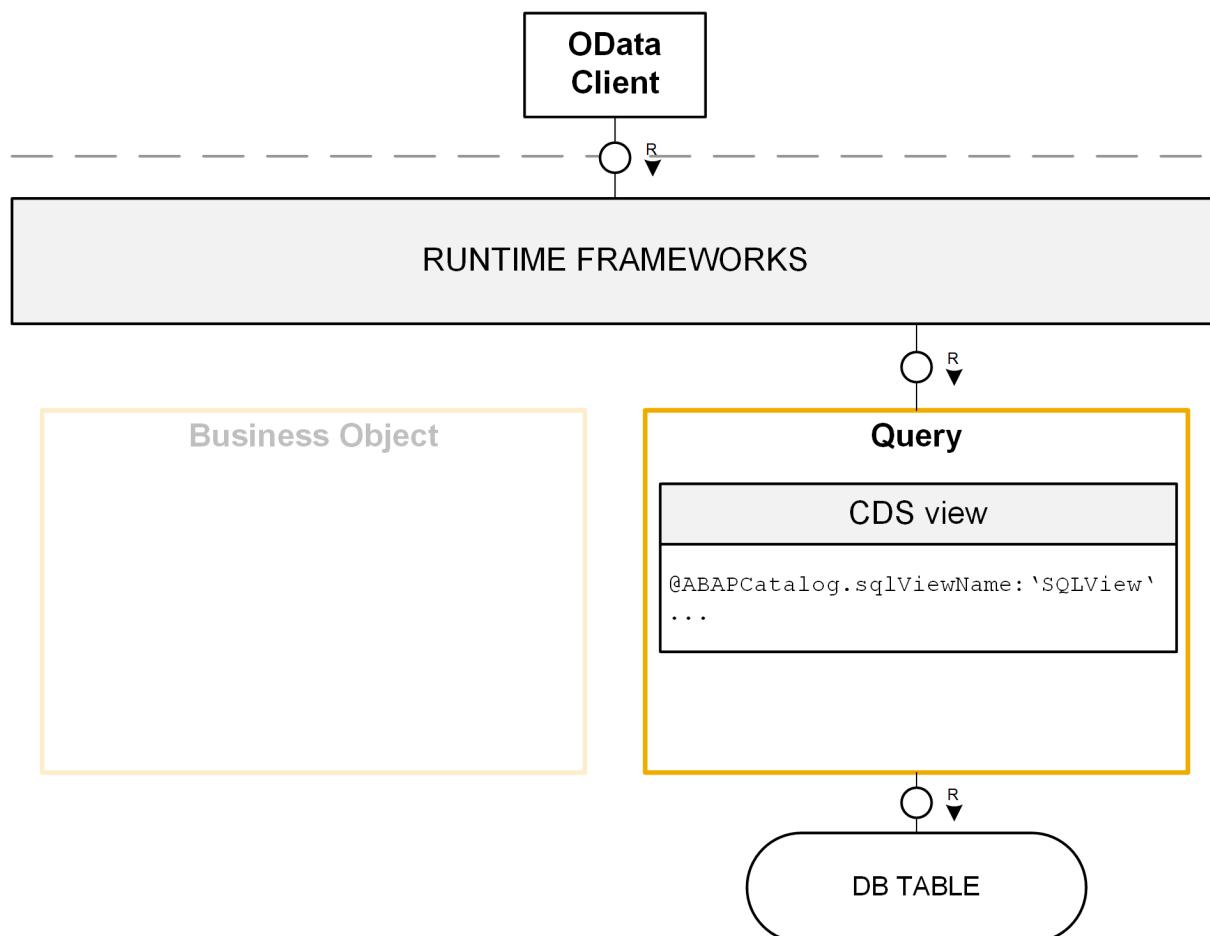
For a detailed runtime diagram, see [Query Runtime \[page 53\]](#).

4.1.1.1 Query Runtime Implementation

Managed Query

The default case for a query is the managed implementation type. In this case, the orchestration framework manages the data access to the database. Query capabilities, which result from OData query options (\$orderby, \$top, \$skip ...) are considered, as well as possible authorizations, which are derived from attached access control. The framework creates an SQL statement for the query that is executed based on the definition in the CDS source code, the query capabilities and the authorizations. For the runtime of the managed query, the application developer does not have to implement anything. The application development tasks are limited to defining the data model and the related access controls during the design time.

The following diagram illustrates the runtime of a query.



Access controls are not illustrated in the preceding diagram. If authorizations are modeled with access controls, they would automatically be evaluated.

Managed Query - Runtime

• Example

An OData request with the query option `$filter` reaches an [OData service \[page 816\]](#). Once transformed into an ABAP consumable object, the orchestration framework triggers the query to be executed. Then, the query framework creates the SQL statement to select the required data from the database. In this case, the query framework extends the SQL statement with a where clause to only select the data sets that match the filter condition. In case, access controls are involved, the query framework also evaluates the involved authorizations.

For a detailed runtime diagram, see [Query Runtime \[page 53\]](#).

Unmanaged Query

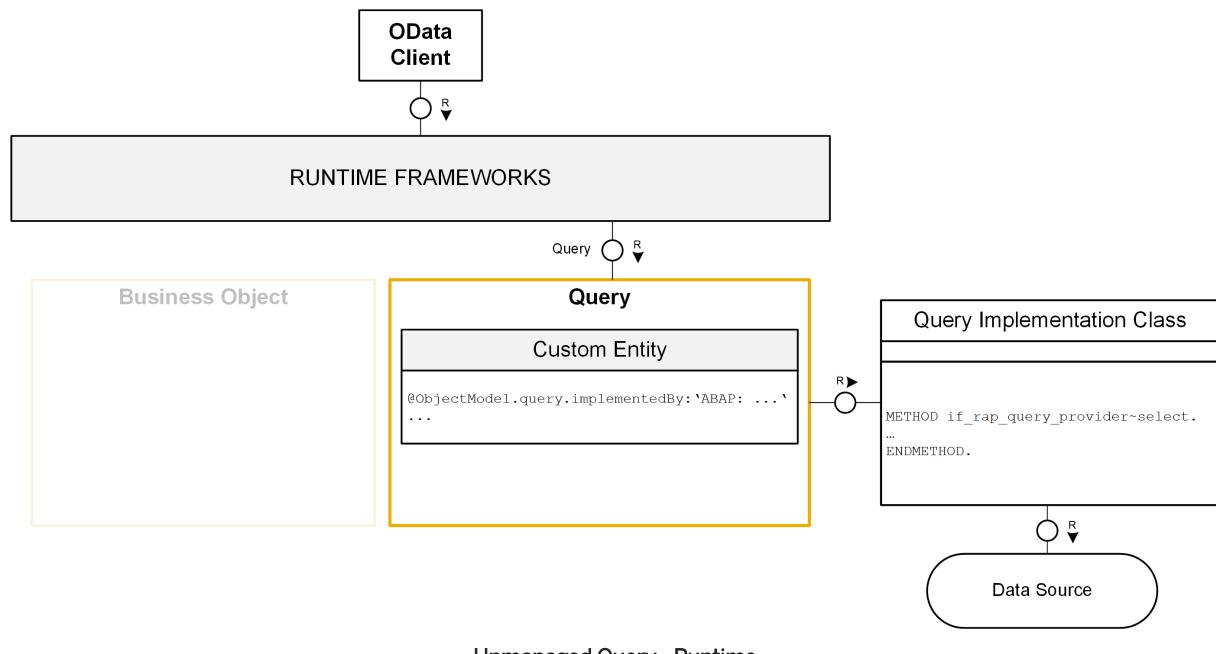
The unmanaged implementation type for a query is used when the standard SQL push-down by the query framework is not sufficient or not usable at all.

Use cases for unmanaged queries are

- the data source for an OData request is not a database table, but, for example another OData service, which is reached by an OData client proxy,
- performance optimization with application specific handling,
- using AMDPs with some query push-down parameters in the SQL script implementation,
- forwarding the call to the analytical engines, or
- enrichment of query result data on property or row level, for example when splitting rows for intermediate sums or condensing the filter result.

The unmanaged query is protocol agnostic. That means, like managed queries, it can be reused for multiple scenarios.

The following diagram illustrates the runtime of an unmanaged query.



Explanation

The data model for an unmanaged query must be defined in a [CDS custom entity \[page 809\]](#). A custom entity defines the structure of the data returned by the query. This is done using CDS syntax in a CDS data definition (DDLS). A CDS custom entity does not have an SQL view to select data from a database. Instead, the custom entity specifies an ABAP class that implements the query. The entity annotation `@ObjectModel.query.implementedBy: 'ABAP:<Query_Impl_Class>'` is used to reference the [query implementation class \[page 818\]](#) in the data definition of the CDS custom query. This annotation is evaluated when the unmanaged query is executed whereby the query implementation class is called to perform the query.

Since no SQL artifact is generated for custom entities and the query is implemented in ABAP, custom entities cannot be used in ABAP SQL or in SQL joins in [data definitions \[page 811\]](#).

The syntax of a CDS custom entity is the following:

```
@EndUserText.label: 'EndUserText'  
@ObjectModel.query.implementedBy: 'ABAP:<Query_Impl_Class>'  
[define] [root] custom entity CustomEntityName  
    [ with parameters  
        ParamName : dtype [, ...]      ]  
{  
    [ @element_annot ]  
    [key] EleName : dtype;  
        EleName : dtype;  
...  
    [ _Assoc : association [cardinality] to TargetEntity on CondExp [with default  
filter CondExp ]   ];  
    [ _Comp : composition [cardinality] of TargetEntity   ];  
    [ _ParentAssoc : association to parent Parent on CondExp   ];  
}
```

A CDS custom entity can have parameters, elements and associations. Like in CDS views, it lists the elements that are used in the data model. For each element, the data type must be specified as it cannot be retrieved from an underlying database representation.

A custom entity can be an entity in a business object, for example a root, a parent, or a child entity using `root` and `composition` relationships. Custom entities may also be used as targets in the definition of [associations \[page 804\]](#) and define associations as a source.

A custom entity cannot be used in ABAP SQL `SELECT` executions as they do not have a database representation. In particular, you cannot use elements of an associated custom entity in the element list of the source CDS entity.

Unmanaged queries are implemented in ABAP classes. The [query implementation class \[page 818\]](#) implements a predefined ABAP interface (`IF_RAP_QUERY_PROVIDER`) to ensure that the required basic OData support is enabled. The interface has a `select` method which imports an interface instance for the request data and one for the response data.

Access control needs to be implemented manually in the query implementation class to ensure that only those records are returned the user is allowed to access. You cannot use an access control object for a custom entity.

In contrast to the managed query, the application developer has to take care for every supported query option in the query implementation class, including possible authorizations that are also implemented in the query implementation class.

• Example

An example on how to use a CDS custom entity and implement an unmanaged query with the interface `IF_RAP_QUERY_PROVIDER` in a query implementation class is given in [Implementing an Unmanaged Query \[page 494\]](#).

The use case of an unmanaged query in combination with the client proxy is explained in the develop scenario [Developing a UI Service with Access to a Remote Service \[page 369\]](#).

For more information about the interface `IF_RAP_QUERY_PROVIDER`, see [Unmanaged Query API \[page 741\]](#).

i Note

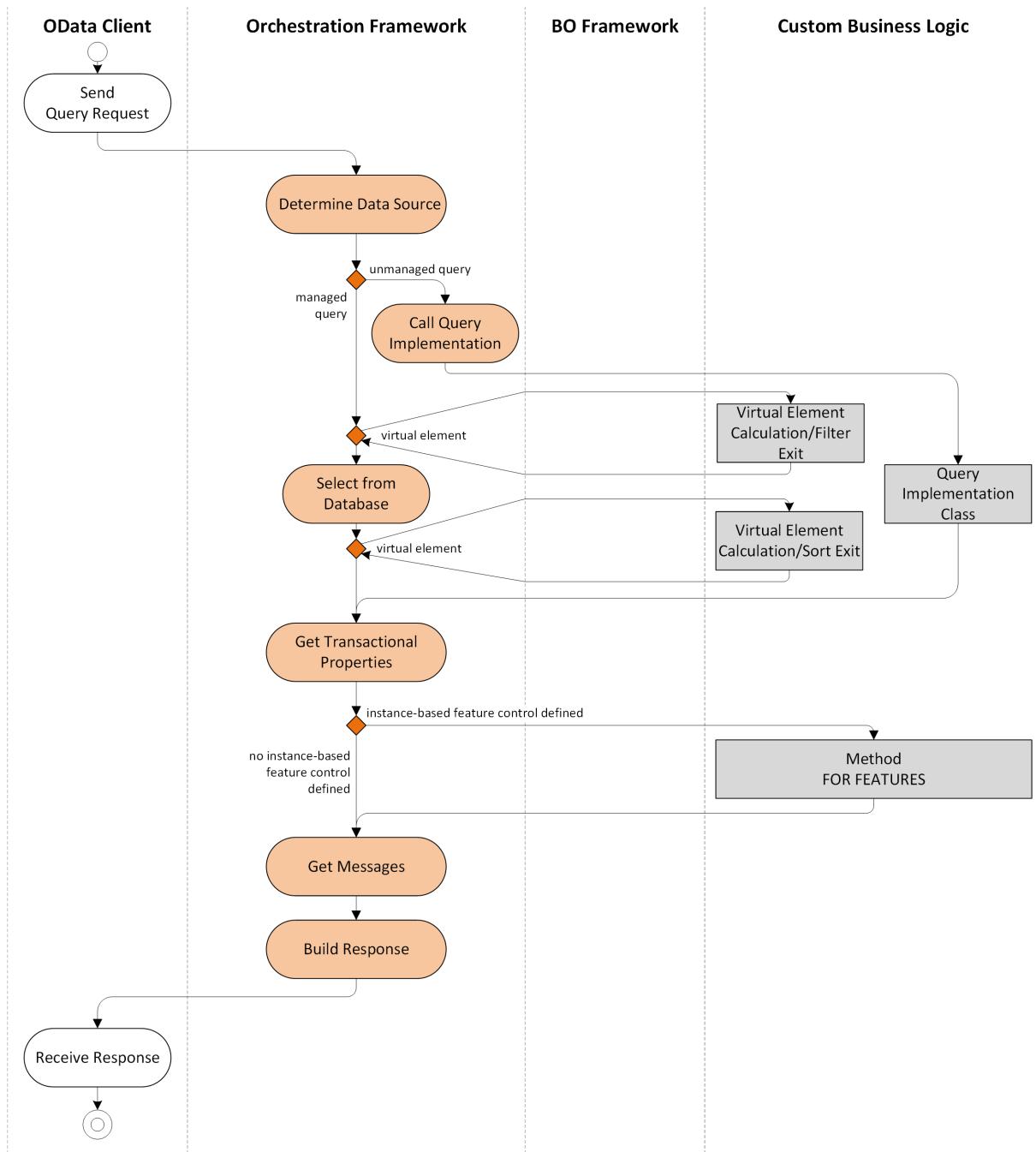
Custom Entities cannot be projected in CDS projection views.

For a detailed runtime diagram, see [Query Runtime \[page 53\]](#).

4.1.1.1 Query Runtime

In RAP, a query is the non-transactional read operation to retrieve data directly from the database.

The following runtime diagram illustrates the main agents' activities when an OData GET (GET ENTITYSET) request is sent.



- Update Operation [page 67]

- [Update Operation \[page 67\]](#)

i Note

This diagram reflects the activities for requesting an entity set (GET ENTITYSET). If you request a single entity with a key, exceptions that are not depicted in the diagram may arise if, for example, the requested key does not exist.

4.1.2 Business Object

Introduction

A **business object (BO)** is a common term to represent a real-world artifact in enterprise application development such as the *Product*, the *Travel*, or the *SalesOrder*. In general, a business object contains several nodes such as Items and ScheduleLines and common transactional operations such as for creating, updating and deleting business data. An additional application-specific operation in the *SalesOrder* business object might be, for example, an *Approve* action allowing the user to approve the sales order. All changing operations for all application-related business objects form the transactional behavior in an application scenario.

When going to implement an application scenario based on business objects, we may distinguish between the external, consumer-related representation of a business object and the internal, provider-related perspective:

- The **external perspective** hides the intrinsic complexity of business objects. Developers who want to create a service on top of the existing business objects for role-based UIs do not need to know in detail on which parts of technical artifacts the business objects are composed of or how runtime implementations are orchestrated internally. The same also applies to all developers who need to implement a consumer on top of the business object's APIs.
- The **internal perspective** exposes the implementation details and the complexity of business objects. This perspective is required for application developers who want to provide new or extend existing business objects for the industries, the globalization and partners.

From a formal point of view, a business object is characterized by

- a structure,
- a behavior and
- the corresponding runtime implementation.

Structure of a Business Object

From a structural point of view, a business object consists of a hierarchical tree of nodes (*SalesOrder*, *Items*, *ScheduleLines*) where the nodes are linked by special kinds of associations, namely by compositions. A composition is a specialized association that defines a whole-part relationship. A composite part only exists together with its parent entity (whole).

Each node of this composition tree is an element that is modeled with a CDS entity and arranged along a composition path. As depicted in the diagram below, a sequence of compositions connecting entities with each other, builds up a composition tree of an individual business object.

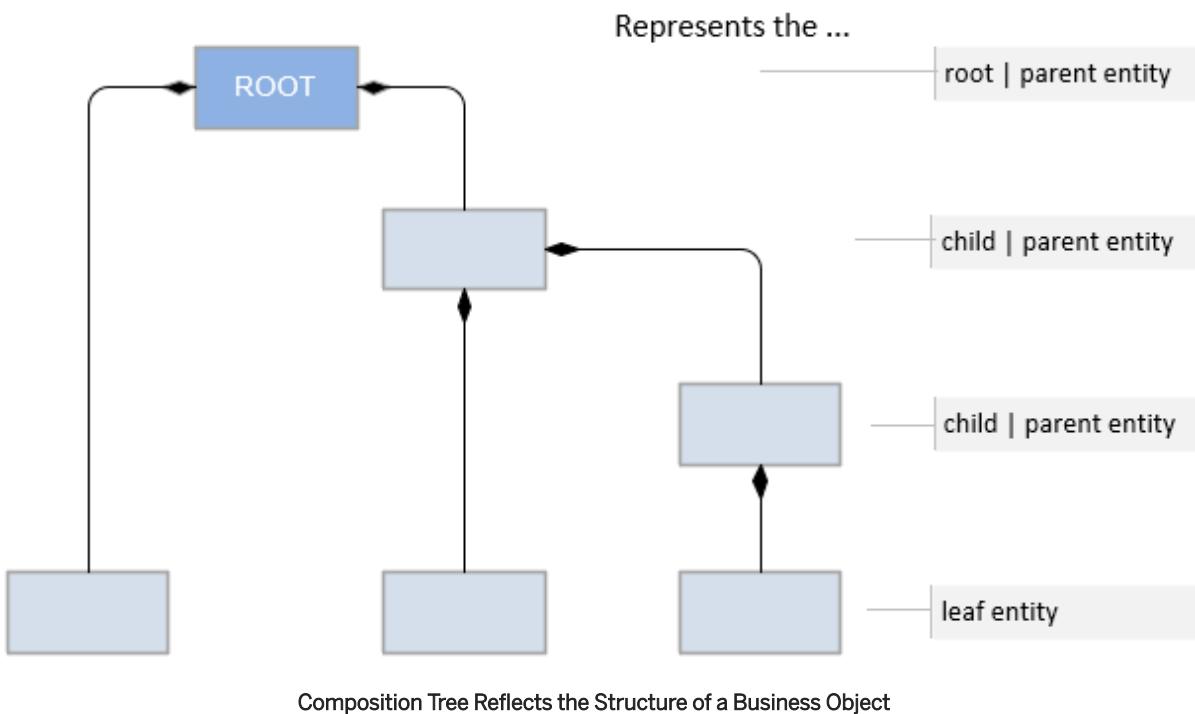
The root entity is of particular importance in a composition tree: The root entity serves as a representation of the business object and defines the top node within a hierarchy in a business object's structure. This is considered in the source code of the CDS data definition with the keyword `ROOT`.

The root entity serves as the source of a composition which is defined using the keyword `COMPOSITION` in the corresponding data definition. The target of this composition defines a direct child entity. On the other hand, CDS entities that represent child nodes of the business object's composition tree, must define an association to their compositional parent or root entity. This relationship is expressed by the keyword `ASSOCIATION TO PARENT`. A to-parent association in ABAP CDS is a specialized association which can be defined to model the child-parent relationship between two CDS entities.

In a nutshell: both, a sequence of compositions and to-parent associations between entities define the structure of a business object with a root entity on top of the composition tree.

All entities - except the root entity - that represent a node of the business object structure serve as a:

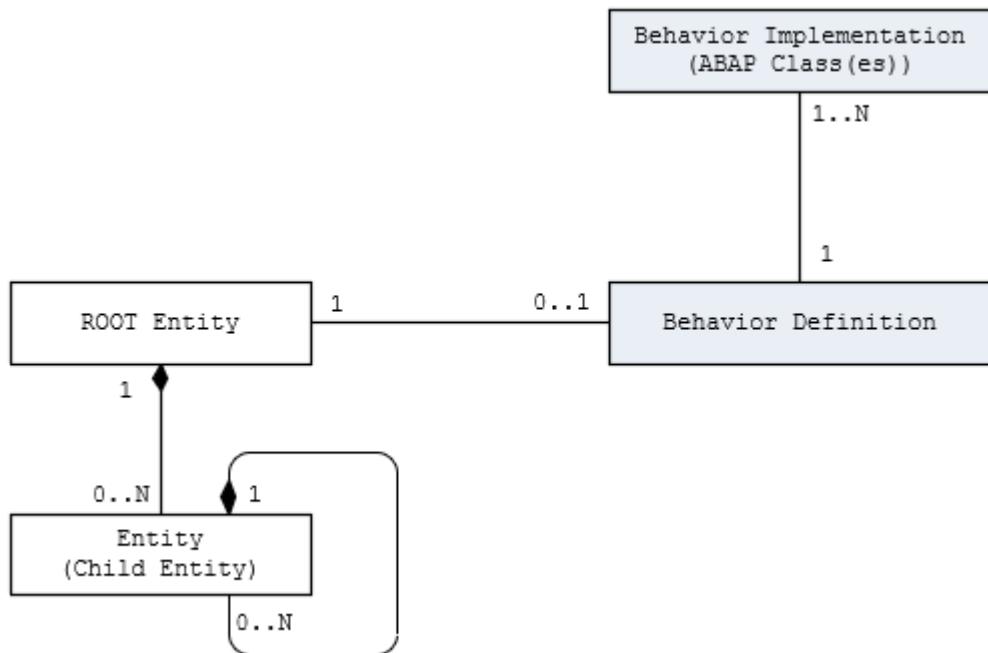
- Parent entity - if it represents a node in a business object's structure that is directly connected to another node when moving towards the root.
- Child entity - if it represents a node in a business object's structure that is directly connected to another node (parent node) when moving away from the root.
- Leaf entity - if it represents a node in a business object's structure without any child nodes. A leaf entity is a CDS entity, which is the target of a composition (a child entity node) but does not contain a composition definition.



Behavior of a Business Object

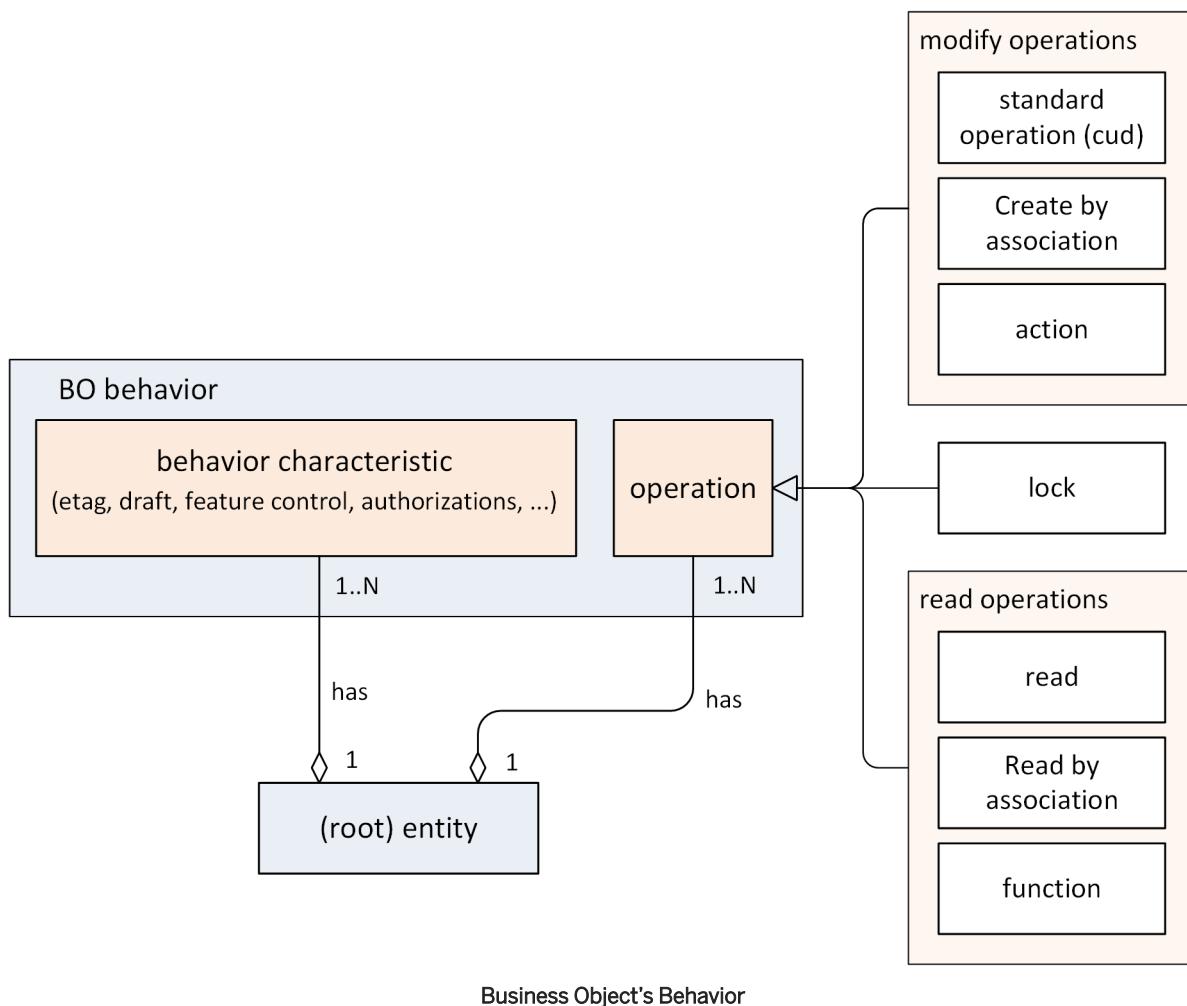
To specify the business object's behavior, the behavior definition as the corresponding development object is used. A business object behavior definition (behavior definition for short) is an ABAP Repository object that describes the behavior of a business object in the context of the ABAP RESTful application programming model. A behavior definition is defined using the Behavior Definition Language (BDL).

A behavior definition always refers to a CDS data model. As shown in the figure below, a behavior definition relies directly on the CDS root entity. One behavior definition refers exactly to one root entity and one CDS root entity has at most one behavior definition (a 0..1 cardinality), which also handles all included child entities that are included in the composition tree. The implementation of a behavior definition can be done in a single ABAP class (behavior pool) or can be split between an arbitrary set of ABAP classes (behavior pools). The application developer can assign any number of behavior pools to a behavior definition (1..N cardinality).



Relationship Between the CDS Entities and the Business Object Behavior

A behavior specifies the operations and field properties of an individual business object in the ABAP RESTful programming model. It includes a behavior characteristic and a set of operations for each entity of the business object's composition tree.



Behavior Characteristic

Behavior characteristic is that part of the business object's behavior that specifies general properties of an entity such as:

[ETag \[page 812\]](#)

Draft handling

[Feature control \[page 812\]](#)

[Numbering \[page 61\]](#)

Authorizations.

These characteristics can be defined for each entity separately.

Operations

Each entity of a business object can offer a set of operations. They can cause business data changes that are performed within a transactional life cycle of the business object. As depicted in the diagram below, these modify operations include the standard operations create(), update() and delete() as well as lock implementations and application-specific operations with a dedicated input and output structure which are called actions. Another kind of operations are the read operations: they do not change any business data in the

context of a business object behavior. Read operations include read, read by association, and functions (that are similar to actions, however, without causing any side effects).

For more information, see

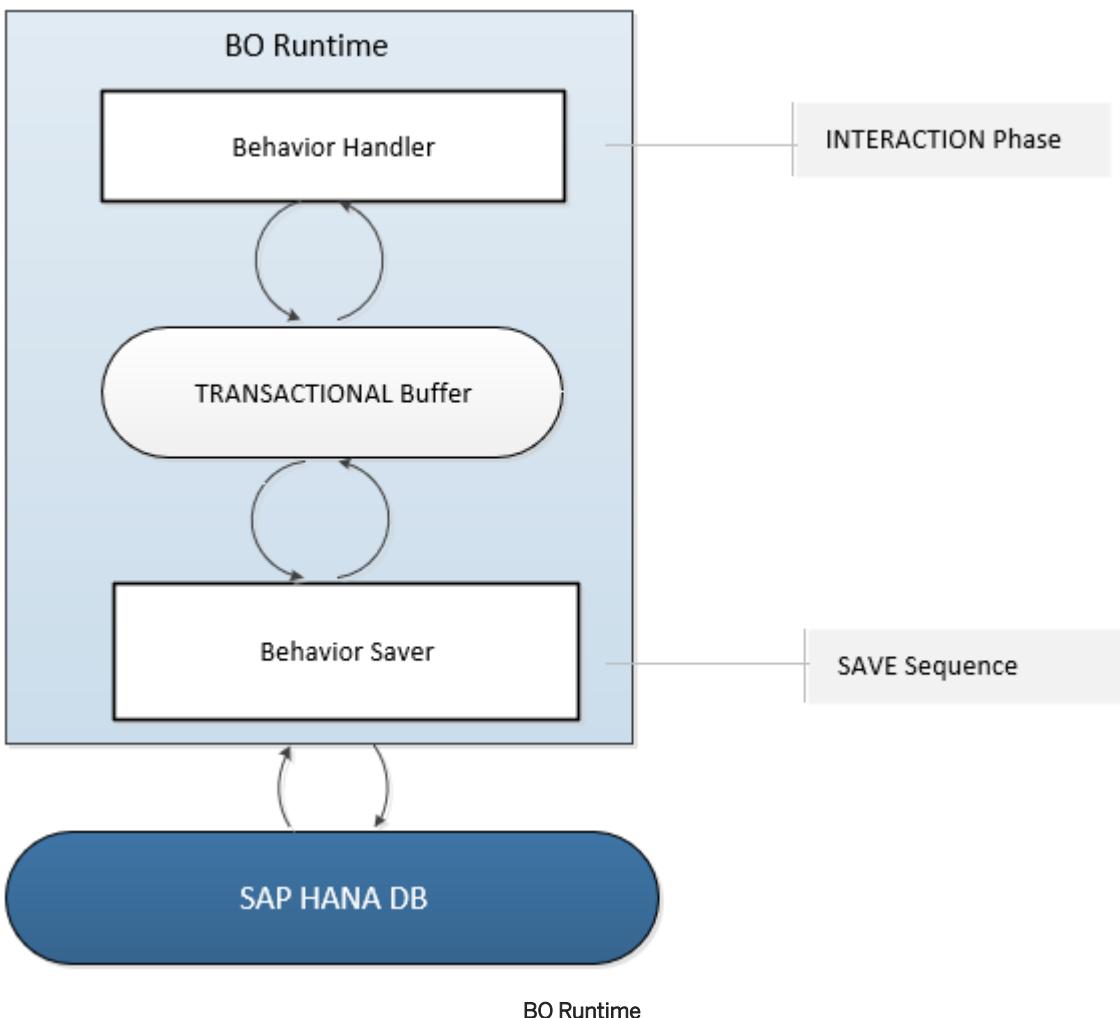
- [Create Operation \[page 65\]](#)
- [Update Operation \[page 67\]](#)
- [Delete Operation \[page 69\]](#)
- [Actions \[page 73\]](#)

Business Object's Runtime

The business object runtime mainly consists of two parts:

The first part is the **interaction phase**, in which a consumer calls the business object operations to change data and read instances with or without the transactional changes. The business object runtime keeps the changes in its internal **transactional buffer** which represents the state of the instance data. This transactional buffer is always required for a business object. After all changes were performed, the data can be persisted.

This is realized with the **save sequence**.



For each operation the transactional runtime is described in detail in the respective runtime diagrams:

- [Create Operation \[page 65\]](#)
- [Update Operation \[page 67\]](#)
- [Delete Operation \[page 69\]](#)

The save sequence has the same structure for each operation. For more information, see [Save Sequence \[page 83\]](#).

4.1.2.1 Behavior Characteristics

4.1.2.1.1 Numbering

Numbering is about setting values for primary key fields of entity instances during runtime.

The primary key of a business object entity can be composed of one or more key fields, which are identified by the keyword `key` in the underlying CDS view of the business object. The set of primary key fields uniquely

identify each instance of a business object. The primary key value of a business object instance cannot be changed after the `CREATE`.

There are various options to handle the numbering for primary key fields depending on when (early or late during the transactional processing) and by whom (consumer, application developer, or framework) the primary key values are set. You can assign a numbering type for each primary key field separately. The following options are available:

- [Early Numbering \[page 62\]](#)
 - [External Numbering \[page 62\]](#)
 - [Internal Numbering \[page 63\]](#)
 - [Managed Early Numbering \[page 63\]](#)
 - [Unmanaged Early Numbering \[page 63\]](#)
 - [Optional External Numbering \[page 63\]](#)
- [Late Numbering \[page 64\]](#)

4.1.2.1.1.1 Early Numbering

The numbering type **early numbering** refers to an early value assignment for the primary key field. In this case, the final key value is available in the transactional buffer instantly after the `MODIFY` request for `CREATE`.

The key value can either be given by the consumer (externally) or by the framework (internally). In both cases, the newly created BO instance is clearly identifiable directly after the `CREATE` operation has been executed. It can be referred to by this value in case other operations are executed on this instance during the interaction phase (for example an `UPDATE`). During the save sequence, the newly created BO instance is written to the database with the primary key value that was assigned earlier on the `CREATE` operation.

External Numbering

We refer to **external numbering** if the consumer hands over the primary key values for the `CREATE` operation, just like any other values for non-key fields. The runtime framework (managed or unmanaged) takes over the value and processes it until finally writing it to the database.

In this scenario, it must be ensured that the primary key value given by the consumer is uniquely identifiable.

Implementation

For external numbering, you must ensure that the primary key fields are not read-only. Otherwise, the consumer cannot provide any value for the primary key field.

- In **managed** scenarios, the processing and saving of the key values is done by the managed runtime framework.
- In **unmanaged** scenarios, the processing and saving of the key values must be done by the application developer using the method for `CREATE` and for `SAVE`.

Internal Numbering

In scenarios with **internal numbering**, the runtime framework assigns the primary key value when creating the new instance. Internal numbering can be managed by the RAP runtime or unmanaged, that is implemented by the application developer.

Managed Internal Numbering

When using managed numbering, a UUID is drawn automatically during the `CREATE` request by the RAP managed runtime.

i Note

Managed early numbering is only possible for key fields with ABAP type `raw(16)` (UUID) of BOs with implementation type managed.

Implementation

Managed numbering is defined in the behavior definition. The key field must be `read only`.

```
[implementation] managed [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
  lock (master|dependent() )
  ...
{
  ...
  field ( read only, numbering:managed ) KeyField1 [,KeyField2, ..., keyFieldn];
  ...
}
```

For more information, see [Automatically Drawing Primary Key Values in Managed BOs \[page 455\]](#).

Unmanaged Numbering

When using unmanaged numbering, the application developer has to implement the numbering logic in application code that is called during the `CREATE` operation.

- In unmanaged BOs, the `CREATE` operation implements the assignment of a primary key value during the `CREATE` modification.

Implementation

The assignment of a new number must be included in your application code.

For more information, see [Implementing the CREATE Operation for Travel Instances \[page 305\]](#).

i Note

With the current version of the ABAP RESTful Programming Model, it is not possible to use unmanaged numbering in managed BOs.

Optional External Numbering

We refer to **optional external numbering** if both, external and internal numbering is possible for the same BO. If the consumer hands over the primary key value (external numbering), this value is processed by the runtime framework. If the consumer does not set the primary key value, the framework steps in and draws the number for the instance on `CREATE`.

i Note

Optional external numbering is only possible for managed business objects with UUID keys.

Use cases for optional external numbering are replication scenarios in which the consumer already knows some of the UUIDs for specific instances to create.

Implementation

Optional external numbering is defined in the behavior definition. The key field must not be `read only`, so that the consumer is able to fill the primary key fields.

```
[implementation] managed [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
  lock (master|dependent() )
  ...
{
  ...
  field ( numbering:managed ) KeyField1 [,KeyField2, ..., keyFieldn];
  ...
}
```

Related Information

[Numbering \[page 61\]](#)

[Late Numbering \[page 64\]](#)

4.1.2.1.1.2 Late Numbering

The numbering type **late numbering** refers to a late value assignment for the primary key fields. The final number is only assigned just before the instance is saved on the database.

To identify a newly created instance for internal processing before the save sequence, a temporary identifier, the preliminary ID (%PID) is assigned. It works as a substitute as long as there is no final key value. The instance is referred to by the PID in case other operations are executed on this instance during the interaction phase (for example an `UPDATE`). To assign the real and final key value to the instance the method `adjust_numbers` is called after the point of no return and before the actual `SAVE` happens. It overwrites the preliminary ID with the final key value. This late value assignment ensures that the instance can be written to the database.

For more information, see [Method ADJUST_NUMBERS \[page 733\]](#).

Late numbering is used for scenarios that need gap-free numbers. As the final value is only set just before the `SAVE`, everything is checked before the number is assigned.

In contrast to [Early Numbering \[page 62\]](#), late numbering can only be done internally, as a consumer cannot interfere anymore during the save sequence.

Late Numbering in Managed Scenarios

With the current version of the ABAP RESTful Programming Model, it is not possible to use late numbering in managed scenarios.

Late Numbering in Unmanaged Scenarios

Late numbering must be defined for each node in the behavior definition of an unmanaged BO.

```
implementation unmanaged;
define behavior for Entity [alias AliasedName]
late numbering
...
{ ...
}
```

In addition, the [Method ADJUST_NUMBERS \[page 733\]](#) must be implemented to assign a final number for each instance.

Related Information

[Numbering \[page 61\]](#)

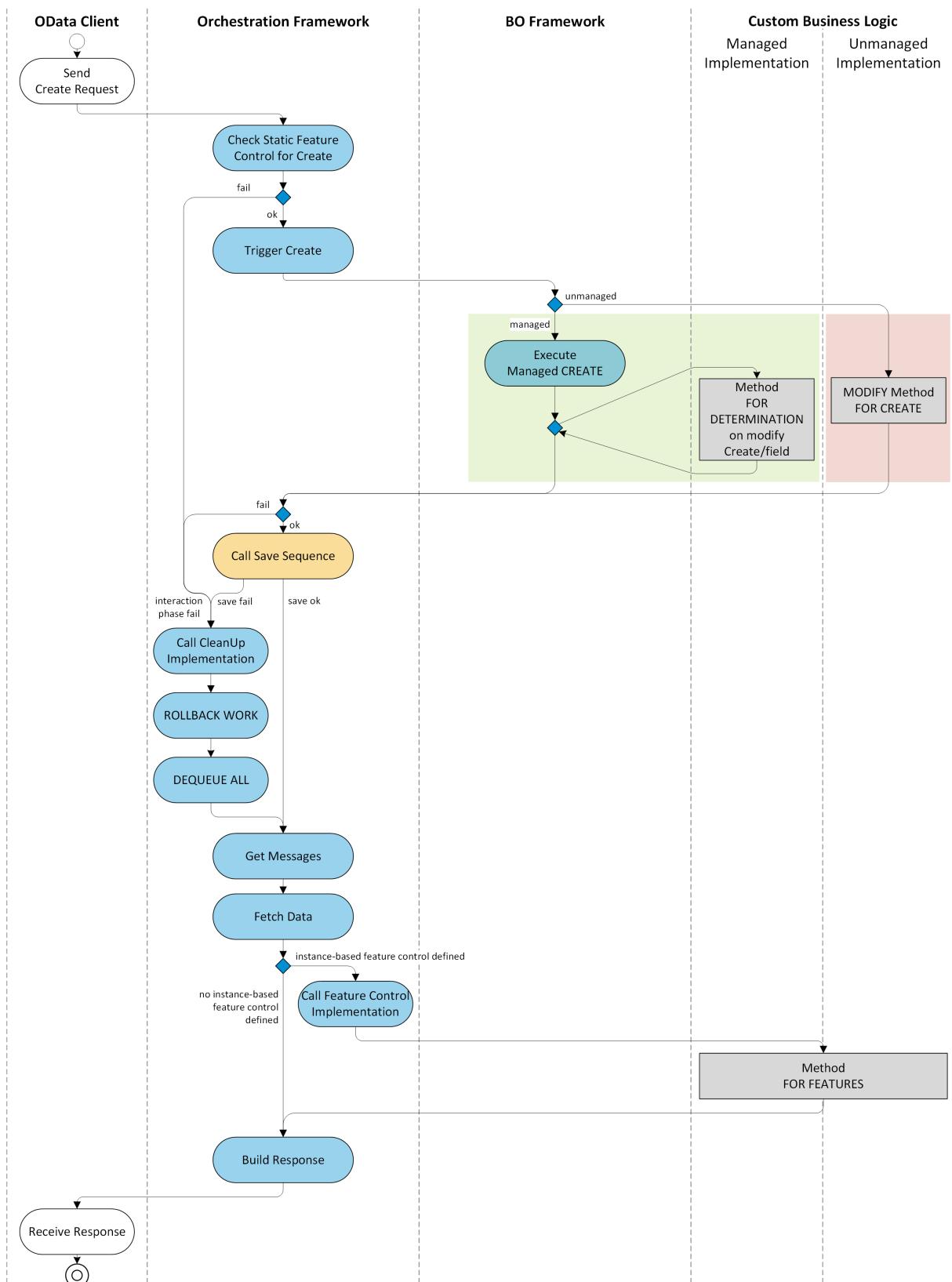
[Early Numbering \[page 62\]](#)

4.1.2.2 Operations

4.1.2.2.1 Create Operation

In RAP, the `create` operation is a standard modifying operation that creates new instances of a business object entity.

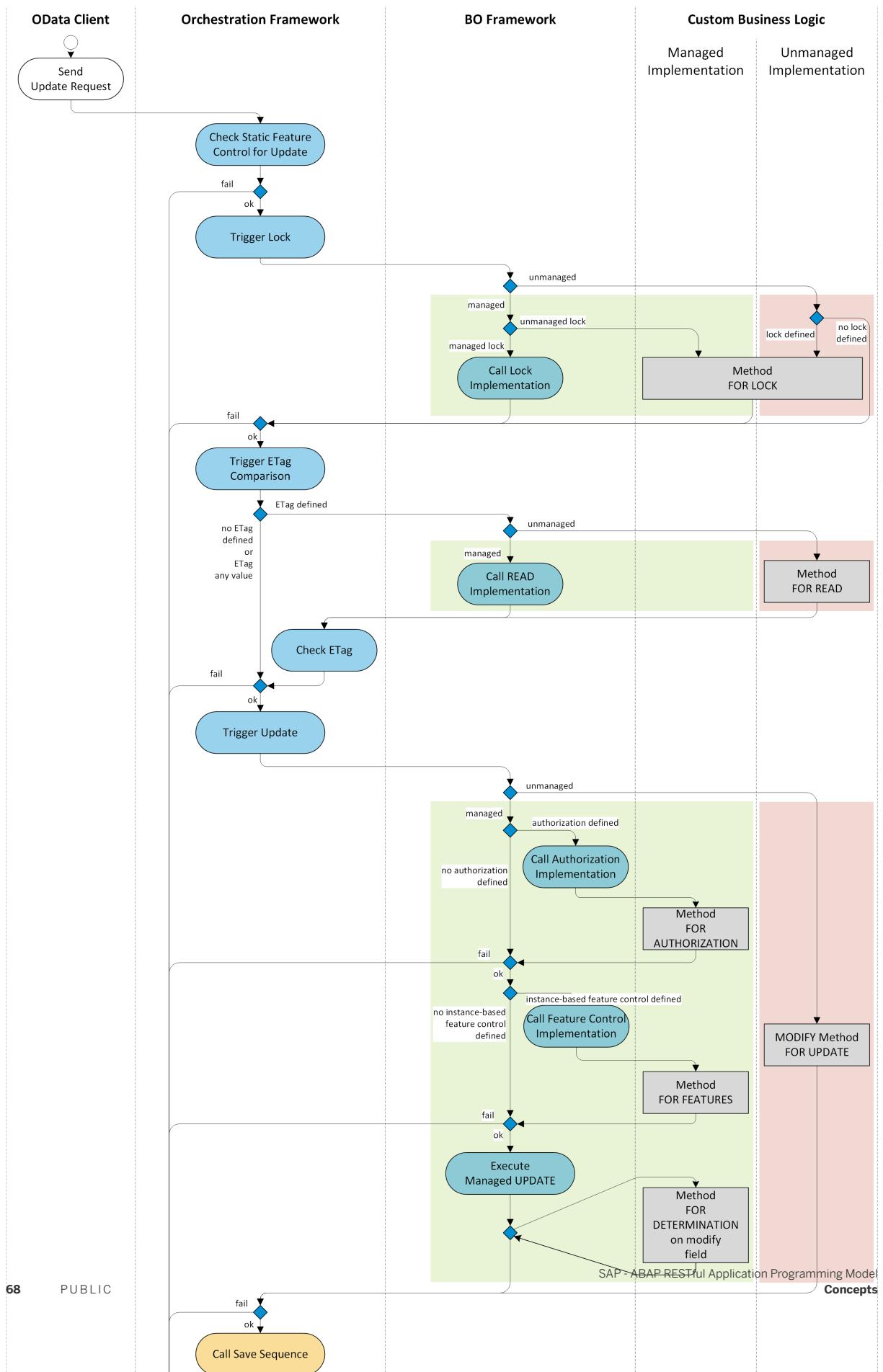
The following runtime diagram illustrates the main agents' activities during the interaction phase of a create operation when an OData request for create (`POST`) is sent. The save sequence is illustrated in a separate diagram, see [Save Sequence \[page 83\]](#).



4.1.2.2.2 Update Operation

In RAP, the update operation is a standard modifying operation that changes instances of a business object entity.

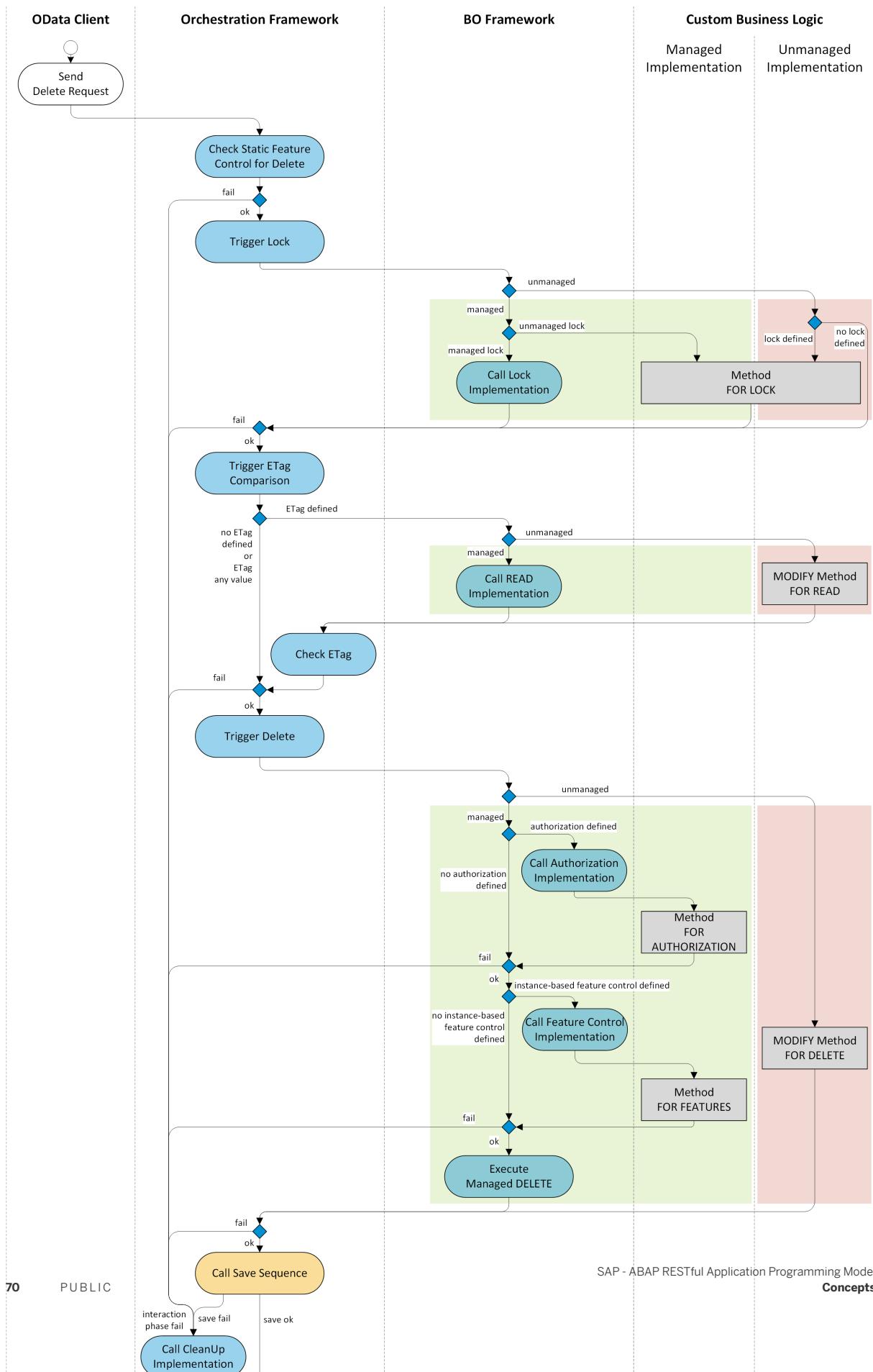
The following runtime diagram illustrates the main agents' activities during the interaction phase of an update operation when an OData request for update (`MERGE`) is sent. The save sequence is illustrated in a separate diagram, see [Save Sequence \[page 83\]](#).



4.1.2.2.3 Delete Operation

In RAP, the delete operation is a standard modifying operation that deletes instances of a business object entity.

The following runtime diagram illustrates the main agents' activities during the interaction phase of a delete operation when an OData request for delete (`DELETE`) is sent. The save sequence is illustrated in a separate diagram, see [Save Sequence \[page 83\]](#).

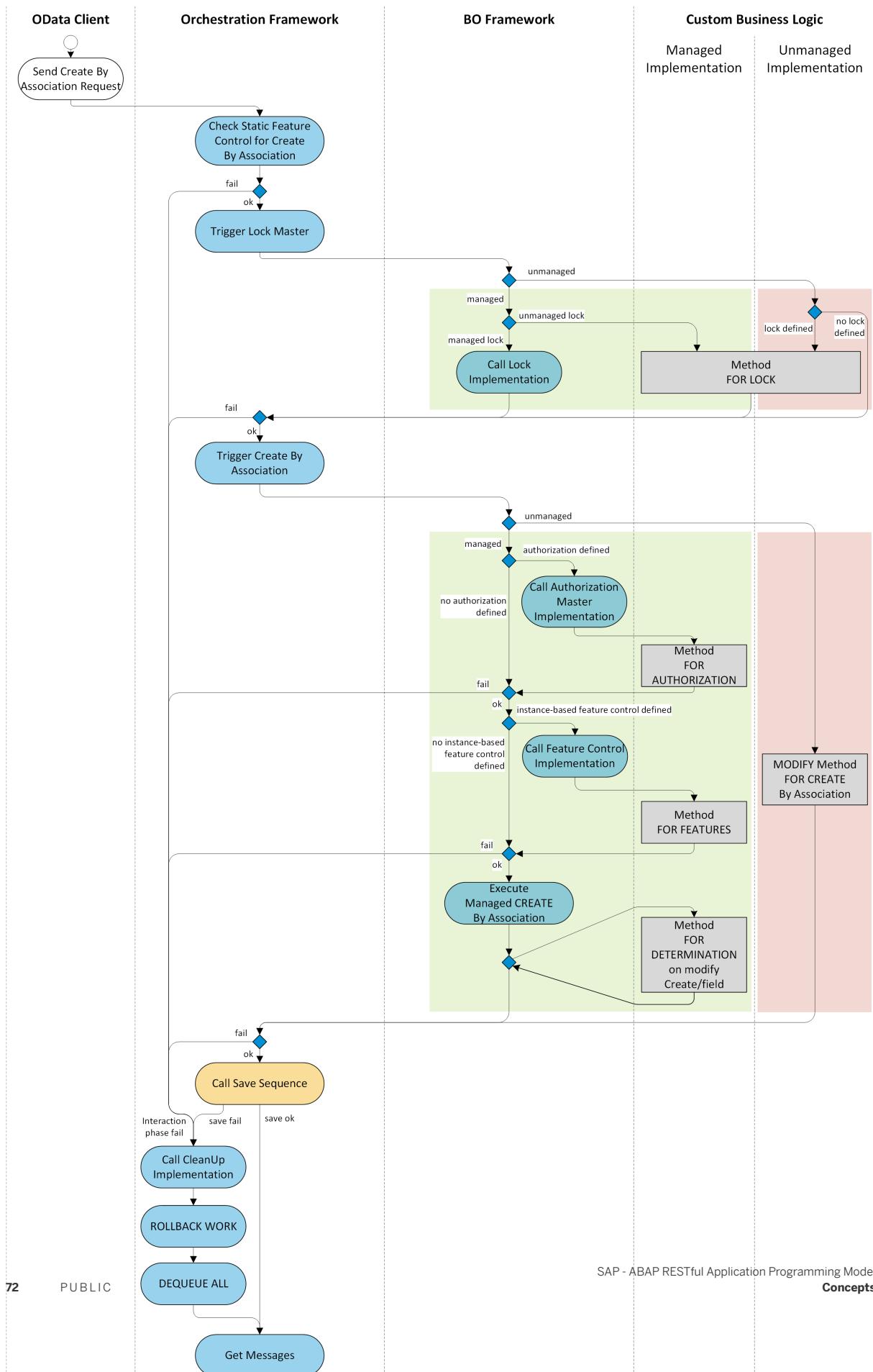


- Delete Operation [page 69]

4.1.2.2.4 Create by Association Operation

In RAP, the `create by association` operation is a modify operation that creates new instances of an associated entity.

The following runtime diagram illustrates the main agents' activities during the interaction phase of a `create by association` operation when an OData request for a create by association (`POST`) is sent. The save sequence is illustrated in a separate diagram, see [Save Sequence \[page 83\]](#).



4.1.2.2.5 Actions

An action in RAP is a non-standard modifying operation that is part of the business logic.

The standard use case of an action is to change specific fields of a business object entity. When using an action, it is not the standard update operation that is called, but the action with the predefined update implementation. On a Fiori UI, this means that the consumer of the Fiori app can change the state of an individual business object instance without having to switch to edit mode. The application developer provides action buttons for the action to be executable directly from a list report or an object page.

In the travel demo scenario, we provide examples of actions for changing the status of a travel instance to booked. Expand the following figure to watch how an action is executed on a Fiori UI.

Action

Setting the Status to booked by an Action

In general, however, actions can have a much wider scope than just changing single values of an instance. You can create instances or implement complete work-flows in an action.

Technically, actions are part of the business logic. They are defined in the behavior definition and implemented in the behavior pool of a business object. Actions are executed by calling the corresponding method `FOR MODIFY` that has typed import and export parameters. They are identified as actions by the syntax `FOR ACTION`.

Triggers for Actions

For an action to be executed, a corresponding trigger is required. Actions can be called by

- A service consumer, for example, by a user interface.
- Internally, for example, by another action or by a determination via EML.
- By other business objects via EML.

Related Information

[Action Definition \[page 75\]](#)

[Action Implementation \[page 79\]](#)

[Action Runtime \[page 81\]](#)

4.1.2.2.5.1 Action Definition

You define actions in the behavior definition of the entity the action is assigned to.

Syntax for Defining Actions

Actions are specified as non-standard operations in behavior definitions by using the following syntax:

Click on the syntax elements to jump to the respective explanation.

```
...
define behavior for CDSEntity [alias AliasedEntityName]
implementation in class ABAP_CLASS_NAME [unique]
[authorization master ( instance )]
...
{
  [internal [page 75]] [static [page 75]] [factory [page 76]]
    action [(
      features: instance [page 76], authorization: none [page 76] )
    ActionName [external 'ExternalActionName' [page 76]]
      [parameter { InputParameter | $self) } [page 76] ]
      [result [page 77] [selective [page 77]] [cardinality] [page 77]
    { $self [page 77] |
      entity OutputParameterEntity [page
    77] |
      OutputParameterStructure [page 78]
    [external 'ExtResultStructureName' [page 78]] } ];
}
```

Internal Action

By default, actions are executable by OData requests as well as by EML from another business object or from the same business object. To only provide an action for the same BO, the option `internal` can be set before the action name, for example, when executing internal status updates. An internal action can only be accessed from the business logic inside the business object implementation such as from a determination or from another action.

Static Action

By default, actions are related to instances of a business object's entity. The option `static` allows you to define a static action that is not bound to any instance but relates to the complete entity.

Factory Actions

With factory actions you can create entity instances by executing an action. Factory actions can be instance-bound or static. Instance-bound factory actions can be useful if you want to copy specific values of an instance. Static factory actions can be used to create instances with default values.

i Note

In the current version of the ABAP RESTful Application Programming Model, factory actions are only supported for the unmanaged implementation type.

Dynamic Action Control

You can enable or disable actions depending on other preconditions of the business object. For example, you might want to offer the action `accept_travel` only if the status is not *rejected* already.

Dynamic action control is defined with the syntax `features: instance` and must be implemented in the [<method> FOR FEATURES \[page 729\]](#).

For more information, see [Adding Feature Control \[page 459\]](#).

Authorization Control

Actions can be checked against unauthorized execution. A precondition for this is the definition of an authorization master in the behavior definition. To exclude the action from authorization checks, you can use the syntax `authorization: none`.

For more information, see [Adding Authorization Control to Managed Business Objects \[page 487\]](#).

External Action Name

By using the syntax `external 'ExternalActionName'`, you can rename the action for external usage. That means, the new name will be exposed in the OData metadata. This external name can be much longer than the actual action name, but is not known by ABAP.

If you want to define an action button for an action with an external name, the external name must be used in the `@UI` annotation. For more information, see [UI Consumption of Actions \[page 80\]](#).

Input Parameter

Actions can pass abstract CDS entities or other structures as input parameters. They are defined by the keyword `parameter`.

You can specify `$self` if the input parameter entity is the same abstract entity the action is assigned to. Input parameters with `$self` are only allowed on static actions. Instance-bound actions always import the key of the instance on which the action is executed. If you import the same entity instance as input parameter, the keys would be imported twice, which will cause a short dump during runtime. For more information about the importing parameter, see [Importing Parameter \[page 79\]](#)

Output Parameter

The output parameter for an action is defined with the keyword `result`. The result parameter for actions is not obligatory. However, if a result parameter is declared in the action definition, it must be filled in the implementation. If it is not filled, the action does not return anything, even if the action is declared with result cardinality greater 0. In such a case, the OData service returns initial values.

Selective Result

By declaring the action result to be `selective` you can define that the action consumer can decide whether the result shall be returned completely or only parts of it, for example the keys only. This can help to improve performance as performance consuming calculated fields can be excluded from the result.

The action method in the behavior pool then receives the additional importing parameter `REQUEST`. This parameter specifies which components of the result shall be returned. For more information about the implementation of actions with selective result parameter, see [Action Importing Parameter \[page 79\]](#).

A Fiori UI requests only the keys of the result when an action with selective result is executed.

Result Cardinality

The `result cardinality` for actions determines the multiplicity of the output. In this way, it indicates whether the action produces 0..1, 1, 0..n, or 1..n output instances. The possible values for cardinality are therefore:

[0..1], or [1], or [0..*], or [1..*].

i Note

The protocol OData V2 does not support actions with result cardinality greater than 1.

Result Parameter

You can define an entity or a structure as a result for actions:

- **Result Entity:** You can return a business object entity as action result.
 - Use the syntax `$self` if the result entity instance is the same instance for which the action is executed..
In a UI service, the UI always stays on the same page where the action is executed. If, like in the demo above, the action is executed on the list report, the UI stays on the list report page after the action is executed. Executing the action from the object page returns the action result on the object page.
 - Use the syntax `entity OutputParameterEntity` to define the action result if the result entity is a different entity of the same or another BO.

i Note

Only actions having output entities that are included in the service definition are exposed in the service.

For action with return type result entity other than `$self`, the Fiori UI does not navigate to the result entity, but stays on the page where the action is executed.

i Note

In a projection behavior definition, result entities other than `$self` must be redefined with the projection result entity. For more information, see [Actions in Projection Behavior Definitions \[page 78\]](#).

- **Result Structure:** Apart from returning entities you can also return ABAP structures. This can be an entity type but also other structures that are defined in the ABAP dictionary. A resulting structure for actions is defined without the keyword `entity`.
Using a structure as a return type is useful if you want to use the result in your further implementation. For result structures it is possible to define an alias to clearly identify the result in the OData metadata. The keyword `external` after the result type defines this OData representation of the action result.

i Note

If the action result is an abstract entity, you have to define the result without the keyword `entity` as abstract entities are generally considered to be structures in ABAP.

Actions in Projection Behavior Definitions

Like any operation, an action must be included in the projection behavior definition if you want to expose it for an OData service.. The following syntax is used:

```
projection; ...
define behavior for CDSEntity [alias AliasedEntityName]
{
  ...
    use action ActionName [result entity ProjResultEntity] [as ActionAlias]
    [external ExtActName];
}
```

The keyword `use` registers the action for the specified business object projection.

Actions that have an entity result other than `$self` must be redirected to the projection entity of the result entity with the syntax `result entity ProjResultEntity`. Otherwise, it may happen that the action is not exposed anymore if the result entity is not included in the service.

You can define an internal alias for the action by using the syntax `as ActionAlias`. This alias is used in EML calls.

You can define an external name for the action with `external ExtActName` that is used in OData. This external name can be much longer than the alias name in ABAP and needs to be used when defining the corresponding UI annotation.

Example

For a fully defined action, see [Defining Actions as Part of the Behavior Definition \[page 180\]](#).

Related Information

[Actions \[page 73\]](#)

[Action Implementation \[page 79\]](#)

[Action Runtime \[page 81\]](#)

4.1.2.2.5.2 Action Implementation

You implement action in the behavior pool with the respective method FOR MODIFY.

As a rule, an action that belongs to a business object's entity is implemented in the behavior pool that is defined in the behavior definition by the keyword implementation in class ABAP_CLASS_NAME [unique].

The concrete implementation of an action is based on the ABAP language in a local handler class as part of the behavior pool.

As depicted in the listing below, each such local class inherits from the base handler class CL_ABAP_BEHAVIOR_HANDLER. The signature of the handler method FOR MODIFY is type based on the entity that is defined by the keyword FOR ACTION followed by AliasedEntityName~ActionName. The alias name is defined in the behavior definition using the additional alias AliasedEntityName that refers to the suitable CDS entity.

Implementing an Action in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS method_name FOR MODIFY  
    IMPORTING it_key_for_action FOR ACTION AliasedEntityName~ActionName  
    [REQUEST it_requested_fields]  
    [RESULT et_action_result].  
ENDCLASS.  
CLASS lhc_handler IMPLEMENTATION.  
METHOD method_name.  
    // Implement method here!  
ENDMETHOD.  
ENDCLASS.
```

Importing Parameter

- Depending on the type of action, the importing parameter it_key_for_action has the following components:

Action Specifics	Importing Parameter Components
instance action	An instance action imports the key of the instance.
static action	An instance action imports %cid_ref. This component is executed on a newly created instance that is filled with the %cid of the instance the action is assigned to.
action with parameter	A static action imports %cid. For static actions, the operation uniquely.
action with result type entity	If the action returns one or more entity instances, the final key is set.
factory action	A factory action imports %cid and %cid_ref. As factory actions always create new instances, the final key is set. If factory actions are instance-bound they also create a new instance to which they are assigned.

- If the result parameter is defined as `selective` in the behavior definition, the action declaration in the behavior pool receives another importing parameter `REQUEST it_requested_field`. In the request parameter all fields of the action result that are selected by the action executor are flagged. Because of this, the action provider knows which fields are expected as a result.

Result Parameter

The components of the result parameter depend on those of the importing structure. The imported values of %cid and %cid_ref are returned if they are imported.

If a result is defined, it has the structure %param to be filled by the action implementation. This component is a table that reflects the type of the defined result type.

For action with selective result, only the field that are requested in REQUEST must be filled in %param.

UI Consumption of Actions

For an action to be consumable by a Fiori Elements UI, you need to define an action button in the relevant CDS view.

An action is always assigned to one business object entity in the behavior definition. In the corresponding CDS view, the action button must be defined in the @UI annotation.

Use `@UI.lineItem: [{type: #FOR_ACTION, dataAction: 'ActionName', label: 'ButtonLabel' }]` to define an action button on a list report page.

Use `@UI.identification: [{type: #FOR_ACTION, dataAction: 'ActionName', label: 'ButtonLabel' }]` to define an action button on the object page.

The ActionName must correspond to the action name in the behavior definition. If an external action name is defined for the action, you have to use this external name.

Example

For a fully implemented action, see [Implementing Actions \[page 181\]](#) and [Enabling Actions for UI Consumption \[page 186\]](#).

Related Information

[Action Runtime \[page 81\]](#)

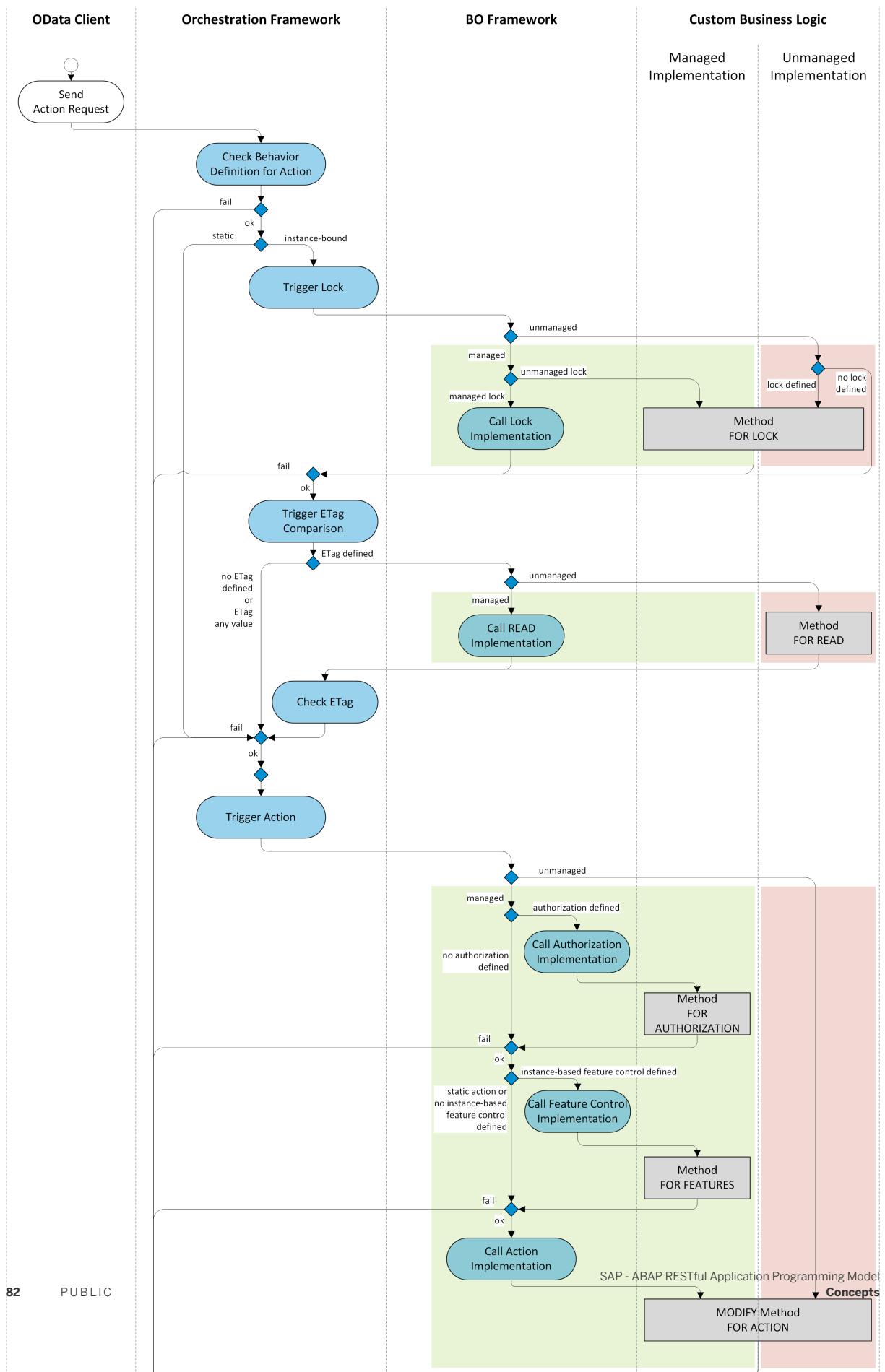
[Actions \[page 73\]](#)

[Action Definition \[page 75\]](#)

4.1.2.2.5.3 Action Runtime

In RAP, an action is a non-standard modify operation.

The following runtime diagram illustrates the main agents' activities during the interaction phase of an action when an OData request to execute an action (POST) is sent. The save sequence is illustrated in a separate diagram, see [Save Sequence \[page 83\]](#).



Related Information

Actions [page 73]

Action Definition [page 75]

Action Implementation [page 79]

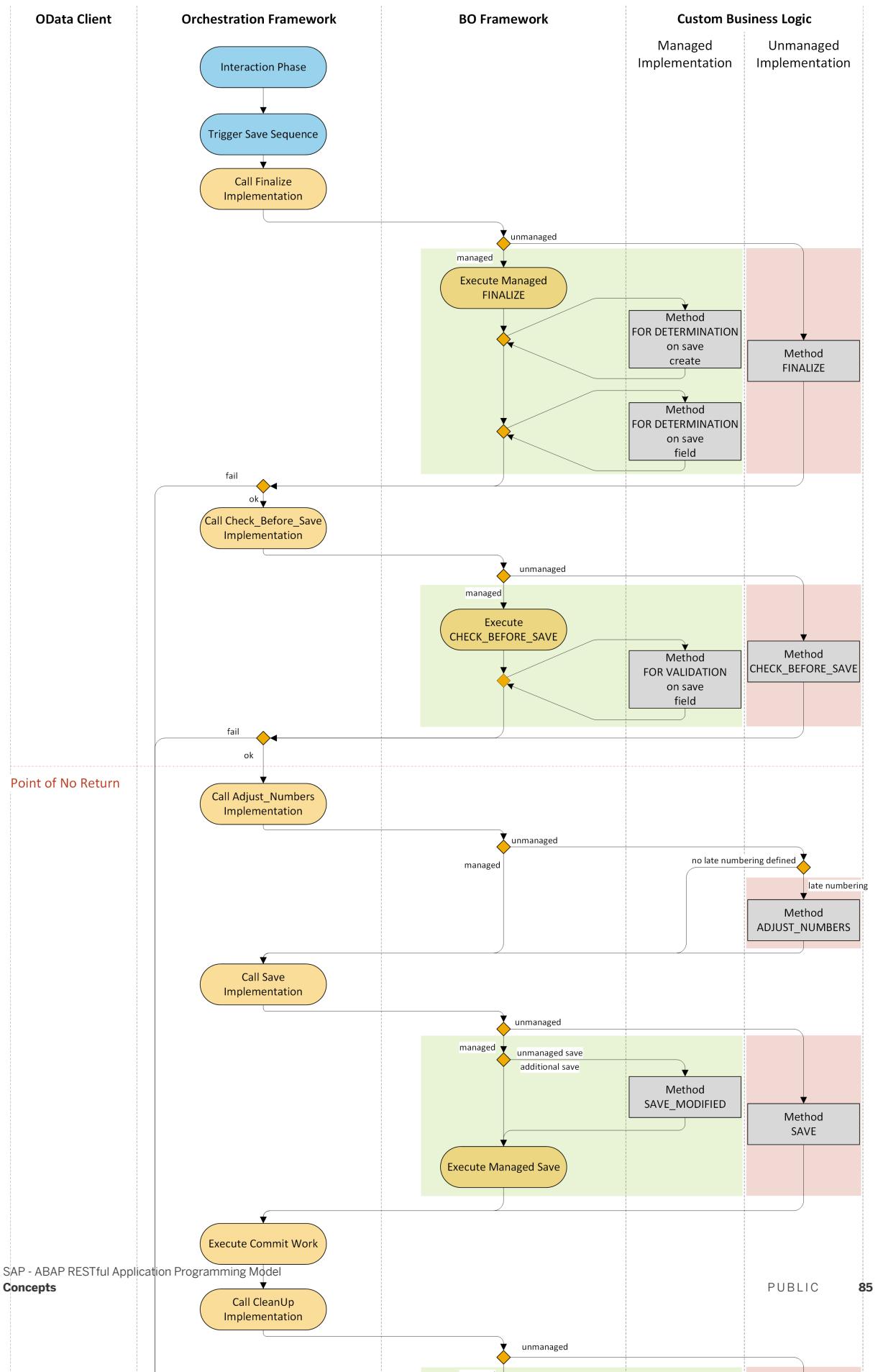
4.1.2.2.6 Save Sequence

The save sequence is part of the business logic and is called when data must be persisted after all changes are performed during the interaction phase.

The following runtime diagram illustrates the main agents' activities during the save sequence in OData requests. The interaction phase of the operations is illustrated in separate diagrams, see

- Create Operation [page 65]
 - Update Operation [page 67]

- Delete Operation [page 69]



4.1.2.3 Concurrency Control

Concurrency control prevents concurrent and interfering database access of different users. It ensures that data can only be changed if data consistency is assured.

RESTful applications are designed to be usable by multiple users in parallel. In particular, if more than one user has transactional database access, it must be ensured that every user only executes changes based on the current state of the data and thus the data stays consistent. In addition, it must be ensured that users do not change the same data at the same time.

There are two approaches to regulate concurrent writing access to data. Both of them must be used in the ABAP RESTful Application Programming Model to ensure consistent data changes.

- Optimistic Concurrency Control [page 87]
 - Pessimistic Concurrency Control (Locking) [page 91]

4.1.2.3.1 Optimistic Concurrency Control

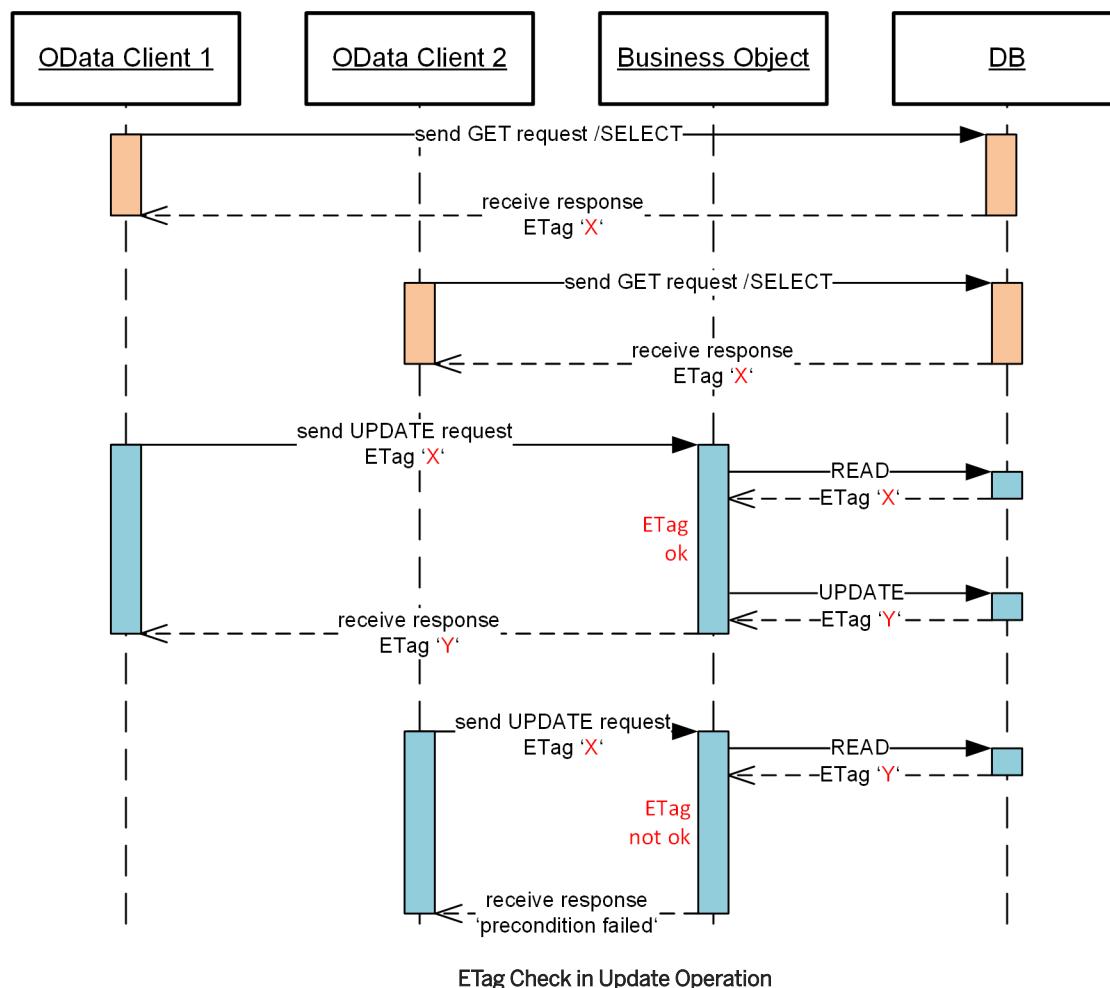
Optimistic concurrency control enables transactional access to data by multiple users while avoiding inconsistencies and unintentional changes of already modified data.

The approach of optimistically controlling data relies on the concept that every change on a data set is logged by a specified ETag field. Most often, the ETag field contains a timestamp, a hash value, or any other versioning that precisely identifies the version of the data set.

When optimistic concurrency control is enabled for RAP business objects, the OData client must send an ETag value with every modifying operation. On each ETag relevant operation, the value of the ETag field is compared to the value the client sends with the request. Only if these values match is the change request accepted and the data can be modified. This mechanism ensures that the client only changes data with exactly the version the client wants to change. In particular, it is ensured that data an OData client tries to change has not been changed by another client between data retrieval and sending the change request. On modifying the entity instance, the ETag value must also be updated to log the change of the instance and to define a new version for the entity instance.

Concurrency control based on ETags is independent of the ABAP session and instances are not blocked to be used by other clients.

The following diagram illustrates the ETag checks for two different clients working on the same entity instance.



In RAP business objects, optimistic concurrency control is defined in the behavior definition by specifying an ETag field. Shortly before data is changed on the database, the orchestration framework reads the ETag field to compare its value to the value that is sent with the change request. The modify operation is accepted if the ETag values match. The modify operation is then executed and a new ETag value is assigned to the entity instance. The modify operation is denied if the ETag values are not identical. To enable the transactional read for reading the ETag value in unmanaged scenarios, the method `FOR READ` must be implemented by the application developer.

For more information about the ETag check during the runtime of a modify operation, see [Update Operation \[page 67\]](#).

Related Information

[ETag Definition \[page 88\]](#)

[ETag Implementation \[page 89\]](#)

4.1.2.3.1.1 ETag Definition

In RAP business objects, ETags are used for optimistic concurrency control. You define the ETag in the behavior definition of the business object entity.

Whenever an ETag is defined for a business object entity in the behavior definition, the ETag check is executed for modifying operations, as described in [Optimistic Concurrency Control \[page 87\]](#). You can define which entities support optimistic concurrency control based on their own ETag field and which entities use the ETag field of other entities, in other words, which are dependent on others.

An ETag is defined using the following syntax elements in the **behavior definition**:

```
...
define behavior for CDSEntity [alias AliasedEntityName]
implementation in class ABAP_CLASS_NAME [unique]
...
etag master ETagField [page 88] | etag dependent by _AssocToETagMaster [page
89]
...
{
  ...
  association _AssocToETagMaster { }
}
```

ETag Master

An entity is an ETag master if changes of the entity are logged in a field that is part of the business object entity. This field must be specified as an ETag field in the behavior definition (`ETagField`). Its value is compared to the value the change request sends before changes on the business entity are executed.

i Note

In the managed scenario, the type of the ETag field must be compatible to `timestamp1`.

Root entities are often ETag masters that log the changes of every business object entity that is part of the BO.

ETag Dependent

An entity is defined as ETag dependent if the changes of the entity are logged in a field of another BO entity. In this case, there must be an association to the ETag master entity. To identify the ETag master, the association to the ETag master entity is specified in the behavior definition (`_AssocToETagMaster`). Whenever changes on the ETag dependent entities are requested, the ETag value of their ETag master is checked.

i Note

You do not have to include the ETag field in ETag dependent entities. Via the association to the ETag master entity, it is ensured that the ETag field can always be reached.

The association that defines the ETag master must be explicitly specified in the behavior definition, even though it is implicitly transaction-enabled due to internal BO relations, for example a child/parent relationship. The association must also be defined in the data model structure in the CDS views and, if needed, redefined in the respective projection views.

An ETag master entity must always be higher in the BO composition structure than its dependents. In other words, a child entity cannot be ETag master of its parent entity.

Projection Behavior Definition

```
projection;
  define behavior for ProjectionView [alias ProjectionViewAlias]
    use etag [page 89]
  {
  ...
    use association _AssocToETagMaster
  }
```

To expose the ETag for a service specification in the projection layer, the ETag has to be used in the projection behavior definition for each entity with the syntax `use etag`. The ETag type (master or dependent) is derived from the underlying behavior definition and cannot be changed in the projection behavior definition.

If the entity is an ETag dependent, the association that defines the ETag master must be used in the projection behavior definition. This association must be correctly redirected in the projection layer.

Related Information

[Optimistic Concurrency Control \[page 87\]](#)

[ETag Implementation \[page 89\]](#)

4.1.2.3.1.2 ETag Implementation

There are two prerequisites that must be fulfilled to make an ETag check work properly:

- The ETag field must be updated reliably with every change on the entity instance.
- The read access to the ETag master field from every entity that uses an ETag must be guaranteed.

If these prerequisites are fulfilled, the actual ETag check is performed by the orchestration framework, see [Update Operation \[page 67\]](#), for example.

Implementation for ETag Field Updates

An ETag check is only possible, if the ETag field is updated with a new value whenever the data set of the entity instance is changed or created. That means, for every modify operation, except for the delete operation, the ETag value must be uniquely updated.

Managed Scenario

The managed scenario updates administrative fields automatically if they are annotated with the respective annotations:

```
@Semantics.user.createdBy: true  
@Semantics.systemDateTime.createdAt: true  
@Semantics.user.lastChangedBy: true  
@Semantics.systemDateTime.lastChangedAt: true
```

If the element that is annotated with `@Semantics.systemDateTime.lastChangedAt: true` is used as an ETag field, it gets automatic updates by the framework and receives a unique value on each update. In this case, you do not have to implement ETag field updates.

i Note

You can only use ETag fields with types that are compatible to `timestamp1` in the managed scenario.

If you choose an element as ETag field that is not automatically updated, you have to make sure that the ETag value is updated on every modify operation via determinations.

Unmanaged Scenario

Unlike in managed scenarios, the application developer in the unmanaged scenario must always ensure that the defined ETag field is correctly updated on every modify operation in the application code of the relevant operations, including for updates by actions.

Implementation for Read Access to the ETag Field

As can be seen in the runtime diagrams of the ETag check-relevant operations (for example [Update Operation \[page 67\]](#)), the ETag check during runtime can only be performed if the transactional READ operation to the relevant ETag master entity is enabled.

For ETag master entities that means the `READ` operation must be defined and implemented, whereas for ETag dependent entities, the `READ by Association` operation to the ETag master entity must be defined and implemented.

Unless you are using groups in your behavior definition, the `READ` operation is always implicitly defined. You cannot explicitly specify it. In groups, however, you have to assign the `READ` operation to one group.

The `READ by Association` must be defined in the behavior definition by the syntax `association AssocName`, see [ETag Definition \[page 88\]](#). It must be ensured that there is an implementation available for the `READ by Association` definition.

Managed Scenario

In the managed scenario, the `READ` operation, as well as the `READ by Association` operation for each entity is provided by the framework. The `READ` operation is always supported for each entity and the `READ by Association` operation is supported as soon as the association is explicitly declared in the behavior definition, see [<method> FOR READ \[page 726\]](#).

Unmanaged Scenario

In the unmanaged scenario, the application developer has to implement the read operations for the ETag check. This includes the `READ` operation for the ETag master entity, as well as the `READ by Association` operation from every ETag dependent entity to the ETag master entity. The corresponding `method` for `READ` must be implemented in the behavior pool of the business object.

For a complete example, see [Implementing the READ Operation for Travel Data \[page 316\]](#) and [Implementing the READ Operation for Associated Bookings \[page 327\]](#).

Related Information

[Optimistic Concurrency Control \[page 87\]](#)

[ETag Definition \[page 88\]](#)

4.1.2.3.2 Pessimistic Concurrency Control (Locking)

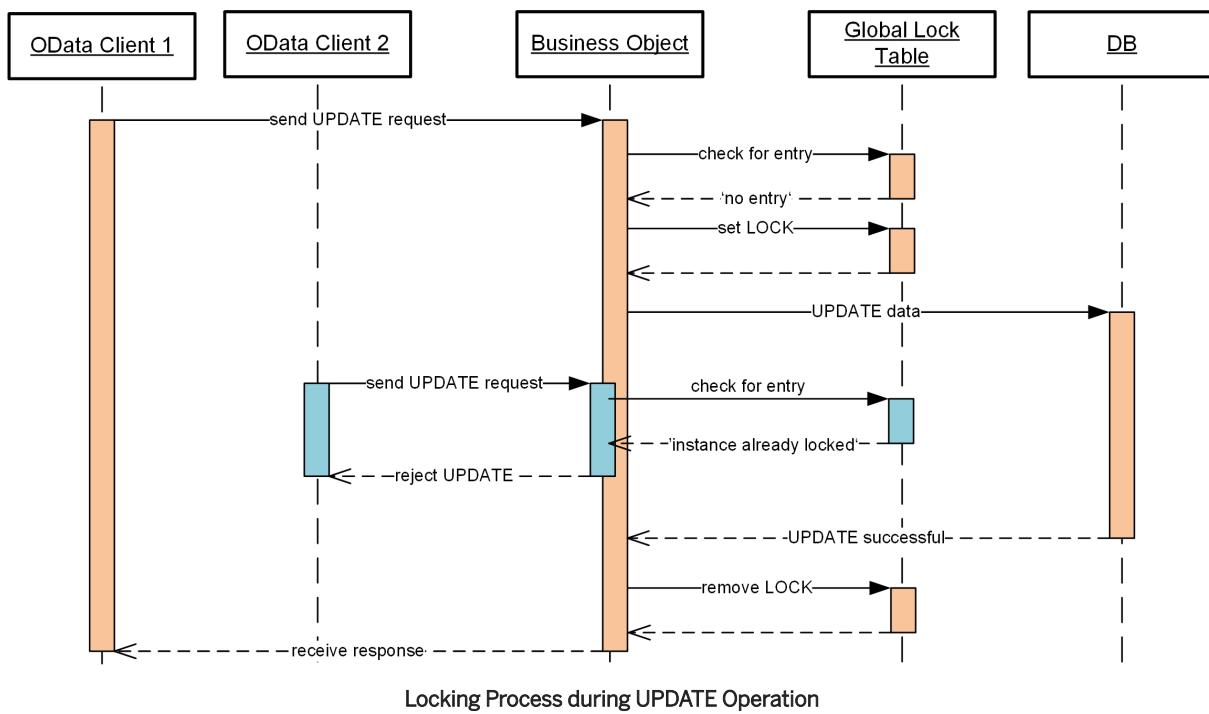
Pessimistic concurrency control prevents simultaneous modification access to data on the database by more than one user.

Pessimistic concurrency control is done by exclusively locking data sets for the time a modification request is executed. The data set that is being modified by one user cannot be changed by another user at the same time.

Technically, this is ensured by using a global lock table. Before data is changed on the database, the corresponding data set receives a lock entry in the global lock table. Every time a lock is requested, the system checks the lock table to determine whether the request collides with an existing lock. If this is the case, the request is rejected. Otherwise, the new lock is written to the lock table. After the change request has been successfully executed, the lock entry on the lock table is removed. The data set is available to be changed by any user again.

The lifetime of such an exclusive lock is tied to the session life cycle. The lock expires once the lock is actively removed after the successful transaction or with the timeout of the ABAP session.

The following diagram illustrates how the lock is set on the global lock table during an `UPDATE` operation.



The transaction of the client that first sends a change request makes an entry in the global lock table. During the time of the transaction, the second client cannot set a lock for the same entity instance in the global lock tables and the change request is rejected. After the successful update of client 1, the lock is removed and the same entity instance can be locked by any user.

For more information, see [SAP Lock Concept](#).

Locking in RAP

If a lock is defined for a RAP BO entity, it is invoked during the runtime of the following modify operations:

- [Update Operation \[page 67\]](#)
- [Delete Operation \[page 69\]](#)
- [Create by Association Operation \[page 71\]](#)
- [Action \[page 81\]](#).

The CREATE operation does not invoke the lock mechanism, as there is no instance whose keys can be written to the global lock table.

i Note

The locking mechanism does not check key values for uniqueness during CREATE. That means, the locking mechanism does not prevent the simultaneous creation of two instances with the same key values.

In the managed scenario, this uniqueness check is executed by the managed BO framework. In the unmanaged scenario, the uniqueness check must be ensured by the application code provided by the application developer.

To prevent simultaneous data changes in RAP business objects, the lock mechanism must be defined in the behavior definition. Before instance data is changed by RAP-modifying operations, the entity instance is then locked to prevent data from being changed by other users or transactions.

In managed scenarios, the business object framework assumes all of the locking tasks. You do not have to implement the locking mechanism in that case. If you do not want the standard locking mechanism by the managed business object framework, you can create an unmanaged lock in the managed scenario. This enables you to implement your own locking logic for the business object.

i Note

Whereas the managed BO runtime executes a uniqueness check for all dependent entities of the lock master entity, an unmanaged implementation must ensure that newly created instances are unique.

In unmanaged scenarios, however, the application developer has to implement the method for lock and implement the locking mechanism including the creation of the lock object. The method for lock is called by the orchestration framework before the relevant modifying operations are executed. The lock method calls the enqueue method of a lock object that was previously created to enter the lock for the relevant entity instance on the lock table. During the save sequence, after data has been successfully saved to the database, the lock is removed during the cleanup method, see [Save Sequence \[page 83\]](#).

Lock Master and Lock Dependent

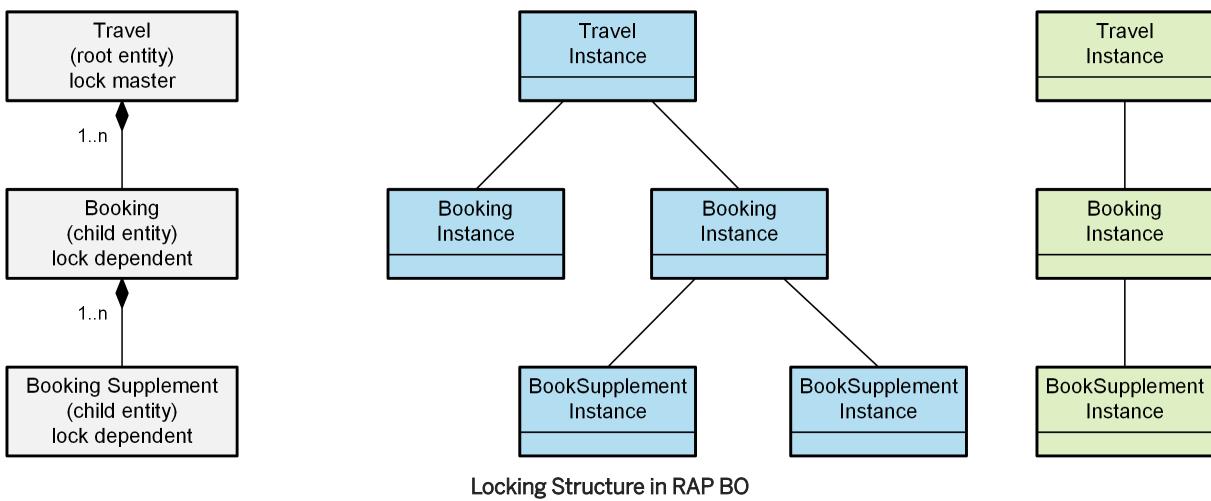
In RAP, locking is not only restricted to the entity instance that is being modified. All related entities in the business object structure are involved if one entity instance is getting locked. The locking structure is defined in the behavior definition with the keywords `lock master` and `lock dependent by`. Every business object that supports locking must have at least one lock master entity. Whenever a lock is requested for a specific entity instance, its lock master and all dependent entity instances are locked for editing by a different user. This mechanism ensures that relevant data is not changed concurrently.

i Note

Currently, only root entities can be lock masters.

Lock dependent entities must always have a lock master that is superior to them in the business object structure.

The following diagram illustrates the structure of a business object with lock master and lock dependent entities.



If one entity instance of the blue BO tree receives a lock request, its lock master, the travel instance, is locked and with it all dependent entity instances of this travel instance. That means if one of the blue instances is locked, all blue instances are locked, but not the green instances of a different lock master entity instance.

Related Information

[Lock Definition \[page 94\]](#)

[Lock Implementation \[page 96\]](#)

4.1.2.3.2.1 Lock Definition

In RAP business objects, enqueue locks are used for pessimistic concurrency control. You define the lock in the behavior definition of the business object entity.

Whenever the locking mechanism is defined for a business object entity in the behavior definition, the orchestration framework calls the lock method to lock the respective data set and its lock dependencies. You define which entities are lock masters and which entities are dependent on other entities. The lock mechanism is only defined in the behavior definition in the interface layer. Its use for a business service must not be specified in a projection behavior definition.

i Note

In managed scenarios, locking must always be enabled. Therefore, the lock definition is always included in the template of the behavior definition.

The lock mechanism is defined using the following syntax elements in the **behavior definition**:

```
...
define behavior for CDSEntity [alias AliasedEntityName]
implementation in class ABAP_CLASS_NAME [unique]
...
lock master [page 95] [unmanaged [page 95]] | lock dependent by
_AsocToLockMaster [page 95]
```

```
...
{
    ...
    association _AssocToLockMaster { }
}
```

Lock Master

Lock master entities are locked on each locking request on one of their lock dependent entities. The method `FOR LOCK` in unmanaged scenarios must be implemented for the lock master entities. The lock implementation must include locking all dependent entities.

i Note

Currently, only root entities are allowed to be lock masters.

Lock dependent entities must always have a lock master that is superior to them in the business object composition structure.

Lock Master Unmanaged

In the managed scenario, you can define an unmanaged lock if you do not want the managed BO framework to assume the locking task. In this case the lock mechanism must be implemented in the method `FOR LOCK` of the behavior pool, just like the lock implementation in the unmanaged scenario, see [Unmanaged Scenario \[page 96\]](#).

Lock Dependent

An entity is defined as lock dependent if locking requests shall be delegated to its lock master entity. The lock master entity of lock dependent entities is identified via the association to the lock master entity. This association must be explicitly specified in the behavior definition, even though it is implicitly transaction-enabled due to internal BO relations, for example a child/parent relationship. The association must also be defined in the data model structure in the CDS views and, if needed, redefined in the respective projection views.

Related Information

[Pessimistic Concurrency Control \(Locking\) \[page 91\]](#)

[Lock Implementation \[page 96\]](#)

4.1.2.3.2.2 Lock Implementation

If a lock mechanism is defined for business objects in the behavior definition, it must be ensured that the lock is set for modifying operations.

Managed Scenario

The lock mechanism is enabled by default for business objects with implementation type `managed`. The template for the behavior definition comes with the definition for at least one lock master entity and the implementation of the lock mechanism is provided by the managed BO framework.

If you define an unmanaged lock for a managed business object, you have to implement the method `FOR LOCK`, just like in the unmanaged scenario. It is then invoked at runtime.

Unmanaged Scenario

Just like any other operation in the unmanaged scenario, the lock must be implemented by the application developer. To enable locking, the method `FOR LOCK` must be implemented.

For a complete example, see [Implementing the LOCK Operation \[page 330\]](#).

4.1.2.3.2.2.1 <method> FOR LOCK

Implements the lock for entities in accordance with the lock properties specified in the behavior definition.

The `FOR LOCK` method is automatically called by the [orchestration framework \[page 817\]](#) framework before a changing (`MODIFY`) operation such as `update` is called.

Declaration of <method> FOR LOCK

In the behavior definition, you can determine which entities support direct locking by defining them as `lock master`.

i Note

The definition of `lock master` is currently only supported for root nodes of business objects.

In addition, you can define entities as `lock dependent`. This status can be assigned to entities that depend on the locking status of a parent or root entity. The specification of `lock dependent` contains the association by which the runtime automatically determines the corresponding `lock master` whose method `FOR LOCK` is executed when change requests for the dependent entities occur.

The declaration of the predefined `LOCK` method in the behavior definition is the following:

```
METHODS lock_method FOR LOCK  
  [IMPORTING] lock_import_parameter FOR LOCK entity.
```

The keyword `IMPORTING` can be specified before the import parameter. The name of the import parameter `lock_import_parameter` can be freely selected.

The placeholder `entity` refers to the name of the entity (such as a CDS view) or to the alias defined in the behavior definition.

Import Parameters

The row type of the import table provides the following data:

- ID fields
All elements that are specified as a key in the related CDS view.

i Note

The compiler-generated structures `%CID`, `%CID_REF`, and `%PID` are not relevant in the context of locking since locking only affects persisted (non-transient) instances.

Changing Parameters

The `LOCK` method also provides the implicit `CHANGING` parameters `failed` and `reported`.

- The `failed` parameter is used to log the causes when a lock fails.
- The `reported` parameter is used to store messages about the fail cause.

You have the option of explicitly declaring these parameters in the `LOCK` method as follows:

```
METHODS lock_method FOR LOCK  
  IMPORTING lock_import_parameter FOR LOCK entity  
  CHANGING failed TYPE DATA  
        reported TYPE DATA.
```

Implementation of `method FOR LOCK`

The RAP lock mechanism requires the instantiation of a lock object. A lock object is an ABAP dictionary object, with which you can enqueue and dequeue locking request. For tooling information about lock objects, see .

The `enqueue` method of the lock object writes an entry in the global lock tables and locks the required entity instances.

An example on how to implement the `method FOR LOCK` is given in [Implementing the LOCK Operation \[page 330\]](#).

Related Information

[Implicit Returning Parameters \[page 738\]](#)

4.2 Business Service

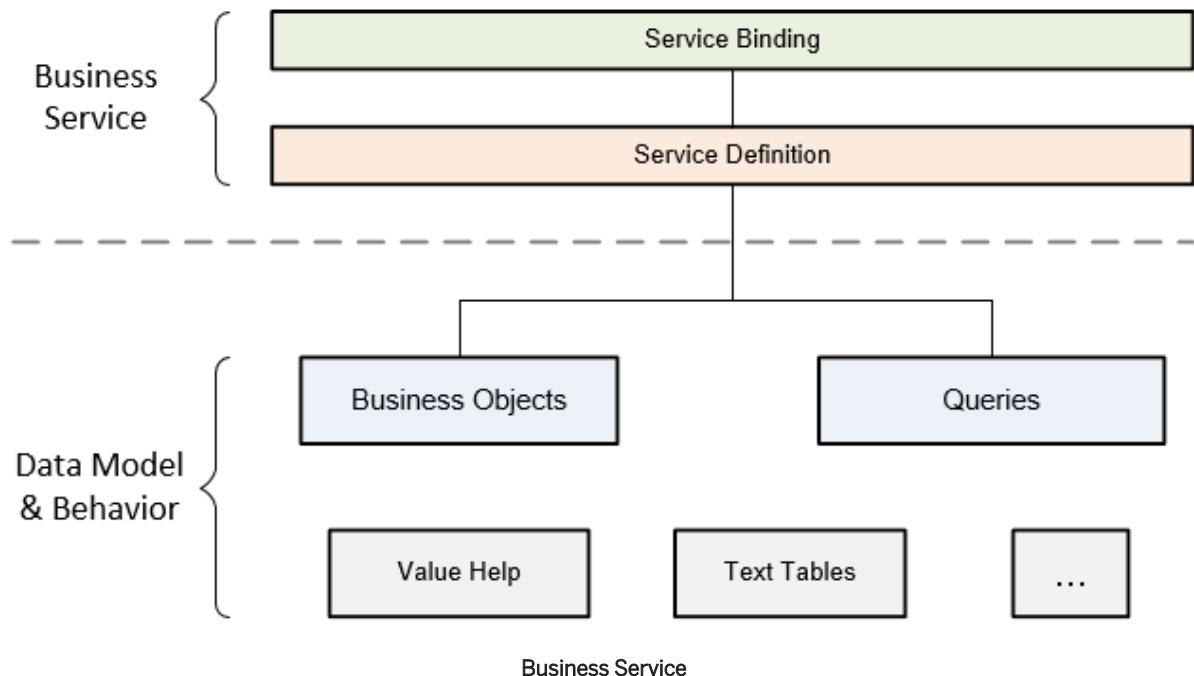
Definition

The ABAP development platform can act in the roles of a **service provider** and a **service consumer** (such as SAP Fiori UI client).

In the context of the ABAP RESTful programming model, a business service is a RESTful service which can be called by a consumer. It is defined by exposing its data model together with the associated behavior. It consists of a service definition and a service binding.

Business Services in the ABAP RESTful Programming Model

As illustrated in the figure below, the programming model distinguishes between the **data model and behavior** and the service that is defined by exposing these data model together with the behavior. The data model and the behavior layer contain domain-specific semantic entities like business objects, list views, and analytical queries, and, in addition, related functionality such as value help, feature control, and reuse objects.



A **business object** (BO) is a common term used to represent a real-world artifact in enterprise application development such as the *Product*, the *SalesOrder* or the *Travel*. In general, a business object contains multiple nodes such as *Items* and *ScheduleLines* (data model) and common transactional operations such as creating, updating and deleting data and additional application-specific operations, such as the *Approve* operation in a *SalesOrder* business object. All modifying operations for all related business objects form the transactional behavior model of an application.

Value help supports the end user when entering data on user interfaces based on these business objects.

In read-only development scenarios, the Fiori UI technology offers a list report (ALV-like) functionality based on a **query** in the back end. A list report UI like this provides a scrollable list of items that are automatically inserted into the list from the underlying data source.

Both the business objects and the queries require value help for the end user to select data according to filter values.

Separation Between the Service Definition and the Service Binding

In SAP Fiori UI, many role-based and task-oriented apps are based on the same data and related functionality must be created to support end users in their daily business and in their dedicated roles. This is implemented by reusable data and behavior models, where the data model and the related behavior is projected in a service-specific way. The **service definition** is a projection of the data model and the related behavior to be exposed, whereas the **service binding** defines a specific communication protocol, such as OData V2, and the kind of service to be offered for a consumer. This separation allows the data models and service definitions to be integrated into various communication protocols without the hassle of re-implementation.

Example

Let us assume that a business object *SalesOrder* is defined and implemented in the data model and the behavior layer with the related value help and authorization management. The service definition might expose

the [SalesOrder](#) and several additional business objects such as the [Product](#) and the [BusinessPartner](#) as they are included in a service binding for an OData V2 service.

Related Information

[Service Definition \[page 108\]](#)

[Service Binding \[page 111\]](#)

4.2.1 Business Object Projection

The business object projection in the ABAP RESTful Programming Model is an ABAP-native approach to project and to alias a subset of the business object for a specific business service. The projection enables flexible service consumption as well as role-based service designs.

Introduction

A service projection layer is required for a flexible service consumption of one business object. The basic [business object \[page 805\]](#) is **service agnostic**. That means, this BO is built independently from any [OData service \[page 816\]](#) application. The basic BO comprises the maximum range of features that can be applicable by a service that exposes this BO. The projection layer is the first layer in the development flow of the ABAP RESTful Programming Model that is **service specific**. When projecting the basic BO, you define the real manifestation of a business object in an OData service. The business object projection entails that part (the subset) of the BO structure and behavior that is relevant for the respective service, including denormalization of the underlying data model. Furthermore, the projection layer contains service-specific fine-tuning which does not belong to the general data model layer, for example UI annotations, value helps, calculations or defaulting.

Why Using Projections?

By using a projection layer for your business object, you gain flexibility in the service consumption. The general business object can be extended without affecting the already existing business service. This layering with projections enables robust application programming. The projection layer exposes the service specific subset of the general business object and thus, the service remains stable under modification of the underlying business object. In addition, aliasing in the projection views allows context-specific adaptions of the business object for a service.

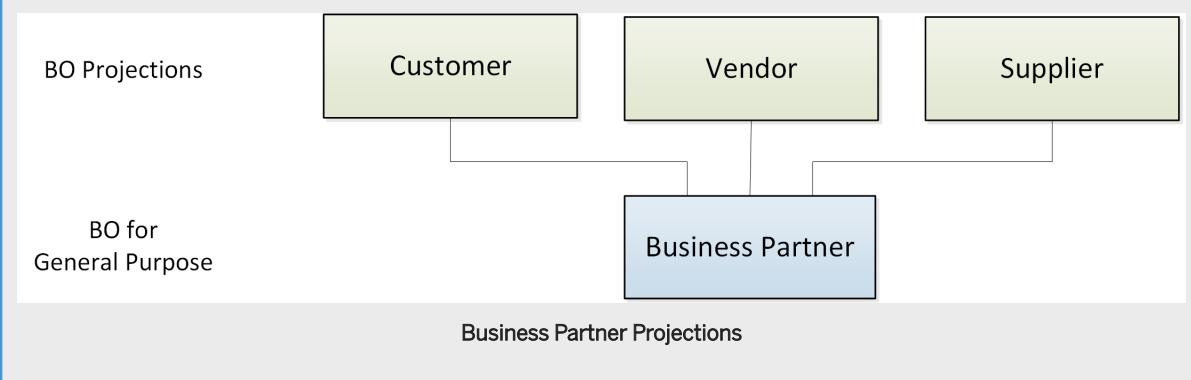
The projection layer also enables one business object to be exposed in an OData service for a Fiori UI and for a stable Web API. The service-specific differences can then be implemented in the respective projection layers. For example, UI specifications are defined only in the BO projection that is exposed for the UI service. Furthermore, with projections, you cannot only define the type of the service, but you can also design role-

based services. One business object for general purpose is exposed for more than one context-specific projection as specialized business object. The most prominent example is the business partner BO, which is exposed as customer, vendor, or supplier. In the projection, you can use that subset of the business partner BO that is relevant for the respective specialization.

• Example

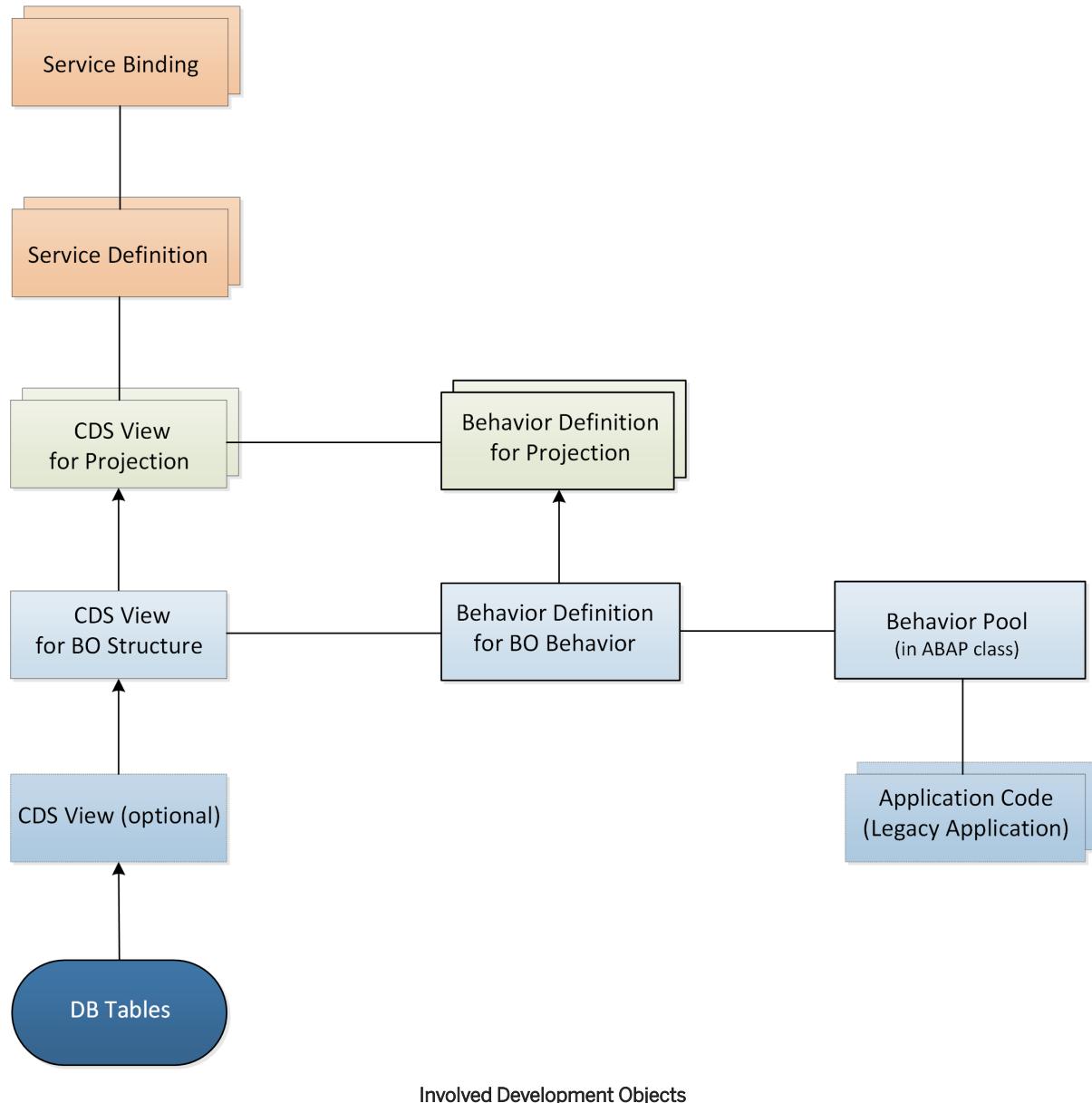
The basic BO of a business partner contains a wide range of CDS elements and behavior options. Depending on the concrete realization of the business partner, that is, depending on which role the business partner is assigned to, the structure of the data model and the behavior in the BO projection might vary. In the role of a customer, which is a typical projection of the business partner, the business partner projection contains the standard data available for business partners and in addition, sales arrangements. Sales arrangements contain data that is related to specific sales areas and used for the purposes of sales. All these characteristics must already be available in the basic BO and are then selected as a subset of the general business partner pool of elements and functionalities.

Imagine the business partner is enriched with characteristics for a new role of a business partner, for example a supplier. You can add the necessary additional elements, for example delivery information, to the data model and the behavior implementation in the business partner BO without affecting the already existing BO projections.



How to Use BO Projections?

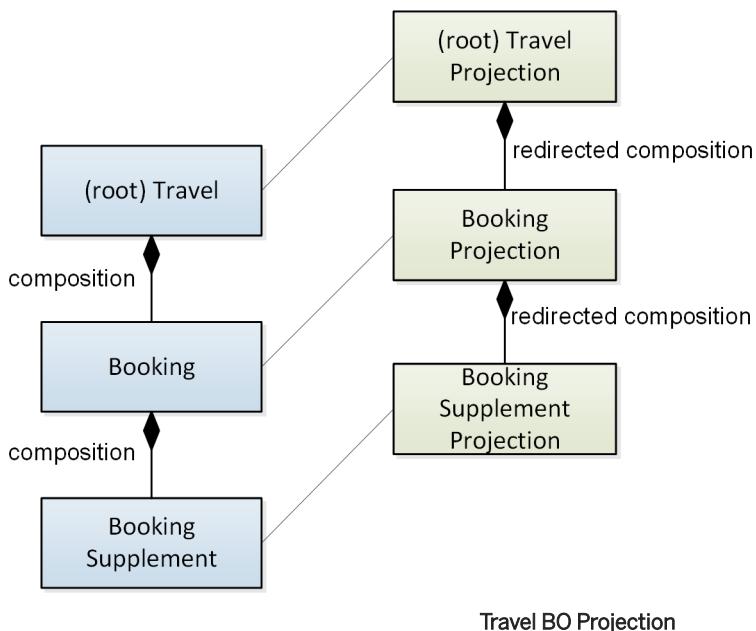
The design time artifacts to create an OData service that includes a projection layer are the following:



To create a projection layer for a business object, you need to create two projection artifacts:

- **CDS Projection Views**

The projection of the data model is done in one or more CDS projection views, depending on the number of nodes of the underlying BO. The CDS projection views use the syntax element `as projection on <ProjectedEntity>` to mark the relationship to the underlying [projected entity \[page 817\]](#). As opposed to the former consumption views, they do not create another SQL view. Since they only provide the consumption representation of the projected entity, they do not need an ABAP Dictionary representation. If one BO entity is projected, the root and all parent entities must be projected as well. The root entity has to stay the root entity and must be defined as root projection view. The compositions are redirected to the new target projection entity.



For a detailed description on CDS projection views and their syntax, see [CDS Projection View \[page 103\]](#).

- **Projection Behavior Definition**

The projection of the behavior is done in a behavior definition of type `projection`, which is declared in the header of the behavior definition. According to this type, only syntactical elements for projections can be used. Only behavior that is defined in the underlying behavior definition can be reused in the projection behavior definition. You cannot define or implement new behavior. The projection behavior always refers back to the behavior implementation of the underlying business object.

For more information on projection behavior definitions and their syntax, see [Projection Behavior Definition \[page 107\]](#).

! Restriction

In the current version of the ABAP RESTful Programming Model, it is not possible to migrate a classic CDS consumption view to a CDS projection view. It is recommended to delete and recreate the CDS consumption view as CDS projection view.

4.2.1.1 CDS Projection View

Projection views provide means within the specific service to define service-specific projections including denormalization of the underlying data model. Fine-tuning, which does not belong to the general data model layer is defined in projection views. For example, UI annotations, value helps, calculations or defaulting.

CDS projection views are defined in data definition development objects. The wizard for data definitions provides a template for projection views. For a detailed description on how to create projection views, see [Creating Projection Views \[page 773\]](#).

For the CDS view projection, a subset of the CDS elements is projected in the projection view. These elements can be aliased, whereas the mapping is automatically done. That means, the elements can be renamed to match the business service context of the respective projection. It is not possible to add new persistent data elements in the projection views. Only the elements, that are defined in the underlying data model can be

reused in the projection. However, it is possible to add virtual elements to projection views. These elements must be calculated by ABAP logic.

You can add new read-only associations in the projection view. This can be relevant to display additional information on the UI, like charts etc. It is not possible, however, to denormalize fields from new associated entity in the projection view. New associated entities cannot be accessed transactionally. Associations, including compositions, that are defined in the projected CDS view can be used in the projection CDS view. However, associations or compositions might change their target, if the target CDS view is also projected. This is especially relevant for compositions as the complete BO is projected and therefore the composition target changes. In case of a changed target, the association or composition must be redirected to the new target. The projection view comes with a new syntax element to express the target change.

Syntax for CDS Projection Views

The syntax of CDS projections views is similar to CDS views. However, some annotations and syntax elements are omitted as a projection view is a direct projection of the underlying CDS views and thus inherits annotations.

To define a CDS projection view, the following syntax is used:

```
@EndUserText.label: 'EndUserText'  
@AccessControl.authorizationCheck: #VALUE  
[@view anno]  
/* Definition of projection view */  
[define] [root] view entity ProjectionViewName  
/* Defines the data source for the projection.*/  
as projection on ProjectedEntity [as ProjectedEntityAlias ]  
/* New read-only associations  
association [min..max] to TargetEntity [as _Alias] on OnCondition  
/* Subset of elements from the projected entity */  
{  
    /* Fields from the projected entity*/  
    [@element_annot]  
    ElemtName           [as ElemtAlias] ,  
    /*Localized element */  
    [ [@element_annot]  
        Assoc.Element2      [as Elemt2Alias] : localized , ]  
    /* Cast element */  
    [ [@element_annot]  
        cast Elemt3Name : {DataElement | ABAPType }      [as Elemt3Alias] , ]  
    [ [@element_annot]  
        virtual Elemt4Name : {DataElement | ABAPType } , ]  
    /* Associations from the projected entity with possible redirections */  
    [ _Association : [redirected to ProjectionViewTarget], ]  
    /* Redirected compositions */  
    [ _Composition : redirected to composition child ChildProjectionView, ]  
    /* Redirected association to parent */  
    [ _ParentAssoc : redirected to parent ParentProjectionView ]  
}  
[WHERE [NOT] Condition ]
```

Explanation

A CDS projection consists of the projection view definition and elements. Further characteristics are the following:

A projection does not define an additional SQL view; it uses the one that is introduced in the projected entity. Likewise, the same set of key elements as in the projected entity must be used in the projection entity.

If the projected entity is the root node of the business object, the projection view must be a root as well.

CDS projection views can only be based on CDS views. Stacking of projection views is not allowed. You cannot create a projection view based on a projection view.

You can define new read-only associations in projection views. Composition or to-parent associations cannot be defined.

The element list of the projection view is a subset of the element list in the projected entity. In this element list, no new fields or associations can be introduced. Elements from the projected entity can be exposed directly. But also elements that are retrieved via a path expression to an associated entity with cardinality 1 can be exposed. The syntax for this is the same as in BO-interface views: `TargetEntity.TargetElement [as ElemAlias]`. If you want to use text elements from an associated view in your service, these elements must be included in the projection view via their association. Language-dependent texts can be denormalized in projection views to allow text search and filtering in Fiori UIs. The keyword `localized` identifies these language-dependent elements.

Elements from the projected view can be cast to a different data type. The syntax is the same as in BO-interface views.

In projection views, you can define virtual elements as additions to the persistent data model. The values for these elements are calculated during runtime with predefined interface methods.

Associations that are defined in the projected entity can be reused. If the target of the association does not change in the projection layer, the associations can be exposed directly. If the target entity is also projected, the association must be redirected to the target projection view. Redirections must also be done for compositions and to-parent-associations as the whole BO must be projected, if one node is projected.

The select list of the CDS projection view can be restricted using `WHERE` with a condition expression. This filter is respected in queries.

i Note

In transactional operations, including the transactional `READ`, the `where` clause in projection views is not respected. Applications must ensure that the instances that are created, updated, or read via the projection conform to the `where` clause.

Annotation Propagation to Projection Views

Annotations that are defined in the projected entity on element level are completely propagated to the projection view. That means, annotation values remain the same in the projection view. Once the same annotation is used on the same elements in the projection view, the values are overwritten and only the new values are valid for the respective element.

If you use an annotation with an element reference in the projected entity and the reference element is aliased in the projection entity, the reference is not drawn to the element in the projection view, due to the name change. In such a case, you have to redefine the annotation in the projection view and use the alias name of the element in the annotation value.

• Example

The amount and currency elements are annotated in the underlying CDS view with @Semantics annotations to support the semantic relationship of the elements.

```
define root view /DMO/I_Travel
...
{
    key travel_id,
...
    @Semantics.amount.currencyCode: 'currency_code'
    total_price,
    @Semantics.currencyCode: true
    currency_code,
...
}
```

Both @Semantics annotations are propagated to the projection view. However, the element currency_code is aliased in the projection view and therefore the reference to the correct element is not established. Hence, the relationship is broken and the metadata of a possible OData service will not resemble this semantic relationship.

To avoid this, you have to reannotate the amount element with the reference to the aliased element.

```
define root view entity /DMO/C_Travel as projection on /DMO/I_Travel
...
{
    key travel_id,
...
    @Semantics.amount.currencyCode: 'CurrencyCode'
    total_price      as TotalPrice,
    currency_code   as CurrencyCode,
...
}
```

Defining UI Specifics in the Projection Views

From a design time point of view, the projection layer is the first service-specific layer. If the resulting OData service is a UI service, all UI specifications or other service-specific annotations must be defined in the CDS projection views via [CDS annotations \[page 809\]](#). The following UI specifics are relevant on the projection BO layer:

- UI annotations defining position, labels, and facets of UI elements
- Search Enablement
- Text elements (language dependent and independent)
- Value Helps

! Restriction

In the current version of the ABAP RESTful Programming Model, CDS projection views can only be used to project CDS view entities. Other entities, such as custom entities are not supported.

Related Information

[Creating Projection Views \[page 773\]](#)

[Providing a Data Model for Projections \[page 239\]](#)

4.2.1.2 Projection Behavior Definition

A projection behavior definition provides means to define service-specific behavior for a BO projection.

The behavior definition with type `projection` is created equally to other types of behavior definitions. When creating a behavior definition based on a CDS projection view, the syntax template directly uses the projection type. For more information, see [Working with Behavior Definitions \[page 760\]](#).

In a behavior definition, only behavior characteristics and operations that are defined in the underlying behavior definition can be defined for the BO projection. The syntax for this is `use <Element>`.

Syntax: Behavior Definition for Projection

The syntax in a projection behavior definition is the following:

```
projection;
  define behavior for ProjectionView alias ProjectionViewAlias
    /* use the same eTag defined in underlying behavior definititon */
    use etag
  {
    /* define static field control */
    field ( readonly ) ProjViewElem1;
    field ( mandatory ) ProjViewElem2;
    /* expose standard operations defined in underlying behavior definition */
    use create;
    use update;
    use delete;
    /* expose actions or functions defined in underlying behavior definition */
    use action|function ActionName [result entity ProjResultEntity][as ProjAction]
[external ExtProjname];
    /* expose create_by_association for child entities defined in underlying
behavior definition */
    use association _Assoc { create; }
  }
```

Explanation

The keyword `use` exposes the following characteristics or operations for the service-specific projection. In the projection, only elements can be used that were defined in the underlying behavior definition. These elements can be

- ETag
- standard operations
- actions
- functions
- `create_by_association`

Every operation that you want to expose to your service must be listed in the projection behavior definition. New aliases can be assigned for actions and functions. Projection behavior definitions do not have a behavior implementation. The complete behavior is realized by mapping it to the underlying behavior.

The definitions that already restrict the character of the underlying BO are automatically applied in the BO projection and cannot be overwritten. This is the case for:

- locking
- authorization
- feature Control

If no static field control is defined in the underlying behavior definition, you can add this definition in the projection behavior definition. If it is already defined in the underlying behavior definition, you cannot define the opposite in the projection layer. If you do, you will get an error during runtime. New dynamic field control cannot be defined in the projection behavior definition, as there is no option to implement the feature.

Related Information

[Working with Behavior Definitions \[page 760\]](#)

[Providing Behavior for Projections \[page 258\]](#)

4.2.2 Service Definition

Definition

A business service definition (short form: service definition) describes which CDS entities of a data model are to be exposed so that a specific business service, for example, Sales Order handling, can be enabled. It is an ABAP Repository object that describes the consumer-specific but protocol-agnostic perspective on a data model. It can directly access the standard *ABAP Workbench* functionality, such as transports, syntax checks, element information, and activation. Its transport type is SRVD.

Use

A service definition represents the service model that is generically derived from the underlying CDS-based data model.

You use a service definition to define which data is to be exposed as a business service using one or more business service bindings (short form: service bindings). A service definition itself is independent from the version or type of the protocol that is used for the business service.

→ Remember

When going to expose a data model as a service, you can make use of a service definition only in connection with at least one service binding.

Syntax: DEFINE SERVICE

```
@EndUserText.label: 'text'  
@<Annotation_1>  
...  
@<Annotation_n>  
  
DEFINE SERVICE service_definition_name  
{  
    EXPOSE cds_entity_1 [AS alias_1];  
    EXPOSE cds_entity_2 [AS alias_2];  
    EXPOSE ...  
    EXPOSE cds_entity_m [AS alias_m];  
}
```

Explanation

The source code of the actual service definition is preceded by the optional CDS annotation `@EndUserText.label` that is available for all objects which can contain CDS annotations. The annotation value is a character string with a maximum of 60 characters. The specified `text` value should consist of a meaningful short text that describes the service in the original language of the source code. The size of the set of service-relevant CDS entities depends on the kind of functionality the service should provide to the application scenario. However, the respective dependencies must be considered:

Depending on the needs of your scenario, further optional annotations `@<Annotation_1>` ... `@<Annotation_n>` can be specified.

The service definition is initiated with the `DEFINE SERVICE` keyword followed by the name for the service definition.

i Note

This name for the service definition follows the naming rules that are common to ABAP Repository objects:

- Names are not case-sensitive.
- A name can have a maximum of 30 characters.
- A name can consist of letters, numbers, underscores (_), and slashes (/).
- A name must start with a letter or a slash character (in the case of namespaces).
- The CDS keywords, as well as the CDS entity names cannot be used as names.
- Corresponding with naming conventions, there is no need for a prefix or suffix in the service definition name. **See also:** [Naming Conventions for Development Objects \[page 780\]](#)

The source code of a service definition is created within a single bracket { ... } that is used to group all the related CDS entities (including their associations with the relevant entities) which are to be exposed as part of an individual service.

The name of each individual CDS entity to be exposed follows the `EXPOSE` keyword. This is followed by an optional alias name, which is initiated by the `AS` keyword. An alias defines an alternative name for a CDS entity to be exposed. As a result, when accessing the service, the alias names are used instead of the current entity names. Thus, you have the option of assigning syntactically uniform identifiers in the service definition and thus decoupling the service semantics from the concrete technical names resulting from the data definition.

Similar to the CDS syntax rules, each statement is completed by a semicolon.

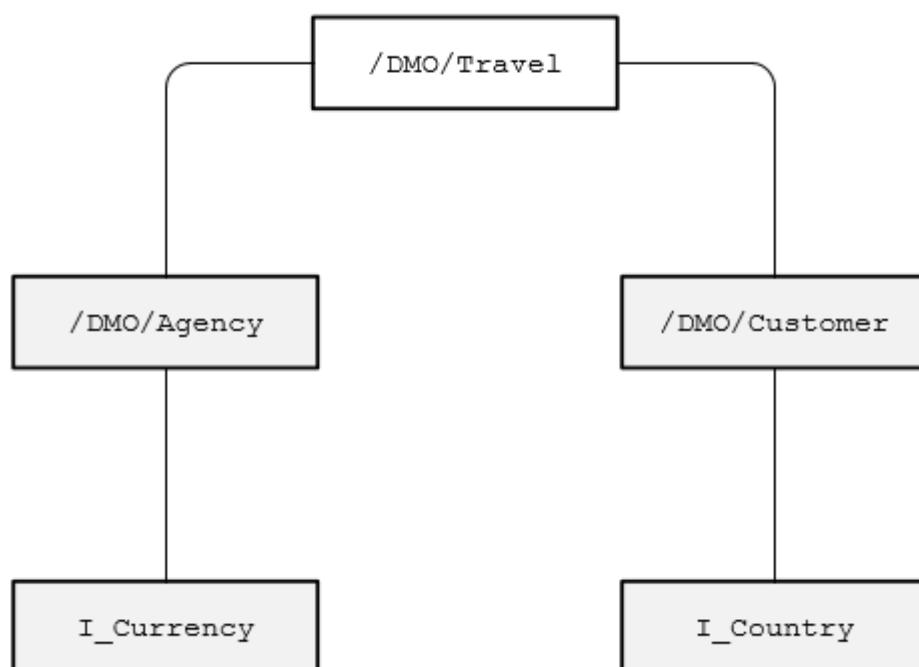
! Restriction

Whenever you edit a service definition in the context of the ABAP RESTful programming model to create a UI service or a Web API, you can only use CDS views or custom entities as entities. **You cannot use abstract CDS entities [page 808]** in service bindings for in this service exposure use case. The usage of abstract entities would cause a dump error.

However, abstract entities can be used in a service definition, but only if they are created using OData client proxy tools (service consumption use case).

Example

/DMO/Travel, defines associations to the entities /DMO/Customer and /DMO/Agency. In addition, associations to the entities I_Currency and I_Country must be included.



Data model for the TRAVEL service

The following example shows the corresponding source code for the service definition /DMO/TRAVEL. The travel management service to be defined in this way includes all dependencies that come from the root entity /DMO/I_TRAVEL.

```
@EndUserText.label: 'Service for managing travels'  
define service /DMO/TRAVEL  
{  
    expose /DMO/I_TRAVEL      as Travel;  
    expose /DMO/I_AGENCY      as TravelAgency;  
    expose /DMO/I_CUSTOMER    as Passenger;  
    expose I_Currency         as Currency;  
    expose I_Country          as Country;  
}
```

i Note

In this example, the service definition is based on CDS entities that originate from different namespaces.

Related Information

[Service Binding \[page 111\]](#)

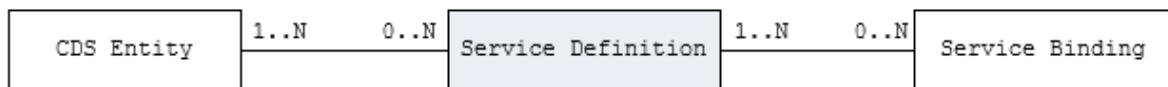
4.2.3 Service Binding

Definition

The business service binding (short form: service binding) is an ABAP Repository object used to bind a service definition to a client-server communication protocol such as OData. Like any other repository object, the service binding uses the proven infrastructure of the ABAP Workbench, including the transport functionality. The transport type of a service binding is SRVB.

Use

As shown in the figure below, a service binding relies directly on a service definition that is derived from the underlying CDS-based data model. Based on an individual service definition, a plurality of service bindings can be created. The separation between the service definition and the service binding enables a service to integrate a variety of service protocols without any kind of re-implementation. The services implemented in this way are based on a separation of the service protocol from the actual business logic.



Relationship Between the Data Model, the Service Definition and the Service Binding

Parameters

The following parameters are used to characterize a service binding:

Service Name

Defines a unique system-wide name for the service and is identical to the name of the service binding.

→ Tip

We recommend using the prefix `API_` for *Web API* services and the prefix `UI_` for *UI* services.

Binding Type

The binding type primarily specifies the specific protocol which is implemented with the service binding. The OData models of the current version of ABAP Platform support the **OData version 2.0** (ODATA V2).

→ Remember

The Open Data Protocol (OData) enables the creation of HTTP-based services, which allow resources identified using Uniform Resource Identifiers (URIs) and defined in an abstract data model to be published and edited by Web clients using HTTP messages. OData is used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems, and traditional Web sites.

This parameter also determines the way a service is offered to a consumer. There are two options:

UI	UI service
A UI service makes it possible to add a SAP Fiori elements UI or other UI clients to the service.	
Web API	Service exposed as Web API
A service that is exposed as Web API is used for unforeseen extensions of the SAP software. It is offered by an application for an unknown consumer for direct interaction with the user.	

Service Version

The versioning of services is made by a version number which is assigned to a service binding.

The next higher version is created by adding another service definition to the existing service binding. By means of this further service definition, functional changes or extensions (compared to the previous version) are exposed. And, vice versa, the version number can be decreased by removing a service definition from the service binding.

Activation State

This parameter determines whether the service is activated as a *local service endpoint* (in the service catalog of the current system) or not.

Service URL

The derived URL (as a part of the service URL) is used to access the OData service starting from the current ABAP system. It specifies the virtual directory of the service by following the syntax: `/sap/opu/odata/<service_binding_name>`.

Example

The following figure shows the key parameters and information that are stored for a service binding in the corresponding editor of *ABAP Development Tools* (ADT). This example is the first version of a UI service that

implements the OData V2 protocol and is intended to manage travel data. The editor also provides information on the entire entity set as well as on the navigation path of the respective entity.

The screenshot shows the SAP ABAP Application Programming Model Concepts interface. The title bar reads "Service Binding: /DMO/UI_TRAVEL_U_V2".

General Information: This section describes general information about this service binding. The binding type is listed as "ODATA V2 (UI - User Interface: Consumed in SAPUI5 Apps)".

Service Versions: Define service versions associated with the service binding. A table lists one version: "0001 /DMO/TRAVEL_U". Buttons for "Add..." and "Remove" are available.

Service Version Details: View information on selected service version. It includes a "Local Service Endpoint" section with "Acti" and "Deactivate" buttons, and a "Service URL" field containing "/sap/opu/odata/DMO/UI_TRA". A "Preview" button is also present.

Maintain Authorization Default Values: This section is currently empty.

Entity Set and Association: A tree view showing associations between entities: TravelAgency, BookingSupplement, Booking, and their associations to BookSupplement, Connection, and Customer. An option "Open Fiori Elements App Preview" is available for the to_BookSupplement association.

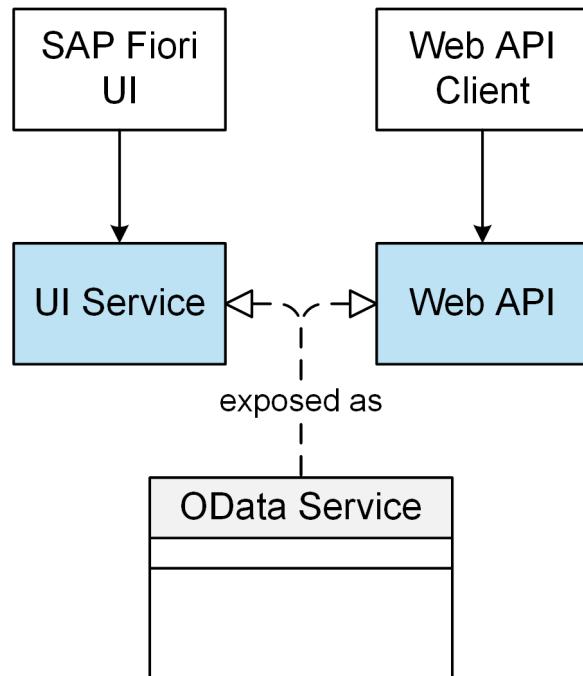
Service Binding in ADT

Related Information

[Service Definition \[page 108\]](#)

4.3 Service Consumption

An OData service [\[page 816\]](#) can be exposed as a UI service, that can be consumed by an SAP Fiori UI, or as a Web API that can be consumed by any OData client.



- <https://help.sap.com/viewer/468a97775123488ab3345a0c48cadd8f/LATEST/en-US/03265b0408e2432c9571d6b3feb6b1fd.html> [https://help.sap.com/viewer/468a97775123488ab3345a0c48cadd8f/LATEST/en-US/03265b0408e2432c9571d6b3feb6b1fd.html]
- [Creating an OData Service \[page 24\]](#)
- [Developing a Web API \[page 365\]](#)

UI service

An OData service that is exposed as a UI service is consumable by an SAP Fiori Elements app. Every front-end configuration, which is manifested in the back-end development object (for example, UI annotations), is exposed within the metadata of the service. That means, a Fiori UI reads the information in the metadata and creates the matching UI for the service. These UI settings can be enhanced and overwritten in the SAP Web IDE.

A UI service can be previewed with the Fiori Elements preview in the [service binding \[page 807\]](#) tool. The preview mocks a real UI app and has the same look and feel as a Fiori Elements app. It is therefore a powerful tool to test the UI of your OData service already in the backend. However, it does not substitute the development in the SAP Web IDE.

For further information about the Fiori Elements preview in the service binding, see [Previewing the Resulting UI Service \[page 33\]](#).

For more information about SAPUI5 to get more information about creating a deployable SAP Fiori app, see [Developing Apps with SAP Fiori Elements](#).

Web API

An OData service that is exposed as a Web API comes without any UI specific information in the metadata. It is the public interface for any OData client to access the OData service. For example, you can consume a Web API from another OData service.

Web APIs require a life cycle management. It must be possible to define the release, the version, and the possible deprecation of the Web API. This functionality is enabled in the [service binding \[page 807\]](#) tool.

For more information about Web APIs, see [Developing a Web API \[page 365\]](#).

4.4 Runtime Frameworks

The runtime frameworks [SAP Gateway](#) and the [Orchestration Framework](#) are the frameworks that manage the generic runtime for OData services built with the ABAP RESTful Programming Model. As a developer you do not have to know the concrete inner functioning of these frameworks, as many development tasks are automatically given. However, the following sections provide a high-level overview.

SAP Gateway

SAP Gateway provides an open, REST-based interface that offers simple access to SAP systems via the Open Data Protocol (OData).

As the name suggests, the gateway layer is the main entry point to the ABAP world. All services that are created with the ABAP RESTful Programming Model provide an OData interface to access the service. However, the underlying data models and frameworks are based on ABAP code. SAP Gateway converts these OData requests into ABAP objects to be consumed by the ABAP runtime.

Orchestration Framework

The orchestration framework dispatches the requests for the business object (BO) or the query. It receives the ABAP consumable OData requests from the Gateway layer, forwards it to the relevant part of the business logic and interprets the matching ABAP calls for it. For transactional requests, the orchestration framework delegates the requests to the BO and calls the respective method of the BO implementation. For query requests, the framework executes the query. Depending on the implementation type, the [BO \[page 805\]](#) or the [query \[page 817\]](#) runtime is implemented by a framework or by the application developer.

If [locks \[page 814\]](#) are implemented, the orchestration framework executes first instance-independent checks and sets locks. For the [eTag \[page 812\]](#) handling, the framework calls the necessary methods before the actual request is executed.

i Note

The orchestration framework is also known under the name [SADL \[page 818\]](#) (Service Adaptation Description Language). Apart from the runtime orchestration, the SADL framework is also responsible for essential parts in the query and BO runtime.

Examples

The OData client sends a `DELETE` request, which is converted to an object that is understandable for ABAP. The orchestration framework analyzes this ABAP object and triggers the `MODIFY` method for `DELETE` of the business object to execute the `DELETE` operation on the database table. Depending on the implementation type (managed or unmanaged), the code for the `MODIFY` method is generically available or must be implemented by the application developer.

Likewise, if an OData request contains a query option, such as `$orderby`, the Gateway layer converts it to the query capability `SORT`. Then, the orchestration framework takes over and delegates the query capability to the query. Depending on the runtime type (managed or unmanaged), the query is executed by the generic framework in case of managed type or by the self-implemented runtime in case of unmanaged type. For a managed query, the generic framework converts the requests to ABAP SQL statements to access the database.

4.5 Entity Manipulation Language (EML)

Entity Manipulation Language (in short: EML) is a part of the ABAP language that is used to control the business object's behavior in the context of ABAP RESTful programming model. It provides a type-safe read and modifying access to data in transactional development scenarios.

Consumption of Business Objects Using EML

Business objects that are implemented with the ABAP RESTful architecture based on the behavior definition and implementation of the interaction phase and save sequence in behavior pools can be consumed not only by means of OData protocol (Fiori UIs, or Web APIs) but also directly in ABAP by using the EML syntax.

There are two flavors of EML available:

- A **standard API**, which uses the signature of the business object related entities
- A **generic API** for dynamic/generic consumption of business objects.

The latter is typically used for generic integration of business objects into other frameworks, such as the [Cloud Data Migration Cockpit](#) or the [Process Test Framework](#).

The standard API is used whenever the "target" business object is statically specified. It provides code completion and static code checks. This typed API provides statements for read-only access to data (`READ ENTITIES`), as well as for modifying data access (`MODIFY ENTITIES`) and for triggering the save sequence (`COMMIT ENTITIES`).

→ Remember

One of the uses cases of EML is the writing of test modules as ABAP Unit tests. As an ABAP application developer, it gives you the ability to test transactional behavior of business objects for each relevant operation that is defined in the behavior definition.

EML Syntax

The EML syntax consists of three major statements:

MODIFY ENTITIES

This statement includes all operations that change data of entities. This is handled by the statement `MODIFY ENTITIES`, which provides the following operations:

- create
- create by association
- update
- delete
- actions, that is, any modify operation that cannot be covered with create, update, or delete, for example, if parameters are required.

Syntax (short form)

```
MODIFY ENTITY EntityName
  CREATE [FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance_c
  CREATE BY \association_name [FIELDS ( field1 field2 ... ) WITH] | [FROM]
  it_instance_cba
  UPDATE [FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance_u
  DELETE FROM it_instance_d
  EXECUTE action_name FROM it_instance_a
    [RESULT et_result_a]
  [FAILED ct_failed]
  [MAPPED ct_mapped]
  [REPORTED ct_reported].
```

You can use the short form of the `MODIFY` statement in special cases when calling modify operations for one entity only - without any relation to a business object. In this case, however, it is required that you specify the full name of the CDS entity instead of the alias name. The keywords for modify operations are: `CREATE`, `UPDATE`, `DELETE`, and `EXECUTE` for actions. Each operation has a table of instances as input parameters. For actions, you can also add a `RESULT` parameter `et_result_a` in case the action is defined to provide a result.

→ Remember

Note that at least one of the operations must specify a valid EML statement. The sequence of the operations is irrelevant.

The `MODIFY` statement provides a `FIELDS (field1 field2 ...) WITH` option with a field list for direct creating, creating by association and updating entity's instance data. That means all fields that are to be updated for an instance or are required for creating an entity's instance are specified in the field list.

The different operations can be mixed within one EML statement. It is possible, for example, to combine a create, update and action operation of an entity (even related to the same instance(s)) in one statement.

You can complete the `MODIFY` statement with the response parameters [FAILED \[page 738\]](#), [MAPPED \[page 739\]](#), and [REPORTED \[page 739\]](#). As is common in ABAP, you can use either existing variables (`ct_failed`) with matching data types or add an inline declaration (`DATA (ct_failed)`).

Syntax (long form)

```
MODIFY ENTITIES OF RootEntityName
  ENTITY entity_1_name
    CREATE [FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance1_c
    CREATE BY \association1_name [FIELDS ( field1 field2 ... ) WITH] | [FROM]
  it_instance1_cba
    UPDATE [FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance1_u
    DELETE FROM it_instance1_d
    EXECUTE action FROM it_instance1_a
      [RESULT et_result_a]
  ENTITY entity_2_name
    CREATE FROM it_instance2_c
    ...
  ENTITY entity_3_name
  ...
  [FAILED ct_failed]
  [MAPPED ct_mapped]
  [REPORTED ct_reported].
```

The long form of the `MODIFY` statement allows you (like a complex OData request) to collect multiple modify operations on multiple entities of one business object that is identified by `RootEntityName`. Grouped by entities, the relevant operations are listed according to the previous short form syntax. If aliases for the entities are defined in the behavior definition, they should be referred in the long form syntax.

Examples:

- Action implementation: [copy_travel \[page 183\]](#)
- Action implementation: [set_status \[page 185\]](#)

READ ENTITIES (Transactional READ)

This statement includes all operations that do not change data of entities (read-only access).

The current version of the ABAP RESTful programming model provides the following read operations:

- read: for read access to entities by using a key
- read by association: for read access to child entities by using parent key(s).

Syntax (short form)

```
READ ENTITY EntityName
  [FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance
  RESULT et_result
  BY \association_name
  [FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance_rba
  RESULT et_result_rba
  LINK et_link_rba
  [FAILED ct_failed].
```

The short syntax directly specifies the `EntityName` (CDS view name). The consumer using EML has therefore read access to data for this entity only. In this case, an alias of the entity cannot be used since no context of a business object is known.

The `READ` statement provides a `FIELDS (field1 field2 ...) WITH` option with a field list for direct reading, reading by association of entity's instance data. The fields of an instance to be read are specified in the field list.

The `READ` statement always has the addition `RESULT`, since this specifies the variable that receives the `READ` operation's results. This variable contains the target instance(s) with all fields of the entity.

The read-by-association operation provides an additional target variable that follows after the `LINK` keyword addition. The `et_link_rba` variable contains only a list of key pairs: key of the source entity and the key of the target entity. Target variables must have either a matching type or are declared inline, for example, `DATA (et_link_rba)`.

You can complete the `READ` statement with the response parameter `FAILED` (table containing error keys).

Examples:

- Action implementation: [copy_travel \[page 183\]](#)
- Action implementation: [set_status \[page 185\]](#)
- Implementing Validations [page 200]
- Implementing Dynamic Feature Control [page 190]

Syntax (long form)

```
READ ENTITIES OF RootEntityName
ENTITY entity_1           " entity alias name
[FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance_1
    RESULT it_result
    BY \association1_name
[FIELDS ( field1 field2 ... ) WITH] | [FROM] it_instance_rba
    RESULT et_result_rba
    LINK et_link_rba
ENTITY entity_2_name      " entity alias name
[FIELDS ( ...
    ...
ENTITY entity_3_name      " entity alias name
    ...
[FAILED ct_failed].
```

The long form `READ ENTITIES` allows you to group read operations for multiple entities of a business object that is specified by `RootEntityName`. The long form allows using aliases defined in the behavior definition for specifying the entities.

COMMIT ENTITIES

[Modify operations \[page 816\]](#) that are executed within a behavior pool or by an ABAP program, do not cause any data changes at the database level. This is because they are applied only to the transactional buffer and the buffer content disappears at the end of the ABAP session. This means the save sequence must be triggered in this case.

The save sequence is triggered by the `COMMIT ENTITIES` statement. The runtime infrastructure translates this statement into the save chain starting with `finalize()` performing the final calculations before data can be persisted. If the subsequent `check_before_save()` call is positive for all transactional changes, the point-of-no-return is reached. From now on, a successful `save()` is guaranteed by all involved BOs. After the

point-of-no-return, the `adjust_numbers()` call can occur to take care of late numbering. The `save()` call persists all BO instance data from the transactional buffer in the database.

In its simplest form, the statement `COMMIT ENTITIES` does not have any parameters:

Syntax (simplest form)

```
COMMIT ENTITIES.
```

Syntax (long form)

```
COMMIT ENTITIES
  [RESPONSE OF root_entity_name_1
    [FAILED ct_failed]
    [REPORTED ct_reported]]
  [RESPONSE OF root_entity_name_2
    [FAILED ct_failed]
    [REPORTED ct_reported]].
```

The syntax of the `COMMIT ENTITIES` statement also provides the `RESPONSE` clause that allows to retrieve the response information of one or more business objects involved that are grouped by their root entity. For each root entity (`root_entity_name_1`, `root_entity_name_2`), a `RESPONSE` clause can contain parameters `FAILED` and `REPORTED`.

`COMMIT ENTITIES` saves all BOs that were changed within the [LUW \[page 815\]](#).

→ Remember

The crucial criteria of success for the `COMMIT ENTITIES` EML statement is not `FAILED` response is empty, but `SY-SUBRC = 0`. Background: Indirect failures (in remote business objects) cannot be foreseen by using the response parameter.

Related Information

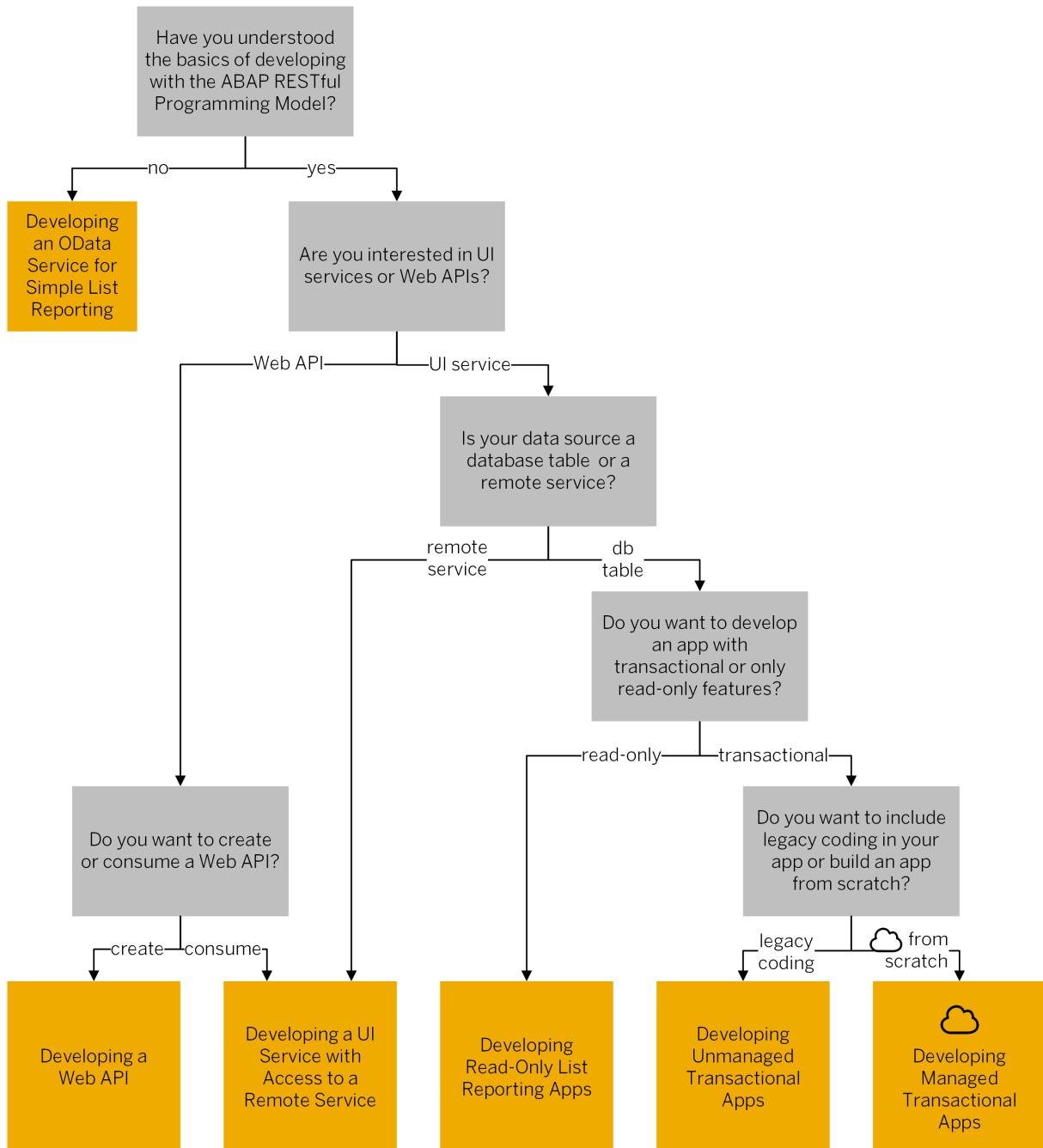
[Consuming Business Objects with EML \[page 469\]](#)

5 Develop

The development guides in this section provide a detailed step-by-step description on how to use and exploit the ABAP RESTful Programming Model in end-to-end scenarios.

The guides in this section focus on specific development tasks. It depends on your initial situation and on the aimed outcome of your development, which guide meets your requirements best.

Follow the path in the diagram and ask yourself the questions to find out which development guide helps you with your development task. You get further information about the steps and the development guides by hovering over the image.



- [Develop \[page 121\]](#)
- [Develop \[page 121\]](#)
- [Develop \[page 121\]](#)
- [Develop \[page 121\]](#)
- [Developing an OData Service for Simple List Reporting \[page 14\]](#)
- [Developing Read-Only List Reporting Apps \[page 123\]](#)
- [Developing Unmanaged Transactional Apps \[page 263\]](#)

- [Developing a UI Service with Access to a Remote Service \[page 369\]](#)
- [Developing a Web API \[page 365\]](#)
- [Develop \[page 121\]](#)
- [Develop \[page 121\]](#)
- [Developing Managed Transactional Apps \[page 144\]](#)

Use the navigation in the image or the following links to navigate to the development guides:

[Developing Read-Only List Reporting Apps \[page 123\]](#)

[Developing Unmanaged Transactional Apps \[page 263\]](#)

[Developing Managed Transactional Apps \[page 144\]](#)

[Developing a Web API \[page 365\]](#)

[Developing a UI Service with Access to a Remote Service \[page 369\]](#)

5.1 Developing Read-Only List Reporting Apps

Based on existing persistent data sources, you create and implement a query for an OData service to get a running app with useful read-only features.

Introduction

In this chapter you learn how to develop an OData service including multiple read-only features. This OData service can be consumed by a Fiori Elements application or by any other OData client.

Starting from the elementary list reporting scenario that was introduced in the [Getting Started \[page 13\]](#) section, you may want to add some further features to the existing elementary OData service. First of all, the end user wants to be able to navigate to a second information layer for a flight connection to retrieve more detailed information about the flight, such as flight dates or plane types. Secondly, if the list report contains a large number of rows, it becomes difficult for end users to find the information they need. To make it easier to find this information, you can implement search capabilities or label the app elements differently than their presets in the data model layer in the back end. You might also want to enable value helps for selection field dialogs. All these features are implemented using specific CDS annotations, which you, as the developer, add to the source code of the respective CDS view.

We assume that you know the development steps to create an OData service, as described in the [Getting Started \[page 13\]](#) section. The following guide uses the CDS view `/DMO/I_Connection_R` and the OData service `/DMO/UI_FLIGHT_R_V2` as the basis for a more elaborate OData service with further read-only features. You learn how to expand the data model with associated CDS views and how to include useful read-only functions in the OData service.

You are guided step-by-step through the application model and expand the OData service that you created in the [Getting Started](#) section with the following features and query capabilities.

- [Expanding the data model with additional CDS views \[page 126\]](#)

- Navigating between CDS views with associations [page 131]
- Changing UI Field Labels and Descriptions [page 133]
- Displaying Text for Unreadable Elements [page 135]
- Providing Value Help for the Selection Fields [page 138]
- Adding Search Capabilities [page 141]

This scenario implements the query case. We firstly define a CDS data model for which we define modeled query capabilities. The service that was created in the [Getting Started](#) scenario is reused and the new CDS entities including their query capabilities are exposed for this service.

i Note

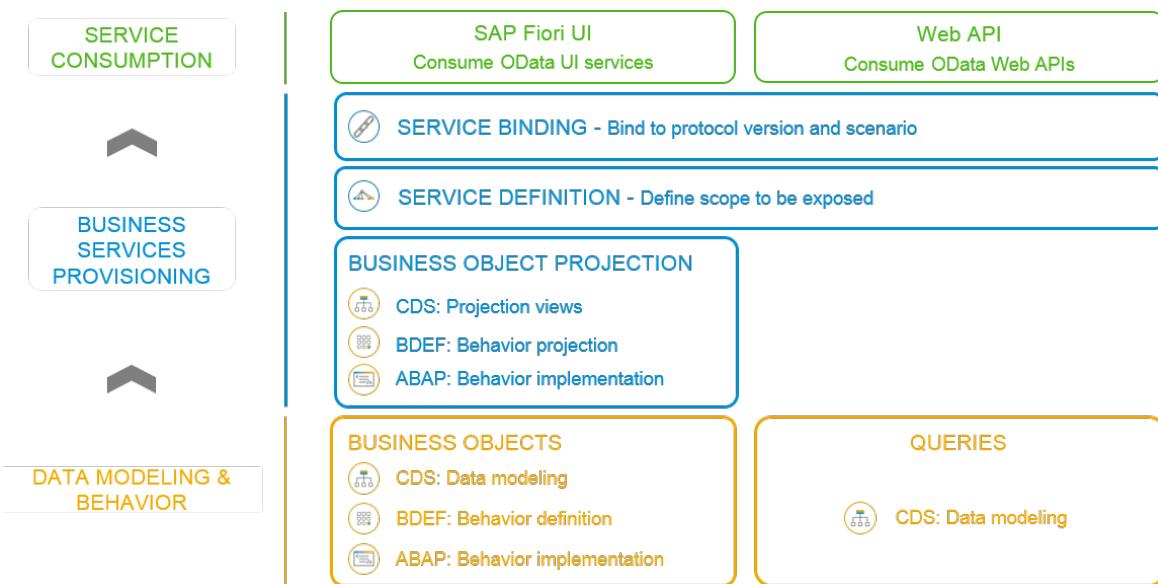
Via ABAPGit You can import the service including the related development objects into your development environment for comparison and reuse. You find the service in the package `/DMO/FLIGHT_READONLY`. The suffix for development objects in this development guide is `_R`.

For information about downloading the ABAP Flight Reference Scenario, see [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

Prerequisites

Developing the scenario that is described in the subsequent chapters requires the following:

-  You have access to and an account for **SAP Cloud Platform, ABAP environment**.
- You have installed ABAP Development Tools (ADT).
SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.
- To recreate the demo scenario, the **ABAP Flight Reference Scenario** must be available in your ABAP system.
You can download the complete reference scenario from [GitHub: Downloading the ABAP Flight Reference Scenario \[page 13\]](#).
- You have understood the development steps to create an OData service as described in [Developing an OData Service for Simple List Reporting \[page 14\]](#).
In particular, you are able to use the existing OData service `/DMO/UI_FLIGHT_R_V2` to check and try out the new implementation with the preview tool.



Objectives

By the end of this development guide you are able to:

- Apply and enhance your knowledge about how to create and expand an OData service
- Implement associations between CDS views
- Expose new CDS views for an existing OData service
- Use `@EndUser.Text` annotations
- Implement text associations
- Develop value helps for input fields
- Implement search capabilities

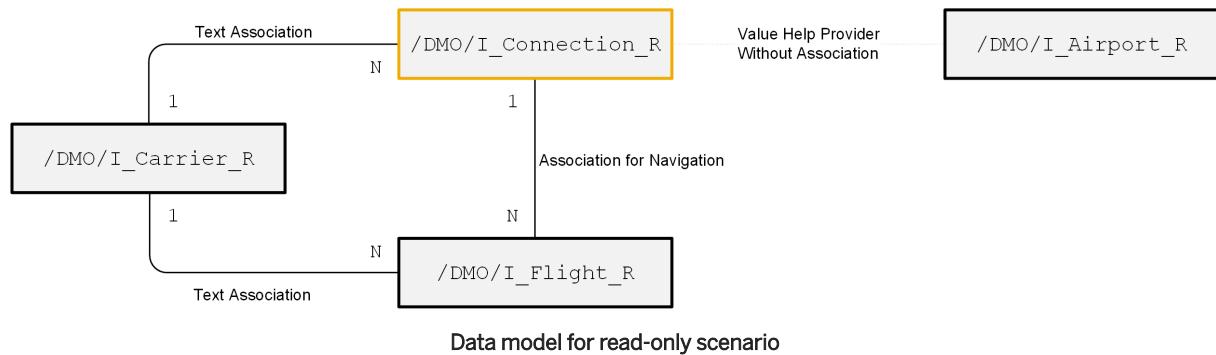
5.1.1 Determining the Data Model for the Read-Only Scenario

Starting Point

We build the list reporting scenario based on the service that was created in the [Getting Started \[page 13\]](#) scenario. The CDS view `/DMO/I_Connection_R` is reused in the following scenario and included in a broader data model.

In the first step of this development guide, you expand the data model with additional CDS views to create a full blown reference app for flights. The resulting running app is able to find flights based on connections and the airports involved. From a technical point of view, this means that the CDS data model is expanded with three other CDS views, one of which is connected using associations to enable navigation from one CDS view to

another. The data model and the relationship between the CDS views that you work on to implement read-only features is illustrated in the following figure. All items represent a self-contained data model of a CDS view that retrieves data from a database table included in the [ABAP Flight Reference Scenario \[page 774\]](#).



As you can see in the figure above, the entry point of the flight scenario is the well-known CDS view `/DMO/I_Connection_R`. It manages data for flight connections. From here, you can navigate to more detailed flight information with flight dates and plane information. The detailed information in the carrier CDS view is used to display the full name of the airline, whereas the CDS view `/DMO/I_Airport_R` is used as a value help provider view for the airport elements in the connection CDS view.

These items only represent a part of the [ABAP Flight Reference Scenario](#). The other items in the reference data model are applied in other development scenarios.

5.1.1.1 Defining CDS Views

To provide a complete data model for a read-only OData service, CDS views have to be defined first.

Context

In the introductory guide, you learned how to define CDS views based on an existing persistent data source. Repeat the process for further CDS views to expand the data model for the flight scenario. We assume that the CDS view `/DMO/I_Connection_R` already exists. This view defines the starting point of the app.

For a detailed description of the following steps, look at [Defining the Data Model with CDS \[page 16\]](#).

Procedure

1. For each new CDS view, do the following:
 - a. Create a data definition using the creation wizard in your development package.
 - b. Implement the CDS view as a data model and define the name of the CDS view as well as a name for the database view as given in the table below.

i Note

Naming CDS views: Since CDS views are (public) interface views, they are prefixed with `I_` in accordance with the VDM (virtual data model) naming convention. In addition, we add the suffix `_R` for the read-only implementation type to the view name.

Data Definition

CDS View Name

Database View Name	Data Source	Description
<code>/DMO/I_FLIGHT_R</code>	<code>/dmo/flight</code>	Provides information about the available flights including flight dates and plane information.
<code>/DMO/I_Flight_R</code>	(DB table)	You can navigate to this view once you have chosen a flight connection.
<code>/DMO/I_CARRIER</code>	<code>/dmo/carrier</code>	Provides information about the airlines that operate the flights.
<code>/DMO/I_Carrier</code>	(DB table)	This view is later used as a value help provider view for the view <code>/DMO/I_CONNECTION_R</code> .
<code>/DMO/ICARRIER_RE</code>		

i Note

As this CDS view is reused in more scenarios, it does not carry a suffix.

<code>/DMO/I_AIRPORT</code>	<code>/dmo/airport</code>	Provides information about the involved airports.
<code>/DMO/I_Airport</code>	(DB table)	
<code>/DMO/IAIRPORT_RE</code>		

i Note

As this CDS view is reused in more scenarios, it does not carry a suffix.

2. For each new CDS view, do the following:
 - a. Insert all elements provided from the data source.
 - b. Add a meaningful alias to each CDS element.
3. Add the relevant `@Semantics` annotations where necessary.

The following table helps you to identify the elements that need a semantic annotation to ensure that semantic data types are consumed in the right way. The elements that must be annotated are currency codes and their corresponding amount as well as units of measure and their corresponding quantities. Whereas the existing CDS view `/DMO/I_Connection_R` contains a unit of measure that needs to be annotated, the new CDS views contain currency codes and the corresponding price elements that need to be annotated.

More information: [Relating Semantically Dependent Elements \[page 21\]](#).

CDS View Name	Element	Annotation
/DMO/I_FLIGHT_R	Price	<pre>@Semantics.amount.currencyCode: '<CurrencyCodeRef>'</pre> <p>This annotation establishes the link between the amount and the currency element. Reference the element with the currency code of the element price CurrencyCode.</p>
	CurrencyCode	<pre>@Semantics.currencyCode: true</pre> <p>This annotation ensures that the element is handled as a currency element.</p>
/DMO/I_CARRIER	CurrencyCode	<pre>@Semantics.currencyCode: true</pre> <p>This annotation ensures that the element is handled as a currency element.</p>

4. Insert the relevant UI annotations to ensure that the user interface is rendered properly where necessary.

Whereas the existing CDS view of the introductory guide in the getting started section requires many UI annotations, since it represents the landing list report page, the new CDS views only require a limited number of UI annotation. In fact, only the CDS view /DMO/I_Flight_R requires the @UI.lineItem: [{}] annotation to be applied to the relevant elements. Since the remaining views only function as text provider or value help provider views, they do not need any UI annotations at all.

A detailed description of this procedure is described in [Designing the User Interface for a Fiori Elements App \[page 35\]](#).

5. Activate the CDS view.

The resulting source code for each data definition is displayed at the end of this topic: [Data Model for Flight App \[page 128\]](#).

6. Include the new CDS entities in the existing service /DMO/UI_FLIGHT_R_V2, which you created in the introductory guide in the getting started section: [Developing an OData Service for Simple List Reporting \[page 14\]](#). To do this, expose the CDS entities in the service definition /DMO/FLIGHT_R. The following codeblock displays the source code of the updated service definition [/DMO/FLIGHT_R](#)

```
@EndUserText.label: 'Read-Only E2E: SD for Flights'
define service /DMO/FLIGHT_R {
    expose /DMO/I_Connection_R as Connection;
    expose /DMO/I_Flight_R as Flight;
    expose /DMO/I_Carrier as Airline;
    expose /DMO/I_Airport as Airport;
}
```

5.1.1.2 Data Model for Flight App

The following listings display the source code of the CDS views that are involved in our read-only scenario.

i Expand the following listing to view the source code of the Connection CDS view.

Connection CDS view /DMO/I_Connection_R

This CDS view is the entry point of the flight reference app. It provides the elements for the connection search that the end user can use to find suitable flights for a journey.

This CDS view /DMO/I_Connection_R was already modeled in the introductory guide in the getting started section.

```
@AbapCatalog.sqlViewName: '/DMO/ICONNECT_R'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E: Data Model Connection'
@UI.headerInfo: { typeName: 'Connection',
                  typeNamePlural: 'Connections' }
define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
{
  @UI.facet: [
    { id: 'Connection',
      purpose: #STANDARD,
      type: #IDENTIFICATION_REFERENCE,
      label: 'Connection',
      position: 10 } ]
  @UI: {
    lineItem: [ { position: 10, label: 'Airline' } ],
    identification:[ { position: 10, label: 'Airline' } ] }
  key Connection.carrier_id      as AirlineID,
  @UI: {
    lineItem: [ { position: 20, label: 'Connection Number' } ],
    identification:[ { position: 20, label: 'Connection Number' } ] }
  key Connection.connection_id    as ConnectionID,
  @UI: {
    lineItem: [ { position: 30, label: 'Departure Airport Code' } ],
    selectionField: [ { position: 10 } ],
    identification:[ { position: 30, label: 'Departure Airport Code' } ] }
  Connection.airport_from_id     as DepartureAirport,
  @UI: {
    lineItem: [ { position: 40, label: 'Destination Airport Code' } ],
    selectionField: [ { position: 20 } ],
    identification:[ { position: 40, label: 'Destination Airport Code' } ] }
  Connection.airport_to_id       as DestinationAirport,
  @UI: {
    lineItem: [ { position: 50, label: 'Departure Time' } ],
    identification: [ {position: 50, label: 'Departure Time'} ] }
  Connection.departure_time      as DepartureTime,
  @UI: {
    lineItem: [ { position: 60, label: 'Arrival Time' } ],
    identification: [ {position: 60, label: 'Arrival Time'} ] }
  Connection.arrival_time        as ArrivalTime,
  @UI: { identification: [ {position: 70, label: 'Distance' } ] }      **
establishes the link between quantity and the corresponding unit of measure
@Semantics.quantity.unitOfMeasure: 'DistanceUnit'
  Connection.distance          as Distance,
  @Semantics.unitOfMeasure: true      ** defines the semantic content of the
element
  Connection.distance_unit       as DistanceUnit
}
```

i Expand the following listing to view the source code of the Flight CDS view.

Flight CDS view /DMO/I_Flight_R

This CDS view provides detailed information about the flights. It is displayed as a second facet in the UI, which means we only need a limited number of UI annotations.

```
@AbapCatalog.sqlViewName: '/DMO/IFLIGHT_R'
```

```

@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E': Data Model Flight
define view /DMO/I_Flight_R
    as select from /dmo/flight as Flight
{
    @UI.lineItem: [ { position: 10, label: 'Airline' } ]
    key Flight.carrier_id as AirlineID,
        @UI.lineItem: [ { position: 20, label: 'Connection Number' } ]
    key Flight.connection_id as ConnectionID,
        @UI.lineItem: [ { position: 30, label: 'Flight Date' } ]
    key Flight.flight_date as FlightDate,
        @UI.lineItem: [ { position: 40, label: 'Price' } ]
    @Semantics.amount.currencyCode: 'CurrencyCode'      ** establishes the link
between amount and currency code
    Flight.price as Price,
    @Semantics.currencyCode: true
    Flight.currency_code as CurrencyCode,      ** defines the semantic content
of the element
    @UI.lineItem: [ { position: 50, label: 'Plane Type' } ]
    Flight.plane_type_id as PlaneType,
    @UI.lineItem: [ { position: 60, label: 'Maximum Seats' } ]
    Flight.seats_max as MaximumSeats,
    @UI.lineItem: [ { position: 70, label: 'Occupied Seats' } ]
    Flight.seats_occupied as OccupiedSeats
}

```

i Expand the following listing to view the source code of the Carrier CDS view.

Carrier CDS view /DMO/I_Carrier

This CDS view is used as a text provider view for the main views of the app. It contains the text for the element AirlineID.

```

@AbapCatalog.sqlViewName: '/DMO/ICARRIER_RE'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E: Data Model Carrier'
define view /DMO/I_Carrier
    as select from /dmo/carrier as Airline
{
    key Airline.carrier_id as AirlineID,
        Airline.name as Name,
        @Semantics.currencyCode: true      ** defines the semantic content of
the element
        Airline.currency_code as CurrencyCode
}

```

i Expand the following listing to view the source code of the Airport CDS view.

Airport CDS view /DMO/I_Airport

This CDS view is used as a value help provider view for the connection view. It contains detailed information about the available airports that can be used for the value help.

```

@AbapCatalog.sqlViewName: '/DMO/IAIRPORT_RE'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E: Data Model Airport'
define view /DMO/I_Airport
    as select from /dmo/airport as Airport
{
    key Airport.airport_id as AirportID,
        Airport.name as Name,

```

```
Airport.city      as City,  
Airport.country   as CountryCode  
}
```

i Note

To obtain the full scope of a data model and to enable flexible service consumption, you can project the data model before creating a business service with CDS projection views. With a projection layer, you can extend the basic data model without affecting the already existing business service.

For more information, see [CDS Projection View \[page 103\]](#).

5.1.2 Implementing Associations for Existing CDS Views

Associations structure the relationships between CDS views. To enable navigation in the UI, associations must be implemented in CDS.

Context

You created the flight CDS view that provides detailed information about the available flights for the connection. To be able to navigate to this view in the UI, you have to implement an association from /DMO/I_Connection_R to /DMO/I_Flight_R.

An association defines a structural and unidirectional relationship between two CDS views. Associations are used to navigate from a source CDS view to a related target CDS view.

Associations are defined in the source CDS view. The connections between the two CDS views are established by a join condition applied to the respective elements of the source and target view. You need to define a cardinality for the target CDS view to define how many records of the target CDS view are associated with one record of the source.

Procedure

1. Open the source CDS view /DMO/I_Connection_R.
2. Define the association after the select statement.
 - a. Introduce the association with the keyword **association**.
 - b. Define `[min .. max]` for the cardinality of the target view.

i Note

The cardinality defines the minimum and maximum number of associated entries of the target view.

In our example, the cardinality is one-to-many, since one entry in /DMO/I_Connection_R is associated to many entries of /DMO/I_Flight_R. This complies with business logic since one connection can be operated by flights on different dates.

```
...
define view /DMO/I_Connection_R as select from /dmo/connection as
Connection
association [1..*]
```

- Specify the target CDS view and define an alias. An alias makes it easy to reference the association.

```
define view /DMO/I_Connection_R as select from /dmo/connection as
Connection
association [1..*] to /DMO/I_Flight_R as _Flight
```

- Specify the mapping condition for the CDS views.

In our example, the associated CDS views are mapped to two elements. Therefore both of them must be stated in the condition expression.

```
define view /DMO/I_Connection_R as select from /dmo/connection
association [1..*] to /DMO/I_Flight_R as _Flight on
$projection.AirlineID = _Flight.AirlineID
                                         and
$projection.ConnectionID = _Flight.ConnectionID
```

- Add the association to the element list in the CDS view.

```
{
  key Connection.carrier_id      as AirlineID,
...
/*Associations*/
    _Flight           ** use the alias to refer to the association
}
```

- For the UI: Provide a second facet to make it possible to navigate from the list report page to the detailed object page of the connection with corresponding flights as line items.

A detailed description of UI annotations can be found in [Defining UI Annotations \[page 36\]](#).

```
@UI.facet: [
  { id: 'Connection',
  purpose: #STANDARD,
  type: #IDENTIFICATION_REFERENCE,
  label: 'Connection',
  position: 10 },
  { id: 'Flight',
  purpose: #STANDARD,
  type: #LINEITEM_REFERENCE,
  label: 'Flight',
  position: 20,
  targetElement: '_Flight' }
]
```

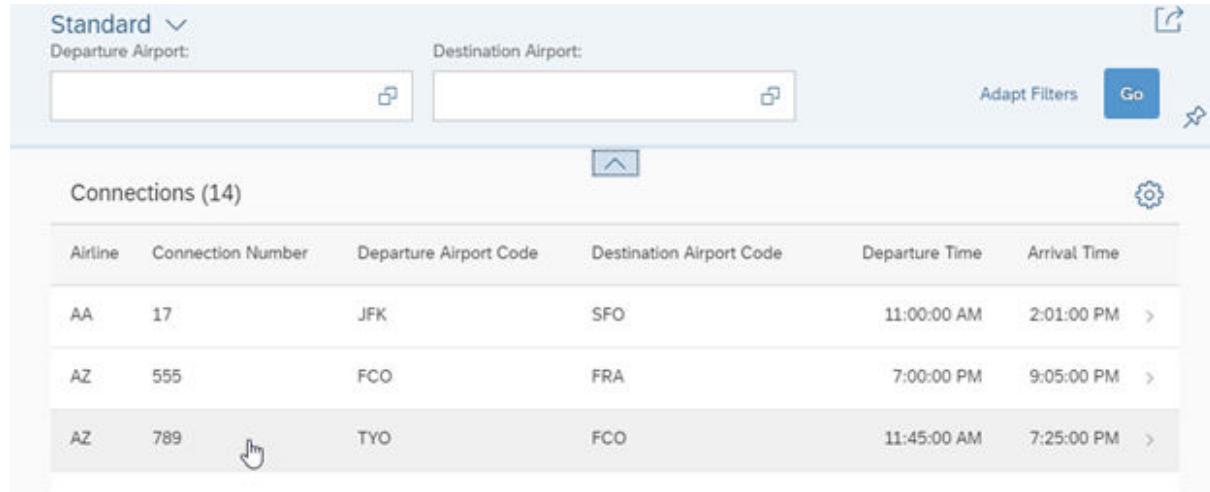
- Activate the CDS view.

Results

You have established a connection between the connection CDS view and the flight CDS view. With the right configuration of UI annotations, the end user can now navigate to the flight information by selecting one

connection in the list report app. The flight information is displayed as line items in the object page as can be seen in the following figures.

Click on one connection entry to navigate to the information for flights.

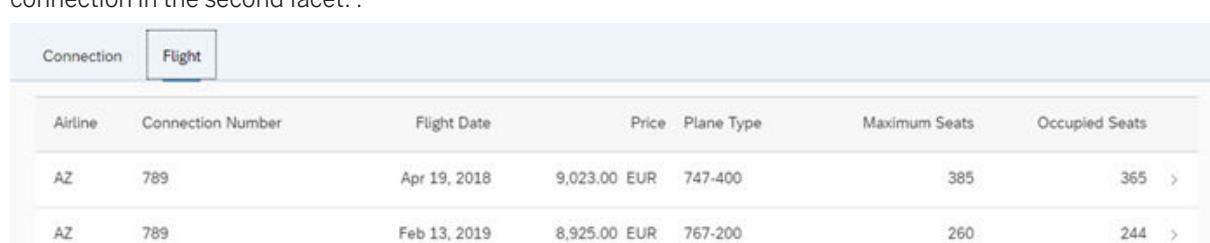


The screenshot shows a SAP Fiori application interface. At the top, there are input fields for 'Departure Airport' and 'Destination Airport' with small file icon buttons next to them. To the right are buttons for 'Adapt Filters', 'Go', and a search icon. Below the header is a table titled 'Connections (14)'. The table has columns: Airline, Connection Number, Departure Airport Code, Destination Airport Code, Departure Time, and Arrival Time. There are three rows of data:

Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time
AA	17	JFK	SFO	11:00:00 AM	2:01:00 PM >
AZ	555	FCO	FRA	7:00:00 PM	9:05:00 PM >

The third row, which contains the connection AZ 789 from TYO to FCO, has a mouse cursor pointing at it. Below the table, the text 'Selection of the connection AZ 789 from TYO to FCO' is displayed.

The object page then displays the information about the selected connection and the related flights for this connection in the second facet. .



The screenshot shows the object page for the selected connection AZ 789. It features two tabs at the top: 'Connection' and 'Flight', with 'Flight' being the active tab. The table below has columns: Airline, Connection Number, Flight Date, Price, Plane Type, Maximum Seats, and Occupied Seats. There are two rows of data:

Airline	Connection Number	Flight Date	Price	Plane Type	Maximum Seats	Occupied Seats
AZ	789	Apr 19, 2018	9,023.00 EUR	747-400	385	365 >
AZ	789	Feb 13, 2019	8,925.00 EUR	767-200	260	244 >

Object page displaying all flights of the connection AZ 789 from TYO to FCO

5.1.3 Changing UI Field Labels and Descriptions

Field labels and description help to customize the UI of the app.

Meaningful descriptions of elements that appear on the user interface are a key concept for working with an app and improving the user experience. It is essential that all items on the UI are readable and understandable for the end user. Although every data element in *ABAP Dictionary* is labeled with descriptions for presentation on the UI, sometimes you want to modify the description for a specific use case and give it a name other than the name predefined on the persistent database layer. In the CDS layer, database description labels can be redefined and given more information by using the annotations `@EndUserText.label: '<text>'` and `@EndUserText.quickInfo: '<text>'`.

The labels that are assigned in the CDS layer overwrite the database field descriptions and are exposed to the OData service. These labels are also propagated to the UI and displayed on the user interface if no UI labeling annotations exist that overwrite the CDS `EndUserText` annotations.

i Note

UI labeling annotations, such as `@UI.lineItem: [{ label: '' }]`, overwrite any `@EndUserText` labeling annotations on the UI. However, they are not manifested in the OData service metadata.

A tooltip can provide additional or more thorough information for the element. The tooltip is displayed on the UI as mouse over text. If no `@EndUserText` annotations are used, the text for the mouse over function is retrieved from the long description of the data element stored in [ABAP Dictionary](#).

In general, every text that is used in `EndUserText` annotations is translated into every relevant language by the SAP translation process, along with the labels that are given to the data elements.

For more information on how to integrate information for the end user in your data model, refer to [Adding Field Labels and Descriptions \[page 513\]](#).

Adding Labels for Elements with Selection Fields

In our flight scenario, we have equipped the CDS elements with `UI` annotations to display them adequately in the UI, which means that all elements are already represented with a meaningful label.

However, there is one use case where `EndUserText` labels are necessary to ensure coherence on the UI. Whereas `@UI.listItem` and `@UI.identification` offer the option to directly label list items and object page items with `UI` annotations, the `@UI.selectionField` annotation lacks this option. The selection field label is therefore retrieved from the database label and might not match the label we have given to the list item and the identification. In this case, it is useful to apply the `@EndUserText` annotations to those elements that represent selection fields to provide consistency.

• Example

In the CDS view `/DMO/I_Connection_R`, we defined two selection fields that are labeled differently than the corresponding list item. We use the `@EndUserText` annotation to acquire matching labels.

```
{...  
  @UI: {  
    lineItem: [ { position: 30, label: 'Departure Airport Code' } ],  
    selectionField: [ { position: 10 } ],  
    identification:[ { position: 30, label: 'Departure Airport Code' } ] }  
  @EndUserText.label: 'Departure Airport Code'      //*** Use the same  
label as in lineItem  
  Connection.airport_from_id  as DepartureAirport,  
  @UI: {  
    lineItem: [ { position: 40, label: 'Destination Airport Code'} ],  
    selectionField: [ { position: 20 } ],  
    identification:[ { position: 40, label: 'Destination Airport Code' } ] }  
  @EndUserText.label: 'Destination Airport Code'    //*** Use the same  
label as in lineItem  
  Connection.airport_to_id    as DestinationAirport,  
  ... }
```

Integrating the Mouse Over Function

If you want additional and longer information about an element, you can use the annotation `@EndUserText.quickInfo: <text>` to display a text when hovering over the element.

i Note

If you do not define a tooltip in CDS, the mouse over text displays the short description of the data element in *ABAP Dictionary*.

The screenshot shows a table titled "Connections (14)". The columns are: Airline, Connection Number, Departure Airport Code, Destination Airport Code, Departure Time, and Arrival Time. A row for AA with Connection Number 17 is selected. The "Connection Number" cell contains the value "17". A tooltip "Flight Reference Scenario: Carrier ID" is displayed above the cell. The table has a header row with column names and a footer row with times.

Mouse over displaying long text of data element

To change the text of the mouse over, use the tooltip.

• Example

```
@UI: {  
    lineItem: [ { position: 10, label: 'Airline' } ],  
    identification:[ { position: 10, label: 'Airline' } ] }  
    @EndUserText.quickInfo: 'Airline that operates the flight.'  
key Connection.carrier_id      as AirlineID,
```

The screenshot shows a table titled "Connections (20) Standard". The columns are: Airline, Connection Number, Departure Airport Code, Destination Airport Code, Departure Time, and Arrival Time. A row for AA with Connection Number 15 is selected. The "Connection Number" cell contains the value "15". A tooltip "Airline that operates the flight." is displayed above the cell. The table has a header row with column names and a footer row with times.

Mouse over displaying tooltip of CDS annotation

Related Information

[Adding Field Labels and Descriptions \[page 513\]](#)

5.1.4 Displaying Text for Unreadable Elements

Use text associations to add readable texts to short forms or identifiers.

Data that is stored in databases is usually kept as short as possible and consequently, many words are shortened or abbreviated with a character code. While this is convenient for storage reasons, it becomes a problem on the UI as the elements might then not be understandable anymore. The following figure displays a UI screen with airline codes that are not commonly known and which makes it difficult (or impossible) to use the app.

In our use case, the airline is abbreviated with the two letter airline code.

Connections (14)						
Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time	
SQ	15	SFO	SIN	4:00:00 PM	2:45:00 AM	>
UA	941	FRA	SFO	2:30:00 PM	5:06:00 PM	>

The full name of the airline is given in the CDS view `/DMO/I_Carrier_R` based on the database `/DMO/carrier`. Using a text association, you can establish a link from the connection CDS view to the carrier CDS view and adopt the names of the airlines as an addition to the airline codes in the UI. This requires the use of annotations from the domains `@semantics` and `@ObjectModel` to mark the names as readable text and to assign the text addition to the airline code elements.

For more detailed information about text elements, refer to [Defining Text Elements \[page 422\]](#).

Getting Airline Names through a Text Association

Prerequisites

A text provider view already exists. In our case, the CDS view `/DMO/I_Carrier` contains the relevant text element `Name`.

Context

The readable names of the airlines can be displayed in the UI together with the two letter code. The text is taken from the text provider view `/DMO/I_Carrier` that contains the airline code as well as the readable name of the airline. We want to display the text for the airlines on the list report page and on the object page. This is why we need to implement the text association for both views, `/DMO/I_Connection_R` and `/DMO/I_Flight_R`. Apart from an association with the text provider view, the text element in that view needs to be specified as text with an `@Semantics` annotation. In addition, the element with the airline code in the source view must be annotated to assign the text from the associated view to the element.

Procedure

1. Open the CDS view `/DMO/I_Carrier`. Use the annotation `@Semantics.text: true` on the element `Name` to identify the annotated element as a text and activate the view.

```
...
define view /DMO/I_Carrier
  as select from /dm0/carrier as Airline
{
  key Airline.carrier_id    as AirlineID,
```

```

    @Semantics.text: true
    Airline.name           as Name,
    @Semantics.currencyCode: true
    Airline.currency_code as CurrencyCode
}

```

Note

In general, you can annotate more than one view field as a text field. However, only the first annotated field is respected in the text consumer view for OData exposure.

2. Open the CDS view /DMO/I_Connection_R, for which you want to display the text. Implement an association to the CDS view /DMO/I_Carrier with the join condition on AirlineID. This association serves as a text association.
The process how to implement associations is described in [Implementing Associations for Existing CDS Views \[page 131\]](#).
3. Use the annotation `@ObjectModel.text.association: '<_AssocToTextProvider>'` on the element AirlineID and reference the association _Carrier as a text association. Then activate the CDS view.

```

...
define view /DMO/I_Connection_R
  as select from /dmo/connection as Connection
  association [1..*] to /DMO/I_Flight_R as _Flight on
  $projection.AirlineID = _Flight.AirlineID
                                         and
  $projection.ConnectionID = _Flight.ConnectionID
  association [1] to /DMO/I_Carrier as _Airline on $projection.AirlineID
= _Airline.AirlineID{
...
  @ObjectModel.text.association: '_Airline'
  key Connection.carrier_id as AirlineID,
...
/*Association*/
  _Airline
}

```

4. Open the CDS view /DMO/I_Flight_R for which you also want to display the text. Use the annotation `@ObjectModel.text.association: '<_AssocToTextProvider>'` on the element AirlineID and reference the association _Airline as a text association. Then activate the CDS view.

```

...
define view /DMO/I_Flight_R
  as select from /dmo/flight as Flight
  association [1] to /DMO/I_Carrier_R as _Airline on
  $projection.AirlineID = _Airline.AirlineID
{
...
  @ObjectModel.text.association: '_Airline'
  key Flight.carrier_id as AirlineID,
...
/*Association*/
  _Airline
}

```

5. Activate all changed CDS views.

Results

You have established a text association from the two CDS views that contain the two-letter airline code. The text is taken from the text provider view and is displayed in the UI together with the two-letter code, as can be seen in the following image. Now, the airlines are clearly identifiable by their full names.

Connections (14) Standard ▾						
Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time	
Alitalia (AZ)	789	TYO	FCO	11:45:00 AM	7:25:00 PM	>
Delta Airlines (DL)	106	JFK	FRA	7:35:00 PM	9:30:00 AM	>
Japan Airlines (JL)	407	NRT	FRA	1:30:00 PM	5:35:00 PM	>
Japan Airlines (JL)	408	FRA	NRT	8:25:00 PM	3:40:00 PM	>
Lufthansa (LH)	400	FRA	JFK	10:10:00 AM	11:34:00 AM	>

i Note

You can also establish a text association with the CDS view `/DMO/I_Airport` to provide a full text for the airport elements `Departure Airport Code` and `Destination Airport Code`.

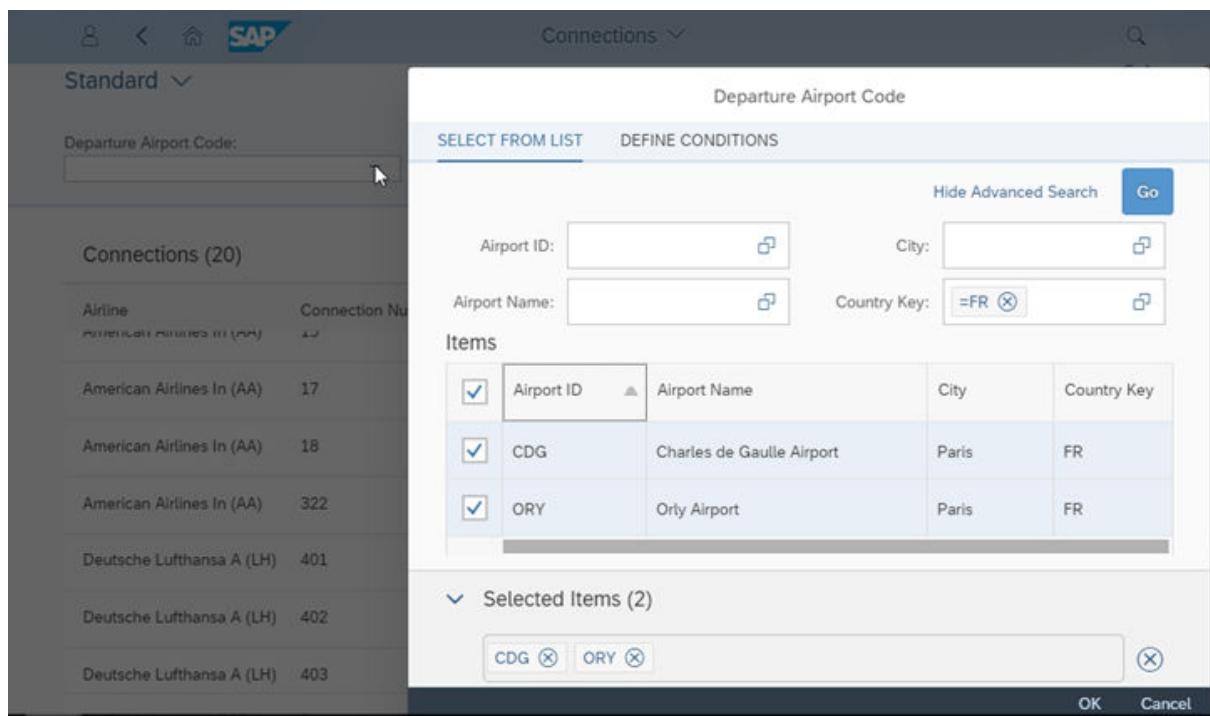
Related Information

[Defining Text Elements \[page 422\]](#)

5.1.5 Providing Value Help for the Selection Fields

Use value helps to make it easier to find the correct value for selection fields on the UI.

We created some selection fields for the UI to enable the end user to find a suitable flight in the flight app. The difficulty, however, is to know the correct three letter airport code to determine the correct departure or destination airport for a flight. You can enable a value help option to assist the end user in finding correct values. The end user then gets direct access to a value help dialog in which he or she can enter values, such as city names or country codes, to find the suitable airport code. This is shown in the following figure.



Adding Annotations for Value Help

Prerequisites

A value help provider view already exists. In our case, the CDS view /DMO/I_Airport contains the relevant fields `Name`, `City` and `CountryCode` to help find the airport code.

Context

To provide a value help for the selection fields of the elements `Departure Airport Code` and `Destination Airport Code` in the CDS view /DMO/I_Connection_R, use the annotation `@Consumption.valueHelpDefinition`.

Procedure

1. Open the CDS view /DMO/I_Connection_R.
2. Annotate the elements `Departure Airport Code` and `Destination Airport Code`. You want to equip these elements with a value help dialog with the annotation
`@Consumption.valueHelpDefinition: [{ entity: { name: '<target_view>' }, element:`

'<target_element>' } }] . In our case, the target view is /DMO/I_Airport with the target element AirportID , which works as the binding condition for both elements in the source view. .

```
{...  
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/  
I_Airport',  
        element: 'AirportID' } }]  
        airport_from_id as DepartureAirport,  
...  
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/  
I_Airport' ,  
        element:  
'AirportID' } }]  
        airport_to_id as DestinationAirport,  
...}
```

i Note

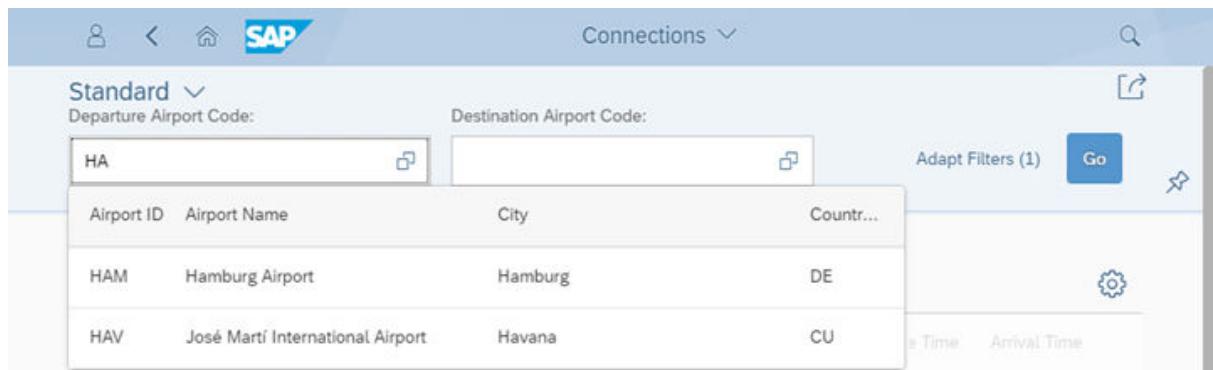
For the value help, you do not need to establish an association between the source and the target view. Nevertheless, you need to make sure that the target view is part of the service.

If not already done, add the value help provider view to the service definition.

3. Activate the CDS view.

Results

You have implemented a value help for the selection fields that refer to the airport code elements. Clicking  opens the value help dialog with the option to use all the elements of the value help provider view /DMO/I_Airport to find the correct three letter airport code (see the image above). Additionally, the selection fields are now equipped with a value completion option. This means that, once you start typing in the selection field, the possible values are displayed. This is shown in the following figure:



The screenshot shows a SAP Fiori application interface. At the top, there's a header bar with icons for user profile, back, home, SAP logo, and search. Below the header, there are two input fields: "Departure Airport Code:" and "Destination Airport Code:", each with a small blue square icon to its right. Underneath these fields is a table with three columns: "Airport ID", "Airport Name", and "City". The first row of the table has a dropdown arrow icon to its right. A dropdown menu is open over this icon, listing the options "HA", "HAM", and "HAV". To the right of the dropdown menu, there are buttons for "Adapt Filters (1)", "Go", and a gear icon. At the bottom of the table, there are additional columns: "Country" and "Time".

Value completion option

Related Information

[Consumption Annotations \[page 562\]](#)

[Providing Value Help \[page 428\]](#)

5.1.6 Adding Search Capabilities

Include a search input field to execute a text and fuzzy search on multiple elements.

Prerequisites

The CDS view must be suitable for text and fuzzy search enabling. For more information, take a look at the corresponding topics in the [SAP HANA Search Developer Guide](#).

Context

In the previous chapter, you implemented a value help for the selection fields, which are based on airport code fields. This means you can easily search for connections from and to certain airports. However, it is not possible to search for connections by specific airlines or flights that are operated on a specific day. Maybe the end user even wants to search for values of different elements, say someone wants to find out which connections are available to or from Frankfurt on, say, February 18, 2019. This is exactly the use case when search capabilities are required.

You use annotations to enable search capabilities for a specific CDS view and also to mark the elements that you want to be included in the search scope. You can also define fuzziness thresholds for the search, so that entries are also found if the values are entered in the search field with incorrect spelling. This makes the search easier and more effective.

The following section explains the process used to implement search capabilities when the end user wants to search for the following:

- Connections operated by one or more certain airlines
- Connections on one or more certain days
- Connections with one or more plane types
- Or a combination of any of these.

Whereas the first two searches operate only on the CDS view `/DMO/I_Connection_R` (because the search target elements `Airline`, `Departure Airport Code` and `Destination Airport Code` are part of this view), the searches for days and plane types require the search to be operated on the associated view, since the elements `Flight Date` and `Plane Type` are part of `/DMO/I_Flight_R`. In addition, we also want to be able to search for the full airline name. That is why the text provider view `/DMO/I_Carrier` must also be search enabled.

Procedure

1. Enable the relevant elements for searches in `/DMO/I_Connection_R`
 - a. Open the CDS view `/DMO/I_Connection_R`.
 - b. Use the annotation `@Search.searchable: true` on the entity level to enable the CDS view for searches and to expose a standard search field on the UI.

i Note

If you use this annotation in a CDS view, you have to assign at least one default search element.

```
@AbapCatalog.sqlViewName: '/DMO/ICONNECT_R'  
@AbapCatalog.compiler.compareFilter: true  
@AccessControl.authorizationCheck: #NOT_REQUIRED  
@EndUserText.label: 'Read-Only E2E: Data Model Connection'  
@UI.headerInfo: { typeName: 'Connection',  
    typeNamePlural: 'Connections' }  
@Search.searchable: true      /** exposes a standard search field on  
the UI  
define view /DMO/I_Connection_R  
    as select from /dmo/connection as Connection  
...
```

- c. Choose the elements that you want to search for and annotate them with
`@Search.defaultSearchElement: true.`
- d. Define a fuzziness threshold for the searchable elements with
`@Search.fuzzinessThreshold: <fuzziness_value>.`

i Note

You can set a fuzziness threshold of 0 to 1. SAP recommends that you use 0.7 to begin with. This means that every item with a 70% match and greater is found. You can then customize the threshold.

The search elements in the connection view are Airline, Departure Airport Code, and Destination Airport Code. As the first one only consists of a two letter code, you do not need to define a fuzziness threshold for this element.

```
{  
...  
    @Search.defaultSearchElement: true  
    key Connection.carrier_id      as AirlineID,  
...  
    @Search.defaultSearchElement: true  
    @Search.fuzzinessThreshold: 0.7  
    Connection.airport_from_id   as DepartureAirport,  
...  
    @Search.defaultSearchElement: true  
    @Search.fuzzinessThreshold: 0.7  
    Connection.airport_to_id     as DestinationAirport,  
... }
```

- e. Annotate the association _Flight with `@Search.defaultSearchElement: true` to enable the search for elements in the associated view.

```
{...  
    @Search.defaultSearchElement: true  
    _Flight,  
... }
```

- f. Activate the CDS view.

Now the end user can search for connections by specific airlines or to or from a specific airport. The following figure illustrates the connections operated by SQ (Singapore Airlines Limited).

Connections (2)		Standard	More		
Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time
Singapore Airlines Limited (SQ)	1	SFO	SIN	1:15:00 AM	11:50:00 AM >
Singapore Airlines Limited (SQ)	2	SIN	SFO	6:30:00 AM	9:15:00 AM >

2. For our use case, we not only want to search for elements in the main view of the app, but also for the fields of the associated view. Enable the relevant elements for search in /DMO/I_Flight_R. As previously described, open the CDS view /DMO/I_Flight_R, mark it as **search enabled**, and then choose the elements to search on with a proper fuzziness threshold.

```
@AbapCatalog.sqlViewName: '/DMO/IFLIGHT_R'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2': Data Model Flight
@Search.searchable: true
define view /DMO/I_Flight_R
  as select from /dmo/flight as Flight
{...
  @Search.defaultSearchElement: true
  @Search.fuzzinessThreshold: 0.7
  key Flight.flight_date as FlightDate,
    @Search.defaultSearchElement: true
    @Search.fuzzinessThreshold: 0.7
    Flight.plane_type_id as PlaneType,
... }
```

The association is now search enabled. This means it is possible to search for connections with a specific plane type and at a specific airport at the same time, even though the elements are not part of the same view. The following figure displays the search results for the search for the machine with the code A380 and the airport with the code FRA.

Connections (2)		Standard	More		
Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time
Japan Airlines Co., Ltd. (JL)	407	NRT	FRA	1:30:00 PM	5:35:00 PM >
Deutsche Lufthansa AG (LH)	401	JFK	FRA	6:30:00 PM	7:45:00 AM >

3. Enable the relevant elements for searches in /DMO/I_Carrier to be able to search for the full airline names and not only for the two letter code. As previously described, open the CDS view /DMO/I_Carrier, mark it as **search enabled**, and then choose the elements to search on with a proper fuzziness threshold.

```
@AbapCatalog.sqlViewName: '/DMO/ICARRIER_R'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Read-Only E2E: Data Model Carrier'
@Search.searchable: true
define view /DMO/I_Carrier_R
  as select from /dmo/carrier as Airline
{...
```

```

    @Search.defaultSearchElement: true
    @Search.fuzzinessThreshold: 0.7
    Airline.name           as Name,
...
}

```

The text association now enables the end user to search for the full names of airlines, even though the full name element `Name` is not exposed on the UI. Even if the full name is not complete in the search field, the search still presents the right results, as can be seen in the following figure.

The screenshot shows a search interface with two input fields: 'Departure Airport Code:' and 'Destination Airport Code:', both containing the placeholder 'United'. Below the inputs is a search button with a magnifying glass icon. To the right are buttons for 'Adapt Filters (1)' and 'Go'. A table below displays search results for 'Connections (3)'. The columns are 'Airline', 'Connection Number', 'Departure Airport Code', 'Destination Airport Code', 'Departure Time', and 'Arrival Time'. The results show three flights from SFO to FRA at different times.

Airline	Connection Number	Departure Airport Code	Destination Airport Code	Departure Time	Arrival Time
United Airlines, Inc. (UA)	58	SFO	FRA	1:45:00 PM	9:55:00 AM >
United Airlines, Inc. (UA)	59	FRA	SFO	1:55:00 PM	4:30:00 PM >
United Airlines, Inc. (UA)	941	FRA	SFO	2:30:00 PM	5:06:00 PM >

Related Information

[Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#)

5.2 Developing Managed Transactional Apps

In this chapter, you will be guided through all steps necessary to develop a travel administration app based on the business object's managed runtime infrastructure.

Introduction

Let's assume you want to develop completely new transactional apps for SAP Fiori UI or for an arbitrary Web API consumer in SAP Cloud Platform or ABAP environment. Let's further assume that you don't have any transactional buffer or business logic implementation or authorization functionality available to you.

In that situation you can benefit from the `managed` implementation type of the ABAP RESTful programming model.

Unlike the unmanaged scenario which aims for reusing the persistence layer and integrating an already existing business logic, the managed scenario addresses use cases where all essential parts of an application must be developed from scratch. However, these new applications can highly benefit from **out-of-the-box support for transactional processing**. Whereas for the unmanaged implementation type, the application developer must implement essential components of the REST contract manually, for the managed scenario, all required standard operations (create, update, delete) must only be specified in the behavior definition to obtain a ready-to-run business object. The technical implementation aspects are taken over by the business object runtime.

infrastructure. In this case, the business object framework implements the interaction phase and the save sequence generically. The application developer can then **focus on business logic** that is implemented using actions, validations and determinations and user interaction. The corresponding BO runtime manages the entire life cycle of your business objects and covers all aspects of your business application development.

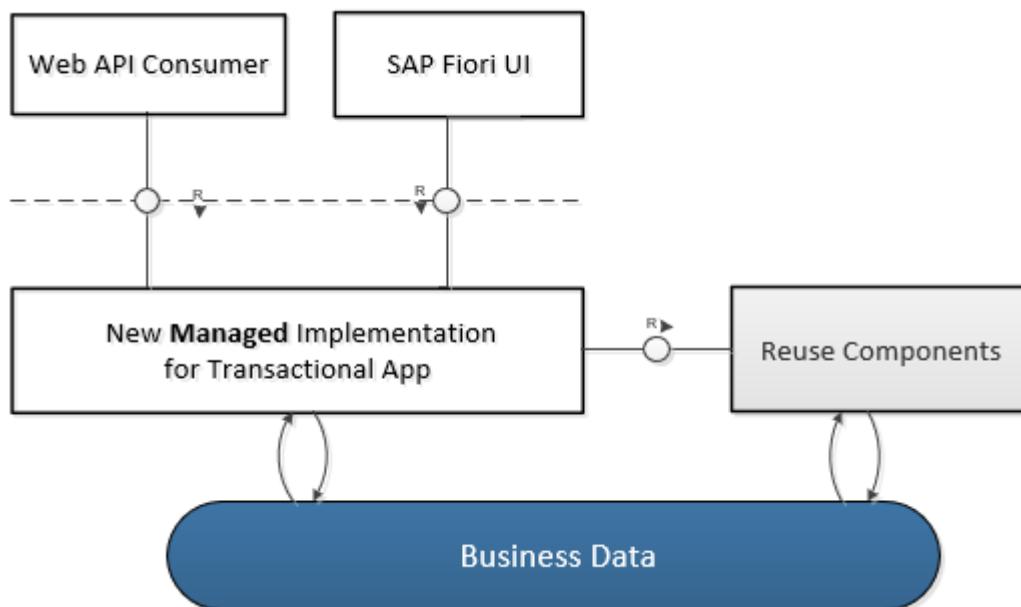
Architecture Overview

The architecture for a managed scenario can be depicted as in the simplified figure below.

The new implementation of business objects and their functionality is a key component in the architecture. In addition to the generic implementation of transactional handling, these objects also provide the application-specific business logic. Even with new developments, you will certainly always use already available reuse functionality whenever possible. For example, our demo application uses value help views and text views for text information retrieval, currency conversion procedures, and message constants as reuse components.

i Note

If you are running ABAP developments on *SAP Cloud Platform*, then you have the option to introduce reuse components after the custom code migration into ABAP Environment.



Architecture Overview - Managed Implementation

Prerequisites

Developing the scenario that is described in the subsequent chapters requires the following:

- You have access to and an account for **SAP Cloud Platform, ABAP environment**.
- You have installed ABAP Development Tools (ADT).

SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.

- To recreate the demo scenario, the [ABAP Flight Reference Scenario](#) must be available in your ABAP system. You can download the complete reference scenario from [GitHub: Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

→ Remember

The namespace `/DMO/` is reserved for the demo content. Apart from the downloaded ABAP Flight Scenario, do not use the namespace `/DMO/` and do not create any development objects in the downloaded packages.

You can access the development objects in `/DMO/` from your own namespace.

However, if you want to recreate all development objects of this demo content, make sure that you use different names from this documentation.

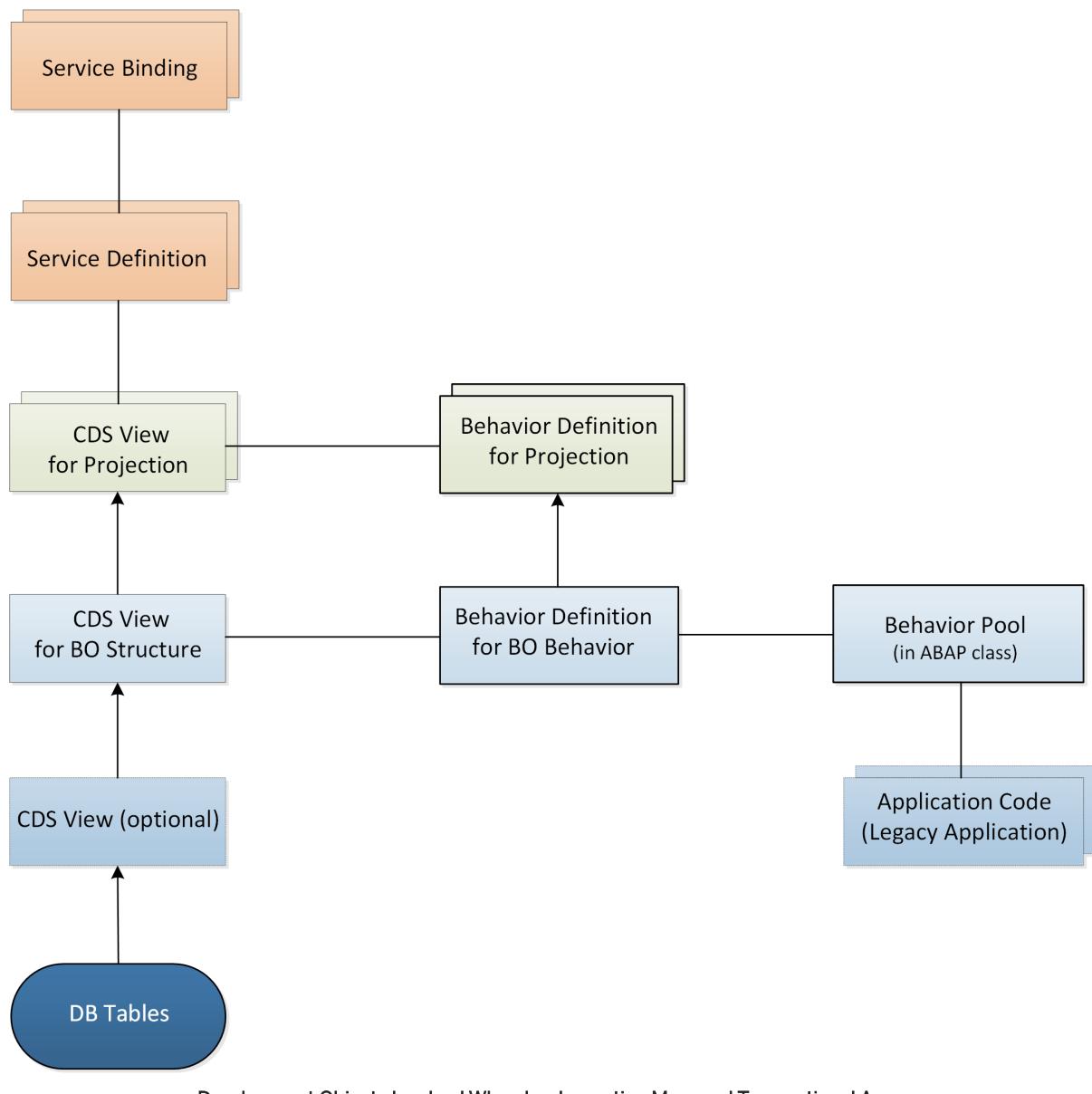
Constraints

When working with the managed implementation type the following constraints are given:

- **Creating Child Entity Instances:** Instances of child entities can only be created by a `create-by-association`.
- **CDS View Fields:** In CDS views, you have to use the same name for the fields as in the underlying database table. No aliasing is supported. You can alias the CDS elements in the CDS projection view.
- **Determinations:**
Changes caused by determinations cannot trigger other determinations.
Determinations that are triggered at entire entity level can only be defined for the `create` operation.
- **Validations:** Validations can only be triggered at field level.
- **Authority:** Only instance-based authorizations are available. That means, static authorizations are not available. Therefore, you cannot apply authorization checks to create operations.
- **Numbering:** Late numbering is not supported.
- **Actions:** Factory actions are not supported.
- **Functions:** Functions are not supported.

Involved Development Objects

The figure below provides an overview of the main development objects involved when creating new transactional apps based on the managed implementation type.



Development Process in Overview

The development of new managed applications mainly requires developers to perform the following fundamental activities:

1. Developing Ready-to-Run Business Objects

In this section, you will create, from scratch, all the components of your application required to run ready-to-run business objects. Using the business object framework for managed implementation type, you will save time during the development cycle because you don't have to implement all the technical details yourself - details such as low-level transaction handling, buffer management, or business logic orchestration. This framework provides a set of generic services and functionalities to speed up, standardize, and modularize your development.

More on this: [Developing a Ready-to-Run Business Object \[page 153\]](#)

2. Developing Business Logic

Using model-driven development approach, you may focus your attention more on the actual business requirements themselves by developing actions, validations, determinations and providing feature control for each entity of the business object structure.

More on this: [Developing Business Logic \[page 175\]](#)

3. Providing Projection Layer for Service Consumption

To enable a flexible consumption of the resulting business service, a [projection \[page 817\]](#) is introduced as a separate layer within the application development. This layer is required for service projections for different consumption scenarios such as Web APIs and UI-related services according to Fiori UI role-based design.

This essentially includes three steps: providing a data model with CDS views for projections, modelling behavior definitions for projections as well as defining business services that expose projections for consumption.

More on this: [Developing a Projection Layer for Flexible Service Consumption \[page 237\]](#)

4. Testing the OData UI Service

Using ABAP Development Tools, you have the option of publishing the service to the service repository of your local system. As soon as the service is published, it is ready for consumption through an OData client, such as an SAP Fiori app. The service binding editor also offers a preview tool that you can use for testing the resulting app within your ABAP development environment.

More on this: [Testing the Business Object \[page 173\]](#)

5.2.1 Reference Business Scenario

In this demo scenario, we will implement a simple travel provider app that can be used to manage flight bookings. A single travel should be booked by the travel provider for an existing customer through a travel agency and include one or multiple flight bookings. In addition, one or more supplements should be able to be booked for each flight.

Requirements

Based on the same data model, two different views of the travel app are to be realized, each corresponding to two different user roles:

- The **processor** acquires all the data relevant to flight bookings: he or she creates individual travel instances, creates and manages individual flights, and adds supplements to flight bookings. The accumulated travel costs should be calculated automatically when the underlying bookings are updated. When editing individual travel data, validation must be made for data consistency and, in the case of an error, be issued as an appropriate message to the user.

Adapt Filters Go

^ X

Travels (19) Standard ▾				
Travel ID	Agency ID	Customer ID	Starting Date	End Date
<input checked="" type="checkbox"/> 1	Sunshine Travel (70001)	Buchholm (7)	Apr 18, 2019	May 2, 2019 >
	Total Price: 837.00 EUR			
	Booking Status: O			
<input type="radio"/> 12	No Return (70020)	Benjamin (451)	Jun 13, 2019	Jun 27, 2019 >
	Total Price: 425.50 EUR			
	Booking Status: O			

List of Travels - UI for Processor Role

*Travel ID [1,...,99999999]:	<input type="text" value="79"/>	*Booking Fee:	<input type="text" value="17.00"/> EUR □
*Agency ID:	<input type="text" value="70049"/> □	Total Price:	<input type="text" value="EUR"/> □
*Customer ID:	<input type="text" value="51"/> □	*Overall Status [O (Open) C (Completed)]:	<input type="text" value="O"/>
*Starting Date:	<input type="text" value="Jun 19, 2019"/> □	Comment:	<input type="text" value="Business trip to WDF"/>
*End Date:	<input type="text" value="Jul 6, 2019"/> □		

Save Cancel

Creating a Travel - UI for Processor Role

- The role of the **approver** is limited to the verification of the recorded travel data entered by the processor. The approver has the option to accept or reject individual travels.

Adapt Filters (1) Go

^ X

Travels (10) Standard ▾				
Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee
<input checked="" type="checkbox"/> 1	O	Sunshine Travel (70001)	Buchholm (7)	13.00 EUR >
	Total Price: 837.00 EUR			
	Description: Vacation AB			
<input type="radio"/> 12	O	No Return (70020)	Benjamin (451)	12.00 EUR >
	Total Price: 425.50 EUR			
	Description: AB			

List of Travels - UI for Approver Role

Data Model

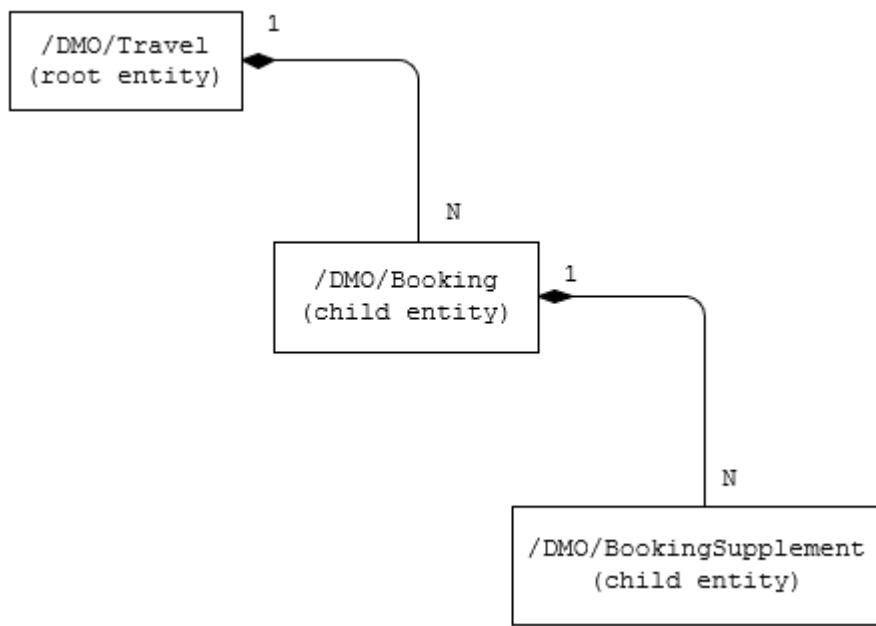
Entities for Transactional Data

As depicted in the figure below, the 3-tier entity hierarchy for managing transactional data in our scenario consists of the following editable entities, with a 1: N cardinality on each level:

- Travel
- Booking
- BookingSupplement

That is, each travel instance has 0..N bookings and each booking has 0..N booking supplements.

The figure below shows the composition relationship between the travel, the (flight) booking and the supplement entities, where the travel entity represents the root of the data model.



Editable Entities of the Data Model

Additional Entities for Master Data (Reuse Components)

To access business data from other entities in our demo content, we are going to reuse master data from the package `/DMO/FLIGHT_REUSE` that is part of the downloaded demo content `/DMO/FLIGHT`. Some of these entities will be used primarily as text provider views for retrieving text information and as value help provider views for individual input fields on the UI.

✓	/DMO/FLIGHT_REUSE (16) <i>Flight Reference Scenario: Reused Entities</i>
✓	Core Data Services (8)
✓	Data Definitions (8) <ul style="list-style-type: none"> > /DMO/I_AGENCY <i>Travel View - CDS Data Model</i> > /DMO/I_AIRPORT <i>Airport View - CDS Data Model</i> > /DMO/I_CARRIER <i>Carrier View - CDS Data Model</i> > /DMO/I_CONNECTION <i>Connection View - CDS Data Model</i> > /DMO/I_CUSTOMER <i>Customer View - CDS Data Model</i> > /DMO/I_FLIGHT <i>Flight View - CDS Data Model</i> > /DMO/I_SUPPLEMENT <i>Supplement view - CDS data model</i> > /DMO/I_SUPPLEMENTTEXT <i>Text provider view - CDS data mode</i>
>	Dictionary (8)

Non-Editable Entities Reused in the Data Model

Behavior

In the following table, we briefly outline the scope of the business logic to be implemented in the demo scenario. Bear in mind that the managed implementation type comes into play with a few new concepts such as validations and determinations, which will be introduced later on over the course of this E2E guide.

Processor Role:

Processor

Entity	Behavior
Travel	Operations: <ul style="list-style-type: none"> • Create, update, delete • Create bookings by association • Action: create travels by template Validations: <ul style="list-style-type: none"> • Validate editable input fields Feature Control: <ul style="list-style-type: none"> • Static and dynamic field control • Dynamic operation control • Dynamic action control

Entity	Behavior
Booking	<p>Operations:</p> <ul style="list-style-type: none"> • Update • Create booking supplements by association <p>Validations:</p> <ul style="list-style-type: none"> • Validate editable input fields <p>Determinations:</p> <ul style="list-style-type: none"> • Calculate total price for bookings <p>Feature Control:</p> <ul style="list-style-type: none"> • Static and dynamic field control • Dynamic operation control
Booking Supplement	<p>Operations:</p> <ul style="list-style-type: none"> • Update <p>Determinations:</p> <ul style="list-style-type: none"> • Calculate total price for supplements <p>Feature Control:</p> <ul style="list-style-type: none"> • Static and dynamic field control

Approver Role:

Approver

Entity	Behavior
Travel	<p>Operations:</p> <ul style="list-style-type: none"> • Update (on limited set of travel fields) • Action: accept travel • Action: reject travel <p>Validations:</p> <ul style="list-style-type: none"> • Validate editable fields <p>Feature Control:</p> <ul style="list-style-type: none"> • Static and dynamic field control • Dynamic action control
Booking	Read operations only

Restrictions

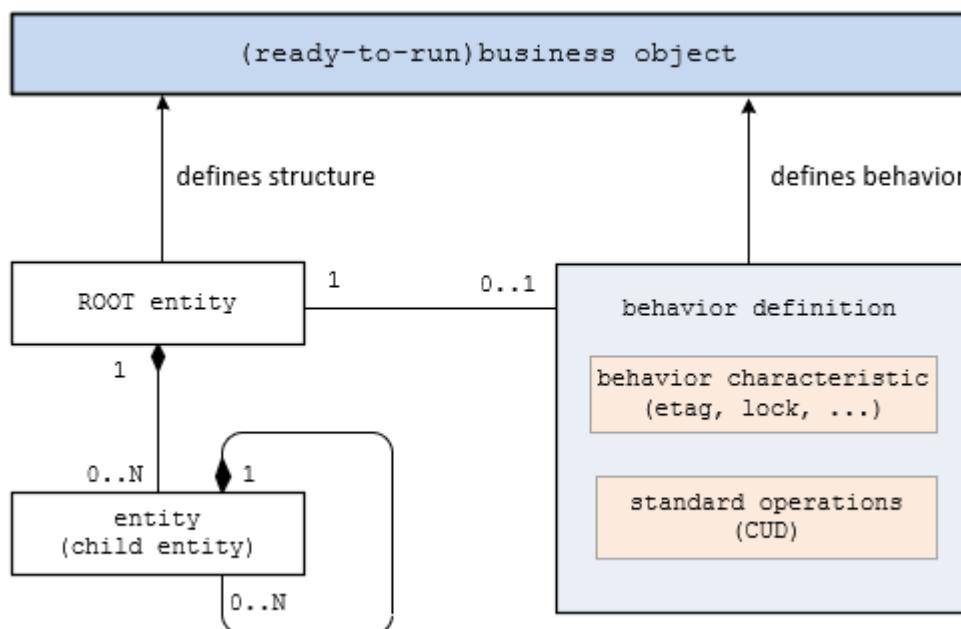
! Restriction

The current version of the ABAP RESTful programming model does not support [late numbering \[page 814\]](#) for managed implementation type. Therefore, in our demo scenario, the numbering is not implemented in the save sequence using the `adjust_numbering()` method when creating instances for travels, bookings, and booking supplements.

5.2.2 Developing a Ready-to-Run Business Object

Each business object is characterized by a structure, a behavior and a runtime implementation.

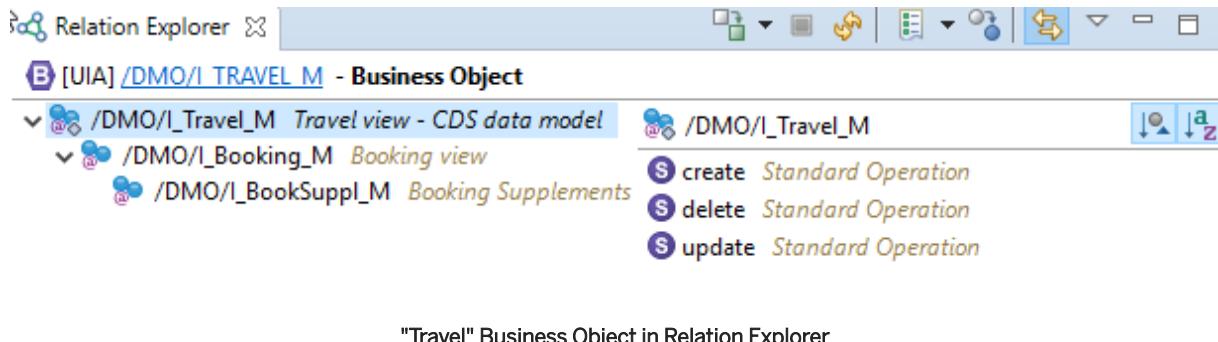
As shown in the figure below, the structure of a business object is defined by the sequence of compositions and to-parent associations between CDS entities with a root entity on top of the composition tree. Each level of a business object's composition tree can offer a set of operations that are specified in the behavior definition that refers to the entities of the CDS data model. In the case of managed implementation type, the standard operations (create, update, delete) must only be specified in the behavior definition to obtain a ready-to-run business object. The business object runtime infrastructure already implements the interaction phase and the save sequence generically and provides an out-of-the-box support for transactional processing.



Data Model and Behavior of a Ready-to-run Business Object

Preview: Resulting Business Object in Relation Explorer

The figure below displays the resulting “Travel” business object in *Relation Explorer* view. If you choose the *Business Object* context, the *Relation Explorer* provides the composition tree of the business object and displays all operations of the selected entity.



Activities Relevant to Developers

1. [Providing Business Object Structure \[page 154\]](#)
 1. [Providing Persistent Tables \[page 156\]](#)
 2. [Creating CDS Data Definitions \[page 160\]](#)
 3. [Defining Data Model and Business Object Structure \[page 161\]](#)
2. [Defining Elementary Behavior for Ready-to-Run Business Object \[page 165\]](#)
3. [Testing the Business Object \[page 173\]](#)

Related Information

[Exploring Business Objects \[page 759\]](#)

5.2.2.1 Providing Business Object Structure

From a structural point of view, a business object consists of a tree of entities that are linked by compositions. Every entity in this composition tree is an element that is modeled with a CDS entity.

For our demo travel booking scenario, we will implement a 3-level hierarchy composition tree.

Entities for Transactional Data

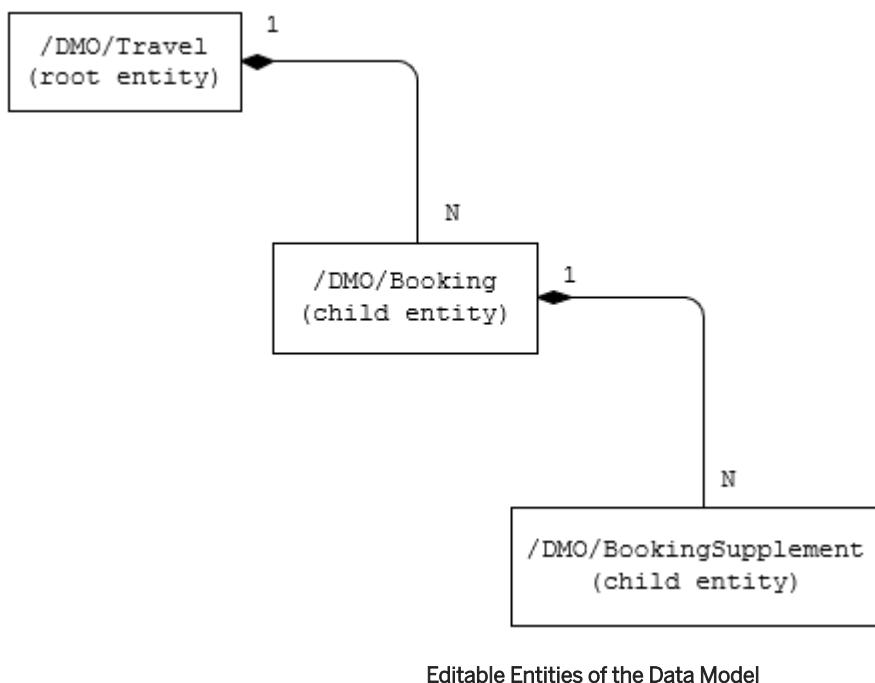
As depicted in the figure below, the 3-tier entity hierarchy for managing transactional data in our scenario consists of the following editable entities, with a 1: N cardinality on each level:

- Travel
- Booking

- BookingSupplement

That is, each travel instance has 0..N bookings and each booking has 0..N booking supplements.

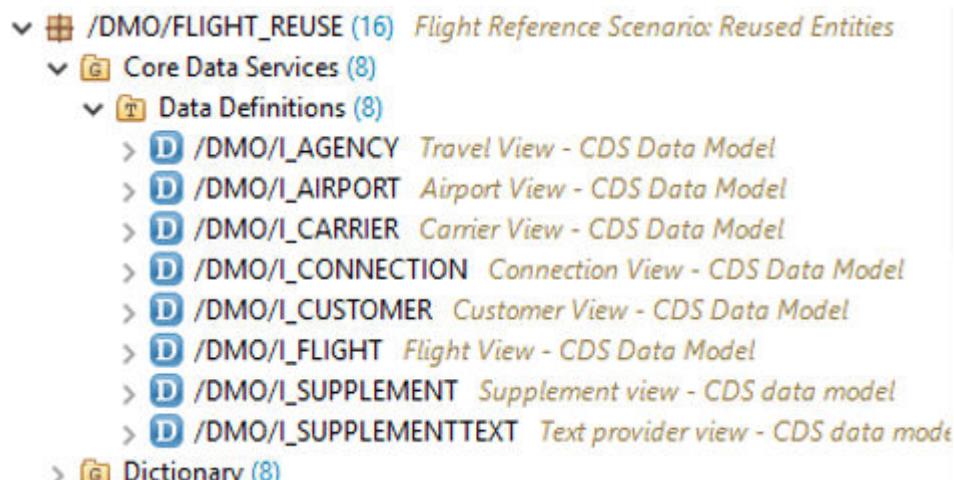
The figure below shows the composition relationship between the travel, the (flight) booking and the supplement entities, where the travel entity represents the root of the data model.



Editable Entities of the Data Model

Additional Entities for Master Data (Reuse Components)

To access business data from other entities in our demo content, we are going to reuse master data from the package /DMO/FLIGHT_REUSE that is part of the downloaded demo content /DMO/FLIGHT. Some of these entities will be used primarily as text provider views for retrieving text information and as value help provider views for individual input fields on the UI.



Non-Editable Entities Reused in the Data Model

i Note

Additional entities for currencies (`I_Currency`) and countries (`I_Country`) are generally available in your system and are included in our data model using associations.

Related Information

[Providing Persistent Tables \[page 156\]](#)

[Creating CDS Data Definitions \[page 160\]](#)

[Defining Data Model and Business Object Structure \[page 161\]](#)

5.2.2.1.1 Providing Persistent Tables

Travel data of our demo application is distributed across multiple database tables: a table for the travel header data, a table for flight bookings, and booking supplements.

To build up our demo scenario from scratch, we will first create a suitable set of database tables using ADT ABAP Dictionary tools and then use these tables as a data source in the new CDS-based data model.

The standard use case is to work with client-dependent tables. That means, the database table has a key field in which the client is maintained. Like this, you can control the access to the database entries based on the client. However, there are use cases in which the instances of the database tables are not client-specific. These tables do not contain a client field. The RAP managed BO runtime also supports scenarios with client-independent database tables. For more information, see [Using Client-Independent Database Tables in Managed Transactional Apps \[page 159\]](#).

Starting Point and Prerequisites

To provide data persistence for our sample travel management scenario, we assume that you...

- Have the standard developer authorization profile to create ABAP development objects with ABAP Development Tools.
- Reuse the data elements from the ABAP Flight Reference Scenario (package: `/DMO/FLIGHT_LEGACY`) when editing table fields for tables to be created.

Procedure: Creating Tables

To create and work with database tables in ABAP Development Tools (ADT), do the following:

1. Open the source-based ABAP Dictionary editor in ADT and create the corresponding database tables listed below. For further information, see: .

2. Edit the database tables with the fields and metadata as depicted in the listings below.
3. Activate the tables

Table /DMO/TRAVEL_M: Persistent Table for Managing Travel Data

This table defines general travel data, such as the key element travel ID, agency ID or customer ID, the date for travel's begin and end, as well as the overall status of the travel bookings, and the total price of an individual travel. In addition, the fields for standard administration data, such as the respective user or the time of creation, are added to the table.

Listing 1: Source Code

```

@EndUserText.label : 'Flight Reference Scenario: Managing Travels'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #LIMITED
define table /dmo/travel_m {
  key client      : abap.clnt not null;
  key travel_id   : /dmo/travel_id not null;
  agency_id       : /dmo/agency_id;
  customer_id    : /dmo/customer_id;
  begin_date     : /dmo/begin_date;
  end_date       : /dmo/end_date;
  @Semantics.amount.currencyCode : '/dmo/travel_data.currency_code'
  booking_fee    : /dmo/booking_fee;
  @Semantics.amount.currencyCode : '/dmo/travel_data.currency_code'
  total_price    : /dmo/total_price;
  currency_code  : /dmo/currency_code;
  description    : /dmo/description;
  overall_status : /dmo/overall_status;
  @AbapCatalog.anonymizedWhenDelivered : true
  created_by     : syuname;
  created_at     : timestamp;
  @AbapCatalog.anonymizedWhenDelivered : true
  last_changed_by : syuname;
  last_changed_at : timestamp;
}

```

Table /DMO/BOOKING_M: Persistent Table for Managing Bookings

This table will be used for managing flight booking data, such the flight connection, the carrier, or the price and flight date, and finally, the status of flight bookings.

Listing 2: Source Code

```

@EndUserText.label : 'Flight Reference Scenario: Booking'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #LIMITED
define table /dmo/booking_m {
  key client      : abap.clnt not null;
  @AbapCatalog.foreignKey.label : 'Travel'
}

```

```

@AbapCatalog.foreignKey.screenCheck : false
key travel_id : /dmo/travel_id not null
  with foreign key [0..*,1] /dmo/travel_m
    where travel_id = /dmo/booking_m.travel_id;
key booking_id : /dmo/booking_id not null;
booking_date : /dmo/booking_date;
customer_id : /dmo/customer_id;
carrier_id : /dmo/carrier_id;
connection_id : /dmo/connection_id;
flight_date : /dmo/flight_date;
@Semantics.amount.currencyCode : '/dmo/booking_data.currency_code'
flight_price : /dmo/flight_price;
currency_code : /dmo/currency_code;
booking_status : /dmo/booking_status;
last_changed_at : timestampl;
}

```

Table /DMO/BOOKSUPPL_M: Persistent Table for Managing Booking Supplement Data

This table is used to add additional products to a travel booking. For example, the customer can book together with a flight, a drink or a meal.

Listing 3: Source Code

```

@EndUserText.label : 'Flight Reference Scenario: Booking Supplement'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #LIMITED
define table /dmo/booksuppl_m {
  key client : abap.clnt not null;
  @AbapCatalog.foreignKey.label : 'Travel'
  @AbapCatalog.foreignKey.screenCheck : false
  key travel_id : /dmo/travel_id not null
    with foreign key [0..*,1] /dmo/travel_m
      where travel_id = /dmo/booksuppl_m.travel_id;
  @AbapCatalog.foreignKey.label : 'Booking'
  @AbapCatalog.foreignKey.screenCheck : false
  key booking_id : /dmo/booking_id not null
    with foreign key [0..*,1] /dmo/booking_m
      where travel_id = /dmo/booksuppl_m.travel_id
        and booking_id = /dmo/booksuppl_m.booking_id;
  key booking_supplement_id : /dmo/booking_supplement_id not null;
  supplement_id : /dmo/supplement_id;
  @Semantics.amount.currencyCode : '/dmo/book_suppl_data.currency_code'
  price : /dmo/supplement_price;
  currency_code : /dmo/currency_code;
  last_changed_at : timestampl;
}

```

Related Information

Database Tables

5.2.2.1.1.1 Using Client-Independent Database Tables in Managed Transactional Apps

The RAP managed BO runtime supports transactional scenarios with client-independent database tables. This topic provides information about things you need to consider when using client-independent tables.

Creating Client-Independent Database Tables

The procedure for creating client-independent database tables does not differ from the process described in [Procedure: Creating Tables \[page 156\]](#). You can simply remove the client field in the template for database tables.

The following codeblock shows the travel database table without a client field.

```
@EndUserText.label : 'Flight Reference Scenario: Managing Travels'  
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE  
@AbapCatalog.tableCategory : #TRANSPARENT  
@AbapCatalog.deliveryClass : #A  
@AbapCatalog.dataMaintenance : #LIMITED  
define table /dmo/travel_m {  
    key travel_id      : /dmo/travel_id not null;  
    agency_id         : /dmo/agency_id;  
    customer_id      : /dmo/customer_id;  
    begin_date       : /dmo/begin_date;  
    end_date         : /dmo/end_date;  
    @Semantics.amount.currencyCode : '/dmo/travel_data.currency_code'  
    booking_fee      : /dmo/booking_fee;  
    @Semantics.amount.currencyCode : '/dmo/travel_data.currency_code'  
    total_price      : /dmo/total_price;  
    currency_code    : /dmo/currency_code;  
    description      : /dmo/description;  
    overall_status   : /dmo/overall_status;  
    @AbapCatalog.anonymizedWhenDelivered : true  
    created_by       : syuname;  
    created_at       : timestamppl;  
    @AbapCatalog.anonymizedWhenDelivered : true  
    last_changed_by : syuname;  
    last_changed_at : timestamppl;  
}
```

Modeling Business Objects with Client-Independent Tables

The business object must be consistent with regard to the client fields in the database tables. This means that CDS views that are used for modeling the data model in scenarios with client-independent tables must not contain a client element. In particular, the generated SQL-view of the CDS view must not contain such an element.

There must be client-consistency among the entities in the business object composition structure. It is not possible to have one entity that is client-dependent and one that is client-independent with a compositional relationship between them.

Business Object Runtime with Client-Independent Tables

The client field in database tables influences the runtime of business objects during lock. While locks are set on entity instances only for one specific client in client-dependent tables, locks in client-independent tables lock instances for all clients.

i Note

In scenarios in which you have client-dependent database tables, but join client-independent fields from other database tables to your CDS view, the managed BO runtime locks the instances specific to the client. This means only the fields from the client-dependent database tables are locked. If you also want to lock the client-independent fields, you have to implement an [unmanaged lock \[page 821\]](#).

i Note

In scenarios, in which you use an [unmanaged save \[page 822\]](#) in the managed scenarios, the managed BO runtime always sets a client-specific lock. If you want to lock client-independently, you have to use an [unmanaged lock \[page 821\]](#).

Related Information

[Lock Definition \[page 94\]](#)

[Integrating Unmanaged Save in Managed Business Objects \[page 227\]](#)

5.2.2.1.2 Creating CDS Data Definitions

In this step you create CDS views as the basis for the data model of our demo scenario. For each data definition a corresponding development object, the related CDS views and the corresponding database views are created too.

Data Definitions and CDS Views to Create

i Note

Naming CDS views: Since CDS views are (public) interface views, they are prefixed with `I_` in accordance with the VDM (virtual data model) naming convention. In addition, we add the suffix `_M` to the view name in case it is specific for our managed implementation type scenario. For detailed information, see: [Naming Conventions for Development Objects \[page 780\]](#)

Data Definitions Required for all Editable Entities of the Data Model

Data Definition

CDS View Name

Database View Name	Data Source	Description
/DMO/I_TRAVEL_M	/ DMO/ TRAVEL_M (DB table)	A Travel entity defines general travel data, such as the agency ID or customer ID, overall status of the travel bookings, and the total price of a travel.
/DMO/I_Travel_M(root entity)		
/DMO/I_TRAVEL_M		
/DMO/I_BOOKING_M	/ DMO/ BOOKING_M (DB table)	The booking entity is used for managing flight booking data, such as the customer, the flight connection, or the price and flight date.
/DMO/I_Booking_M(child entity)		
/DMO/I_BOOKING_M		
/DMO/I_BOOKSUPPL_M	/ DMO/ BOOKSUPPL_M (DB table)	This entity is used to add additional products to a travel booking.
/DMO/I_BookSuppl_M(child entity)		
/DMO/I_BOOKSUPPL_M		

Procedure: Creating a Data Definition

To launch the wizard tool for creating a data definition, do the following:

1. Launch *ABAP Development Tools*.
2. In your ABAP project (or ABAP cloud project), select the relevant package node in *Project Explorer*.
3. Open the context menu and choose ► *New* ► *Other ABAP Repository Object* ► *Core Data Services* ► *Data Definition* ▶

Further information: (Tool Reference)

5.2.2.1.3 Defining Data Model and Business Object Structure

Travel Root View /DMO/I_Travel_M

The listing 1 (below) provides you with the implementation of the CDS data model for managing travel instances, where the database table /dmo/travel_m serves as the data source for the corresponding CDS

view /DMO/I_Travel_M. This CDS view defines the root entity of the data model and represents the root of compositional hierarchy for the travel business object to be created.

To define a [composition](#) [page 808] relationship from the root to a child entity, the keyword COMPOSITION is used. In our example, we specify the /DMO/I_Booking_M as child entity in the composition _Booking. As a result, the booking node is defined as a direct child entity to the business object's root. The cardinality [0 .. *] specifies that any number of booking instances can be assigned to each travel instance.

To be able to access master data from other entities, a set of associations is defined in the CDS source code. These associations refer to CDS entities (/DMO/I_Agency, /DMO/I_Customer) that are part of our demo application scenario. Some of these views are used primarily as text views for retrieving text information and as value help provider views for specific UI fields.

Finally, the fields for standard administration data are added to the select list, where the persistent field last_changed_at plays a special role as an [ETag](#) [page 812] field.

To ensure uniform data processing on the consumer side, all administrative as well as quantity and currency fields are provided with appropriate @Semantics annotations. These annotations are necessary to support managed ETag handling, that is the automatic update of the relevant ETag field on every operation.

i [Expand the following listing to view the source code.](#)

Listing 1: Source Code of the CDS Root View /DMO/I_Travel_M

```
@AbapCatalog.sqlViewName: '/DMO/ITRAVEL_M'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Travel view - CDS data model'
define root view /DMO/I_Travel_M
    as select from /dmo/travel_m           as Travel -- the travel table is the
    data source for this view
    composition [0..*] of /DMO/I_Booking_M as _Booking
        association [0..1] to /DMO/I_Agency   as _Agency   on
        $projection.agency_id = _Agency.AgencyID
        association [0..1] to /DMO/I_Customer  as _Customer on
        $projection.customer_id = _Customer.CustomerID
        association [0..1] to I_Currency       as _Currency on
        $projection.currency_code = _Currency.Currency

    {
        key travel_id,
        agency_id,
        customer_id,
        begin_date,
        end_date,
        @Semantics.amount.currencyCode: 'currency_code'
        booking_fee,
        @Semantics.amount.currencyCode: 'currency_code'
        total_price,
        @Semantics.currencyCode: true
        currency_code,
        overall_status,
        description,
        @Semantics.user.createdBy: true
        created_by,
        @Semantics.systemDateTime.createdAt: true
        created_at,
        @Semantics.user.lastChangedBy: true
        last_changed_by,
        @Semantics.systemDateTime.lastChangedAt: true
        last_changed_at,          -- used as etag field
    }
```

```

/* Associations */
Booking,
Agency,
Customer,
Currency
}

```

Booking View /DMO/I_Booking_M

Listing 2 (below) provides you with a data model implementation of the booking entity. All fields that are declared in the database table are used in the CDS view. The administrative field `last_changed_at` is used for optimistic concurrency control on all nodes. That means, every business object entity stores its own eTag.

In the data definition of the root entity `/DMO/I_Travel_M`, we specified the booking entity `/DMO/I_Booking_M` as a composition child entity. Reversely, this relationship requires an association to their compositional parent entity – from the child entity. This relationship is expressed by the keyword `ASSOCIATION TO PARENT`.

To provide a data model for 3-tier entity hierarchy, we also define a composition relationship from booking to a booking supplement entity. In our example, we specify `/DMO/I_BookSuppl_M` as child entity in the `composition_BookSupplement`. The cardinality `[0 .. *]` expresses that any number of booking supplement instances can be assigned to each booking instance.

To be able to access data from other entities, a set of additional associations (`_Customer`, `_Carrier`, and `_Connection`) is defined in the CDS source code.

The `SELECT` list includes all elements of a booking entity that are relevant for consumption in a user interface.

i [Expand the following listing to view the source code.](#)

Listing 2: Source Code of the CDS View /DMO/I_Booking_M

```

@AbapCatalog.sqlViewName: '/DMO/IBOOKING_M'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Booking view'
define view /DMO/I_Booking_M
  as select from /dm0/booking_m as Booking
    association to parent /DMO/I_Travel_M as _Travel      on
    $projection.travel_id = Travel.travel_id
    composition [0..*] of /DMO/I_BookSuppl_M      as _BookSupplement
    association [1..1] to /DMO/I_Customer        as _Customer      on
    $projection.customer_id = Customer.CustomerID
    association [1..1] to /DMO/I_Carrier         as _Carrier      on
    $projection.carrier_id = Carrier.AirlineID
    association [1..1] to /DMO/I_Connection      as _Connection    on
    $projection.connection_id = Connection.ConnectionID
    and
  $projection.connection_id = _Connection.ConnectionID
{
  key travel_id,
  key booking_id,
  booking_date,
  customer_id,
  carrier_id,
  connection_id,

```

```

    flight_date,
    @Semantics.amount.currencyCode: 'currency_code'
    flight_price,
    @Semantics.currencyCode: true
    currency_code,
    booking_status,

    @Semantics.systemDateTime.lastChangedAt: true
    last_changed_at, -- used as etag field
    /* Associations */
    _Travel,
    _BookSupplement,
    _Customer,
    _Carrier,
    _Connection
}

```

Booking Supplement View /DMO/I_BookSuppl_M

The 3-tier composition relationship requires an association for the booking supplement child entity /DMO/I_BookSuppl_M entity to their compositional parent entity.

Like the booking entity, the booking supplement entity also uses the field `last_changed_at` to store eTag values.

To access master data from other entities, additional associations `_Product` and `_SupplementText` are defined in the CDS source code.

i *Expand the following listing to view the source code.*

Listing 3: Source Code of the CDS View /DMO/I_BookSuppl_M

```

@AbapCatalog.sqlViewName: '/DMO/IBOOKSUP_M'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Booking Supplement View - CDS data model'
define view /DMO/I_BookSuppl_M
  as select from /dmo/booksuppl_m as BookingSupplement
    association      to parent /DMO/I_Booking_M as _Booking      on
    $projection.travel_id = _Booking.travel_id
                                         and
    $projection.booking_id = _Booking.booking_id
    association [1..1] to /DMO/I_Travel_M as _Travel      on
    $projection.travel_id = Travel.travel_id
    association [1..1] to /DMO/I_Supplement as _Product      on
    $projection.supplement_id = Product.SupplementID
    association [1..*] to /DMO/I_SupplementText as _SupplementText on
    $projection.supplement_id = _SupplementText.SupplementID
{
  key travel_id,
  key booking_id,
  key booking_supplement_id,
  supplement_id,
  @Semantics.amount.currencyCode: 'currency_code'
  price,
  @Semantics.currencyCode: true
  currency_code,

  @Semantics.systemDateTime.lastChangedAt: true
  last_changed_at, -- used as etag field
}

```

```

/* Associations */
_Travel,
_Booking,
_Product,
_SupplementText
}

```

5.2.2.2 Defining Elementary Behavior for Ready-to-Run Business Object

In this step, we will limit our focus to modeling an elementary behavior in which only the standard operations `create()`, `update()`, and `delete()` are defined for each entity. These operations, along with some basic properties (behavior characteristics), should already be sufficient to obtain a ready-to-run business object.

Procedure: Creating a Behavior Definition /DMO/I_TRAVEL_M

To launch the wizard tool for creating a behavior definition, do the following:

1. Launch *ABAP Development Tools*.
2. In the *Project Explorer* view of your ABAP project (or ABAP Cloud Project), select the node for the data definition that defines the root entity (`/DMO/I_TRAVEL_M`).
3. Open the context menu and choose *New Behavior Definition* to launch the creation wizard.

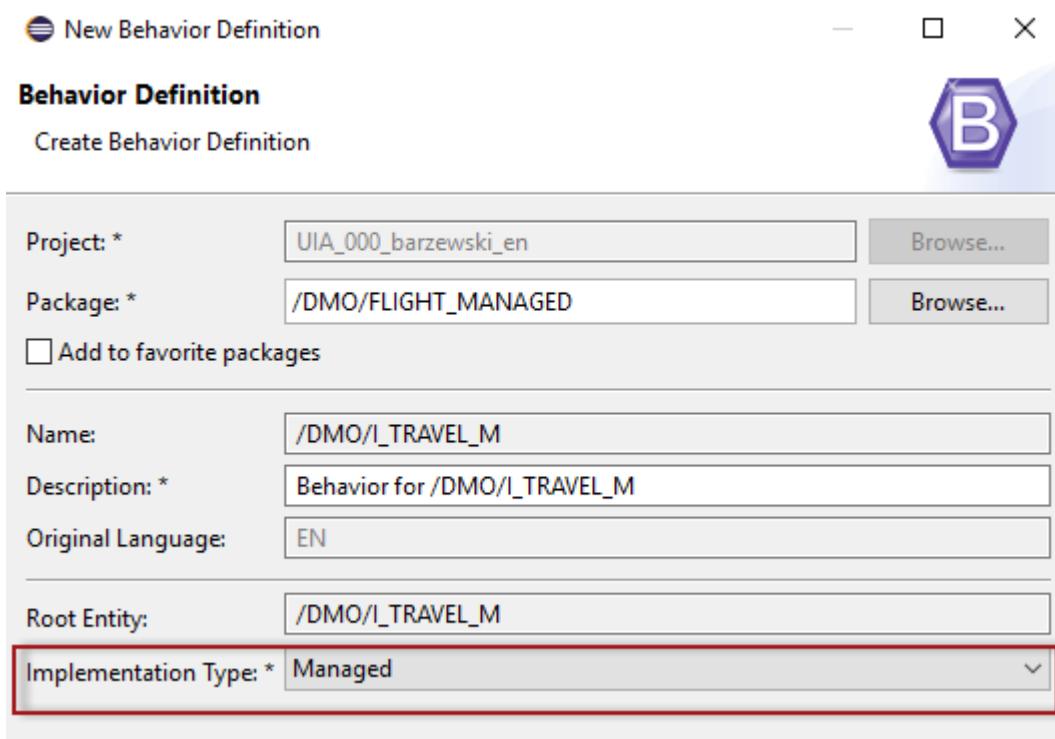


Figure: First Page of the New Behavior Definition Wizard

Further information: [\(Tool Reference\) \[page 761\]](#)

Parenthesis: Syntax for Defining Managed Transactional Behavior

To define the transactional behavior for a CDS Entity for managed implementation type, the following syntax is used:

```
/* Header of behavior definition */
[implementation] managed;
/* Definition of entity behavior */
define behavior for CDSEntity [alias AliasName]
/* Entity properties */
[implementation in class ABAP_CLASS_NAME [unique] ]
persistent table DB_TABLE_NAME
[etag {master Field | dependent by _Association}]
lock {master Field [unmanaged] | dependent by _Association}
[authorization {master(instance) | dependent by _Association}]
{
/* Static field control */
[field (read only | mandatory) field1[, field2, ..., fieldn];]
/* Dynamic field control */
[field (features: instance) field1[, field2, ..., fieldn];]
/* Managed Early Numbering */
field ( [readonly,] numbering:managed ) keyfield1[, keyfield2, ..., keyfieldn];];
/* Mapping entity's field types with table field types */
mapping for DB_TABLE_NAME corresponding;
/* Standard operations */
[internal] create;
[internal] update;
[internal] delete;
/* Actions */
action ActionName;
/* Associations */
[association AssociationName [abbreviation AbbreviationName] {[create;] } ]
/* Validations */
[validation ValidationName on save { field1, ..., fieldn; } ]
/* Determinations */
// (1) determination for triggers at field level
[determination DeterminationName on (save | modify) { field f1[, f2, ...,
fn] ; }]
// (2) determination for triggers at entity level
[determination DeterminationName on (save | modify) { create; }]
}
```

Explanation

A behavior definition consists of a header information and a set of definitions for entity behavior. Each entity of the composition tree can be referred in the behavior definition at most once.

→ Remember

Consider that if an entity does not occur in the behavior definition, then it would not be modifiable within the ABAP RESTful application programming model.

The header specifies the implementation type of the business object:

Implementation Type

Values	Effect
managed	<p>This implementation type addresses use cases where all essential parts of an application must be developed from scratch.</p> <p>Whereas for the unmanaged implementation type, the application developer must implement essential components of the REST contract manually, for the managed scenario, on the other hand, all required standard operations (create, update, delete) must only be specified in the behavior definition to obtain a ready-to-run business object. New applications can highly benefit from out-of-the-box support for transactional processing, since the technical implementation aspects are taken over by the business object runtime infrastructure. In this case, the business object framework implements generically the interaction phase and the save sequence. The application developer can then focus on business logic that is implemented using actions, validations, determinations and user interaction.</p>

The behavior description is divided into a section with behavior characteristic that describes several properties for each entity, followed by information on any operations, validations, and determinations enclosed in the brackets { ... }.

The `AliasName` defined in the behavior definition for `CDSEntity` gives you the option of introducing a more concise name than the entity name that is hence easier to read. Note that the length of the `AliasName` is restricted to 20 characters. The `AliasName` becomes visible in the implementation part when implementing the BO's business logic.

The [Behavior Definition Language \(BDL\) \[page 805\]](#) allows you to add the following properties to a behavior definition:

Behavior Characteristic

Characteristic	Effect
<code>implementation in class ...unique</code>	<p>When modelling the behavior for an individual entity, you have the option of assigning a specific behavior pool that implements the behavior for this entity. Behavior for the entity in question is implemented in a behavior pool with the specified name <code>ABAP_CLASS_NAME</code>.</p> <p>By including the restriction <code>implementation in class ... unique</code> for the implementation, you can protect the application against multiple implementations so that each operation can only be implemented once for the relevant entity. Any other class (program) that attempts this, raises an ABAP compiler error.</p>
<code>persistent table ...</code>	This property defines the database table <code>DB_TABLE_NAME</code> for storing <code>CDSEntity</code> 's data changes that result from transactional behavior (in managed implementation type).

Characteristic	Effect
etag master Field etag dependent by _Association	<p>An ETag is used for optimistic concurrency control in the OData protocol to help prevent simultaneous updates of a resource from overwriting each other.</p> <p>The managed scenario updates administrative fields automatically if they are annotated with the respective annotations:</p> <pre>@Semantics.user.createdBy: true @Semantics.systemDateTime.createdAt: true @Semantics.user.lastChangedBy: true @Semantics.systemDateTime.lastChangedAt: true</pre> <p>If the element that is annotated with <code>@Semantics.systemDateTime.lastChangedAt: true</code> is used as an ETag field, it gets automatic updates by the framework and receives a unique value on each update. In this case, you do not have to implement ETag field updates.</p> <p>i Note</p> <p>You can only use ETag fields with types that are compatible to <code>timestampl</code> in the managed scenario.</p> <p>For more information, see Optimistic Concurrency Control [page 87].</p>
lock master / lock master unmanaged /lock dependent by _Association	<p>In the behavior definition, you can determine which entities support direct locking (<code>lock master</code>) and which entities depend on the locking status of a parent or root entity (<code>lock dependent by</code>). For lock dependents it is required to specify which association is used to determine the lock master. This association must be explicitly defined in the behavior definition with association <code>_AssociationToLockMasterEntity</code>.</p> <p>If you want to implement your own locking logic in the managed scenario, you can define an unmanaged lock with <code>lock master unmanaged</code>. In this case, you have to implement the method <code>FOR LOCK</code> just like in the unmanaged scenario, see Implementing the LOCK Operation [page 330].</p> <p>For more information, see Pessimistic Concurrency Control (Locking) [page 91].</p>

Characteristic	Effect
authorization master / authorization dependent by _Association	<p>To protect data from unauthorized access, you can add authorization checks for modifying operations to each entity of the business object. For standard operations such as update, delete, as well as for create by associations and actions, the authorization control is then checked by the BO runtime as soon as the relevant operation is executed.</p> <p>→ Remember</p> <p>With the current release, the root entity is always defined as <code>authorization master</code>, whereas all child entities are defined as <code>authorization dependent</code>. If a child entity is modified (update, delete, create by association) or an action is invoked on that entity, the authorization check (that is implemented in the behavior class) of the master is triggered to check if the operation is allowed for being accessed.</p>
	<p>! Restriction</p> <p>With the current release, only instance-based authorization control is supported: <code>authorization master (instance)</code>. This means, static authorization is not yet available. Therefore, you cannot apply authorization checks to create operation (static operation).</p>
	<p>i Note</p> <p>The operations create (by association), update, delete, and actions on child entities are treated as an update on the corresponding root entity (<code>authorization master</code>). Thus, the authorization check implementation is triggered to check the authorization for update at master level - despite of the fact that it was a create (by association), update, delete, or action request at dependent level.</p>

Field Properties

Property	Effect
<code>field (readonly)</code>	<p>Defines that the consumer must not create or update the values of the specified fields. The BO runtime rejects modifying requests when creating or updating the specified fields.</p> <p>The specified fields are grayed out on Fiori UI.</p>
<code>field (mandatory)</code>	<p>Defines that the specified fields must be filled with values – at least at save point of time.</p> <p>The BO runtime rejects saving request in case of an empty value.</p> <p>The specified fields are marked with a specific (red star) icon to indicate them as mandatory.</p>
<code>field (numbering : managed)</code>	<p>Defines that the managed BO framework draws the key value for the determined key field automatically after the <code>CREATE</code> request is executed.</p> <p>If you set the key field to <code>readonly</code>, only internal numbering is possible.</p>

Property	Effect
field (features: instance)	<p>Defines that specified field is handled by dynamic field control at instance level.</p> <p>The dynamic field control must be implemented in a handler class of the behavior pool.</p> <p>More on this: Modeling Static and Dynamic Feature Control [page 187]</p>
mapping for ... corresponding	<p>The addition corresponding automatically maps field types of entity fields to types of DB table fields with the same name. The corresponding fields of different names can be specified through explicitly listed field pairs.</p> <p>More on this: Using Type and Control Mapping [page 477]</p>
Operations, Validations and Determinations	
Operation, ...	Meaning
create update delete	<p>An important part of the transactional behavior of a business object are the standard operations create, update and delete (CUD). Whenever an entity can be created, updated, or deleted, these operations must be declared in the behavior definition.</p> <p>To only provide an operation without exposing it to consumers, the option <code>internal</code> can be set before the operation, for example, <code>internal update</code>. An internal operation can only be accessed from the business logic inside the business object implementation such as from a determination.</p>
association	<p>All compositions that form the business object's structure must also be declared in the behavior definition as associations. An abbreviation <code>AbbreviationName</code> needs to be defined if the composition name in the CDS view is longer than 11 characters. The keyword <code>{create;}</code> is used to declare that the association is create-enabled, which means that instances of the associated entity can be created by the source of the association.</p>
<p>i Note</p> <p>The <code>create_by_association</code> expects that the parent key fields of the child entity that is to be created are <code>read-only</code>.</p>	
action	<p>Actions can be specified as non-standard operations in behavior definitions.</p> <p>For more information, see Actions [page 73].</p>

Operation, ...	Meaning
validation	<p>A validation is an implicitly executed function that checks the consistency of entity instances that belong to a business object.</p> <p>The BO framework implicitly evaluates all validations if the validation's trigger condition is fulfilled at a certain validation time. A trigger condition consists of trigger operation (create, update, create by association) and list of entity fields (trigger elements) belonging to the same entity the validation is assigned to. For validations, the current version of the programming model supports only the save phase as validation time:</p> <pre>validation ValidationName on save { field field1, ..., fieldn; }]</pre> <p>Validations are not allowed to modify data; they can reject the save of inconsistent data and return messages.</p>
determination	<p>A determination is an implicitly executed function that is used to handle side effects of modifications by changing instances and returning messages.</p> <p>The BO framework implicitly invokes a determination if the determination's trigger condition is fulfilled at a certain determination time.</p> <p>(1) The option on (save modify) { field f1[, f2, ..., fn]; } defines the determination time on save (before save) or on modify (immediately after modification) and the trigger fields, which (together with the create or update operation) form the trigger condition for the determination.</p> <p>(2) The option on (save modify) { create; } defines the determination time on save or on modify for the entire entity (all entity fields are trigger fields), which together with the create operation (no update operation!) form the trigger condition for the determination. .</p>

Procedure: Modeling the Behavior for Ready-to-Run Travel BO

As depicted in the listing below, the source code of the behavior definition consists of a header information and three definitions for entity behavior: one for the root travel entity and two for the child entities booking and booking supplements – corresponding to the composition tree of the business object. Note that for each entity of the composition tree, the transactional behavior can be defined in the behavior definition at most once. All required transactional operations of an individual business object's node are specified in the same behavior definition (that is introduced by the keyword `DEFINE BEHAVIOR FOR ...`).

The header specifies managed implementation type of our business object's provider since we are going to implement all essential parts of an application from scratch.

For this implementation type, all required standard operations (create, update, delete) and create by association must only be specified in the behavior definition to obtain a ready-to-run business object.

i Note

The `create_by_association` expects that the parent key fields of the child entity that is to be created are `read-only` as the parent key fields are filled automatically during runtime.

Our TRAVEL business object refers to the underlying CDS data model, which is represented by root entity `/DMO/I_Travel_M`. All data changes related to this entity that result from transactional behavior are stored in the database table `/DMO/TRAVEL_M`. The transactional handling of the business object's root entity travel is mainly determined by the standard operations `create`, `update`, and `delete`. The fact that in our scenario new instances of the booking child entity should also be created for a specific travel instance is considered by the addition of the `_Booking` association. The keyword `{create;}` declares that this association is create-enabled what exactly means that instances of the associated bookings can only be created by a travel instance.

The sub node of TRAVEL business object structure refers to the corresponding data model for bookings that is represented by the child entity `/DMO/I_Booking_M`. The transactional handling of booking child entity is determined by the standard operation `update`. In addition, the create-enabled association `_BookSupplement` is defined for creation of supplements as part of associated booking instances. Similarly, we model the behavior of the booking supplement entity `/DMO/I_BookSuppl_M`. Since we reach the end of the composition tree with this entity, there is no need to define a create-enabled association.

When providing modifying operations, we also must take care for locking support for all relevant entities of the composition tree. For this purpose, the root entity `travel` is defined as `lock master` and the child entities as `lock dependent`. In the latter case, the binding information is specified via the association leading from child entity instance to its lock master instance.

In the managed scenario, all entities receive a separate eTag master field to store eTag values. Like this, every entity can be accessed and modified independently of others.

i [Expand the following listing to view the source code.](#)

Listing 1: Behavior Definition

```
managed;
define behavior for /DMO/I_Travel_M alias travel
persistent table /DMO/TRAVEL_M
with additional save
etag master last_changed_at
lock master
{
  mapping for /DMO/TRAVEL_M corresponding;
  create;
  update;
  delete;
  association _Booking { create; }
}
define behavior for /DMO/I_Booking_M alias booking
persistent table /DMO/BOOKING_M
etag master last_changed_at
lock dependent by _Travel
{
  field ( readonly ) travel_id;
  mapping for /DMO/BOOKING_M corresponding;
  update;
  association _BookSupplement { create; }
  association _Travel { }
}
define behavior for /DMO/I_BookSuppl_M alias booksuppl
persistent table /DMO/BOOKSUPPL_M
etag master last_changed_at
lock dependent by _Travel
{
  field ( readonly ) travel_id, booking_id;
  mapping for /DMO/BOOKSUPPL_M corresponding;
  update;
  association _Travel { }
```

}

5.2.2.3 Testing the Business Object

At this point, you can now test the newly created travel business object for its basic functionality by creating some new instances for the 3 entities (travel, booking, booking supplement) that correspond to the business object's structure, modifying the existing data sets or deleting existing instances.

These options are available to you:

- Testing **with Fiori UI** by using the integrated Fiori UI preview function
- Testing **without Fiori UI**:
 - By executing automated ABAP Unit tests, starting from a generated test class
 - By consuming the newly created business object's entities with EML.

Procedures

To use the integrated Fiori UI preview, perform the following steps:

1. Add required UI annotations to the relevant CDS views.
Add, at least, the @UI.identification annotation on each element to be editable when creating or editing entity instances on the Fiori UI object pages.
2. Create the service definition.
3. Specify which CDS entities are exposed as a UI service.
4. Create a service binding for UI consumption.
5. Activate the UI service endpoint in the local service repository.
6. Run the resulting app by using the integrated UI preview function.

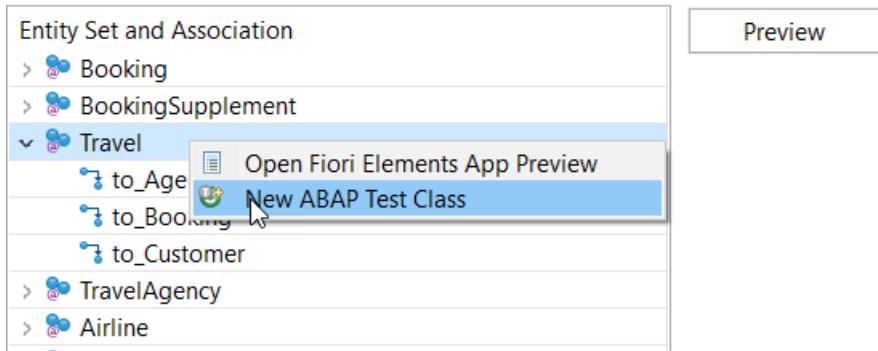
→ Tip

Since UI annotations are not yet defined in the CDS views provided, you can benefit from the generic templates by choosing the *Settings* () button on the (initial) screen that appears. In the *Settings* dialog, click on the *Select all* box for *Columns* and confirm with *OK*. Then choose the *Go* button.

More on this: [Creating an OData Service \[page 24\]](#) and [Designing the User Interface for a Fiori Elements App \[page 35\]](#)

To generate a test class for an individual entity of the OData service, perform the following steps:

1. Create the service definition.
2. Specify which CDS entities are to be exposed as a service.
3. Create a service binding for service consumption.
4. Activate the service endpoint in the local service repository.
5. To launch the test class creation wizard, select the relevant entity in the service binding editor and choose [New ABAP Test Class](#) from context menu.



Creating a Test Class for Travel Entity

For the selected entity, the wizard creates a test class with a source code template for ABAP unit tests. After completing the test code, you can perform CUD operations on relevant entity.

i Note

You can either create a separate test class for each entity or copy and paste the generated code, then change the name of the entity accordingly for writing ABAP unit tests for other entity.

To test the business object functionality by implementing EML consumer class, you can, for example, proceed as follows:

1. Create an EML consumer class.
You can use, for example, the source code template from listing below.
2. Implement MODIFY calls for creating, updating, or deleting instances of business object's entities.

LISTING 1: Template for EML consumer class

```

CLASS MY_EML_CONSUMER_CLASS DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_oo_adt_classrun.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
CLASS /dmo/cl_eml_travel_update IMPLEMENTATION.
  METHOD if_oo_adt_classrun~main.
    " To implement the MODIFY call, add EML code here!
  ENDMETHOD.
ENDCLASS.

```

→ Tip

To check the results of the MODIFY call implemented in the consumer class, run the main method of the consumer class by pressing **F9** key in the class editor and then search for the created, or updated (and deleted) travel, booking and booking supplement instances in the data preview tool (**F8**).

More on this: Entity Manipulation Language (EML) [page 116] and Consuming Business Objects with EML [page 469]

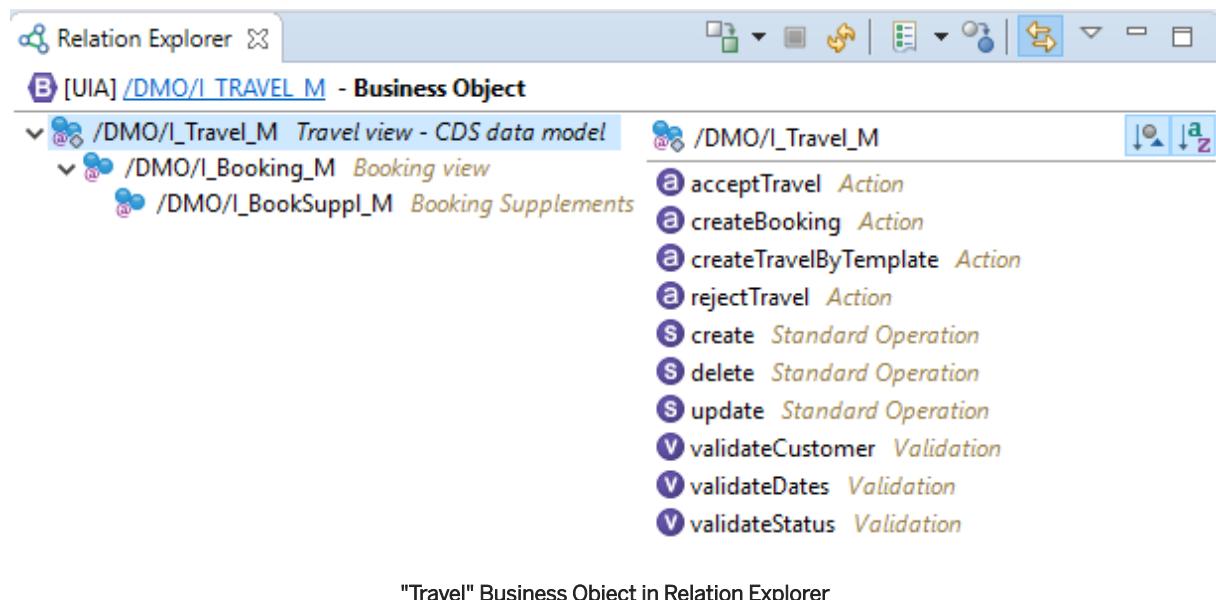
5.2.3 Developing Business Logic

The managed scenario addresses use cases where all essential parts of an application are to be developed from scratch. New applications like this can highly benefit from out-of-the-box support for transactional processing. The corresponding BO runtime manages the entire life cycle of your business objects and covers all aspects of your business application development.

In a managed scenario, the business object framework implements generically the interaction phase and the save sequence. You, as an application developer can then **focus on business logic** that is implemented by adding actions, validations ad determinations and user interaction.

Preview: Resulting Business Object in Relation Explorer

The figure below displays the resulting "Travel" business object in the *Relation Explorer* view. If you choose the *Business Object* context, the *Relation Explorer* provides the composition tree of the business object and displays all operations (including actions), determinations and validations defined by the selected entity.



Contents

- [Creating ABAP Classes for Behavior Implementation \[page 176\]](#)
- [Developing Actions \[page 179\]](#)
- [Adding Static and Dynamic Feature Control \[page 187\]](#)
- [Developing Validations \[page 197\]](#)
- [Developing Determinations \[page 206\]](#)
- [Integrating Additional Save in Managed Business Objects \[page 217\]](#)
- [Integrating Unmanaged Save in Managed Business Objects \[page 227\]](#)

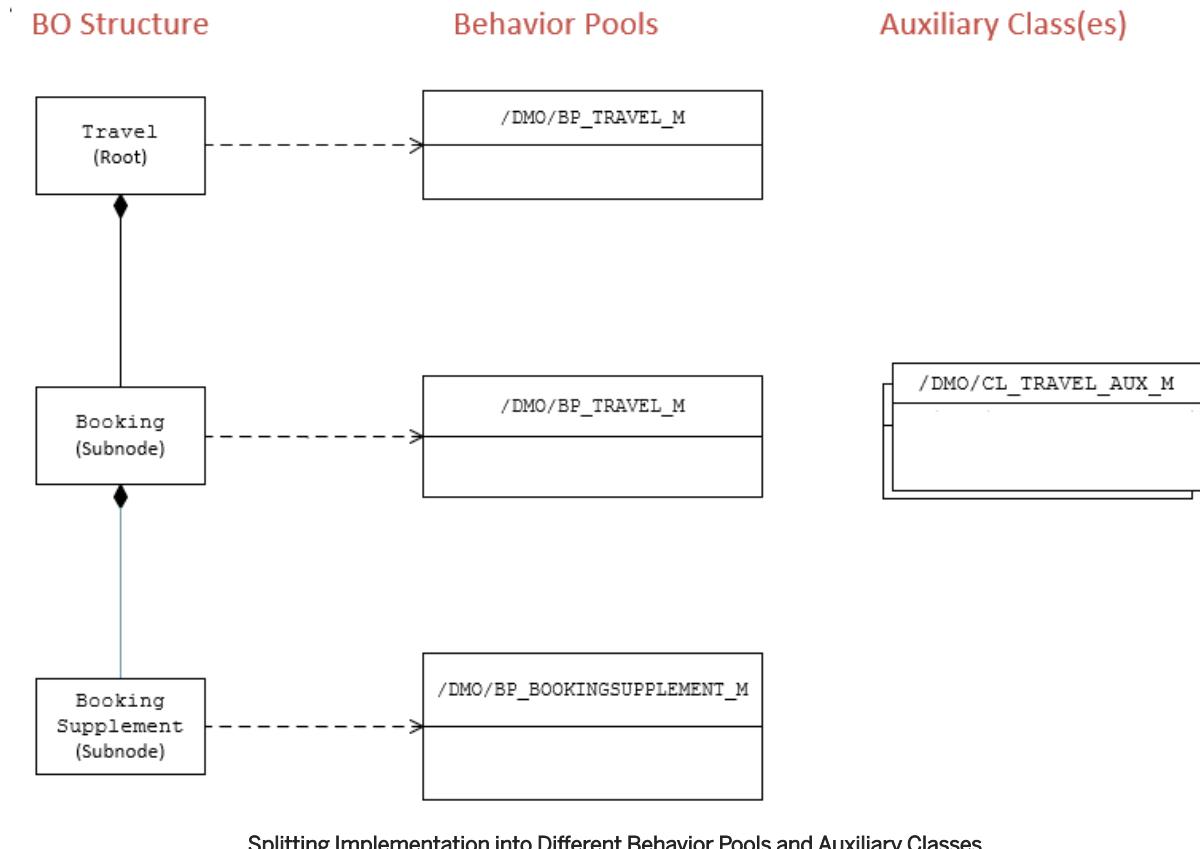
5.2.3.1 Creating ABAP Classes for Behavior Implementation

Until this step, we got by without any line of ABAP code. This was also not necessary, as in case of managed implementation type the technical implementation aspects are taken over by the business object runtime infrastructure itself. In this case, the business object framework implements generically the interaction phase and the save sequence.

However, to provide our application with specific business logic, we will on the one hand extend the behavior definition with actions, feature control, validations and determinations and on the other hand implement it in ABAP code.

In this step, you create the ABAP classes required for extending behavior artifacts of the corresponding behavior definition that you created earlier.

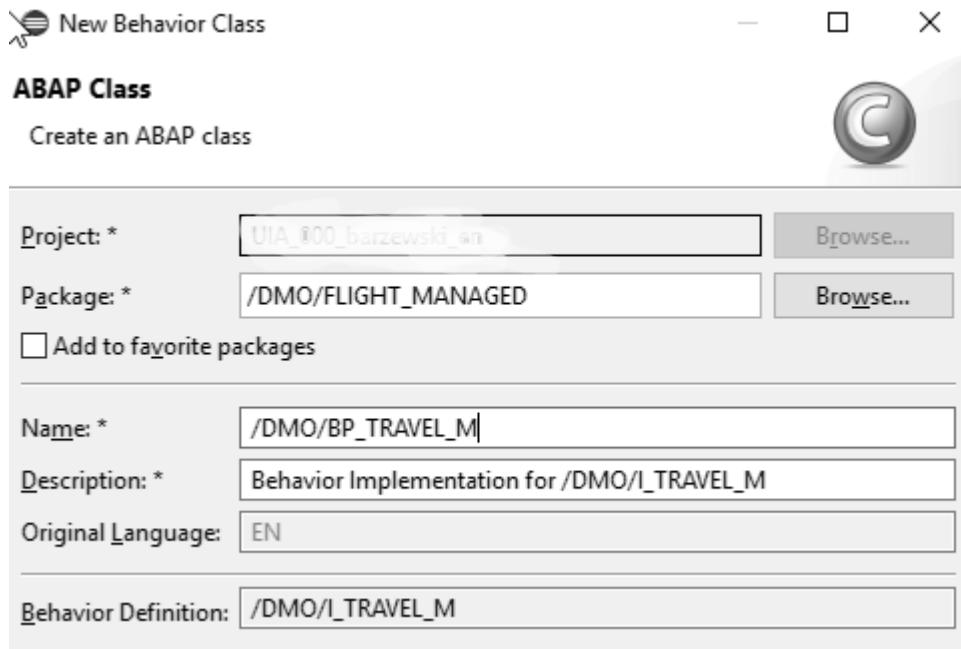
In doing so, we apply the contribution pattern and split the behavior implementation into different behavior pools, one for the travel root entity and the others for the booking and booking supplement child entities (as shown in figure below). In addition, we will make use of a separate auxiliary class for implementing helper methods (such as for methods reused in different handlers) that can be reused in each behavior implementation class.



Procedure 1: Create a Behavior Pool /DMO/BP_TRAVEL_M

To launch the wizard tool for creating a behavior implementation, do the following:

1. In your *ABAP project* (or *ABAP Cloud Project*), select the relevant behavior definition node (*/DMO/I_TRAVEL_M*) in *Project Explorer*.
2. Open the context menu and choose *New Behavior Implementation* to launch the creation wizard.



Creating the Behavior Pool /DMO/BP_TRAVEL_M

Further information: [Naming Conventions for Development Objects \[page 780\]](#)

Compared to the standard ABAP class, the generated behavior pool (in our case /DMO/BP_TRAVEL_M) provides you with an extension FOR BEHAVIOR OF.

```

▶ G /DMO/BP_TRAVEL_M ▶
1① CLASS /dmo/bp_travel_m DEFINITION PUBLIC ABSTRACT FINAL FOR BEHAVIOR OF /dmo/i_travel_m.
2 ENDCLASS.
3
4② CLASS /dmo/bp_travel_m IMPLEMENTATION.
5 ENDCLASS.

```

The screenshot shows the SAP IDE ABAP code editor with the title 'New Global Behavior Pool'. The code area contains the following ABAP code:

```

▶ G /DMO/BP_TRAVEL_M ▶
1① CLASS /dmo/bp_travel_m DEFINITION PUBLIC ABSTRACT FINAL FOR BEHAVIOR OF /dmo/i_travel_m.
2 ENDCLASS.
3
4② CLASS /dmo/bp_travel_m IMPLEMENTATION.
5 ENDCLASS.

```

Below the code, there are tabs for 'Global Class', 'Class-relevant Local Types', 'Local Types', 'Test Classes (non existent)', and 'Macros'. The 'Local Types' tab is selected.

New Global Behavior Pool

The real substance of a behavior pool is located in *Local Types* (there is currently no implementation yet). Here you can implement special local classes, namely handler classes for additional operations (such as actions), validations and determinations that are triggered at specific points in time within the interaction phase.

Note that these classes can be instantiated or invoked only by the *ABAP runtime environment (virtual machine)* [page 804].

Procedure 2: Create a Behavior Pool /DMO/BP_BOOKING_M

Create another behavior pool class /DMO/BP_BOOKING_M, for example, by duplicating the just-created class /DMO/BP_TRAVEL_M .

Listing 1: Global Class /DMO/ BP_BOOKING_M

```
CLASS /dmo/bp_booking_m DEFINITION PUBLIC ABSTRACT FINAL FOR BEHAVIOR OF /dmo/i_travel_m.  
ENDCLASS.  
CLASS /dmo/bp_booking_m IMPLEMENTATION.  
ENDCLASS.
```

i Note

The name behind FOR BEHAVIOR OF always refers to the root entity of the business object.

Procedure 3: Create a Behavior Pool /DMO/BP_BOOKINGSUPLEMENT_M

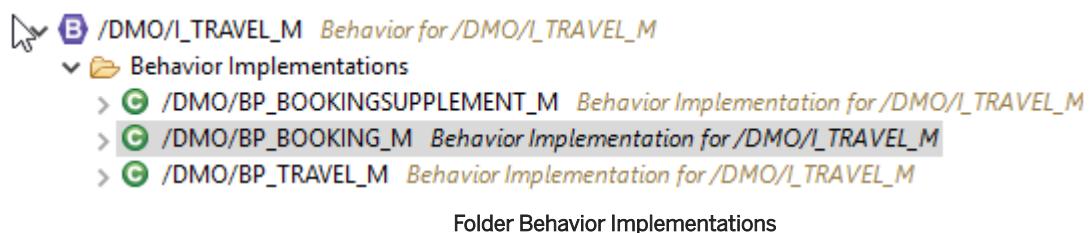
Create another behavior pool class /DMO/BP_BOOKINGSUPPLEMENT_M, for example, by duplicating the class /DMO/BP_TRAVEL_M

Listing 2: Global Class /DMO/ BP_BOOKINGSUPPLEMENT_M

```
CLASS /dmo/bp_bookingsupplement_m DEFINITION PUBLIC ABSTRACT FINAL FOR BEHAVIOR OF /dmo/i_travel_m.  
ENDCLASS.  
CLASS /dmo/bp_bookingsupplement_m IMPLEMENTATION.  
ENDCLASS.
```

Results

The newly created behavior pools are located in the *Behavior Implementations* folder of the corresponding behavior definition node /DMO/I_TRAVEL_M.



Procedure 4: Create an Auxiliary Class /DMO/CL_TRAVEL_AUXILIARY_M

To launch the wizard tool for creating a standard ABAP class, do the following:

1. In your *ABAP project* (or *ABAP Cloud Project*), select the *Source Code Library* ➔ *Classes* node of the relevant package.
2. To launch the creation wizard, open the context menu and choose *New ABAP Class*.

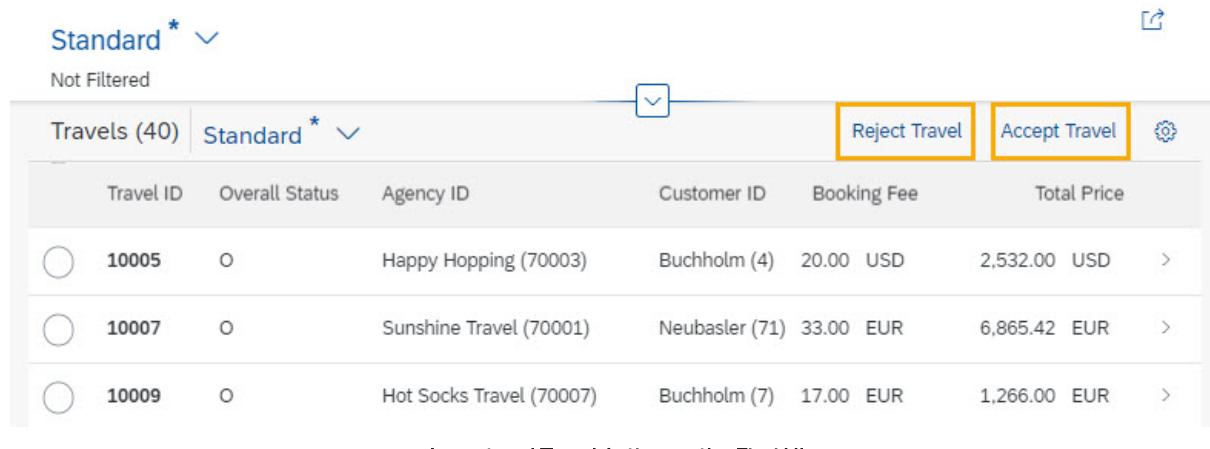
We will use this auxiliary class to offload parts of the handler code and thus reuse them in different behavior pools.

5.2.3.2 Developing Actions

An action is assigned to an individual entity of a business object and is used to implement a modifying non-standard operation as part of the business logic.

This demo scenario implements three actions for the travel entity.

The action `acceptTravel` sets the overall status of the travel entity to accepted (A). The action `rejectTravel` sets the overall status of the travel entity to rejected (X). These actions are an instance action as there is one specific instance of the travel entity for which the status is changed. They return exactly one instance, namely the instance on which the action is executed. The result is therefore defined as `result [1] $self`.



Travels (40)						
Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee	Total Price	
10005	O	Happy Hopping (70003)	Buchholm (4)	20.00 USD	2,532.00 USD	>
10007	O	Sunshine Travel (70001)	Neubasler (71)	33.00 EUR	6,865.42 EUR	>
10009	O	Hot Socks Travel (70007)	Buchholm (7)	17.00 EUR	1,266.00 EUR	>

Accept and Travel Action on the Fiori UI

The action `createTravelByTemplate` creates a new travel instance based on the values of an already existing travel instance. This action is an instance action as the end user has to select one instance which serves as the basis for the entity instance to be created. Of course, a different key is assigned. It returns exactly one instance of the same entity type the action is assigned for. The result is therefore defined as `result [1] $self`.

Standard * ▾

Not Filtered

Travels (40) Standard ▾											
Travel ID	Agency ID	Customer ID	Starting Date	End Date	Total Price	Booking Status					
27	Hot Socks Travel (70007)	Buchholm (7)	Jun 3, 2019	Jun 10, 2019	1,266.00	EUR	X			>	
44	Fly High (70002)	Buchholm (4)	Jun 14, 2019	Jun 21, 2019	2,488.97	EUR	X			>	
53	Happy Hopping (70003)	Buchholm (4)	Jun 13, 2019	Jun 27, 2019	2,532.00	EUR	X			>	

Create Travel By Template Action

For more information about actions in general, see [Actions \[page 73\]](#).

Activities Relevant to Developers

1. [Defining Actions as Part of the Behavior Definition \[page 180\]](#)
2. [Implementing Actions \[page 181\]](#)
3. [Enabling Actions for UI Consumption \[page 186\]](#)

5.2.3.2.1 Defining Actions as Part of the Behavior Definition

Procedure: Adding Actions to the Behavior Definition

Corresponding to the listing below, add the following actions to the behavior defintion /DMO/I_TRAVEL_M.

Listing: Added Actions to /DMO/I_Travel_M

All three actions acceptTravel, rejectTravel, and createTravelByTemplate have a similar syntax: they have no input parameters and the output parameter is the same entity for which the action is executed.

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
{
  ...
  // instance actions
```

```

action acceptTravel result [1] $self;
action rejectTravel result [1] $self;
// instance action for copying travel instances
action createTravelByTemplate result [1] $self;
...
}
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
...
{
  ...
  // No actions defined for bookings
}
define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
...
{
  ...
  // No actions defined for booking supplements
}

```

5.2.3.2.2 Implementing Actions

In this topic we demonstrate how you can implement the two actions `createTravelByTemplate` and `acceptTravel`.

Action `createTravelByTemplate`

This action provides a template for creating a new travel based on an already existing travel instance.

UI Preview

When we run the final app, the UI screen provides the button *Create Travel by Template* for the action as shown in the figure below.

Travels (2) Standard ▾					Create Travel by Template	Delete	Create
Travel ID	Agency ID	Customer ID	Starting Date	End Date			
1	Happy Hopping (70003)	Buchholm (6)	Jul 5, 2019	Jul 13, 2019			
Total Price: 1,587.04 EUR							
Booking Status: A							

Accessing the Action on Fiori UI

In change mode (after the user clicks the *Edit* button on Fiori UI's object page), the end user is able to change the relevant travel fields as shown in the figure below.

Travel ID [1,...,99999999]: 3

Agency ID: 70003

Customer ID: 6

Starting Date: Jul 12, 2019

End Date: Aug 11, 2019

Booking Fee: 234.00 EUR

Total Price: 1,587.04 EUR

Status: O

Description: Enter your comments here

Modifying Travel Data in Edit Mode (Object Page)

As soon as the user chooses the **Save** button on the object page, the data is persisted in the corresponding database table and a travel instance with a new travel ID is created.

Travels (3) Standard ▾					Create Travel by Template	Delete	Create
Travel ID	Agency ID	Customer ID	Starting Date	End Date			
1	Happy Hopping (70003)	Buchholm (6)	Jul 5, 2019	Jul 13, 2019	Total Price: 1,587.04 EUR		
2	Sunshine Travel (70001)	Lautenbach (591)	Jul 8, 2019	Aug 7, 2019	Total Price: 9,940.46 EUR		
3	Happy Hopping (70003)	Buchholm (6)	Jul 13, 2019	Aug 11, 2019	Total Price: 1,587.04 EUR		

Created Travel Instance

Definition

In the behavior definition, the action `createTravelByTemplate` is defined as follows:

```
managed;
define behavior for /DMO/I_Travel_M alias travel
  ...
{
  ...
  action createTravelByTemplate result [1] $self;
  ...
}
```

Procedure: Implementing `createTravelByTemplate` Action in the Handler Class

The `createTravelByTemplate` action is implemented in the `copy_travel` method within the handler class `lhc_travel` which is part (local types) of the behavior pool `/DMO/BP_TRAVEL_M`.

The local handler class `lhc_travel` inherits from class `cl_abap_behavior_handler` and is automatically instantiated by the framework.

The signature of `copy_travel` method includes the importing parameter keys for referring to the `travel` (root) instances, which contains the template data to be copied into the travel root instances to be created. To identify the root entity, the alias `travel` is used - according to the alias that is specified in the behavior definition. The action is then addressed with `FOR ACTION travel~createTravelByTemplate`.

As given in the listing below, the basic structure of the `copy_travel` method includes:

- The maximum of travel number from all existing travel instances is determined in the `SELECT MAX` statement.
- The EML read operation provides read access to the selected travel instance by using the key. The result of this read operation is stored in `lt_read_result`. To access data of the relevant entity fields, the `FIELDS` statement is used. Based on the resulting data, the parameter `lt_create` defines the new data set as a template for the travel instance to be created.
- The actual creation of new travel instance is implemented using EML `MODIFY` statement. The modifying operation takes place in `LOCAL MODE`. As a result, the `create` operation is excluded from the authorization checks (that may be implemented later on).

Listing 1: Action Implementation

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS copy_travel FOR MODIFY IMPORTING keys FOR ACTION  
travel~createTravelByTemplate RESULT result.  
...  
ENDCLASS.  
CLASS lhc_travel IMPLEMENTATION.  
...  
METHOD copy_travel.  
    SELECT MAX( travel_id ) FROM /dmo/travel_m INTO @DATA(lv_travel_id).  
    READ ENTITY /dmo/i_travel_m  
        FIELDS ( travel_id  
                agency_id  
                customer_id  
                booking_fee  
                total_price  
                currency_code )  
        WITH VALUE #( FOR travel IN keys ( %key = travel-%key ) )  
    RESULT DATA(lt_read_result)  
    FAILED failed  
    REPORTED reported.  
    DATA(lv_today) = cl_abap_context_info=>get_system_date( ).  
    DATA lt_create TYPE TABLE FOR CREATE /DMO/I_Travel_M\travel.  
    lt_create = VALUE #( FOR row IN lt_read_result INDEX INTO idx  
        ( travel_id      = lv_travel_id + idx  
        agency_id       = row-agency_id  
        customer_id    = row-customer_id  
        begin_date     = lv_today  
        end_date       = lv_today + 30  
        booking_fee    = row-booking_fee  
        total_price    = row-total_price  
        currency_code  = row-currency_code  
        description    = 'Enter your comments here' )  
    ).
```

```

        overall_status = 'O' ) ) . " Open
MODIFY ENTITIES OF /DMO/I_TRAVEL_M IN LOCAL MODE
ENTITY travel
  CREATE FIELDS (
    travel_id
    agency_id
    customer_id
    begin_date
    end_date
    booking_fee
    total_price
    currency_code
    description
    overall_status )
  WITH lt_create
  MAPPED mapped
  FAILED failed
  REPORTED reported.
  result = VALUE #( FOR create IN lt_create INDEX INTO idx
    ( %cid_ref = keys[ idx ]-%cid_ref
      %key      = keys[ idx ]-travel_id
      %param    = CORRESPONDING #( create ) ) ) .
ENDMETHOD.

...
ENDCLASS.

```

Action acceptTravel

This action provides the end user with the option of accepting individual travels without switching to EDIT mode.

UI Preview

If you run the app, the resulting UI screen provides you with the label *Accept Travel* for the new action - as shown in the figure below.

Travels (2) Standard ▾					Reject Travel	Accept Travel	⚙			
Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee						
1	O ←	Happy Hopping (70003)	Buchholm (6)	234.00 EUR >						
Total Price: 1,587.04 EUR										
Description: Vacation										

Action on Fiori UI

Travels (2) Standard ▾					Reject Travel	Accept Travel	⚙			
Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee						
1	A ←	Happy Hopping (70003)	Buchholm (6)	234.00 EUR						
Total Price: 1,587.04 EUR										
Description: Vacation										

Changed Status as a Result of Action Execution

Definition

Remember, in the behavior definition, the action acceptTravel is defined as follows:

```
managed;
define behavior for /DMO/I_Travel_M alias travel
  ...
{
  ...
  action acceptTravel result [1] $self;
  ...
}
```

Procedure: Implementing acceptTravel Action in the Handler Class

The acceptTravel action is implemented in the set_status_completed method within the handler class lhc_travel which belongs to *local types* component of the behavior pool /DMO/BP_TRAVEL_M.

To update overall_status data of a selected travel instance, operation UPDATE is specified in the EML MODIFY statement. Note that the modifying operation takes place in LOCAL MODE. As a result, the update operation is excluded from the authorization checks (that may be implemented later on).

The UPDATE call allows to trigger delta updates on consumer side where only the key field key-travel_id and the new value need to be supplied. From provider side, it allows to identify which fields are overwritten and which need to be kept according to the DB data.

To complete the action implementation, the result parameter must be filled. In this case, the result parameter is \$self, meaning the same entity on which the action is executed. To fill the concrete values of the result entity, an EML read must be implemented.

Listing 2: Action Implementation

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
  METHODS set_status_completed FOR MODIFY IMPORTING keys FOR ACTION
travel~acceptTravel RESULT result.
  ...
ENDCLASS.
CLASS lhc_travel IMPLEMENTATION.
  ...
  METHOD set_status_completed.
    " Modify in local mode: BO-related updates that are not relevant for
authorization checks
    MODIFY ENTITIES OF /DMO/I_TRAVEL_M IN LOCAL MODE
      ENTITY travel
        UPDATE FIELDS ( overall_status )
          WITH VALUE #( for key in keys ( travel_id      = key-travel_id
                                         overall_status = 'A' ) ) "
Accepted
  FAILED failed
  REPORTED reported.
  " read changed data for result
  MODIFY ENTITIES OF /DMO/I_TRAVEL_M IN LOCAL MODE
    ENTITY travel
      UPDATE FIELDS ( overall_status )
        WITH VALUE #( for key in keys ( travel_id      = key-travel_id
                                         overall_status = 'A' ) ) "
Accepted
  FAILED failed
  REPORTED reported.
  " Read changed data for action result
  READ ENTITIES OF /DMO/I_Travel_M IN LOCAL MODE
```

```

ENTITY travel
  FIELDS ( agency_id
            customer_id
            begin_date
            end_date
            booking_fee
            total_price
            currency_code
            overall_status
            description
            created_by
            created_at
            last_changed_at
            last_changed_by )
  WITH VALUE #( for key in keys ( travel_id = key-travel_id ) )
RESULT DATA(lt_travel).
result = VALUE #( for travel in lt_travel ( travel_id = travel-travel_id
                                              %param      = travel
                                              ) .
ENDMETHOD.

```

→ Tip

The implementation of the `rejectTravel` action takes place analogous to the `acceptTravel` action in the handler class `lhc_travel` which belongs to local types component of the behavior pool `/DMO/BP_TRAVEL_M`. You can view the corresponding method implementation in the package `/DMO/FLIGHT_MANAGED`.

5.2.3.2.3 Enabling Actions for UI Consumption

Actions are often directly related to business objects' instances that you can, for example, trigger for a set of travels instances. End users can select an individual line item (that represents an instance) and execute certain actions on the selected line item.

You can use the following UI annotation to expose actions to the consumer:

Syntax (for LineItem)

```

define view <CDSEntity> as select from <DATA_SOURCE> as ..
{
  @UI.lineItem: [ ...
    { type: #FOR_ACTION, dataAction: 'action_1', label: 'label 1', position: 10 },
    { type: #FOR_ACTION, dataAction: 'action_2', label: 'label 2', position:20 },
    ... ]
}

```

→ Remember

The `dataAction` element references the name of an action as it is defined in the behavior definition.

5.2.3.3 Adding Static and Dynamic Feature Control

As an application developer you may want to determine, which entities of your business object should be create-, delete- and update-enabled, so that they can be modified during consumption using [EML \[page 812\]](#) or [OData services \[page 816\]](#). In addition, you may also want to control which (UI) fields of an entity are read-only or which actions in which usage scenarios are enabled or disabled for execution by the end users.

In ABAP RESTful application programming model, feature control is precisely the way to accomplish such tasks. It allows you to control the visibility and changeability of fields, operations or entire entities.

Depending on whether feature control refers to specific instances or is independent of each entity instance, we distinguish between **instance-bound** and **static feature control**.

The availability of feature control values is modeled in a behavior definition. Unlike static feature control, instance-bound feature control requires not only a definition but also an implementation in a handler class of the behavior pool. Therefore, we also talk about **dynamic feature control** in case of instance-bound feature control.

Feature control can be related to an entire entity or to individual elements of an entity, such as individual fields or operations.

Activities Relevant to Developers

1. [Modeling Static and Dynamic Feature Control \[page 187\]](#)
2. [Implementing Dynamic Feature Control \[page 190\]](#)

5.2.3.3.1 Modeling Static and Dynamic Feature Control

Both, static and dynamic feature control is defined for different levels (entity, field, or action level) in the behavior definition by using the following syntax:

Syntax: Feature Control in the Behavior Definition

```
[implementation] managed;
define behavior for CDSEntity [alias AliasName]
implementation in class ABAP_CLASS [unique]
...
{
/* (1) Feature control at entity level */
/* Static operation control*/
    internal create
    internal update
    internal delete
/* or (instance-based) dynamic operation control: implementation required! */
    update (features: instance);
```

```

    delete (features: instance);
    association {create (features:instance); }
/*-(2) Feature control at field level */
/* Static field control */
    field (read only | mandatory) f1[, f2, ..., fn];
/* or dynamic field control: implementation required! */
    field (features: instance) f1[, f2, ..., fn];
/* (3) Feature control for actions */
/* Static action control */
    internal action ActionName [...]
/* or dynamic action control: implementation required! */
    action ( features: instance ) ActionName [... ]
}

```

(1) Feature Control at Entity Level

To manage a transactional behavior, an entity of a business object offers standard operations create, update, and delete for external consumption using EML or OData services. To only provide these operations without exposing them to consumers, the option `internal` can be set before the operation name. An **internal operation** can only be accessed from the business logic inside the business object implementation such as from an action, a validation or a determination.

For dynamic control of operations acting on individual entity instances, the option `(features: instance)` must be added to the operation in question. However, an implementation in the referenced class pool `ABAP_CLASS` is necessary for this. For each relevant operation, you can specify in the implementing handler of the class pool the following values:

- `ENABLED` - if the operation is enabled
- `DISABLED` - if the operation is disabled.

More on this: [Implementing Dynamic Feature Control \[page 190\]](#)

(2) Feature Control at Element Level

Within the bracket of `define behavior for CDSEntity { ... }`, you can specify for fields of an entity if they should have certain access restrictions.

For static feature control, it is sufficient to define these restrictions in the behavior definition alone:

- The option `field (read only) f1` does not allow the `f1` field to be changed in create and update operations.
- The option `field (mandatory) f2` requires the corresponding field `f2` to be supplied with a value in create operations. For update operations, the field must not have a non-initial value.

i Note

To classify multiple fields in the same way, the comma notation can be used:

```
field(read only) f1, f2, f3;
```

More on this: [Simple Value Help \[page 430\]](#)

For defining instance-based field control, the option `(features: instance)` must be added to the field in question. In this case however, the implementation of dynamic feature control in the referenced class pool `ABAP_CLASS` is required. When implementing dynamic field control you have the option of specifying the following values for each field that is notated with `(features: instance)`:

- `UNRESTRICTED` – field has no restrictions

- MANDATORY – field is mandatory
- READ_ONLY – field is read-only
- ALL – All restrictions are requested.

More on this: [Implementing Dynamic Feature Control \[page 190\]](#)

(3) Feature Control for Actions

Specific operations of an entity of a business object can be defined using actions. Similar to standard operations, you can define internal actions in the behavior definition by adding the option internal to the operation name. Internal actions can only be accessed from the business logic inside the business object implementation such as from validations, determinations, or from other non-internal actions.

For dynamic control of actions acting on individual entity instances, the option (features: instance) must be added to the relevant action in the behavior definition. The required implementation must be provided in the referenced class pool ABAP_CLASS. For each relevant action, you can specify in the implementing handler of the class pool the following values:

- ENABLED - if the action is enabled
- DISABLED - if the action is disabled.

More on this: [Implementing Dynamic Feature Control \[page 190\]](#)

Procedure: Adding Feature Control to Behavior Definition /DMO/I_Travel_M

Corresponding to the listing below, add the static and dynamic feature control to each entity in the behavior definition.

i [Expand the following listing to view the source code.](#)

Listing: Added Feature Control to BDEF /DMO/I_Travel_M

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
...
{
    // administrative fields: read only
    field ( readonly ) last_changed_at, last_changed_by, created_at, created_by;
    // mandatory fields that are required to create a travel
    field ( mandatory ) agency_id, overall_status, booking_fee, currency_code;
    // dynamic field control
    field (features : instance ) travel_id;
    // dynamic action control
    action ( features: instance ) acceptTravel result [1] $self;
    action ( features: instance ) rejectTravel result [1] $self;
    // dynamic operation control
    association _Booking { create (features:instance); }
}
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
...
{
    // static field control
    field ( mandatory ) carrier_id, connection_id, flight_date, booking_status;
    field ( readonly ) travel_id;
```

```

// dynamic field control
field (features : instance ) booking_id, booking_date, customer_id;
// dynamic operation control
association _BookSupplement { create (features:instance); }
...
}
define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
...
{
  // static field control
  field ( mandatory ) supplement_id, price;
  field ( readonly ) travel_id, booking_id;
  // dynamic field control
  field (features : instance ) booking_supplement_id;
...
}

```

For the travel entity, we define all admin fields as read-only, whereas the fields that are required for creating a travel instance are defined as mandatory.

The key field `travel_id` plays a special role and is intended for dynamic field control. In the corresponding implementation, we will distinguish whether the travel instance already exists (in case of EDIT) or whether it should still be created.

Examples of dynamic action control are the two methods `acceptTravel` and `rejectTravel`. Depending on the status value (`overall_status`), these actions can become enabled or disabled.

Creating bookings by association is dynamically controlled. You can only create new bookings if the `overall_status` of the corresponding travel instance is not rejected.

For the other entities (booking and booking supplement), there is some feature control on field level. Similar to creating bookings by association, booking supplements can only be created if the corresponding booking instance status is not booked.

5.2.3.3.2 Implementing Dynamic Feature Control

The implementation of dynamic feature control is based on the ABAP language in a local handler class (`lhc_handler`) as part of the behavior pool. As depicted in the listing below, each such local class inherits from the base handler class `CL_ABAP_BEHAVIOR_HANDLER`.

The dynamic feature control for an entity is implemented in this handler class using the method `FOR FEATURES`. For more information, see [<method> FOR FEATURES \[page 729\]](#).

The output parameter `result` is used to return the feature control values.

These include

- field control information: `%field-fieldx`
- action control information: `%features-%action-action_name`
- standard operation control information for update and delete: `%features- (%update|%delete)`
- operation control information for create by association: `%assoc-assoc_name`.

Listing 1: Implementing Feature Control in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS feature_ctrl_method FOR FEATURES  
    IMPORTING keys REQUEST requested_features FOR aliasedEntity RESULT result.  
ENDCLASS.  
CLASS lhc_handler IMPLEMENTATION.  
METHOD feature_ctrl_method.  
    " Read data required  
    READ ENTITY CDSEntityName  
        FIELDS ( field1 field2 )  
            WITH VALUE #( FOR keyval IN keys ( %key = keyval-%key ) )  
        RESULT DATA(lt_result_variable).  
  
    " Return feature control information  
    result = VALUE #( FOR ls_variable IN lt_result_variable  
        ( %key  
            = ls_variable-%key  
    " Field control information  
        %field-field1  
        %field-field2  
            = if_abap_behv=>fc-f-read_only  
            = if_abap_behv=>fc-f-mandatory  
    " Action control information  
        %features-%action-action_name = COND #( WHEN condition  
            THEN if_abap_behv=>fc-o-  
disabled  
            ELSE if_abap_behv=>fc-o-  
enabled )  
    " Operation (example: update) control information  
        %features-%update  
            = COND #( WHEN condition  
            THEN if_abap_behv=>fc-o-  
disabled  
            ELSE if_abap_behv=>fc-o-  
enabled )  
    " Operation control information for create by association  
        %assoc-Assoc  
            = COND #( WHEN condition  
            THEN if_abap_behv=>fc-o-  
disabled  
            ELSE if_abap_behv=>fc-o-  
enabled )  
        ) ).  
ENDMETHOD.  
ENDCLASS.
```

Dynamic Feature Control for Travel Entity

In the following step, we will apply the feature control specifically to our demo scenario.

UI Preview

The figure below shows the realization of the static and dynamic field control using the example of the travel object page that has been switched to edit mode. All mandatory fields are marked with a red star. Since it is an already existing travel instance selected, a re-assignment of travel ID number is suppressed in edit mode.

Travel Booking

Travel ID [1,...,99999999]: 2	*Starting Date: Jul 8, 2019 <input type="button" value="Calendar"/>	Total Price: 9,940.46 EUR
*Agency ID: 70001 <input type="button" value="Edit"/>	*End Date: Aug 7, 2019 <input type="button" value="Calendar"/>	*Status [O(Open) A(Accepted) X(Canceled)]: O <input type="button" value="Edit"/>
*Customer ID: 591 <input type="button" value="Edit"/>	*Booking Fee: 234.00 <input type="button" value="EUR"/> <input type="button" value="Edit"/>	Description: Business Trip USA

Static and Dynamic Feature Control on Field Level

The following figure shows the effect of dynamic control on the two action buttons *Accept Travel* and *Reject Travel*: Since the selected travel instance has a status of *A* (Accepted), the action button *Accept Travel* is disabled.

Travels (2) Standard ▾					Reject Travel	Accept Travel
Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee		
<input checked="" type="checkbox"/> 1	A	Happy Hopping (70003)	Buchholm (6)	234.00 EUR		<input type="button" value="Accept Travel"/>
	Total Price: 1,587.04 EUR Description: Vacation					
<input type="radio"/> 2	O	Sunshine Travel (70001)	Lautenbach (591)	234.00 EUR		<input type="button" value="Accept Travel"/>
	Total Price: 9,940.46 EUR Description: Business Trip					

Dynamic Feature Control for Actions

Another dynamic feature control is defined for the creating bookings by associations and creating booking supplements by association. In this case the *CREATE* button for creating associated booking instance is displayed if the travel instance's `overall_status` is not rejected. It is hidden, if the travel is set to rejected. Likewise, the *CREATE* button for bookings supplements is displayed depending on the booking status of the corresponding booking instance.

Dynamic Feature Control for Create By Association

Definition

In the behavior definition, the feature control for travel entity is defined as follows:

```
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
...
{
  field ( readonly ) last_changed_at, last_changed_by, created_at, created_by;
  field ( mandatory ) agency_id, overall_status, booking_fee, currency_code;
  field (features : instance ) travel_id;
  action ( features: instance ) acceptTravel result [1] $self;
  action ( features: instance ) rejectTravel result [1] $self;
  association _Booking { create (features:instance); }
...
}
```

Procedure: Implementing Dynamic Feature Control for Travel Entity

The method implementation begins with reading the `travel_id` field that is designated for dynamic field control. This read access is implemented by the EML read operation that provides access to the selected travel instance by using the `%key` component that contains all key elements of an entity. The result of this read operation is stored in `lt_travel_result`.

Depending on the value of `overall_status` field, the actions `rejectTravel` and `acceptTravel` are enabled or disabled, and the create by association is possible or not.

i [Expand the following listing to view the source code.](#)

Listing 2: Implementation of Feature Control for Travel Entity

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
    ...  
    METHODS get_features FOR FEATURES IMPORTING keys REQUEST requested_features  
FOR travel RESULT result.  
ENDCLASS.  
CLASS lhc_travel IMPLEMENTATION.  
    ...  
    METHOD get_features.  
        READ ENTITY /dmo/i_travel_m  
            FIELDS ( travel_id overall_status description )  
                WITH VALUE #( FOR keyval IN keys ( %key = keyval-%key ) )  
            RESULT DATA(lt_travel_result).  
        result = VALUE #( FOR ls_travel IN lt_travel_result  
            ( %key  
                %field-travel_id  
                = ls_travel-%key  
                = if_abap_behv=>fc-f-  
read_only  
                %features-%action-rejectTravel = COND #( WHEN ls_travel-  
overall_status = 'X'  
                    THEN  
if_abap_behv=>fc-o-disabled ELSE if_abap_behv=>fc-o-enabled )  
                %features-%action-acceptTravel = COND #( WHEN ls_travel-  
overall_status = 'A'  
                    THEN  
if_abap_behv=>fc-o-disabled ELSE if_abap_behv=>fc-o-enabled  
                %assoc_Booking = COND #( WHEN ls_travel-  
overall_status = 'X'  
                    THEN  
if_abap_behv=>fc-o-disabled ELSE if_abap_behv=>fc-o-enabled  
                ) ).  
    ENDMETHOD.  
    ...  
ENDCLASS.
```

Dynamic Feature Control for Booking Entity

UI Preview

The following figure shows the effect of static and dynamic field control after switching the Fiori UI object page for bookings in edit mode.

Booking Booking Supplement

Booking Number: *Flight Number:
1 1537

Booking Date: *Flight Date:
Jul 4, 2019 Jul 9, 2019

Customer ID: Flight Price:
7 555.00 EUR

*Airline ID: *Status [N(New)| X(Canceled)| B(Booked)]:
UA N

Static and Dynamic Feature Control at Field Level

Definition

```
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
...
{
    // static field control
    field ( mandatory ) carrier_id, connection_id, flight_date, booking_status;
    field ( readonly ) travel_id;
    // dynamic field control
    field (features : instance ) booking_id, booking_date, customer_id;

    // dynamic feature control create booking supplement by association
    association _BookSupplement { create (features:instance); }
    ...
}
```

Procedure: Implementing Dynamic Feature Control for Booking Entity

The `get_features` method implements dynamic field control for the fields `booking_id`, `booking_date`, `customer_id` and the dynamic feature control for the create booking supplements by association.

i [Expand the following listing to view the source code.](#)

Listing 3: Dynamic feature control for bookings implemented in behavior class /dmo/bp_booking_m

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
...
METHODS get_features FOR FEATURES IMPORTING keys REQUEST requested_features FOR
booking RESULT result.
ENDCLASS.
CLASS lhc_travel IMPLEMENTATION.
...
METHOD get_features.
    READ ENTITY /dmo/i_booking_m
        FIELDS ( booking_id booking_date customer_id booking_status )
        WITH VALUE #( FOR keyval IN keys ( %key = keyval-%key ) )
        RESULT DATA(lt_booking_result).
    result = VALUE #( FOR ls_travel IN lt_booking_result

```

```

( %key          = ls_travel-%key
  %field-booking_id = if_abap_behv=>fc-f-read_only
  %field-booking_date = if_abap_behv=>fc-f-read_only
  %field-customer_id = if_abap_behv=>fc-f-read_only
  %assoc- _BookSupplement = COND #( WHEN ls_travel-
booking_status = 'B'                               THEN if_abap_behv=>fc-
o-disabled ELSE if_abap_behv=>fc-o-enabled    )
) .
ENDMETHOD.
ENDCLASS.

```

Dynamic Feature Control for Booking Supplement Entity

UI Preview

The following figure shows the effect of static and dynamic field control after switching the Fiori UI object page for booking supplements in edit mode.

Booking Supplement

Book. Supp. Number: *Product Price:

1 17.00 EUR

*Product ID: ML-0012

Save Cancel

Static and Dynamic Feature Control at Field Level

Definition

```

define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
...
{
  // static field control
  field ( mandatory ) supplement_id, price;
  field ( readonly ) travel_id, booking_id;
  // dynamic field control
  field (features : instance ) booking_supplement_id;
}

```

Procedure: Implementing Dynamic Feature Control for Booking Supplement Entity

The `get_features` method is implemented analogous to the one of the previous case of bookings.

i *Expand the following listing to view the source code.*

Listing 4: Dynamic feature control for booking supplements /dmo/bp_bookingsupplement_m

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
    ...  
    METHODS get_features FOR FEATURES IMPORTING keys REQUEST requested_features FOR  
booksuppl RESULT result.  
ENDCLASS.  
CLASS lhc_travel IMPLEMENTATION.  
    ...  
    METHOD get_features.  
        READ ENTITY /dmo/i_bookspl_m FROM VALUE #( FOR keyval IN keys  
            ( %key =  
keyval-%key  
                %control-booking_supplement_id =  
if_abap_behv=>mk-on  
                    ) )  
            RESULT DATA(lt_bookspl_result).  
        result = VALUE #( FOR ls_travel IN lt_bookspl_result  
            ( %key = ls_travel-%key  
                %field-booking_supplement_id = if_abap_behv=>fc-f-  
read_only  
            ) ).  
    ENDMETHOD.  
ENDCLASS.
```

5.2.3.4 Developing Validations

A validation is an implicitly executed function intended to check the data consistency of an existing instance of an entity (consistency validation). It is implicitly invoked by the business object's framework as soon as a trigger condition at a predefined point in time is fulfilled. Validations can return messages to the consumer and reject inconsistent instance data from being saved.

Example

A validation is implemented to check if the customer ID referenced in the Travel order is valid. This validation is assigned to the "Travel" entity having defined update as the trigger operation on entity "Travel". As soon as a customer ID is updated by the consumer, the validation will check it and return a warning message if the ID is unknown.

Remember

Validations never modify any instance data but return the messages and keys of failed (inconsistent) entity instances.

Note

When working with determinations and validations you have to consider the following runtime specifics:

- The determination result must not change if the determination is executed several times under the same conditions (idempotence).
- The execution order of validations and determination is not determined. If there is more than one determination/validation on one operation you cannot know which determination/validation is executed first.

- It is not allowed to use EML modify statements in validations.
- If you create or update an instance and delete it with the same request, it can happen that an EML read operation in a determination on modify fails as instances with the given key cannot be found.

How do Consistency Validations Work in General?

Consistency validations check if an instance is consistent regarding the consistency criteria imposed by the business requirements. Such validations are called at predefined points within the business object's transaction cycle to ensure that business object entities are persisted in a consistent state. Each such validation defines a trigger condition that is checked by the business object framework during the save phase of a transaction. If the trigger condition is fulfilled, the consistency validation is executed. Otherwise, if there are inconsistent instance data, the validation sends messages to the consumer and prevents the transaction from being saved until the inconsistency is corrected.

A validation is determined by the following properties:

Assigned Entity

A validation must always be assigned to the entity for which it may return state messages. If, for example, a validation is defined for the root entity, the validation implementation must not return messages that refer to instances of child entities.

Trigger Operations

Validations can only be triggered by create and update operations.

Trigger Time

Validations are executed at predefined points within the business object's transaction cycle. Note that the current version of the programming model supports the on save as the only point in time for triggering validations.

While the save phase is reached, the validation is triggered implicitly by the business object framework during the `Check_Before_Save` point of time.

Trigger Field

Field belonging to the same entity the validation is assigned to.

Trigger condition

A trigger condition consists of trigger operation and list of trigger fields belonging to the same entity the validation is assigned to.

Activities Relevant to Developers

1. [Defining Validations \[page 199\]](#)
2. [Implementing Validations \[page 200\]](#)

5.2.3.4.1 Defining Validations

Validations are specified for individual business object's entities in the behavior definition by using the following syntax:

Syntax for Defining Validations

```
[implementation] managed;
define behavior for CDSEntity [alias AliasedName]
implementation in class ABAP_CLASS [unique]
...
{
    validation ValidationName on save { field f1[, f2, ..., fn]; }
...
}
```

The definition of a validation of a CDSEntity is initiated with the `validation` keyword, followed by `ValidationName`.

The option `on save { field f1[, f2, ..., fn]; }` defines the trigger time and the trigger fields, which (together with the create or update operation) form the trigger condition for the validation.

A validation that belongs to a business object's entity is implemented in the behavior pool that is specified in the behavior definition by the keyword `implementation in class ABAP_CLASS [unique]`.

Procedure: Adding Validations to Behavior Definition /DMO/I_Travel_M

Corresponding to the listing below, add the required validations to each entity in the behavior definition.

 *Expand the following listing to view the source code.*

Listing: Added Validations to /DMO/I_Travel_M Behavior Definition managed;

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
...
{
    ...
    validation validateCustomer on save { field customer_id; }
    validation validateAgency   on save { field agency_id; }
    validation validateDates   on save { field begin_date, end_date; }
    validation validateStatus  on save { field overall_status; }
}
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
...
{
    ...
    validation validateStatus on save { field booking_status; }
}
```

```

define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
...
{
    ...
    // No validations
}

```

For the travel entity, we define the following three validations:

- validateCustomer – checks if the customer ID that is entered by the consumer is valid
- validateAgency – checks if the agency ID that is entered by the consumer is valid
- validateDates – checks for the travel instance if the value of travel's begin_date is in future and if value of the end_date is after begin_date
- validateStatus - checks for the travel instance if the value of overall_status field is valid.

For the booking entity, also a validation validateStatus is defined. However, in contrast to the validation with the same name in the travel entity, booking_status is specified as trigger field.

5.2.3.4.2 Implementing Validations

Implementing Validations in the Local Handler Class of the Behavior Pool

A validation that is assigned to a business object's entity is implemented in the behavior pool, which is referred in the behavior definition by the keyword implementation in class ABAP_CLASS_NAME [unique].

The implementation of a validation is based on the ABAP language (which has been extended compared to the standard using an additional syntax) in a local handler class as part of the behavior pool.

As depicted in the listing below, each such local class inherits from the base handler class CL_ABAP_BEHAVIOR_HANDLER. The signature of the corresponding handler method lhc_handler is typed based on the entity that is defined by the keyword FOR VALIDATION followed by AliasedEntityName~ValidationName.

To identify the instance of an entity, it_key is used as input parameter.

Listing 1: Implementing an Action in a Local Handler Class

```

CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS method_name FOR VALIDATION AliasedEntityName~ValidationName
        IMPORTING it_key FOR AliasedEntityName.
ENDCLASS.

CLASS lhc_handler IMPLEMENTATION.
METHOD method_name.
// Implement method for validation here!
ENDMETHOD.
ENDCLASS.

```

Validations on Travel Entity

UI Preview

If the user enters an invalid Customer ID (an ID that is not available in the customer database table /DMO/Customer) the validation is initiated at the save time. As a result saving the instance data is rejected and a corresponding message is returned to the user.



Customer ID field validation on Fiori UI

Definition

In the behavior definition, the validations on the travel entity are defined as follows:

```
managed;
define behavior for /DMO/I_Travel_M alias travel
  ...
{
  ...
    validation validateCustomer on save { field customer_id; }
    validation validateAgency   on save { field agency_id; }
    validation validateDates   on save { field begin_date, end_date; }
    validation validateStatus  on save { field overall_status; }
}
```

Procedure: Implementing Validations on Travel Entity

It should come as no surprise that the signatures of all three methods for implementing validations are very similar.

In the following listing 2, you see details about the implementation of the `validate_customer` method:

(1) First, the EML read operation `READ ENTITY` provides read access to the selected travel instance by using the key.

To access data of the relevant entity fields, the `FIELDS ()` addition is used. As a result of the read operation the entered (changed) value of the `customer_id` field for the selected travel instance are written into the table `lt_travel`. Only this value is relevant for the validation.

(2) In the following lines of code, we prepare an optimization for the following database select. By using the sorted table `lt_customer`, we ensure that only data records with non-initial customer IDs are considered for database access.

(3) By accessing the contents of the database table `/dmo/customer`, we can check whether the entered customer ID exists on the database at all.

(4) If the validation detects inconsistencies (customer ID is not valid or is initial), we must provide the key of all inconsistent instances as failed key and return error messages to the consumer. For all failed instances, also corresponding messages are created by calling the framework's `new_message` method. For access to suitable message texts, a message class `/DMO/CM_FLIGHT_LEGAC` from the `/DMO/FLIGHT` package is reused.

i *Expand the following listing to view the source code.*

Listing 2: Implementation of validate_customer Method

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
  ...  
  METHODS validate_customer          FOR VALIDATION travel~validateCustomer  
    IMPORTING keys FOR travel.  
  METHODS validate_agency           FOR VALIDATION travel~validateAgency  
    IMPORTING keys FOR travel.  
  METHODS validate_dates            FOR VALIDATION travel~validateDates  
    IMPORTING keys FOR travel.  
  METHODS validate_travel_status    FOR VALIDATION travel~validateStatus  
    IMPORTING keys FOR travel.  
ENDCLASS.  
CLASS lhc_travel IMPLEMENTATION.  
*****  
*  
* Validate customer data when saving travel data  
*  
*****  
METHOD validate_customer.  
  " (1) Read relevant travel instance data  
  READ ENTITIES OF /DMO/I_Travel_M IN LOCAL MODE  
  ENTITY travel  
    FIELDS ( customer_id )  
    WITH CORRESPONDING #( keys )  
  RESULT DATA(lt_travel).  
  DATA lt_customer TYPE SORTED TABLE OF /dmo/customer WITH UNIQUE KEY  
customer_id.  
  " (2) Optimization of DB select: extract distinct non-initial customer IDs  
  lt_customer = CORRESPONDING #( lt_travel DISCARDING DUPLICATES MAPPING  
customer_id = customer_id EXCEPT * ).  
  DELETE lt_customer WHERE customer_id IS INITIAL.  
  IF lt_customer IS NOT INITIAL.  
    " (3) Check if customer ID exists  
    SELECT FROM /dmo/customer FIELDS customer_id  
      FOR ALL ENTRIES IN @lt_customer  
      WHERE customer_id = @lt_customer->customer_id  
      INTO TABLE @DATA(lt_customer_db).  
  ENDIF.  
  " (4) Raise msg for non existing customer id  
  LOOP AT lt_travel INTO DATA(ls_travel).  
    IF ls_travel->customer_id IS INITIAL  
      OR NOT line_exists( lt_customer_db[ customer_id = ls_travel->customer_id ] ).  
        APPEND VALUE #( travel_id = ls_travel->travel_id ) TO failed.  
        APPEND VALUE #( travel_id = ls_travel->travel_id  
                      %msg = new_message( id      = '/DMO/CM_FLIGHT_LEGAC'  
                                         number = '002'  
                                         v1     = ls_travel->customer_id )
```

```

        severity =
if_abap_behv_message=>severity-error )
    %element-customer_id = if_abap_behv=>mk-on )
    TO reported.
ENDIF.
ENDLOOP.
ENDMETHOD.
```

The validation to check if the agency ID is valid has exactly the same structure as validate_customer. The code can be downloaded from ABAP Flight Reference Scenario in /DMO/BP_TRAVEL_M.

In the following listing 3, you see details about the implementation of the validate_dates method:

- (1) The EML read operation READ ENTITY provides read access to data from trigger fields begin_date and end_date. As a result of the read operation the entered (changed) values of the begin_date and end_date fields for the selected travel instance are written into the table row lt_travel.
- (2), (3) The validation detects inconsistencies if the date value of end_date is before the date value of begin_date or if the date value of begin_date is in the past. Each validation can produce failed keys and messages. Any failed keys are stored in the table FAILED whereas the REPORTED table includes all instance-specific messages.

i [Expand the following listing to view the source code.](#)

Listing 3: Implementation of validate_dates Method

```

*****+
*
* Check validity of dates
*
*****+
METHOD validate_dates.
  " (1) Read relevant travel instance data
  READ ENTITY '/DMO/I_Travel_M\travel' FIELDS ( begin_date end_date ) WITH
    VALUE #( FOR <root_key> IN keys ( %key = <root_key> ) )
    RESULT DATA(lt_travel_result).
  LOOP AT lt_travel_result INTO DATA(ls_travel_result).
    " (2) Check if end_date is not before begin_date
    IF ls_travel_result-end_date < ls_travel_result-begin_date.
      APPEND VALUE #( %key      = ls_travel_result-%key-
                     travel_id   = ls_travel_result-travel_id ) TO failed.
      APPEND VALUE #( %key      = ls_travel_result-%key-
                     %msg       = new_message( id      = /dmo/
                     cx_flight_legacy=>end_date_before_begin_date-msgid
                                         number   = /dmo/
                     cx_flight_legacy=>end_date_before_begin_date-msgno
                                         v1       = ls_travel_result-
                     begin_date
                                         v2       = ls_travel_result-
                     end_date
                                         v3       = ls_travel_result-
                     travel_id
                                         severity =
if_abap_behv_message=>severity-error )
                     %element-begin_date = if_abap_behv=>mk-on
                     %element-end_date  = if_abap_behv=>mk-on ) TO reported.
    " (3) Check if begin_date is in the future
    ELSEIF ls_travel_result-begin_date <
cl_abap_context_info=>get_system_date( ).
      APPEND VALUE #( %key      = ls_travel_result-%key-
                     travel_id   = ls_travel_result-travel_id ) TO failed.
```

```

APPEND VALUE #( %key = ls_travel_result-%key
               %msg = new_message( id      = /dmo/
cx_flight_legacy=>begin_date_before_system_date-msgid
                           number   = /dmo/
cx_flight_legacy=>begin_date_before_system_date-msgno
                           severity =
if_abap_behv_message=>severity-error )
               %element-begin_date = if_abap_behv=>mk-on
               %element-end_date   = if_abap_behv=>mk-on ) TO reported.
ENDIF.
ENDLOOP.
ENDMETHOD.
```

In the following listing 4, you see details about the validate_travel_status method:

Checking the validity of overall_status values is performed within a case loop. The valid values O, X, and A are specified directly in the source code.

i [Expand the following listing to view the source code.](#)

Listing 4: Implementation of validate_travel_status Method

```

***** ****
*
* Validate travel status when saving travel data
*
***** ****
METHOD validate_travel_status.
  READ ENTITY /DMO/I_Travel_M\travel FIELDS ( overall_status ) WITH
    VALUE #( FOR <root_key> IN keys ( %key = <root_key> ) )
    RESULT DATA(lt_travel_result).
  LOOP AT lt_travel_result INTO DATA(ls_travel_result).
    CASE ls_travel_result-overall_status.
      WHEN 'O'.  " Open
      WHEN 'X'.  " Cancelled
      WHEN 'A'.  " Accepted
      WHEN OTHERS.
        APPEND VALUE #( %key = ls_travel_result-%key ) TO failed.
        APPEND VALUE #( %key = ls_travel_result-%key
                       %msg = new_message( id      = /dmo/
cx_flight_legacy=>status_is_not_valid-msgid
                           number   = /dmo/
cx_flight_legacy=>status_is_not_valid-msgno
                           v1       = ls_travel_result-
overall_status
                           severity =
if_abap_behv_message=>severity-error )
                       %element-overall_status = if_abap_behv=>mk-on ) TO
reported.
        ENDCASE.
    ENDLOOP.
ENDMETHOD.
```

Validation on Booking Entity

UI Preview

In this case, we want to validate the status values of booking instances. If a user enters the wrong value **K**, according to the figure below, the instance with its data is not saved and an error message is displayed instead.

The screenshot shows a Fiori application interface. At the top, there are two tabs: "Booking" (which is selected) and "Booking Supplement". Below the tabs, there are several input fields:

- Booking Number: 1
- Flight Number: 1537
- Booking Date: Jun 27, 2019
- Flight Date: Jul 2, 2019
- Customer ID: 7
- Flight Price: 422.00 EUR
- Airline ID: UA
- Status: K (highlighted with a dotted border)

At the bottom right of the form are "Save" and "Cancel" buttons. Below the form is a "Messages" panel:

1
⚠ Travel status not valid for travel K

Below the messages panel is the caption: "Booking Status Validation on Fiori UI".

Definition

```
define behavior for /DMO/I_Booking_M alias booking
...
{
  ...
  validation validateStatus on save { field booking_status; }
```

Procedure: Implementing Validation on Booking Entity

The `validate_booking_status` method is implemented analogous to the one of the previous case when checking travel status.

i *Expand the following listing to view the source code.*

Listing 5: Implementation of validate_booking_status Method

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS validate_booking_status FOR VALIDATION booking~validateStatus  
IMPORTING keys FOR booking.  
...  
ENDCLASS.  
CLASS lhc_travel IMPLEMENTATION.  
METHOD validate_booking_status.  
READ ENTITY /DMO/I_Travel_M\booking  
FIELDS ( booking_status )  
WITH VALUE #( FOR <root_key> IN keys ( %key = <root_key> ) )  
RESULT DATA(lt_booking_result).  
LOOP AT lt_booking_result INTO DATA(ls_booking_result).  
CASE ls_booking_result-booking_status.  
WHEN 'N'. " New  
WHEN 'X'. " Canceled  
WHEN 'B'. " Booked  
WHEN OTHERS.  
    APPEND VALUE #( %key = ls_booking_result-%key ) TO failed.  
    APPEND VALUE #( %key = ls_booking_result-%key  
        %msg = new_message( id      = /dmo/  
cx_flight_legacy=>status_is_not_valid-msgid  
cx_flight_legacy=>status_is_not_valid-msgno  
        number   = /dmo/  
        v1       = ls_booking_result-  
booking_status  
        severity =  
if_abap_behv_message=>severity-error )  
        %element-booking_status = if_abap_behv=>mk-on ) TO  
reported.  
    ENDCASE.  
ENDLOOP.  
ENDMETHOD.  
...  
ENDCLASS.
```

Developing Determinations

A determination is an implicitly executed action that handles side effects of modified entity instances. It is invoked by the business object's framework as soon as a determination's **trigger condition** at a predefined point in time, the **determination time**, is fulfilled.

Determinations are triggered internally based on changes made to the entity instance of a business object. The trigger conditions are checked by the business object framework at different points during the transaction cycle, depending on the determination time and the changing operations on the relevant entity instances. For each determination, it is necessary to specify both the determination time and the changes that form the trigger condition. A trigger condition consists of a list of fields belonging to the same entity the determination is assigned to and the changing operations that include creating or updating entity instances. We call these operations **trigger operations**.

In case a field is changed (after creation or update), the condition is fulfilled. The framework evaluates the triggering condition of all determinations at certain points in time (determination time). For determinations, this is currently either after each modification or during the save phase.

You can use a determination primarily to compute data that is derived from the values of other fields. The determined fields and the determining fields either belong to the same entity or to different entities of a business object.

As a result, a determination can modify entity instances and return transition messages (error, warning, information, success).

As a typical example, a determination is used to calculate the invoice amount depending from a changed price or quantity of an item. The determination would be assigned to the "Item" entity having defined the trigger operation `UPDATE` and `CREATE` on entity "Item". As soon as the consumer creates a new item entity instance or updates the quantity and price of an existing one, the determination will run and update that item and re-calculate the invoice amount to the new value.

i Note

When working with determinations and validations you have to consider the following runtime specifics:

- The determination result must not change if the determination is executed several times under the same conditions (idempotence).
- The execution order of validations and determination is not determined. If there is more than one determination/validation on one operation you cannot know which determination/validation is executed first.
- It is not allowed to use EML modify statements in validations.
- If you create or update an instance and delete it with the same request, it can happen that an EML read operation in a determination on modify fails as instances with the given key cannot be found.

Trigger Operations

Depending on the use case in question, the business object framework checks the triggers of a determination at specific points during the transaction cycle.

A determination time defines at what time in the transaction cycle the trigger condition of that determination should be evaluated. For example, the re-calculation of the invoice amount should take place every time after a modification is performed (determination time: `on modify`), but only if there are instances of a child entity that were updated (trigger operation: `update`).

The following list shows which trigger operations are evaluated at which determination time:

Determination Time	Trigger Operation: CREATE	Trigger Operation: UPDATE
On Modify	X	X
On Save	X	X

The following list shows which trigger operations are evaluated at the field level (field trigger) and for the entire entity (entity trigger):

Trigger Type	Trigger Operation: CREATE	Trigger Operation: UPDATE
Field Trigger	X	X

Trigger Type	Trigger Operation: CREATE	Trigger Operation: UPDATE
Entity Trigger	X	-

! Restriction

The current version of the ABAP RESTful programming model does not support `DELETE` as trigger operation in determinations. We suggest using the following workaround in this case: Implement the `DELETE` operation for child entities in a regular action, which implements the semantics of the determination, for example a calculation) and disable the `DELETE` operation by dynamic feature control – if required.

This restriction is the reason why the `DELETE` standard operation is not defined for booking and booking supplement entities in our demo scenario. The deletion only takes place at the root level, for entire travel instances.

Activities Relevant to Developers

1. [Defining Determinations \[page 208\]](#)
2. [Implementing Determinations \[page 210\]](#)

5.2.3.5.1 Defining Determinations

Determinations are specified for individual business object's entities in the behavior definition by using the following syntax:

Syntax for Defining Determinations

```
[implementation] managed;
define behavior for CDSEntity [alias AliasedName]
implementation in class ABAP_CLASS [unique]
...
{
    // (1) determination for triggers at field level
    determination DeterminationName on (save | modify) { field f1[, f2, ..., fn] ; }
    // (2) determination for triggers at entity level
    determination DeterminationName on (save | modify) { create; }
...
}
```

The definition of a determination that is assigned to a CDSEntity is initiated with the determination keyword, followed by DeterminationName.

Depending on whether the trigger condition is defined at the field or the entity level, the following options are available:

(1) The option on (save | modify) { field f1[, f2, ..., fn]; } defines the determination time on save (before save) or on modify (immediately after modification) and the trigger fields, which (together with the create or update operation) form the trigger condition for the determination.

(2) The option on (save | modify) { create; } defines the determination time on save or on modify for the entire entity (all entity fields are trigger fields), which together with the create operation (no update operation!) form the trigger condition for the determination.

A determination that belongs to a business object's entity is implemented in the behavior pool that is specified in the behavior definition by the keyword implementation in class ABAP_CLASS [unique].

Procedure: Adding Determinations to Behavior Definition /DMO/I_Travel_M

As depicted in the listing below, add the following determinations to relevant entities in the behavior definition.

i [Expand the following listing to view the source code.](#)

Listing: Added Determinations to /DMO/I_Travel_M Behavior Definition

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
...
{
  ...
  // No determinations for travel entity
}
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
...
{
  ...
  // determination for calculation of total flight price
  determination calculateTotalFlightPrice on modify { field flight_price,
  currency_code; }
}
define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
...
{
  ...
  // determination for calculation of total suppl. price
  determination calculateTotalSupplmPrice on modify { field price,
  currency_code; }
}
```

The determination calculateTotalFlightPrice on the booking entity is intended to handle the calculation of total price of all flight bookings that belong to the selected travel. The determination will be triggered by on modify as determination time when creating new booking instances or updating the flight price value or when

changing the currency. In other words: both fields `flight_price` and `currency_code` serve as trigger fields and form, together with create and update operations, the trigger condition for the determination.

For the booking supplement entity, the determination `calculateTotalSupplmPrice` is defined analogously. This determination is used to calculate the total price of all supplements assigned to an individual flight booking instance.

All calculated values are finally used to re-calculate the total travel price at the root entity level.

5.2.3.5.2 Implementing Determinations

Implementing Determinations in the Local Handler Class of the Behavior Pool

A determination that is assigned to a business object's entity is implemented in the behavior pool, which is referred in the behavior definition by the keyword `implementation in class ABAP_CLASS_NAME [unique]`.

The implementation of a determination is based on the ABAP language in a local handler class as part of the behavior pool.

As depicted in the listing below, each such local class inherits from the base handler class `CL_ABAP_BEHAVIOR_HANDLER`. The signature of the corresponding handler method `lhc_handler` is typed based on the entity that is defined by the keyword `FOR DETERMINATION` followed by `AliasedEntityName~DeterminationName`.

To identify the instance of an entity, `it_key` is used as input parameter of the handler method.

Listing 1: Implementing a Determination in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS method_name FOR DETERMINATION AliasedEntityName~DeterminationName  
    IMPORTING it_key FOR AliasedEntityName.  
ENDCLASS.  
CLASS lhc_handler IMPLEMENTATION.  
METHOD method_name.  
// Implement method for determination here!  
ENDMETHOD.  
ENDCLASS.
```

Determination on Booking Entity

UI Preview

The figure below refers to the starting point of viewing with a newly created travel instance with the initial amount (*Total Price*) and the travel currency *0.00 EUR*.

Travels (1) Standard ▾						Create Travel by Template	Delete	Create	⋮
Travel ID	Agency ID	Customer ID	Starting Date	End Date	Total Price	Booking Status			
1	Sunshine Travel (70001)	Neubasler (71)	Jul 24, 2019	Aug 21, 2019	0.00 EUR	O			>

The Newly Created Travel with a Total Price 0.00 EUR.

If a user adds a flight booking to the travel, then also the travel's *Total Price* is updated.

The screenshot illustrates the SAP Fiori user interface for managing travel and bookings. It consists of two main sections: the Travel page and the Booking page.

Travel Page: This section shows a single travel entry with the following details:

- Travel ID: 1
- Agency ID: Sunshine Travel (70001)
- Customer ID: Neubasler (71)
- Starting Date: Jul 24, 2019
- End Date: Aug 21, 2019
- Total Price: 0.00 EUR
- Booking Status: O

Booking Page: This section shows a single booking entry with the following details:

- Booking Number: 1
- Booking Date: Jul 24, 2019
- Customer ID: 7
- Airline ID: United Airlines, Inc. (UA)
- Flight Number: 1537
- Flight Date: Jul 29, 2019
- Flight Price: 422.00 EUR
- Status: N

The "Total Price" field on the Travel page is highlighted with a yellow box, indicating it has been updated. The "Flight Price" field on the Booking page is also highlighted with a yellow box, indicating it triggered the update.

Added Bookings Triggers an Update on Travel Data

If the user switches the booking's object page to edit mode and then changes the *Flight Price*, then the *Total Price* is also updated at root level.

The screenshot shows a flight booking entry in a Fiori application. At the top, there's a table with columns: Booking Number, Booking Date, Customer ID, Airline ID, Flight Number, Flight Date, and Flight Price. The Flight Price row is highlighted with a yellow box. Below this is a status field containing 'N'. A large orange arrow points down from the Flight Price field to a summary table at the bottom. The summary table has columns: Travel ID, Agency ID, Customer ID, Starting Date, End Date, Total Price, and Booking Status. The Total Price row is also highlighted with a yellow box. The bottom right of the screen shows 'Save' and 'Cancel' buttons.

Booking Number	Booking Date	Customer ID	Airline ID	Flight Number	Flight Date	Flight Price
1	Jul 24, 2019	7	UA	1537	Jul 29, 2019	411 00 EUR
Status: N						

Travel ID	Agency ID	Customer ID	Starting Date	End Date	Total Price	Booking Status
1	Sunshine Travel (70001)	Neubasler (71)	Jul 24, 2019	Aug 21, 2019	411.00 EUR	O

Updated Total Price at Root Level

Definition

In the behavior definition, the determination on the booking entity is defined as follows:

```
managed;
define behavior for /DMO/I_Booking_M alias booking
  ...
{
  ...
  determination calculateTotalFlightPrice on modify { field flight_price,
  currency_code; }
```

Procedure: Implementing the Determination Code in the Auxiliary Class

Since the pricing calculation is required for both determinations and we will access them from different handler classes, we outsource the more generic code of the `calculate_price` method to a separate class, which we already created in one of the previous steps as auxiliary class `/dmo/cl_travel_auxiliary_m`.

i Expand the following listing to view the source code.

Listing 2: Implementation of `calculate_price` Method as Part of the Auxiliary Class

```
CLASS /dmo/cl_travel_auxiliary_m DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
  * Type definition for import parameters -----
    TYPES tt_travel_id          TYPE TABLE OF /dmo/travel_id.
    TYPES tt_travel_reported    TYPE TABLE FOR REPORTED /dmo/i_travel_m.
    TYPES tt_booking_reported   TYPE TABLE FOR REPORTED /dmo/
i_booking_m.
    TYPES tt_bookingsupplement_reported TYPE TABLE FOR REPORTED /dmo/
i_booksuppl_m.
  * Method for price calculation (used in determination calls) -----
    CLASS-METHODS calculate_price IMPORTING it_travel_id TYPE tt_travel_id.
  ENDCLASS.

CLASS /dmo/cl_travel_auxiliary_m IMPLEMENTATION.
  METHOD calculate_price.
    DATA: total_book_price_by_trav_curr  TYPE /dmo/total_price,
```

```

        total_suppl_price_by_trav_curr TYPE /dmo/total_price.
IF it_travel_id IS INITIAL.
  RETURN.
ENDIF.
* (1) Read relevant travel instance data
-----
READ ENTITIES OF /dmo/i_travel_m
  ENTITY travel
    FROM VALUE #( FOR lv_travel_id IN it_travel_id (
      travel_id = lv_travel_id
      %control-currency_code = if_abap_behv=>mk-on ) )
    RESULT DATA(lt_read_travel).
* (2) Read relevant booking instance data by using the association (travel to
booking)-----
READ ENTITIES OF /dmo/i_travel_m
  ENTITY travel BY \booking
    FROM VALUE #( FOR lv_travel_id IN it_travel_id (
      travel_id = lv_travel_id
      %control-flight_price = if_abap_behv=>mk-on
      %control-booking_id = if_abap_behv=>mk-on
      %control-currency_code = if_abap_behv=>mk-on ) )
    RESULT DATA(lt_read_booking_by_travel).
LOOP AT lt_read_booking_by_travel INTO DATA(ls_booking)
  GROUP BY ls_booking-travel_id INTO DATA(ls_travel_key).
  ASSIGN lt_read_travel[ KEY entity COMPONENTS travel_id = ls_travel_key ]
    TO FIELD-SYMBOL(<ls_travel>).
  CLEAR <ls_travel>-total_price.
  LOOP AT GROUP ls_travel_key INTO DATA(ls_booking_result)
    GROUP BY ls_booking_result-currency_code INTO DATA(lv_curr).
    total_book_price_by_trav_curr = 0.
    LOOP AT GROUP lv_curr INTO DATA(ls_booking_line).
      total_book_price_by_trav_curr += ls_booking_line-flight_price.
    ENDLOOP.
    IF lv_curr = <ls_travel>-currency_code.
      <ls_travel>-total_price += total_book_price_by_trav_curr.
    ELSE.
* (2') Call procedure for currency conversion
-----
/dmo/cl_flight_amdp=>convert_currency(
  EXPORTING
    iv_amount           = total_book_price_by_trav_curr
    iv_currency_code_source = lv_curr
    iv_currency_code_target = <ls_travel>-currency_code
    iv_exchange_rate_date =
  cl_abap_context_info=>get_system_date( )
  IMPORTING
    ev_amount           = DATA(total_book_price_per_curr)
  ).
  <ls_travel>-total_price += total_book_price_per_curr.
  ENDIF.
ENDLOOP.
ENDLOOP.
* (3) Read relevant supplement data by using the association (booking to
booking supplement)-----
READ ENTITIES OF /dmo/i_travel_m
  ENTITY booking BY \BookSupplement
    FROM VALUE #( FOR ls_travel IN lt_read_booking_by_travel (
      travel_id          = ls_travel-travel_id
      booking_id         = ls_travel-booking_id
      %control-price     = if_abap_behv=>mk-on
      %control-currency_code = if_abap_behv=>mk-on ) )
    RESULT DATA(lt_read_booksuppl).
LOOP AT lt_read_booksuppl INTO DATA(ls_booking_suppl)
  GROUP BY ls_booking_suppl-travel_id INTO ls_travel_key.
  ASSIGN lt_read_travel[ KEY entity COMPONENTS travel_id = ls_travel_key ]
  TO <ls_travel>.
  LOOP AT GROUP ls_travel_key INTO DATA(ls_bookingsuppl_result)
    GROUP BY ls_bookingsuppl_result-currency_code INTO lv_curr.

```

```

total_suppl_price_by_trav_curr = 0.
LOOP AT GROUP lv_curr INTO DATA(ls_booking_suppl2).
  total_suppl_price_by_trav_curr += ls_booking_suppl2-price.
ENDLOOP.
IF lv_curr = <ls_travel>-currency_code.
  <ls_travel>-total_price += total_suppl_price_by_trav_curr.
ELSE.
* (3') Call procedure for currency conversion
-----
  /dmo/cl_flight_amdp=>convert_currency(
    EXPORTING
      iv_amount                      = total_suppl_price_by_trav_curr
      iv_currency_code_source        = lv_curr
      iv_currency_code_target        = <ls_travel>-currency_code
      iv_exchange_rate_date          =
  cl_abap_context_info=>get_system_date( )
    IMPORTING
      ev_amount                      = DATA(total_suppl_price_per_curr)
    .
    <ls_travel>-total_price += total_suppl_price_per_curr.
  ENDIF.
ENDLOOP.
* (4) Update the total_price value for the relevant travel instance
-----
MODIFY ENTITIES OF /dmo/i_travel_m
ENTITY travel
  UPDATE FROM VALUE #( FOR travel IN lt_read_travel (
    travel_id                     = travel-travel_id
    total_price                    = travel-total_price
    currency_code                  = travel-currency_code
    %control-total_price           = if_abap_behv=>mk-on ) ) .
ENDMETHOD.
ENDCLASS.

```

Procedure: Implementing the Determination on Booking Entity

The implementation of the determination method `calculate_total_flight_price` in the handler class `lhc_travel` of the corresponding class pool `/dmo/bp_booking_m` is now reduced to the method call `/dmo/cl_travel_auxiliary_m=>calculate_price()`.

i [Expand the following listing to view the source code.](#)

Listing 3: Implementation of calculate_total_flight_price Method

```

*****
*
* Calculates total booking price
*
*****
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
  ...
  METHODS calculate_total_flight_price FOR DETERMINATION
    booking~calculateTotalFlightPrice
      IMPORTING keys FOR booking.
  ENDCLASS.
  CLASS lhc_travel IMPLEMENTATION.
    METHOD calculate_total_flight_price.
      IF keys IS NOT INITIAL.
        /dmo/cl_travel_auxiliary_m=>calculate_price(
          it_travel_id = VALUE #(  FOR GROUPS <booking> OF booking_key IN keys
                                    GROUP BY booking_key-travel_id WITHOUT
MEMBERS

```

```

        ENDIF.
ENDMETHOD.
...
ENDCLASS.

```

Validation on Booking Supplement Entity

UI Preview

If the user adds a supplement to a given flight booking, then the travel amount is re-calculated.

Booking Number:	Flight Number:
1	1537

Booking Date:	Flight Date:
Jul 24, 2019	Jul 29, 2019

Customer ID:	Flight Price:
7	411.00 EUR

Airline ID:	Status [N(New) X(Canceled) B(Booked)]:
United Airlines, Inc. (UA)	N

Booking Supplement	
Book. Supp. Number	Product ID
1	Hamburg Salad with Fresh Shrimps (ML-0012)
	Product Price
	17.00 EUR

Added Supplement Changes the Travel Amount

The updated travel amount is displayed in as new value of Total Price.

Travel ID	Agency ID	Customer ID	Starting Date	End Date	Total Price	Booking Status
1	Sunshine Travel (70001)	Neubasler (71)	Jul 24, 2019	Aug 21, 2019	428.00 EUR	O

Updated Total Price of the Travel Instance

In addition to the price, the currency was also defined as another trigger field for determination. In this way, we want to ensure that currency conversion is also carried out when the total amount is re-calculated. In our case, the supplement price with the current currency ([USD](#)) is converted into the travel currency ([EUR](#)).

The screenshot shows a booking supplement entry screen. At the top, there's a search bar and a toolbar with icons. Below that is a table with columns: Book. Supp. Number, Product ID, and Product Price. The Product Price field contains '17.00' with a dropdown arrow, and a yellow box highlights the entire row. Below the table is a dark button bar with 'Save' and 'Cancel' buttons. In the main area, there's another table with columns: Travel ID, Agency ID, Customer ID, Starting Date, End Date, Total Price, and Booking Status. The Total Price column shows '429.08 EUR' with a yellow box highlighting it. An orange arrow points from the highlighted 'Product Price' field to the highlighted 'Total Price' field.

Book. Supp. Number	Product ID	Product Price
1	ML-0012	17.00 USD

Travel ID	Agency ID	Customer ID	Starting Date	End Date	Total Price	Booking Status
<input type="radio"/> 1	Sunshine Travel (70001)	Neubasler (71)	Jul 24, 2019	Aug 21, 2019	429.08 EUR	O

Currency Conversion of Supplement Price and Updated Total Price in EUR

Definition

In the behavior definition, the determination on the booking supplement entity is defined as follows:

```
define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
...
{
  ...
  determination calculateTotalSupplmPrice on modify { field price,
  currency_code; }
}
```

Procedure: Implementing the Determination on Booking Supplement Entity

The implementation of the determination method `calculate_total_price` in the handler class `lhc_travel` of the corresponding class pool `/dmo/bp_booking supplement_m` is now reduced to the method call `/dmo/cl_travel_auxiliary_m=>calculate_price()`.

i [Expand the following listing to view the source code.](#)

Listing 4: Implementation of `calculate_total_price` Method

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
  METHODS calculate_total_price FOR DETERMINATION
    booksuppl~calculateTotalSupplmPrice
      IMPORTING keys FOR booksuppl.
  ...
ENDCLASS.
CLASS lhc_travel IMPLEMENTATION.
*****{*}
* Calculates total total flight price - including the price of supplements
*
*****{*}
METHOD calculate_total_price.
  IF keys IS NOT INITIAL.
    /dmo/cl_travel_auxiliary_m=>calculate_price(
      it_travel_id = VALUE #( FOR GROUPS <booking_suppl> OF booksuppl_key
    IN keys
```

```
MEMBERS
      GROUP BY booksuppl_key-travel_id WITHOUT
      ( <booking_suppl> ) ) .
ENDIF.
ENDMETHOD.
ENDCLASS.
```

5.2.3.6 Integrating Additional Save in Managed Business Objects

This section explains how you can integrate Additional Save within the transactional life cycle of managed business objects.

Use Case

In some application scenarios, an external functionality must be invoked during the save sequence, after the managed runtime has written the changed data of business object's instances to the database but before the final commit work has been executed.

For example, reuse services like **change documents** and the **application log** must be triggered during the save sequence and the changes of the current transaction must be written to change requests.

In real-life business applications, the data of business objects may change frequently. It is often helpful, and sometime even necessary, to be able to trace or reconstruct changes for objects that are critical, for example for investigation or auditing purposes. The ABAP Application Server records changes to business data objects in change documents.

Application events can be centrally recorded in the application log. The entries of an application log contain information about who gave rise to a given event at what time and with which program.

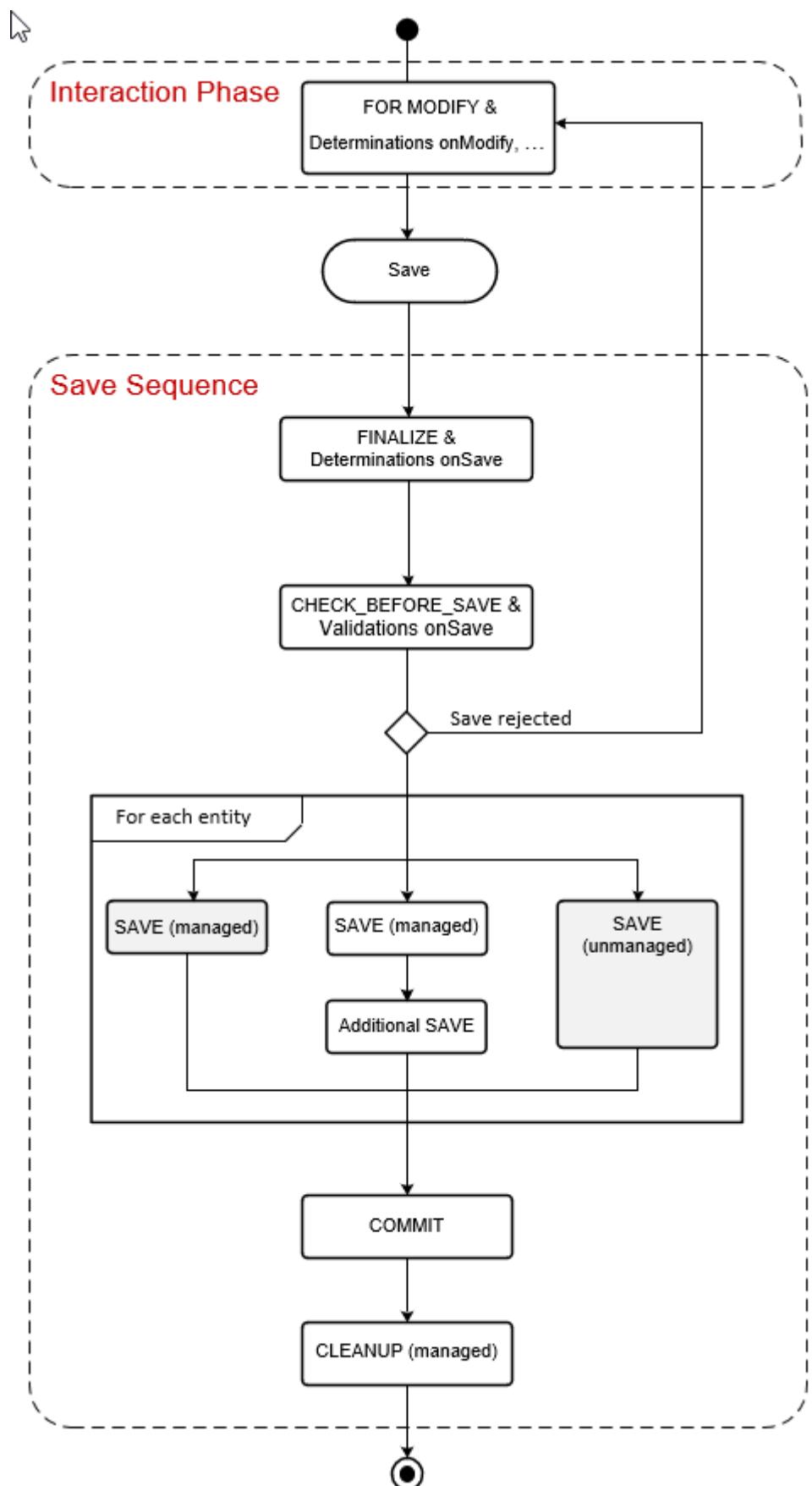
In order to integrate the additional save into the save sequence as a part of the managed runtime, you must first add the corresponding syntax to the behavior definition and then implement the saver handler method as a part of the behavior pool.

Note

If you would like to replace the managed runtime from saving the entity's data and reuse your own save logic instead, you can integrate the unmanaged save instead. **More on this:** [Integrating Unmanaged Save in Managed Business Objects \[page 227\]](#).

Additional Save Within the Transactional Life Cycle

The following figure depicts the additional save within the transactional life cycle of a managed business object.



Additional Save within the Transactional Processing

The save sequence is triggered for each business object after at least one successful modification (create, update, delete) was performed and saving data has been explicitly requested by the consumer. The save sequence starts with the FINALIZE processing step performing the final calculations and determinations before data changes can be persisted.

If the subsequent CHECK_BEFORE_SAVE call, including all onSave validations (validations with the [trigger time \[page 821\]](#) on save), is positive for all transactional changes, the point-of-no-return is reached. From now on, a successful save is guaranteed by all involved BOs.

If, on the other hand, the result of the checks is negative at the time of CHECK_BEFORE_SAVE, a save is denied and the save sequence is interrupted. The consumer has now the option of modifying business object data and then trigger the save sequence again.

After the **point-of-no-return**, the save call persists all BO instance data from the transactional buffer in the database.

For each entity of an individual business object, the following options are available to execute the SAVE processing step:

- Managed save (default)
- Managed save in conjunction with additional save
- Unmanaged save (to prevent the managed runtime from saving the entities data)

All change requests of the current [LUW \[page 815\]](#) are committed. The actual save execution is finished by COMMIT WORK.

The final CLEANUP clears all transactional buffers of all business objects involved in the transaction.

Activities Relevant to Developers

- [Defining Additional Save in the Behavior Definition \[page 219\]](#)
- [Implementing Additional Save \[page 221\]](#)

5.2.3.6.1 Defining Additional Save in the Behavior Definition

In this topic, you will learn the syntax for defining the additional save for managed business objects.

Syntax for Defining Additional Save for Each Individual Entity

In general, an additional save can be defined for each entity of a given business object with managed implementation type. This is done in the behavior definition of the business object by adding the keyword `with additional save` - after specifying the persistent table `DB_TABLE`.

The database table `DB_TABLE` is used in managed implementation type for storing entity's business data changes that result from the transactional life cycle.

The actual implementation of an additional save is based on the ABAP language and takes place in a local saver class as a part of the behavior pool. [More on this: Implementing Additional Save \[page 221\]](#)

```
[implementation] managed;
define behavior for Entity [alias AliasedName]
implementation in class ABAP_CLASS [unique]
persistent table DB_TABLE
with additional save
...
{
    ...
}
```

Short Syntax for Defining Additional Save for All Entities

The following compact notation for defining the additional save is useful as an alternative if you want to define an additional save for all entities of a business object and the saver implementation is carried out in a single behavior pool ABAP_CLASS. In this case, the keyword with additional save is already specified in the header of the business object's behavior definition.

```
[implementation] managed with additional save
implementation in class ABAP_CLASS [unique]
define behavior for Entity [alias AliasedName]
persistent table DB_Table
...
{
    ...
}
```

Example

In the following behavior definition, the additional save is defined for the travel (root) entity and the booksuppl child entity, whereas for the booking child entity, the (default) managed save is defined.

Listing: Behavior Definition with Additional Save

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
persistent table /DMO/TRAVEL_M
with additional save
...
{
    ...
}
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
persistent table /DMO/BOOKING_M
...
{
    ...
}
define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
```

```
persistent table /DMO/BOOKSUPPL_M  
with additional save  
...  
{  
    ...  
}
```

5.2.3.6.2 Implementing Additional Save

The additional save of the relevant business object's entity is implemented in the behavior pool (`ABAP_CLASS`) that is specified in the behavior definition by the keyword `implementation in class ABAP_CLASS [unique]`.

The implementation takes place in a local saver class as a part of the behavior pool. As depicted in the listing below, each such local class inherits from the base saver class `CL_ABAP_BEHAVIOR_SAVER`. This superclass provides the predefined method `save_modified` that needs to be redefined in the local saver class `lhcsaver`.

General Implementation Steps

The following listing provides a template with the main steps for implementing additional save within the `save_modified` method.

The essential elements of this method are the predefined, implicit parameters:

- `CREATE-EntityName`
- `UPDATE-EntityName`
- `DELETE-EntityName`

These parameters contain not only the table type of the entity to be modified, but also the `%control` structure that can be used for identifying which elements are requested by the consumer.

```

    IF create-booksuppl IS NOT INITIAL.
      lt_booksuppl_"
      CALL FUNCTION
    ENDIF.

    " (2) Get insta
    IF update-books
      lt_booksuppl_
      " Read all fi
      SELECT * FROM
        WHERE
        INTO
      " Take over f
      LOOP AT updat
      ASSIGN lt_b

```

booksuppl	type standard table of
travel_id	type /dmo/travel_id
booking_id	type /dmo/booking_id
booking_supplement_id	type /dmo/booking_supplement_id
supplement_id	type /dmo/supplement_id
price	type /dmo/supplement_price
currency_code	type /dmo/currency_code
last_changed_at	type timestamppl
%control	
travel_id	type abp_behv_flag
booking_id	type abp_behv_flag
booking_supplement_id	type abp_behv_flag
supplement_id	type abp_behv_flag
price	type abp_behv_flag
currency_code	type abp_behv_flag
last_changed_at	type abp_behv_flag

Accessing Element Information for the Implicit Parameter Type (F2)

Listing: Template for Implementing Additional Save

```

CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS save_modified REDEFINITION.
  ENDCLASS.
CLASS lcl_saver IMPLEMENTATION.
  METHOD save_modified.
    IF CREATE-EntityName IS NOT INITIAL.
      " Provide table of instance data of all instances that have been created
      during current transaction
      " Use %CONTROL to get information on what entity fields have been set when
      creating the instance
    ENDIF.
    IF UPDATE-EntityName IS NOT INITIAL.
      " Provide table of instance data of all instances that have been updated
      during current transaction
      " Use %CONTROL to get information on what entity fields have been updated
    ENDIF.
    IF DELETE-EntityName IS NOT INITIAL.
      " Provide table with keys of all instances that have been deleted during
      current transaction
      " NOTE: There is no information on fields when deleting instances
    ENDIF.
  ENDMETHOD.
  ...
ENDCLASS.

```

Example

The following example shows you in detail how you can implement additional save based on our [travel demo scenario \[page 774\]](#).

Since in the current release of SAP CP, ABAP Environment, the reuse services like **change documents** and the **application log** are not yet available, we will demonstrate a simplified example for integrating the for additional

save. In this example, all essential changes to the root instance of the travel business object should be recorded in a log table. This is defined in such a way that it can hold different travel instance data and contains the following fields:

Log Table

Log Table Field	Description
change_id	Identifier for an individual change
travel_id	Key field of the travel entity
changing_operation	Standard operation for travel instances: create, update and delete
changed_field_name	Name of the field that has been created or changed
changed_value	Value of a created or changed entity field
created_at	Date and time of instance data creation, update or deletion

The following listing shows you the definition of the corresponding table /DMO/LOG_TRAVEL in the table editor.

Listing: Source code with table definition

```
@EndUserText.label : 'flight reference scenario: log changes to travel entity'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #LIMITED
define table /dmo/log_travel {
    key client          : abap.clnt not null;
    key change_id       : abap.raw(16);
    travel_id          : /dmo/travel_id not null;
    changing_operation : abap.char(10);
    changed_field_name : abap.char(32);
    changed_value      : abap.char(32);
    created_at         : timestamppl;
}
```

UI Preview

Travel ID [1,...,99999999]:
1

*Booking Fee:
11.00 EUR

*Agency ID:
70001

*Customer ID:
1

Total Price:
19,869.20 EUR

*Status [O(Open)|A(Accepted)|X(Canceled)]:
O

Description:
Changed description

Starting Date:
Oct 7, 2019

End Date:
Oct 14, 2019

changed fields

Save Cancel

Consumer modifies business data on Fiori UI

CHANGING_OP...	CHANGED_FI...	CHANGED_VAL...	CREATED_AT
UPDATE	customer_id	000001	20191011143741....
UPDATE	description	Changed descrip	20191011143741....

After successful save, the relevant entries are recorded in the log table

Definition

In the behavior definition, the additional save for the root entity may be defined as follows:

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
persistent table /DMO/TRAVEL_M
with additional save
...
{
  ...
}
```

Implementation

The source code of our example implementation is divided into three sections:

Each of these sections is initiated with an IF statement, each of which checks whether

- the travel instances have been created (IF create-travel IS NOT INITIAL),
- their elements have been modified (IF update-travel IS NOT INITIAL), or

- deleted (IF delete-travel IS NOT INITIAL) by a consumer.

The relevant instance data is written to the internal table `lt_travel_log` and passed to the log table for persistent storage on the database (`INSERT /dmo/log_travel...`).

When creating new travel instances and updating instance data, the `%control` structure is used to get information on what fields have been provided or updated by the consumer. The `%control` structure contains a flag `if_abap_behv=>mk-on` for each field, which indicates whether the field was provided (or changed) by the consumer or not.

When creating instances (1), the new values for relevant fields of the travel entity are written into the internal table `lt_travel_log_c`. In our demo code, we select the two fields `booking_fee` and `overall_status` as an example. Their values are transferred as separate rows by means of `APPEND` into `lt_travel_log_c` and finally written into the log table `/dmo/log_travel` with `INSERT`.

Similarly, in the update case (2), we also select two fields, namely `customer_id` and `description`, as relevant fields for recording. So whenever the value of one of these fields is changed for an existing travel instance by a consumer, a new table row (with the corresponding change ID) is appended to the internal table `lt_travel_log_c`.

The last section (3) deals with the deletion of travel instances. However, in this case we are only interested in the information of which instances have been deleted. Therefore, there is no information of fields available when deleting instances.

Listing: The method `save_modified` implements the additional save

```

CLASS lcl_save DEFINITION INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS save_modified REDEFINITION.
  ENDCLASS.

CLASS lcl_save IMPLEMENTATION.
  METHOD save_modified.
    DATA lt_travel_log    TYPE STANDARD TABLE OF /dmo/log_travel.
    DATA lt_travel_log_c TYPE STANDARD TABLE OF /dmo/log_travel.
    DATA lt_travel_log_u TYPE STANDARD TABLE OF /dmo/log_travel.
    " (1) Get instance data of all instances that have been created
    IF create-travel IS NOT INITIAL.
      " Creates internal table with instance data
      lt_travel_log = CORRESPONDING #( create-travel ).
      LOOP AT lt_travel_log ASSIGNING FIELD-SYMBOL(<fs_travel_log_c>).
        <fs_travel_log_c>-changing_operation = 'CREATE'.
        " Generate time stamp
        GET TIME STAMP FIELD <fs_travel_log_c>-created_at.
        " Read travel instance data into ls_travel that includes %control
        structure
        READ TABLE create-travel WITH TABLE KEY entity COMPONENTS travel_id =
        <fs_travel_log_c>-travel_id INTO DATA(ls_travel).
        IF sy-subrc = 0.
          " If new value of the booking_fee field created
          IF ls_travel-%control-booking_fee = cl_abap_behv=>flag_changed.
            " Generate uuid as value of the change_id field
            TRY.
              <fs_travel_log_c>-change_id =
              cl_system_uuid=>create_uuid_x16_static( ) .
              CATCH cx_uuid_error.
                "handle exception
              ENDTRY.
              <fs_travel_log_c>-changed_field_name = 'booking_fee'.
              <fs_travel_log_c>-changed_value = ls_travel-booking_fee.
              APPEND <fs_travel_log_c> TO lt_travel_log_c.
            ENDIF.
        ENDIF.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

```

```

    " If new value of the overall status field created
    IF ls_travel->%control-overall_status = cl_abap_behv=>flag_changed.
        " Generate uuid as value of the change_id field
        TRY.
            <fs_travel_log_c>-change_id =
cl_system_uuid=>create_uuid_x16_static( ) .
            CATCH cx_uuid_error.
                "handle exception
            ENDTRY.
            <fs_travel_log_c>-changed_field_name = 'overall_status'.
            <fs_travel_log_c>-changed_value = ls_travel-overall_status.
            APPEND <fs_travel_log_c> TO lt_travel_log_c.
        ENDIF.
        " IF ls_travel->%control-...
    ENDIF.
    ENDLOOP.
    " Inserts rows specified in lt_travel_log into the DB table /dmo/log_travel
    INSERT /dmo/log_travel FROM TABLE @lt_travel_log_c.
ENDIF.
" (2) Get instance data of all instances that have been updated during the
transaction
IF update-travel IS NOT INITIAL.
    lt_travel_log = CORRESPONDING #( update-travel ).
    LOOP AT update-travel ASSIGNING FIELD-SYMBOL(<fs_travel_log_u>).
        ASSIGN lt_travel_log[ travel_id = <fs_travel_log_u>-travel_id ] TO FIELD-
SYMBOL(<fs_travel_db>).
        <fs_travel_db>-changing_operation = 'UPDATE'.
        " Generate time stamp
        GET TIME STAMP FIELD <fs_travel_db>-created_at.
        IF <fs_travel_log_u>-%control-customer_id = if_abap_behv=>mk-on.
            <fs_travel_db>-changed_value = <fs_travel_log_u>-customer_id.
            " Generate uuid as value of the change_id field
            TRY.
                <fs_travel_db>-change_id =
cl_system_uuid=>create_uuid_x16_static( ) .
                CATCH cx_uuid_error.
                    "handle exception
                ENDTRY.
                <fs_travel_db>-changed_field_name = 'customer_id'.
                APPEND <fs_travel_db> TO lt_travel_log_u.
            ENDIF.
            IF <fs_travel_log_u>-%control-description = if_abap_behv=>mk-on.
                <fs_travel_db>-changed_value = <fs_travel_log_u>-description.
                " Generate uuid as value of the change_id field
                TRY.
                    <fs_travel_db>-change_id =
cl_system_uuid=>create_uuid_x16_static( ) .
                    CATCH cx_uuid_error.
                        "handle exception
                    ENDTRY.
                    <fs_travel_db>-changed_field_name = 'description'.
                    APPEND <fs_travel_db> TO lt_travel_log_u.
                ENDIF.
                "IF <fs_travel_log_u>-%control-...
            ENDLOOP.
            " Inserts rows specified in lt_travel_log into the DB table /dmo/log_travel
            INSERT /dmo/log_travel FROM TABLE @lt_travel_log_u.
        ENDIF.
        " (3) Get keys of all travel instances that have been deleted during the
transaction
        IF delete-travel IS NOT INITIAL.
            lt_travel_log = CORRESPONDING #( delete-travel ).
            LOOP AT lt_travel_log ASSIGNING FIELD-SYMBOL(<fs_travel_log_d>).
                <fs_travel_log_d>-changing_operation = 'DELETE'.
                " Generate time stamp
                GET TIME STAMP FIELD <fs_travel_log_d>-created_at.
                " Generate uuid as value of the change_id field

```

```

TRY.
  <fs_travel_log_d>-change_id =
cl_system_uuid=>create_uuid_x16_static( ) .
  CATCH cx_uuid_error.
    "handle exception
ENDTRY.
ENDLOOP.
" Inserts rows specified in lt_travel_log into the DB table /dmo/log_travel
INSERT /dmo/log_travel FROM TABLE @lt_travel_log.
ENDIF.
ENDMETHOD.
ENDCLASS.

```

5.2.3.7 Integrating Unmanaged Save in Managed Business Objects

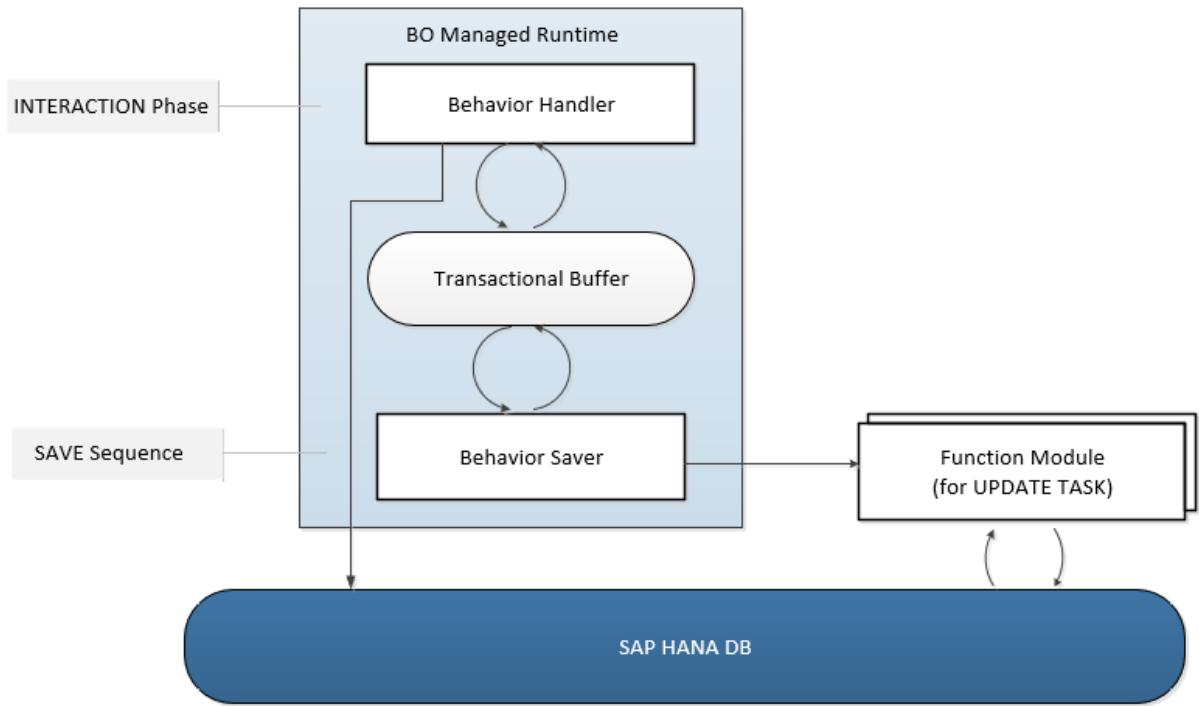
This section explains how you can integrate unmanaged save within the transactional life cycle of managed business objects.

Use Case

In certain use cases you might be requested to prevent business object's managed runtime from saving business data (changes). By default, the managed runtime saves all changed instances of business object's entity in the database table that is specified as `persistent table DB_TABLE` in the behavior definition (managed save). However, you define for each entity of the business object or for the entire business object whether the complete save is done by the managed runtime or by the unmanaged save instead. This implementation flavor of a managed scenario may be relevant to you if you need to implement the interaction phase for your application anyway, but the update task function modules are already available.

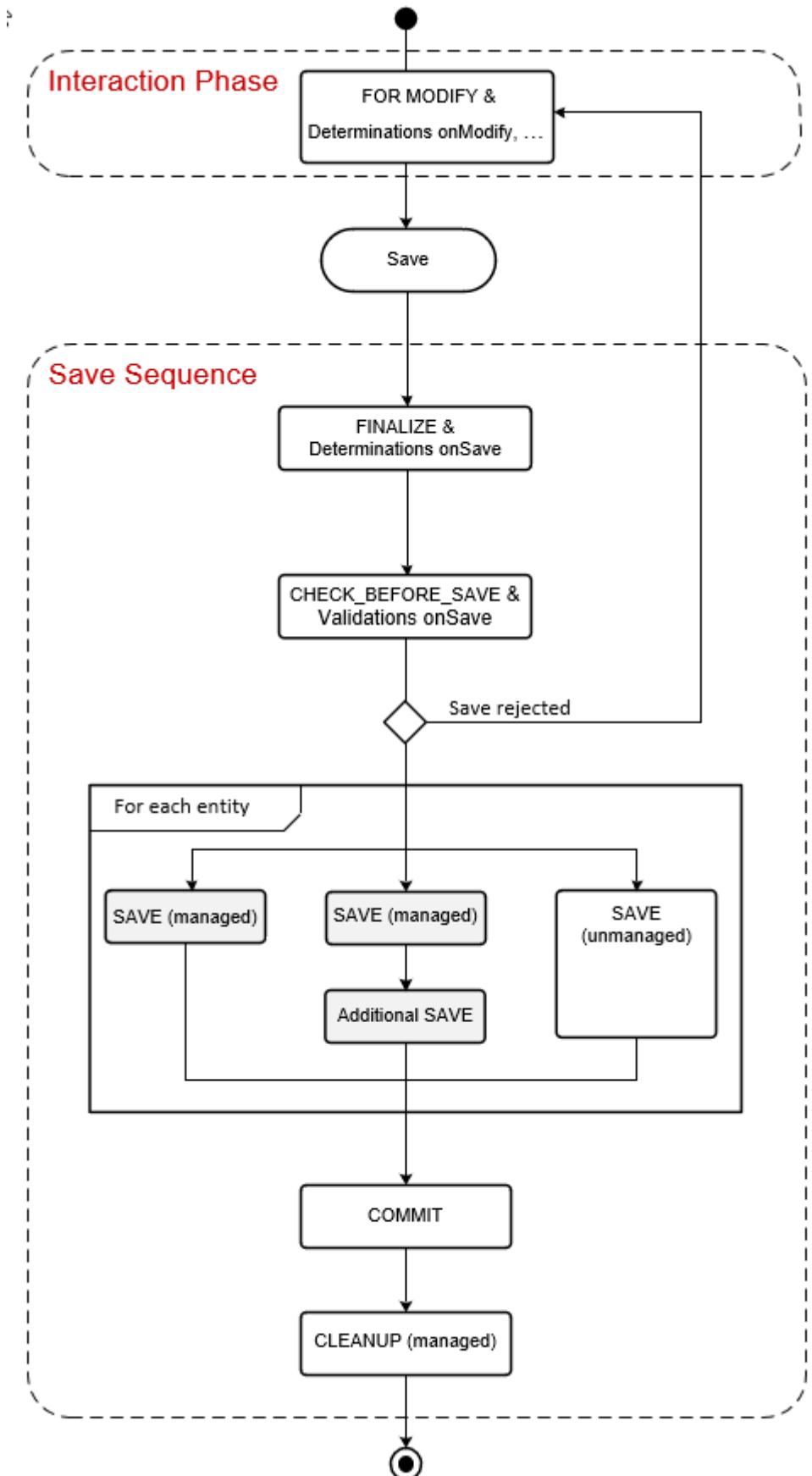
The following figure outlines the main components of business objects managed runtime that integrates function modules for persistent save of business data changes. Within the interaction phase, a consumer calls the business object operations to change business data and read instances with or without the transactional changes. The business object runtime keeps the changes in its internal transactional buffer which represents the state of instance data. After all changes on the related entity were performed, the instance data can be persisted. This is realized during the save sequence. To prevent the managed runtime from saving the data, the function modules (for the update task) are called to save data changes of the relevant business object's entity (unmanaged save). In order to persist the business data changes, the function modules access the corresponding tables of the HANA database.

Note that the behavior handler can also directly access table data from the database during the interaction phase: Authorization checks, for example, require direct access to the table data on the database.



Unmanaged Save in Transactional Life Cycle

The following figure depicts the unmanaged save within the transactional life cycle of a managed business object.



Unmanaged Save within the Transactional Processing

The save sequence is triggered for each business object after at least one successful modification (create, update, delete) was performed and saving data has been explicitly requested by the consumer. The save sequence starts with the FINALIZE processing step performing the final calculations and determinations before data changes can be persisted.

If the subsequent CHECK_BEFORE_SAVE call, including all onSave validations (validations with the [trigger time \[page 821\]](#) on save), is positive for all transactional changes, the point-of-no-return is reached. From now on, a successful save is guaranteed by all involved BOs.

If, on the other hand, the result of the checks is negative at the time of CHECK_BEFORE_SAVE, a save is denied and the save sequence is interrupted. The consumer has now the option of modifying business object data and then trigger the save sequence again.

After the **point-of-no-return**, the save call persists all BO instance data from the transactional buffer in the database.

For each entity of an individual business object, the following options are available to execute the SAVE processing step:

- Managed save (default)
- Managed save in conjunction with additional save
- Unmanaged save (to prevent the managed runtime from saving the entities data)

All change requests of the current [LUW \[page 815\]](#) are committed. The actual save execution is finished by COMMIT WORK.

The final CLEANUP clears all transactional buffers of all business objects involved in the transaction.

Activities Relevant to Developers

In order to integrate unmanaged save into the save sequence as a part of the managed runtime, you must first add the corresponding syntax to the behavior definition and then implement the saver handler method as a part of the behavior pool.

- [Defining Unmanaged Save in the Behavior Definition \[page 230\]](#)
- [Implementing the Save Handler Method \[page 232\]](#)

5.2.3.7.1 Defining Unmanaged Save in the Behavior Definition

In this topic, you will learn the syntax for defining unmanaged save for managed business objects.

Syntax for Defining Unmanaged Save for Each Individual Entity

In general, unmanaged save can be defined for each entity of a given business object with managed implementation type. This is done in the behavior definition of the business object by adding the keyword `with`

unmanaged save. Note that persistent table DB_TABLE does not apply in unmanaged save. In this case, a function module in update task is used for storing entity's business data changes that result from the transactional life cycle.

The actual implementation of the unmanaged save is based on the ABAP language and takes place in a local saver class as a part of the behavior pool. **More on this:** [Implementing the Save Handler Method \[page 232\]](#)

```
[implementation] managed;
define behavior for Entity [alias AliasedName]
implementation in class ABAP_CLASS [unique]
with unmanaged save
...
{
    ...
}
```

Short Syntax for Defining Unmanaged Save for All Entities

The following compact notation for defining the unmanaged save is useful as an alternative if you want to define an unmanaged save for all entities of a business object and the saver implementation is carried out in a single behavior pool ABAP_CLASS. In this case, the keyword with unmanaged save is already specified in the header of the business object's behavior definition.

```
[implementation] managed with unmanaged save
implementation in class ABAP_CLASS [unique]
define behavior for Entity [alias AliasedName]
...
{
    ...
}
```

Example

In the following behavior definition, the unmanaged save is defined for the travel (root) entity, whereas for the child entity booking the (default) managed save and for the child entity booksuppl an additional save is defined.

Listing: Behavior Definition with Unmanaged Save

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
with unmanaged save
...
{
    ...
}
define behavior for /DMO/I_Booking_M alias booking
implementation in class /DMO/BP_BOOKING_M unique
persistent table /DMO/BOOKING_M
...
{
```

```

    ...
}

define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
persistent table /DMO/BOOKSUPPL_M
with additional save
...
{
    ...
}

```

5.2.3.7.2 Implementing the Save Handler Method

The unmanaged save of the relevant business object's entity is implemented in the behavior pool (ABAP_CLASS) that is specified in the behavior definition by the keyword `implementation in class ABAP_CLASS [unique]`.

The implementation takes place in a local saver class as a part of the behavior pool. As depicted in the listing below, each such local class inherits from the base saver class `CL_ABAP_BEHAVIOR_SAVER`. This superclass provides the predefined method `save_modified` that needs to be redefined in the local saver class `lhcsaver`.

→ Remember

Convention: The local saver class that implements the `save_modified` method is either a separate global class or a part of the root implementation (behavior pool for the root entity).

General Implementation Steps

The following listing provides a template with the main steps for implementing unmanaged save within the `save_modified` method.

The essential elements of this method are the predefined, implicit parameters:

- CREATE-EntityName
- UPDATE-EntityName
- DELETE-EntityName

These parameters contain not only the table type of the entity to be modified, but also the [%control \[page 741\]](#) structure that can be used for identifying which elements have been changed during the current transaction..

```

IF create-booksuppl IS NOT INITIAL.
  lt_booksuppl_"
    CALL FUNCTION
      booksuppl type standard table of
        travel_id          type /dmo/travel_id
        booking_id         type /dmo/booking_id
        booking_supplement_id type /dmo/booking_supplement_id
        supplement_id      type /dmo/supplement_id
        price              type /dmo/supplement_price
        currency_code      type /dmo/currency_code
        last_changed_at    type timestamp
      %control
        travel_id          type abp_behv_flag
        booking_id         type abp_behv_flag
        booking_supplement_id type abp_behv_flag
        supplement_id      type abp_behv_flag
        price              type abp_behv_flag
        currency_code      type abp_behv_flag
        last_changed_at    type abp_behv_flag
  ENDIF.

  " (2) Get instance
  IF update-books
    lt_booksuppl_
    " Read all fields
    SELECT * FROM
      WHERE
      INTO
  " Take over fields
  LOOP AT update-books
    ASSIGN lt_b

```

Accessing Element Information for the Implicit Parameter Type (F2)

The actual implementation steps of the `save_modified` method are shown in the template below:

Listing: Template for Implementing Unmanaged Save

```

CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS save_modified REDEFINITION.
  ENDCLASS.

CLASS lcl_saver IMPLEMENTATION.
  METHOD save_modified.
    IF CREATE-EntityName IS NOT INITIAL.
      " Provide table of instance data of all instances that have been created
      during current transaction
      " Use %CONTROL to get information on what entity fields have been set or
      updated during the current transaction
    ENDIF.
    IF UPDATE-EntityName IS NOT INITIAL.
      " Provide table of instance data of all instances that have been updated
      during current transaction
      " Use %CONTROL to get information on what entity fields have been updated
    ENDIF.
    IF DELETE-EntityName IS NOT INITIAL.
      " Provide table with keys of all instances that have been deleted during
      current transaction
      " NOTE: There is no information on fields when deleting instances
    ENDIF.
  ENDMETHOD.
  ...
ENDCLASS.

```

Example

The following example shows you in detail how you can implement unmanaged save specifically based on our [travel demo scenario \[page 774\]](#). In particular, the previous implementation must be extended in such a way

that the available function modules are used to save changes to business data of the booking supplement child entity.

Function Modules

In our example, the corresponding function modules for creating, changing, and deleting instances of the booking supplement entity are already available in the corresponding function group of the demo package /DMO/FLIGHT_MANAGED.

The screenshot shows the SAP Fiori Launchpad. The path selected is /DMO/FLIGHT > /DMO/FLIGHT_MANAGED > Function Groups > /DMO/TRAVEL_UPDATE_TASK > Function Modules. The visible function modules are /DMO/FLIGHT_BOOKSUPPL_C, /DMO/FLIGHT_BOOKSUPPL_D, and /DMO/FLIGHT_BOOKSUPPL_U. Each module is described with its purpose: Create for booking supplements, Delete for booking supplements, and Update for booking supplements respectively.

- ▼ /DMO/FLIGHT <partly loaded> (?) *Flight Reference Scenario*
- ▼ /DMO/FLIGHT_MANAGED <partly loaded> (?) *Flight Reference Scenario: TX managed E2E Guide*
- ▼ Source Code Library <partly loaded> (?)
- > Classes <partly loaded> (?)
- ▼ Function Groups (1)
 - ▼ /DMO/TRAVEL_UPDATE_TASK *Flight update task*
 - ▼ Function Modules
 - /DMO/FLIGHT_BOOKSUPPL_C *Create for booking supplements*
 - /DMO/FLIGHT_BOOKSUPPL_D *Delete for booking supplements*
 - /DMO/FLIGHT_BOOKSUPPL_U *Update for booking supplements*

Available Function Modules

Listing: Function module /DMO/FLIGHT_BOOKSUPPL_C

```
FUNCTION /dmo/flight_booksuppl_c
  IMPORTING
    VALUE(values) TYPE /dmo/tt_booksuppl_m.
  INSERT /dmo/booksuppl_m FROM TABLE @values.
ENDFUNCTION.
```

i Note

To use this source code, the table type /dmo/tt_booksuppl_m is also required for the values importing parameter.

The screenshot shows the SAP Dictionary Tool interface for defining a table type. The top navigation bar indicates the table type is /DMO/TT_BOOKSUPPL_M. The main area is divided into several sections:

- Row Type**: Contains fields for Category (Dictionary Type), Type Name (/DMO/BOOKSUPPL_M), Initial Number of Rows (0), and Access (Standard Table).
- Initialization and Access**: This section is partially visible on the right.
- Key Overview**: A section for defining primary and secondary keys. It contains icons for creating and deleting keys, and a tree view where <Primary Key> is selected.
- Primary Key Details**: A section for specifying key attributes. It includes fields for Key Definition (* Standard Key), Key Category (* Non-Unique), and Alias (empty).

Definition of Table Type /DMO/TT_BOOKSUPPL_M in the Dictionary Tool

The following listing provides you with the function module's source code for persistent storage of individual elements of existing booking supplement instances.

Listing: Function module /DMO/FLIGHT_BOOKSUPPL_U

```
FUNCTION /dmo/flight_booksuppl_u
  IMPORTING
    VALUE(values) TYPE /dmo/tt_booksuppl_m.
    UPDATE /dmo/booksuppl_m FROM TABLE @values.
ENDFUNCTION.
```

The following listing provides you with the source code of the function module for deleting booking supplement instances.

Listing: Function module /DMO/FLIGHT_BOOKSUPPL_D

```
FUNCTION /dmo/flight_booksuppl_d
  IMPORTING
    VALUE(values) TYPE /dmo/tt_booksuppl_m.
    DELETE /dmo/booksuppl_m FROM TABLE @values.
ENDFUNCTION.
```

Definition

In the behavior definition, the unmanaged save for the child entity booksuppl may be defined as follows:

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
...
{
  ...
}
```

```

}

...
define behavior for /DMO/I_BookSuppl_M alias booksuppl
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_M unique
with unmanaged save
...
{
    ...
}

```

Implementation

As shown in the listing below, the source code of our example implementation is divided into three sections:

Each of these sections is initiated with an `IF` statement, each of which checks whether the booking supplement instances have been created, their elements have been modified, or deleted by a consumer. The relevant instance data is written to the internal table `lt_booksuppl_db` and passed to the respective function module for persistent storage on the database.

In case of updating instance data (`IF update-booksuppl IS NOT INITIAL`), the `%control` structure is used to get information on what fields have been updated by the consumer. The `%control` structure contains a flag `if_abap_behv=>mk-on` for each field, which indicates whether the field was provided (changed) by the consumer or not.

To eliminate the option that the unchanged fields are overwritten with default values, we must ensure that they are kept according to the database data. This is the reason why we read the current data set from the database using the statement `SELECT * FROM /dmo/booksuppl_m FOR ALL ENTRIES IN @lt_booksuppl_db...`

Listing: The method `save_modified` implements the unmanaged save

```

CLASS lcl_save DEFINITION INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS save_modified REDEFINITION.
  ENDCLASS.

CLASS lcl_save IMPLEMENTATION.
  METHOD save_modified.
    DATA lt_booksuppl_db TYPE STANDARD TABLE OF /DMO/BOOKSUPPL_M.
    " (1) Get instance data of all instances that have been created
    IF create-booksuppl IS NOT INITIAL.
      lt_booksuppl_db = CORRESPONDING #( create-booksuppl ).
      CALL FUNCTION '/DMO/FLIGHT_BOOKSUPPL_C' EXPORTING values =
    lt_booksuppl_db .
    ENDIF.
    " (2) Get instance data of all instances that have been updated during the
    transaction
    IF update-booksuppl IS NOT INITIAL.
      lt_booksuppl_db = CORRESPONDING #( update-booksuppl ).
      " Read all field values from database
      SELECT * FROM /dmo/booksuppl_m FOR ALL ENTRIES IN @lt_booksuppl_db
        WHERE booking_supplement_id = @lt_booksuppl_db-
          booking_supplement_id
          INTO TABLE @lt_booksuppl_db .
      " Take over field values that have been changed during the transaction
      LOOP AT update-booksuppl ASSIGNING FIELD-SYMBOL(<ls_unmanaged_booksuppl>).
        ASSIGN lt_booksuppl_db[ travel_id = <ls_unmanaged_booksuppl>-travel_id
          booking_id = <ls_unmanaged_booksuppl>-booking_id
          booking_supplement_id = <ls_unmanaged_booksuppl>-
            booking_supplement_id
          ] TO FIELD-SYMBOL(<ls_booksuppl_db>).
        IF <ls_unmanaged_booksuppl>-%control-supplement_id = if_abap_behv=>mk-on.

```

```

        <ls_booksuppl_db>-supplement_id = <ls_unmanaged_booksuppl>-
supplement_id.
ENDIF.
IF <ls_unmanaged_booksuppl>-%control-price = if_abap_behv=>mk-on.
<ls_booksuppl_db>-price = <ls_unmanaged_booksuppl>-price.
ENDIF.
IF <ls_unmanaged_booksuppl>-%control-currency_code = if_abap_behv=>mk-on.
<ls_booksuppl_db>-currency_code = <ls_unmanaged_booksuppl>-
currency_code.
ENDIF.
ENDLOOP.
" Update the complete instance data
CALL FUNCTION '/DMO/FLIGHT_BOOKSUPPL_U' EXPORTING values =
lt_booksuppl_db .
ENDIF.
" (3) Get keys of all travel instances that have been deleted during the
transaction
IF delete-booksuppl IS NOT INITIAL.
lt_booksuppl_db = CORRESPONDING #( delete-booksuppl ).
CALL FUNCTION '/DMO/FLIGHT_BOOKSUPPL_D' EXPORTING values =
lt_booksuppl_db .
ENDIF.
ENDMETHOD.
ENDCLASS.
```

5.2.4 Developing a Projection Layer for Flexible Service Consumption

For a more flexible service consumption, every transactional business object is projected onto a service specific context. In other words, only those elements of the data model and those behavior characteristics and operations that are needed in the relevant business service context are exposed for the service. By means of projections, you can expose one BO in different business contexts by using different BO subsets. The general business logic is defined in the BO whereas the BO projection adopts a subset of the business logic.

A layering with projections enables robust application programming. You can change or enhance the BO without changing the exposed service as the scope of the service is defined in the projection layer. Enhancing the business object with additional structure or behavior does not have any effect on the resulting service.

Projection Layers in the Travel Business Scenario

The business object that you developed with the help of the previous sections is ready to run. It uses the managed runtime for CRUD (create, read, update, delete) operations. In addition, the relevant business logic for managing travels with action, determinations and validations was implemented. The BO CDS entities expose every element that might be relevant for any business service scenario. The behavior is defined and implemented for any kind of business service.

This demo scenario uses two projections with a different service scope. The resulting apps represent two role-based approaches for managing travels:

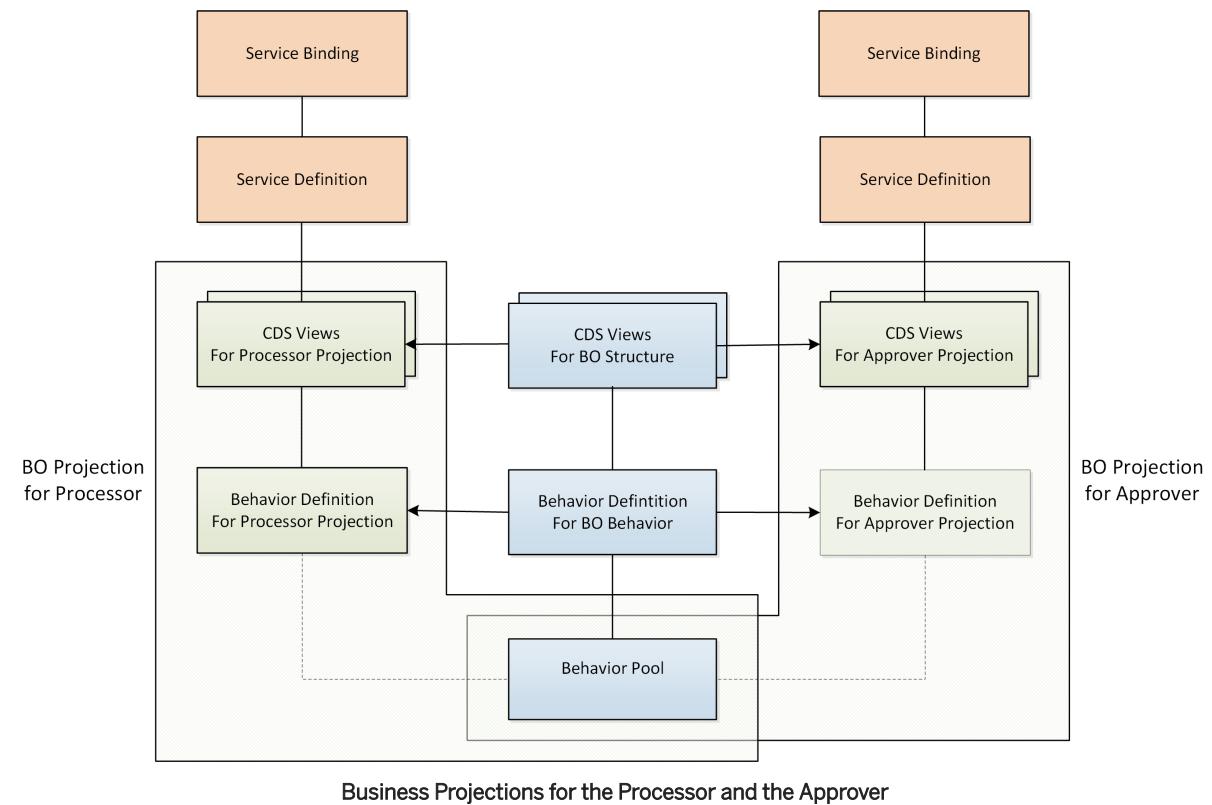
One business object projection uses the BO characteristics and the operations that are relevant for processing travel data. This resulting UI service serves the role of a data processor. The responsible person can enter the

information about travels, bookings and booking supplements into the role-based app for processing travel data. This person needs functionality to create, update and delete entries on all three tiers of the business object. In addition, the action `CreateByTravelTemplate` is designed to facilitate the creation of new travel entries. The instance-bound action reads the values of the selected entry and creates a new entry based on these values with some deviations for the `TravelID` and the travel dates.

The other business object projection is the basis for a UI service that contains the functionality that is relevant for an approver. Imagine a person, maybe a manager of a travel agency, that approves the data that was entered by the processor. That means, this person sees the travel information and the corresponding booking data. Based on this data, the approver can either accept or reject the travel. For a minimal scope of fields in the travel entity the approver is enabled to edit the values, for example the `BookingFee` or the `Description`. The information about bookings is set to read-only for the approver. The approver is not allowed to change the fields of the booking entity.

See [Business Scenario \[page 151\]](#) for a detailed list of features for the respective roles.

The design time artifacts that you need for these projection scenarios are illustrated in the following figure. The CDS views, as well as the behavior definition must be projected for both roles. To expose the BO projections for a UI service, you need to create a service definition and binding for both BO projections. The behavior implementation is not projected. Every behavior characteristic or operation that is used in the BO projection must be implemented in the underlying BO implementation. You cannot define new behavior that needs implementation in the BO projection.



Activities Relevant to Developers

1. [Providing a Data Model for Projections \[page 239\]](#)
 1. [Projection Views for the Processor BO Projection \[page 240\]](#)

2. [Projection Views for the Approver BO Projection \[page 252\]](#)
2. [Providing Behavior for Projections \[page 258\]](#)
 1. [Behavior for the Processor BO Projection \[page 259\]](#)
 2. [Behavior for the Approver BO Projection \[page 261\]](#)
3. [Defining Business Services Based on Projections \[page 262\]](#)

Related Information

[Business Object Projection \[page 100\]](#)

5.2.4.1 Providing a Data Model for Projections

The data model for the BO projection is defined in CDS projection views. Projection views are data definition artifacts with a syntax that differs slightly from CDS views. With projection views, you define the consumption-specific data model.

Syntax: CDS Projection View

To define a CDS projection view, the following syntax is used:

```

@EndUserText.label: 'EndUserText'
@AccessControl.authorizationCheck: #VALUE
[@view_anno]
/* Definition of projection view */
[define] [root] view entity ProjectionViewName
/* Defines the data source for the projection.*/
as projection on ProjectedEntity [as ProjectedEntityAlias ]
/* New read-only associations
association [min..max] to TargetEntity [as _Alias] on OnCondition
/* Subset of elements from the projected entity */
{
    /* Fields from the projected entity*/
    [@element_annot]
    ELEM1NAME [as ELEM1Alias] ,
    /*Localized element */
    [ [@element_annot]
        Assoc.Element2 [as ELEM2Alias] : localized , ]
    /* Cast element */
    [ [@element_annot]
        cast ELEM3NAME : {DataElement | ABAPType } [as ELEM3Alias] , ]
    [ [@element_annot]
        virtual ELEM4NAME : {DataElement | ABAPType } , ]
    /* Associations from the projected entity with possible redirections */
    [ _Association : [redirected to ProjectionViewTarget], ]
    /* Redirected compositions */
    [ _Composition : redirected to composition child ChildProjectionView, ]
    /* Redirected association to parent */
    [ _ParentAssoc : redirected to parent ParentProjectionView ]
}
[WHERE [NOT] Condition ]

```

For a detailed explanation of the syntax, see [CDS Projection View \[page 103\]](#).

Data Model in the Travel Scenario

For our travel scenario, the data models for the two projections have to be defined. For the processor BO, all three entities of the underlying BO are projected; the approver BO only uses the travel entity and the booking entity. All elements are aliased as an automatic mapping is provided for the elements in the projection views. For both projections, we use all elements from the underlying CDS views and the associations that are defined in the projected entity. For the processor BO, the only language-dependent text element in the booking supplement entity must be localized to get the description in the relevant language.

For both BO projections, the compositions have to be redirected.

UI Specifics for the Travel Scenario

Since the projection layer is the first service-specific layer, all UI specification must be defined in the CDS projection views. In the travel scenario, the following UI specifics are relevant on the projection layer:

- UI annotations defining position, labels, and facets of UI elements
- Search Enablement
- Text elements (language dependent and independent)
- Value Helps

These features have to be defined via annotations in the projection views.

The following sections provide a detailed description on how to project the existing BO to define a data model for one business object that is tailored to expose a UI service for a data processor and one that is tailored for a data approver.

- [Projection Views for the Processor BO Projection \[page 240\]](#)
- [Projection Views for the Approver BO Projection \[page 252\]](#)

Related Information

[CDS Projection View \[page 103\]](#)

5.2.4.1.1 Projection Views for the Processor BO Projection

To define a data model for the BO projection that defines the scope for the processor application, the following tasks need to be done:

- [Creating the Projection CDS Views for the Processor \[page 241\]](#)
- [Defining the Data Model for the Processor Projection Views \[page 241\]](#)

5.2.4.1.1.1 Creating the Projection CDS Views for the Processor

A data processor needs to be able to create, update, and delete entries for the travel entity, the booking entity, and the booking supplement entity. That means, all three nodes of the composition structure must be projected.

For the following CDS views, create the corresponding projection views by choosing the projection view template in the creation wizard for data definitions.

CDS views for BO structure	/ DMO/ I_TRAVEL_M	/ DMO/ I_BOOKING_M	/ DMO/ I_BOOKSUPPL_M
<hr/>			
CDS views for BO projection	/ DMO/ C_TRAVEL_PRO CESSOR_M	/ DMO/ C_BOOKING_PR OCESSOR_M	/ DMO/ C_BOOKSUPPL PROCESSOR_M

Note

The names are assigned according to the naming conventions for projection views: [Naming Conventions for Development Objects \[page 780\]](#).

For more information, see [Creating Projection Views \[page 773\]](#).

The resulting CDS projection views must have the following syntax:

```
define root view entity <projection_view> as projection on <projected_view>
```

For more information about the syntax in projection views, see [Syntax for CDS Projection Views \[page 239\]](#)

5.2.4.1.1.2 Defining the Data Model for the Processor Projection Views

The following topics provide you with a detailed description on how to define the data model for the CDS projection views that are used in the BO projection for the processor.

- [Travel Projection View /DMO/C_TRAVEL_PROCESSOR_M \[page 242\]](#)
- [Booking Projection View /DMO/C_BOOKING_PROCESSOR_M \[page 245\]](#)
- [Booking Supplement Projection View /DMO/C_BOOKSUPPL_PROCESSOR_M \[page 249\]](#)

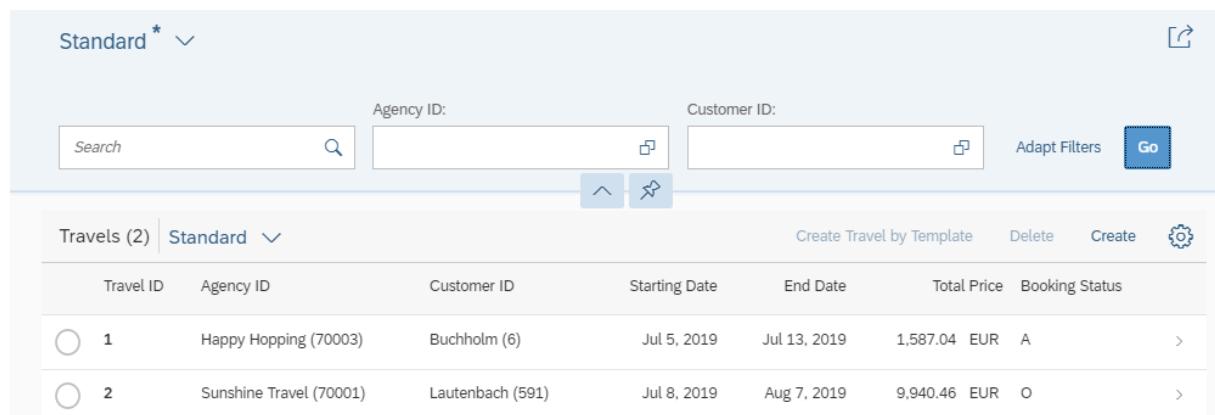
5.2.4.1.1.2.1 Travel Projection View /DMO/C_TRAVEL_PROCESSOR_M

For the service specific projection, the elements as well as all the UI specifics need to be defined.

The data model defines which elements are exposed for the UI service. In addition, in data definitions for projection views you define all UI specifications.

The following UI is achieved by implementing the corresponding features in the CDS Travel projection view for the processor.

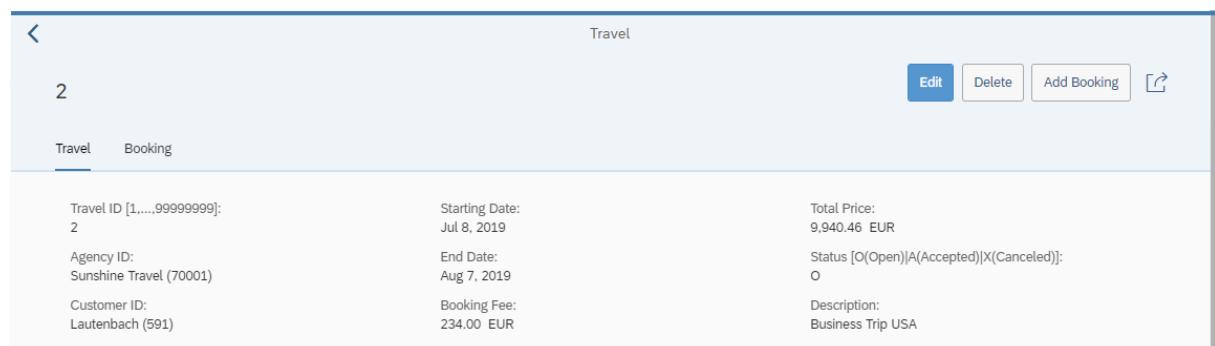
Preview: UI Application for Processor



The screenshot shows a standard UI application interface. At the top, there are search fields for 'Agency ID' and 'Customer ID' with a 'Search' button, and a 'Go' button for filters. Below the search bar is a navigation bar with tabs: 'Travels (2)' (selected), 'Standard', and a dropdown arrow. To the right of the tabs are buttons for 'Create Travel by Template', 'Delete', 'Create', and a gear icon. The main area displays a table of travel records:

Travel ID	Agency ID	Customer ID	Starting Date	End Date	Total Price	Booking Status
1	Happy Hopping (70003)	Buchholm (6)	Jul 5, 2019	Jul 13, 2019	1,587.04 EUR	A
2	Sunshine Travel (70001)	Lautenbach (591)	Jul 8, 2019	Aug 7, 2019	9,940.46 EUR	O

Travel List Report



The screenshot shows a report view for travel details. At the top, there is a back arrow, a title 'Travel', and buttons for 'Edit', 'Delete', 'Add Booking', and a refresh icon. Below the title, there are tabs for 'Travel' (selected) and 'Booking'. The main area displays travel details for travel ID 2:

Travel ID [1,...,99999999]: 2	Starting Date: Jul 8, 2019	Total Price: 9,940.46 EUR
Agency ID: Sunshine Travel (70001)	End Date: Aug 7, 2019	Status [O(Open) A(Accepted) X(Canceled)]: O
Customer ID: Lautenbach (591)	Booking Fee: 234.00 EUR	Description: Business Trip USA

Travel Object Page

i Expand the following listing to view the source code of the travel projection view /DMO/C_TRAVEL_PROCESSOR_M that results in the previously shown UI:

/DMO/C_TRAVEL_PROCESSOR_M

```
@EndUserText.label: 'Travel projection view'  
@AccessControl.authorizationCheck: #NOT_REQUIRED  
@UI: {  
    headerInfo: { typeName: 'Travel', typeNamePlural: 'Travels', title: { type: #STANDARD, value: 'TravelID' } } }  
    @Search.searchable: true  
    define root view entity /DMO/C_Travel_Processor_M  
        as projection on /DMO/I_Travel_M  
    {  
        @UI.facet: [ { id: 'Travel',
```

```

        purpose:      #STANDARD,
        type:        #IDENTIFICATION_REFERENCE,
        label:       'Travel',
        position:    10 },
        { id:          'Booking',
        purpose:      #STANDARD,
        type:        #LINEITEM_REFERENCE,
        label:       'Booking',
        position:    20,
        targetElement: '_Booking' }]

@UI: {
    lineItem:      [ { position: 10, importance: #HIGH } ],
    identification: [ { position: 10, label: 'Travel ID
[1,...,99999999]' } ] }
    @Search.defaultSearchElement: true
key travel_id           as TravelID,
@UI: {
    lineItem:      [ { position: 20, importance: #HIGH } ],
    identification: [ { position: 20 } ],
    selectionField: [ { position: 20 } ] }
    @Consumption.valueHelpDefinition: [{ entity : {name: '/DMO/I_Agency',
element: 'AgencyID' } }]
    @ObjectModel.text.element: ['AgencyName']
    @Search.defaultSearchElement: true
agency_id             as AgencyID,
_Agency.Name         as AgencyName,
@UI: {
    lineItem:      [ { position: 30, importance: #HIGH } ],
    identification: [ { position: 30 } ],
    selectionField: [ { position: 30 } ] }
    @Consumption.valueHelpDefinition: [{ entity : {name: '/DMO/I_Customer',
element: 'CustomerID' } }]
    @ObjectModel.text.element: ['CustomerName']
    @Search.defaultSearchElement: true
customer_id           as CustomerID,
_Customer.LastName as CustomerName,
@UI: {
    lineItem:      [ { position: 40, importance: #MEDIUM } ],
    identification: [ { position: 40 } ] }
begin_date            as BeginDate,
@UI: {
    lineItem:      [ { position: 41, importance: #MEDIUM } ],
    identification: [ { position: 41 } ] }
end_date              as EndDate,
@UI: {
    identification: [ { position: 42 } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
booking_fee           as BookingFee,
@UI: {
    lineItem:      [ { position: 43, importance: #MEDIUM } ],
    identification: [ { position: 43, label: 'Total Price' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
total_price           as TotalPrice,
    @Consumption.valueHelpDefinition: [{entity: {name: 'I_Currency', element:
'Currency' }}]
    currency_code      as CurrencyCode,
@UI: {
    lineItem:      [ { position: 50, importance: #HIGH },
                    { type: #FOR_ACTION, dataAction:
'createTravelByTemplate', label: 'Create Travel by Template' } ],
    identification: [ { position: 45, label: 'Status [O(Open)|A(Accepted)|
X(Canceled)]' } ],
    overall_status     as TravelStatus,
@UI: {
    identification: [ { position: 46 } ] }
description           as Description,
@UI.hidden: true
last_changed_at       as LastChangedAt,

```

```

/* Associations */
Booking : redirected to composition child /DMO/C_Booking_Processor_M,
_Agency,
_Customer
}

```

Explanation

For the data model of the travel projection view in our scenario, you can adopt all elements of the projected view, except for `created_by`, `created_at` and `last_changed_by`. Those element are not needed for our service use cases. The element `last_changed_at`, however, is needed to store the eTag, but the other administrative elements are not needed in the scenario. The other elements for travel information are used to process travel data.

→ Remember

The eTag is needed for optimistic concurrency check. In the travel BO, all nodes use their own master eTag.

All elements of the projection can be given an alias with an automatic mapping done by the service framework.

The travel projection view uses a subset of the associations that are defined in the projected view. `_Agency` and `_Customer` are needed for text provisioning. These associations can simply be adopted in the projection view. On the other hand, the composition to the child entity booking must be redirected as the target entity changes in the projection layer. The association `_Currency` is not necessary in the projection view. It is only defined in the underlying BO data model to represent a complete data model structure.

i Note

Before you can activate the travel projection root view for the processor, you need to create the booking projection view with the redirection of the composition to parent in the booking projection child view. Compositions must always be consistent from parent to child and vice-versa.

UI Specifics

The UI header information is given in an entity annotation to label the list report page.

The travel processor projection view is the root node of the BO projection. When opening the travel processing app, the travel entries are displayed as list items with a navigation to their object page and the corresponding bookings. From this object page, it is possible to navigate to the booking supplements. In the back end, navigating is implemented by compositions. For the UI to enable navigation, UI facets need to be defined in the travel projection view for the identification reference of the travel entity on the object page and for the line item reference of the booking entity.

In addition, the elements for list items and the identification reference for the object page need to be annotated in the projection view with the respective UI annotations to define position, importance and possible labels.

To indicate the number range for the `TravelID` element, the range is added to the label. In the same manner, the possible values for the element `TravelStatus` are added.

For more information about UI navigation and positioning of elements, see [Designing the User Interface for a Fiori Elements App \[page 35\]](#) or [UI Annotations \[page 610\]](#).

The annotations that are used in the projected entity are propagated to the projection view. You do not need to reannotate elements with the same annotations as in the projected entity. However, if an annotation draws reference to a specific element and the name of that specific element is changed with an alias in the projection view, the propagated annotation keeps the reference that was given in the projected entity. A semantic

relationship between two elements can then be lost. In such a case, you have to reuse the same annotation and use the alias name in the element reference of the annotation.

In our example scenario, this is the case for the semantic relationship between `CurrencyCode` and `TotalPrice` or `BookingFee`. In the projection view, you do not need to annotate `CurrencyCode` with `@Semantics.currencyCode: true` as this annotation is inherited from the projected entity. The annotation `@Semantics.amount.currencyCode: 'currency_code'` is inherited as well, but the name of the field has changed in the projection view. So you need to reannotate the element with the new alias name: `@Semantics.amount.currencyCode: 'CurrencyCode'`.

→ Tip

Check the [Active Annotations View](#) to find out which annotations are active for the current CDS view and what the values of the active annotations are. For more information, see .

To be able to search for a specific data set, the travel projection view must be search enabled and at least one element must be assigned as default search element. In addition, you can define selection fields to provide a filter bar for certain elements. For more information about search enabling, see [Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#). For more information about selection fields, see [List Report Header \[page 38\]](#).

To access the corresponding texts or descriptions for ID elements, the relationship between the elements `AgencyID` and `CustomerID` and their text elements in the associated text provider views must be established. The text elements of the text provider view must be integrated in the projection view. The text provider view must be associated to the projection view and the text element in the text provider view must be annotated with `@Semantics.text: true`. For more information about text povationing, see [Defining Text Elements \[page 422\]](#).

Especially for a data processing role, value helps are particularly important to find the necessary values for the ID elements `AgencyID`, `CustomerID` and `CurrencyCode`. Value helps are defined with the annotation `@Consumption.valueHelpDefinititon`. The value help provider view does not have to be associated to get the value help as the entity and the element are referenced in the annotation. For more information, see [Providing Value Help \[page 428\]](#).

In the projection view, you also have to define the position of the execution button of actions, that you have defined in the behavior definition. On the list report page, the position of the button for the action `createTravelByTemplate` is defined. For more information about the action, see [Developing Actions \[page 179\]](#).

5.2.4.1.1.2.2 ☁ Booking Projection View /DMO/C_BOOKING_PROCESSOR_M

For the service-specific projection, the elements as well as all the UI specifics must be defined.

The data model defines which elements are exposed for the UI service. In addition, in data definitions you have to define all UI specifications.

The following UI is achieved by implementing the corresponding features in the CDS Booking projection view for the processor.

Preview: UI Application for Processor

Booking								<input type="text" value="Search"/>	Delete	Create		
Booking Number	Booking Date	Customer ID	Airline ID	Flight Number	Flight Date	Flight Price	Status					
<input type="radio"/> 1	Jul 8, 2019	7	United Airlines, Inc. (UA)	1537	Jul 13, 2019	422.00	EUR	N				>
<input type="radio"/> 2	Jul 8, 2019	7	Singapore Airlines Limited (SQ)	11	Apr 19, 2019	4,880.00	SGD	N				>
<input type="radio"/> 3	Jul 8, 2019	7	United Airlines, Inc. (UA)	59	Apr 17, 2019	6,053.00	USD	N				>

Booking List Report Page

< Booking 2 / 1

[Edit](#) [Delete](#) [Add Supplement](#) [Print](#) [Up](#) [Down](#)

[Booking](#) [Booking Supplement](#)

Booking Number:	Airline ID:	Flight Price:
1	United Airlines, Inc. (UA)	422.00 EUR
Booking Date:	Flight Number:	Status [N(New) X(Canceled) B(Booked)]:
Jul 8, 2019	1537	N
Customer ID:	Flight Date:	
7	Jul 13, 2019	

Booking Object Page

i Expand the following listing to view the source code of the booking projection view /DMO/C BOOKING PROCESSOR M that results in the previously shown UI:

/ DMO/C BOOKING PROCESSOR M

```

@EndUserText.label: 'Booking projection view'
@AccessControl.authorizationCheck: #NOT_REQUIRED
@UI: {
    headerInfo: { typeName: 'Booking',
                  typeNamePlural: 'Bookings',
                  title: { type: #STANDARD, value: 'BookingID' } } }
@Search.searchable: true
define view entity /DMO/C_Booking_Processor_M as projection on /DMO/I_Booking_M
{
    @UI.facet: [ { id:                 'Booking',
                   purpose:          #STANDARD,
                   type:              #IDENTIFICATION_REFERENCE,
                   label:             'Booking',
                   position:          10 },
                  { id:                 'BookingSupplement',
                   purpose:          #STANDARD,
                   type:              #LINEITEM_REFERENCE,
                   label:             'Booking Supplement',
                   position:          20,
                   targetElement:     '_BookSupplement' } ]
    @Search.defaultSearchElement: true
    key travel_id
                    as TravelID,
    @UI: { lineItem:      [ { position: 20, importance: #HIGH } ] }
}

```

```

identification: [ { position: 20 } ],
@Search.defaultSearchElement: true
key booking_id           as BookingID,
@UI: { lineItem:      [ { position: 30, importance: #HIGH } ],
       identification: [ { position: 30 } ] }
booking_date             as BookingDate,
@UI: { lineItem:      [ { position: 40, importance: #HIGH } ],
       identification: [ { position: 40 } ] }
@Consumption.valueHelpDefinition: [{entity: {name: '/DMO/I_Customer',
element: 'CustomerID'}}]
@Search.defaultSearchElement: true
customer_id              as CustomerID,
@UI: { lineItem:      [ { position: 50, importance: #HIGH } ],
       identification: [ { position: 50 } ] }
@Consumption.valueHelpDefinition: [{entity: {name: '/DMO/I_Carrier',
element: 'AirlineID'}}]
@ObjectModel.text.element: ['CarrierName']
carrier_id                as CarrierID,
_carrier.Name              as CarrierName,
@UI: { lineItem:      [ { position: 60, importance: #HIGH } ],
       identification: [ { position: 60 } ] }
@Consumption.valueHelpDefinition: [{entity: {name: '/DMO/I_Flight',
element: 'ConnectionID'}},
                                   additionalBinding: [ { localElement:
'FlightDate',   element: 'FlightDate' },
{ localElement:
'CarrierID',   element: 'AirlineID' },
{ localElement:
'FlightPrice', element: 'Price' },
{ localElement:
'CurrencyCode', element: 'CurrencyCode' } ] ]
connection_id              as ConnectionID,
@UI: { lineItem:      [ { position: 70, importance: #HIGH } ],
       identification: [ { position: 70 } ] }
@Consumption.valueHelpDefinition: [{entity: {name: '/DMO/I_Flight',
element: 'FlightDate'}},
                                   additionalBinding: [ { localElement:
'ConnectionID', element: 'ConnectionID' },
{ localElement:
'CarrierID',   element: 'AirlineID' },
{ localElement:
'FlightPrice', element: 'Price' },
{ localElement:
'CurrencyCode', element: 'CurrencyCode' } ] ]
flight_date                as FlightDate,
@UI: { lineItem:      [ { position: 80, importance: #HIGH } ],
       identification: [ { position: 80 } ] }
@Semantics.amount.currencyCode: 'CurrencyCode'
flight_price               as FlightPrice,
@Consumption.valueHelpDefinition: [{entity: {name: 'I_Currency', element:
'Currency'}}]
currency_code              as CurrencyCode,
@UI: { lineItem:      [ { position: 90, importance: #HIGH, label:
>Status' } ],
       identification: [ { position: 90, label: 'Status [N(New) |
X(Canceled) | B(Booked)]' } ] }
booking_status              as BookingStatus,
@UI.hidden: true
last_changed_at             as LastChangedAt,

```

```

    /* Associations */
    _Travel: redirected to parent /DMO/C_Travel_Processor_M,
    _BookSupplement: redirected to composition child /DMO/
C_BookSuppl_Processor_M,
    _Customer,
    _Carrier
}

```

Explanation

For the data model of the booking projection view, you can adopt all elements of the projected view.

All elements of the projection can be given an alias with an automatic mapping done by the service framework.

The booking projection view uses a subset of the associations that are defined in the projected view. The associations `_Customer` and `_Carrier` are needed for text provisioning. These associations can simply be adopted in the projection view. On the other hand, the compositions to the parent entity `_Travel` and to the child entity `_BookSupplement` must be redirected as the target entities change in the projection layer. The association `_Connection` is not necessary in the projection view. It is defined in the underlying BO data model to complete the BO data model structure.

i Note

Before you can activate the booking projection view for the processor, you need to create the booking supplement projection view with the redirection to the composition to parent from the booking supplement projection child view. Compositions must always be consistent from parent to child and vice-versa.

UI Specifics

Like in the travel projection view, the UI header information for the booking projection view is given in an entity annotation.

For the UI to enable navigation from the Booking to the BookingSupplement entity, you need to define UI facets. The booking entity must be defined as identification reference and the BookingSupplement as line item reference.

In addition, the elements for list items and identification reference for the second navigation need to be annotated in the booking projection view with the respective UI annotations to define position, importance, and possible labels.

For more information about UI navigation and positioning of elements, see [Designing the User Interface for a Fiori Elements App \[page 35\]](#) or [UI Annotations \[page 610\]](#).

As in the travel projection view, the annotation `@Semantics.amount.currencyCode: 'CurrencyCode'` needs to be repeated in the projection view, since the annotation value changes due to aliasing.

To be able to search for a specific data set, the booking projection view must be search enabled and at least one element must be assigned as default search element. For more information about search enabling, see [Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#).

To access the corresponding texts or descriptions for ID elements, the relationship between the element `CarrierID` and its text element in the associated text provider view must be established. Therefore, you need the association to the text provider view. The text element of the text provider view must be integrated in the projection view. The text provider view must be associated to the projection view and the text element in the text provider view must be annotated with `@Semantics.text: true`. For more information about text provisioning, see [Defining Text Elements \[page 422\]](#).

Especially for a data processing role, value helps are important to find the necessary values for the ID elements CustomerID, CarrierID, ConnectionID and to find adequate values for FlightDate and CurrencyCode. Value helps are defined with the annotation @Consumption.valueHelpDefinititon. The value help for ConnectionID and FlightDate use additional bindings, so that only those values appear that match the entries in the given local elements. The value help provider view does not have to be associated to get the value help as the entity and the element are referenced in the annotation. However, it needs to be included in the service definition. For more information, see [Providing Value Help \[page 428\]](#).

The administrative field last_changed_at is only used for concurrent processing and does not have to be displayed on the UI. The annotations @UI.hidden = true is used for that purpose.

5.2.4.1.1.2.3 Booking Supplement Projection View /DMO/

C_BOOKSUPPL_PROCESSOR_M

For the service-specific projection, the elements as well as all the UI specifics must be defined.

The data model defines which elements are exposed for the UI service. In addition, in data definitions you have to define all UI specifications.

The following UI is achieved by implementing the corresponding features in the CDS Booking Supplement projection view for the processor.

Preview: UI Application for Processor

Booking Supplement		
	Book. Supp. Number	Product ID
<input type="radio"/>	1	Mango Juice (BV-0007)
<input type="radio"/>	2	Apple Pie (ML-0003)
<input type="radio"/>	3	Bulky goods like sports equipment (LU-0004)
<input type="radio"/>	4	Pear Pie (ML-0004)
<input type="radio"/>	5	Hamburg Salad with Fresh Shrimps (ML-0012)

Booking Supplement List Report Page

Expand the following listing to view the source code of the booking projection view /DMO/C_BOOKSUPPL_PROCESSOR_M that results in the previously shown UI:

/DMO/C_BOOKSUPPL_PROCESSOR_M

```
@EndUserText.label: 'Booking supplement projection view'
@AccessControl.authorizationCheck: #NOT_REQUIRED
@UI: { headerInfo: { typeName: 'Booking Supplement',
                     typeNamePlural: 'Booking Supplements',
                     title: { type: #STANDARD,
                               label: 'Booking Supplement',
                               value: 'BookingSupplementID' } } }
```

```

@Search.searchable: true
define view entity /DMO/C_BookSuppl_Processor_M as projection on /DMO/
I_BookSuppl_M
{
    @UI.facet: [ { id:                      'BookingSupplement',
                  purpose:          '#STANDARD',
                  type:             '#IDENTIFICATION_REFERENCE',
                  label:            'Booking Supplement',
                  position:         10 } ]
    @Search.defaultSearchElement: true
    key travel_id                         as TravelID,
        booking_id                        as BookingID,
        @UI: { lineItem: [ { position: 10, importance: #HIGH } ],
                identification: [ { position: 10 } ] }
        key booking_supplement_id         as BookingSupplementID,
        @UI: { lineItem: [ { position: 20, importance: #HIGH } ],
                identification: [ { position: 20 } ] }
        @Consumption.valueHelpDefinition: [ {entity: {name: '/DMO/I_SUPPLEMENT',
element: 'SupplementID'}},
                                             additionalBinding: [ { localElement:
'Price', element: 'Price' },
                                             { localElement:
'CurrencyCode', element: 'CurrencyCode' } ] ]
        @ObjectModel.text.element: ['SupplementDescription']
        supplement_id                     as SupplementID,
        _SupplementText.Description       as SupplementDescription:
localized,
        @UI: { lineItem: [ { position: 30, importance: #HIGH } ],
                identification: [ { position: 30 } ] }
        @Semantics.amount.currencyCode: 'CurrencyCode'
        price                           as Price,
        @Consumption.valueHelpDefinition: [ {entity: {name: 'I_Currency', element:
'Currency' }}]
        currency_code                   as CurrencyCode,
        @UI.hidden: true
        last_changed_at                 as LastChangedAt,
    /* Associations */
        _Travel : redirected to /DMO/C_Travel_Processor_M,
        _Booking : redirected to parent /DMO/C_Booking_Processor_M,
        _SupplementText
}

```

For the data model of the booking supplement projection view, you can adopt all elements of the projected view.

All elements of the projection can be given an alias with an automatic mapping done by the service framework.

The booking supplement projection view uses a subset of the associations that are defined in the projected view. The association `_SupplementText` is needed for text provisioning. This association can simply be adopted in the projection view. On the other hand, the composition to the parent entity `_Booking` and to the root entity `_Travel` must be redirected as the target entity changes in the projection layer. The association `_Supplement` is not necessary in the projection view of the service context. It is defined in the underlying BO data model to complete the BO data model structure.

i Note

Now, that all compositions are redirected, you can activate the three projection views for the processor.

UI Specifics

Like in the travel and the booking projection view, the UI header information for the booking supplement projection view is given in an entity annotation.

! Restriction

The Fiori Elements Preview does not support the navigation to more than one child entity. Hence, when accessing the preview via entity set `Travel` and association `to_Booking`, it is not possible to navigate to the object page of the `BookingSupplement` entity. That means, some of the UI annotations are not relevant if you only use the preview to test your application. For example, the UI annotations referring to `identification` cannot be shown in the preview, when testing via the root node `Travel`.

However, if you want to develop a real application or test your service with the Web IDE, you can configure the application to enable navigation to any number of child entities. That is why, the UI annotation concerning `identification` are included in the following description.

You can imitate the behavior of the Web IDE for the second-level navigation by accessing the Fiori Elements Preview via entity set `Booking` and Association `to_BookSupplement`.

To show the entries of the booking supplement entity on its object page, the UI facet for `#IDENTIFICATION_REFERENCE` must be defined.

In addition, the elements for list items (to appear on the object page of the booking entity) and identification must be annotated in the Booking Supplement projection view with the respective UI annotation to define position, importance, and possible labels.

For more information about UI navigation and positioning of elements, see [Designing the User Interface for a Fiori Elements App \[page 35\]](#) or [UI Annotations \[page 610\]](#).

As in the travel projection view, the annotation `@Semantics.amount.currencyCode: 'CurrencyCode'` needs to be repeated in the projection view, since the annotation value changes due to aliasing.

To be able to search for a specific data set, the booking supplement projection view must be search enabled and at least one element must be assigned as default search element. For more information about search enabling, see [Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#).

To access the corresponding texts or descriptions for ID elements, the relationship between the element `SupplementID` and its text element in the associated text provider view must be established. Therefore, you need the association to the text provider view. The text element of the text provider view must be integrated in the projection view. The text provider view must be associated to the projection view and the text element in the text provider view must be annotated with `@Semantics.text: true`. For more information about text provisioning, see [Defining Text Elements \[page 422\]](#).

Especially for a data processing role, value helps are important to find the necessary values for the ID element `SupplementID` as well as for `CurrencyCode`. Value helps are defined with the annotation `@Consumption.valueHelpDefinititon`. The value help for `SupplementID` uses additional binding, so that only those values appear that match the entry in the field `CurrencyCode` field. The value help provider view does not have to be associated to get the value help as the entity and the element are referenced in the annotation. For more information, see [Providing Value Help \[page 428\]](#).

The administrative field `last_changed_at` is only used for concurrent processing and does not have to be displayed on the UI. The annotations `@UI.hidden = true` is used for that purpose.

5.2.4.1.2 Projection Views for the Approver BO Projection

To define a data model for the BO projection that defines the scope for the approver application, the following tasks need to be done:

- [Creating the Projection CDS Views for the Approver \[page 252\]](#)
- [Defining the Data Model for the Approver Projection Views \[page 253\]](#)

5.2.4.1.2.1 Creating the Projection CDS Views for the Approver

The scope of the UI service for the approver is more limited than for the processor. The approver can only modify the travel entity with accepting or rejecting the travel entries. The values in the corresponding booking entries are the basis for this decision-making. Only these two entities are relevant for the approver app. That means, only these two entities must be projected for the approver BO projection.

For the following CDS views, create the corresponding projection views by choosing the projection view template in the creation wizard for data definitions.

CDS views for BO structure	/ DMO/ I_TRAVEL_M	/ DMO/ I_BOOKING_M
CDS views for BO projection	/ DMO/ C_TRAVEL_APP ROVER_M	/ DMO/ C_BOOKING_AP PROVER_M

Note

The names are assigned according to the naming conventions for projection views: [Naming Conventions for Development Objects \[page 780\]](#).

For more information, see [Creating Projection Views \[page 773\]](#).

The resulting CDS projection views must have the following syntax:

```
define root view entity <projection_view> as projection on <projected_view>
```

For more information about the syntax in projection views, see [Syntax for CDS Projection Views \[page 239\]](#)

5.2.4.1.2.2 Defining the Data Model for the Approver Projection Views

The following topics provide you with a detailed description on how to define the data model for the CDS projection views that are used in the BO projection for the approver.

- [Travel Projection View /DMO/C_TRAVEL_APPROVER_M \[page 253\]](#)
- [Booking Projection View /DMO/C_BOOKING_APPROVER_M \[page 256\]](#)

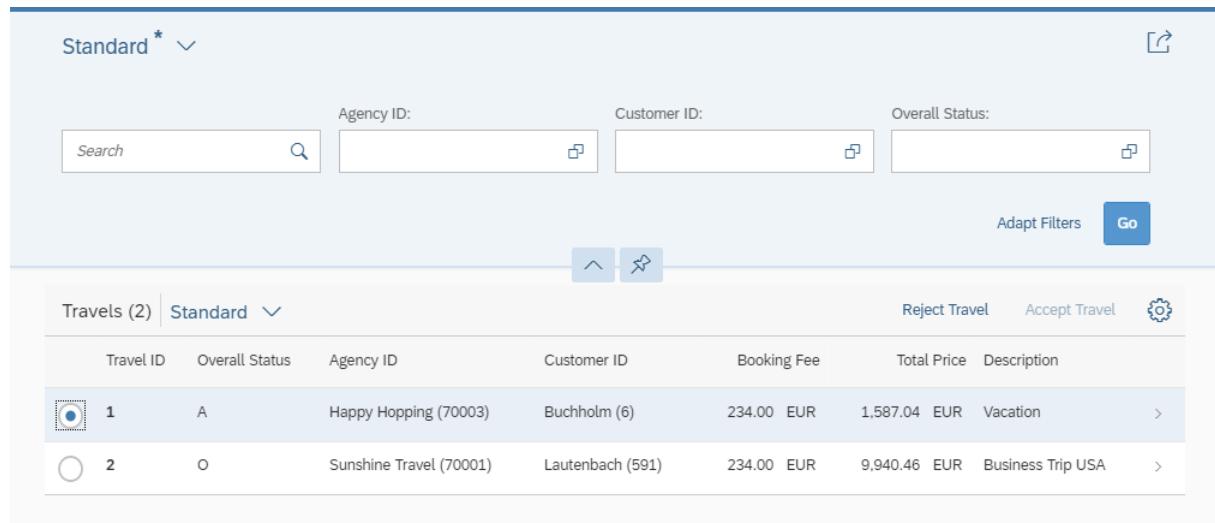
5.2.4.1.2.2.1 Travel Projection View /DMO/C_TRAVEL_APPROVER_M

For the service-specific projection, the elements as well as all the UI specifics for the approver BO projection must be defined.

The data model defines which elements are exposed for the UI service. In addition, in data definitions you have to define all UI specifications.

The following UI is achieved by implementing the corresponding features in the CDS Travel projection view for the approver.

Preview: UI Application for Approver



Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee	Total Price	Description
1	A	Happy Hopping (70003)	Buchholm (6)	234.00 EUR	1,587.04 EUR	Vacation
2	O	Sunshine Travel (70001)	Lautenbach (591)	234.00 EUR	9,940.46 EUR	Business Trip USA

Travel List Report Page

Travel Object Page

i Expand the following listing to view the source code of the travel projection view /DMO/C_TRAVEL_APPROVER_M that results in the previously shown UI:

```
/DMO/C_TRAVEL_APPROVER_M
```

```
@EndUserText.label: 'Travel projection view'
@AccessControl.authorizationCheck: #NOT_REQUIRED
@UI: {
    headerInfo: { typeName: 'Travel', typeNamePlural: 'Travels', title: { type: #STANDARD, value: 'TravelID' } } }
@Search.searchable: true
define root view entity /DMO/C_Travel_Approver_M
    as projection on /DMO/I_Travel_M
{
    @UI.facet: [ { id: 'Travel',
                    purpose: #STANDARD,
                    type: #IDENTIFICATION_REFERENCE,
                    label: 'Travel',
                    position: 10 },
                  { id: 'Booking',
                    purpose: #STANDARD,
                    type: #LINEITEM_REFERENCE,
                    label: 'Booking',
                    position: 20,
                    targetElement: '_Booking' } ]
    @UI: {
        lineItem: [ { position: 10, importance: #HIGH } ],
        identification: [ { position: 10 } ] }
    @Search.defaultSearchElement: true
key travel_id as TravelID,
    @UI: {
        lineItem: [ { position: 20, importance: #HIGH } ],
        identification: [ { position: 20 } ],
        selectionField: [ { position: 20 } ] }
    @Consumption.valueHelpDefinition: [ { entity : { name: '/DMO/I_Agency',
element: 'AgencyID' } } ]
    @ObjectModel.text.element: ['AgencyName']
    @Search.defaultSearchElement: true
    agency_id as AgencyID,
    _Agency.Name as AgencyName,
    @UI: {
        lineItem: [ { position: 30, importance: #HIGH } ],
        identification: [ { position: 30 } ] },
```

```

        selectionField: [ { position: 30 } ] }
    @Consumption.valueHelpDefinition: [ { entity : {name: '/DMO/I_Customer',
element: 'CustomerID' } } ]
    @ObjectModel.text.element: ['CustomerName']
    @Search.defaultSearchElement: true
    customer_id           as CustomerID,
    Customer.LastName as CustomerName,
    @UI: {
        identification:[ { position: 40 } ] }
    begin_date            as BeginDate,
    @UI: {
        identification:[ { position: 41 } ] }
    end_date              as EndDate,
    @UI: {
        lineItem:      [ { position: 42, importance: #MEDIUM } ],
        identification: [ { position: 42 } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    booking_fee           as BookingFee,
    @UI: {
        lineItem:      [ { position: 43, importance: #MEDIUM } ],
        identification: [ { position: 43, label: 'Total Price' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    total_price           as TotalPrice,
    @Consumption.valueHelpDefinition: [ {entity: {name: 'I_Currency', element:
'Currency' } } ]
    currency_code         as CurrencyCode,
    @UI: {
        lineItem:      [ { position: 15, importance: #HIGH },
                        { type: #FOR_ACTION, dataAction: 'acceptTravel',
label: 'Accept Travel' },
                        { type: #FOR_ACTION, dataAction: 'rejectTravel',
label: 'Reject Travel' },
                        identification: [ { position: 15 },
                            { type: #FOR_ACTION, dataAction: 'acceptTravel',
label: 'Accept Travel' },
                            { type: #FOR_ACTION, dataAction: 'rejectTravel',
label: 'Reject Travel' } ],
                        selectionField: [ { position: 40 } ] }
    @EndUserText.label: 'Overall Status'
    overall_status        as TravelStatus,
    @UI: {
        lineItem: [ { position: 45, importance: #MEDIUM } ],
        identification:[ { position: 45 } ] }
    description           as Description,
    @UI.hidden: true
    last_changed_at       as LastChangedAt,
    /* Associations */
    _Booking : redirected to composition child /DMO/C_Booking_Approver_M,
    _Agency,
    _Customer
}

```

Explanation

Except for the actions, which are different in the processor and the approver projection, the CDS projection views for the processor and the approver BO are identical. Refer to [explanation \[page 244\]](#) for a thorough description on the travel projection view.

Minor changes can be detected in the field label of the field `TravelID` and `TravelStatus`. This results from the fact, that the approver does not create new travel entries. It is not necessary for this role to know the number range of the `TravelID` or the possible values of the `TravelStatus`. In addition, the approver BO projection has gained a selection field for the `TravelStatus` to make it easier for the approver to filter for open/accepted/rejected travels.

Actions

The position and the label for the action button must be defined in the CDS projection views. In the case of an approver, the available actions concerning the travel entity set are Accept Travel and Reject Travel. The implementation of these actions is done in the behavior pool, see [Developing Actions \[page 179\]](#). It is simply the UI appearance that needs to be configured in the projection view. The action buttons for the respective actions are designed to appear on the travel list report page and on the travel object page. That is why the annotations are used in the list item and identification UI annotation. When executing the action on the list report page, a travel instance must be selected to assign an instance for the instance-bound action. On the object page, the instance for which the action shall be executed is clear. For more information about the annotations to define the action buttons, see [Enabling Actions for UI Consumption \[page 186\]](#).

5.2.4.1.2.2.2 Booking Projection View /DMO/C_BOOKING_APPROVER_M

The data model defines which elements are exposed for the UI service. In addition, in data definitions you have to define all UI specifications.

Preview: UI Application Approver

Booking List Report Page								
Booking Number	Booking Date	Customer ID	Airline ID	Flight Number	Flight Date	Flight Price	Status	
1	Jul 5, 2019	7	United Airlines, Inc. (UA)	1537	Jul 10, 2019	500.00	AUD	N >
2	Jul 5, 2019	7	United Airlines, Inc. (UA)	1537	Jul 10, 2019	422.00	USD	N >
3	Jul 8, 2019	7	United Airlines, Inc. (UA)	1537	Jul 13, 2019	422.00	EUR	N >
4	Jul 8, 2019	7	United Airlines, Inc. (UA)	1537	Jul 13, 2019	422.00	EUR	N >

Booking List Report Page

Booking

1 / 1

Booking

Booking Number:	Flight Number:
1	1537
Booking Date:	Flight Date:
Jul 5, 2019	Jul 10, 2019
Customer ID:	Flight Price:
7	500.00 AUD
Airline ID:	Status [N(New) X(Canceled) B(Booked)]:
United Airlines, Inc. (UA)	N

Booking Object Page

i Expand the following listing to view the source code of the travel projection view /DMO/C_BOOKING_APPROVER_M that results in the previously shown UI:

```
/DMO/C_BOOKING_APPROVER_M

@EndUserText.label: 'Booking projection view'
@AccessControl.authorizationCheck: #NOT_REQUIRED
@UI: {
    headerInfo: { typeName: 'Booking',
                  typeNamePlural: 'Bookings',
                  title: { type: #STANDARD, value: 'BookingID' } }
}
@Search.searchable: true
define view entity /DMO/C_Booking_Approver_M as projection on /DMO/I_Booking_M
{
    @UI.facet: [ { id: 'Booking',
                   purpose: #STANDARD,
                   type: #IDENTIFICATION_REFERENCE,
                   label: 'Booking',
                   position: 10 } ]

    @Search.defaultSearchElement: true
    key travel_id as TravelID,
        @UI: { lineItem: [ { position: 20, importance: #HIGH } ],
               identification: [ { position: 20 } ] }

    @Search.defaultSearchElement: true
    key booking_id as BookingID,
        @UI: { lineItem: [ { position: 30, importance: #HIGH } ],
               identification: [ { position: 30 } ] }
    booking_date as BookingDate,

    @UI: { lineItem: [ { position: 40, importance: #HIGH } ],
           identification: [ { position: 40 } ],
           selectionField: [ { position: 10 } ] }
    @Search.defaultSearchElement: true
    customer_id as CustomerID,
```

```

@UI: { lineItem: [ { position: 50, importance: #HIGH } ],
        identification: [ { position: 50 } ] }
@ObjectModel.text.element: ['CarrierName']
carrier_id
    as CarrierID,
_Carrier.Name
    as CarrierName,

@UI: { lineItem: [ { position: 60, importance: #HIGH } ],
        identification: [ { position: 60 } ] }
connection_id
    as ConnectionID,

@UI: { lineItem: [ { position: 70, importance: #HIGH } ],
        identification: [ { position: 70 } ] }
flight_date
    as FlightDate,

@UI: { lineItem: [ { position: 80, importance: #HIGH } ],
        identification: [ { position: 80 } ] }
@Semantics.amount.currencyCode: 'CurrencyCode'
flight_price
    as FlightPrice,
currency_code
    as CurrencyCode,
@UI: { lineItem: [ { position: 90, importance: #HIGH, label:
'Status' } ],
        identification: [ { position: 90, label: 'Status [N(New) | X(Canceled) | B(Booked)]' } ] }
booking_status
    as BookingStatus,
/* Admininstrative fields */
@UI.hidden: true
last_changed_at
    as LastChangedAt,
/* Associations */
_Travel: redirected to parent /DMO/C_Travel_Approver_M,
_Customer,
_Carrier
}

}

```

Explanation

The CDS projection views for the processor and the approver BO are almost identical. Refer to [Booking Projection View /DMO/C_BOOKING_PROCESSOR_M \[page 245\]](#) for a thorough description on the booking projection view.

Value helps are not necessary for the interpretation of the approver role in this scenario. As the booking entity is a read-only entity in this scenario and selection fields with value helps cannot be defined for a sub entity, value helps cannot be applied and thus are not necessary to be defined for the booking entity.

5.2.4.2 Providing Behavior for Projections

The behavior for the BO projection is defined in a behavior definition of type `projection`. The type is defined in the behavior definition header. The projection behavior definition provides the behavior for the projection CDS view. All characteristics and operations that you want to include in the BO projection must be listed explicitly. The keyword for this is `use`.

Syntax: Behavior Definition for Projection

The syntax in a projection behavior definition is the following:

```
projection;
```

```

define behavior for ProjectionView alias ProjectionViewAlias
/* use the same eTag defined in underlying behavior definititon */
use etag
{
  /* define static field control */
  field ( readonly ) ProjViewElem1;
  field ( mandatory ) ProjViewElem2;
  /* expose standard operations defined in underlying behavior definition */
  use create;
  use update;
  use delete;
  /* expose actions or functions defined in underlying behavior definition */
  use action|function ActionName [result entity ProjResultEntity][as ProjAction]
[external ExtProjname];
  /* expose create_by_association for child entities defined in underlying
behavior definition */
  use association _Assoc { create; }
}

```

For a detailed explanation of the syntax, see [Projection Behavior Definition \[page 107\]](#).

Defining the BO Projection Behavior in the Travel Scenario

As described in [Reference Business Scenario \[page 148\]](#), the BO projections for the processor and the approver differ with regard to their behavior. The following sections provide a detailed description on how to project the existing BO to define a behavior for one business object that is tailored to expose a UI service for a data processor and one that is tailored for a data approver.

- [Behavior for the Processor BO Projection \[page 259\]](#)
- [Behavior for the Approver BO Projection \[page 261\]](#)

Related Information

[Projection Behavior Definition \[page 107\]](#)

5.2.4.2.1 Behavior for the Processor BO Projection

The behavior for the BO projection that defines the scope for the processor application is defined in a behavior definition with type projection.

5.2.4.2.1.1 Creating a Behavior Definition for the Processor BO Projection

The easiest way to create a projection behavior definition is to use the context menu in the project explorer by selecting the relevant projection root view /DMO/C_TRAVEL_PROCESSOR_M and choosing [New Behavior Definition](#). The behavior definition always uses the same name as the corresponding root view.

For a more detailed description, see [Creating Behavior Definitions \[page 761\]](#).

As the behavior definition is created on the basis of the root projection view, the template with type projection is generated.

5.2.4.2.1.2 Defining the Behavior for the Processor BO Projection

When creating the behavior definition based on the projection view, the template automatically creates the type projection and lists all available characteristics and operations of the underlying behavior definition. That means, if nothing is done explicitly the BO projection has exactly the same behavior as the underlying BO.

For the processor projection, only the following elements are used:

```
projection;
define behavior for /DMO/C_Travel_Processor_M alias TravelProcessor
use etag
{
  field ( readonly ) TotalPrice;
  field ( mandatory ) BeginDate, EndDate, CustomerID;
  use create;
  use update;
  use delete;
  use action createTravelByTemplate;
  use association _BOOKING { create; }
}
define behavior for /DMO/C_Booking_Processor_M alias BookingProcessor
use etag
{
  use update;
  // use delete; // workaround for missing determination on delete
  use association _BOOKSUPPLEMENT { create; }
}
define behavior for /DMO/C_BookSuppl_Processor_M alias BookSupplProcessor
use etag
{
  use update;
  // use delete; // workaround for missing determination on delete
}
```

Explanation

Only the characteristics and operations that are relevant for the processor are used in the projection behavior definition. This is only a subset of the behavior that was defined in the underlying BO. See [Developing Business Logic \[page 175\]](#) to compare the projection BO to the underlying one.

The ETag handling that was defined in the underlying BO is used for all three entities. Especially for the processor role, which is enabled to modify, it is necessary to have a concurrency check. By using a master ETag on all entities, concurrent processing is enabled for the travel BO.

The static field control that was defined for the underlying BO cannot be modified. However, new controls are added to correspond the processor role. The field `TotalPrice` is set to `readonly` as its value is calculated by a determination. The basic elements for a travel entry `BeginDate`, `EndDate` and `CustomerID` are set to `mandatory`.

All standard operations are used for the processor on all the root entity. The child entities can only be created via a `create_by_association`. The `delete` is not enabled for the view `Booking` and `BookingSupplement` as the determination to calculate the total flight price is not triggered on `delete`.

For the travel entity, the action to create a travel entry by a template is enabled for the processor. This action copies certain values from an existing travel entry to create a new travel entity.

5.2.4.2.2 Behavior for the Approver BO Projection

The behavior for the BO projection that defines the scope for the approver application id defined in a behavior definition with type projection.

5.2.4.2.2.1 Creating a Behavior Definition for the Approver BO Projection

The easiest way to create a projection behavior definition is to use the context menu in the project explorer by selecting the relevant projection root view /DMO/C_TRAVEL_APPROVER_M and choosing [New Behavior Definition](#). The behavior definition always uses the same name as the corresponding root view.

For a more detailed description, see [Creating Behavior Definitions \[page 761\]](#).

As the behavior definition is created on the basis of the root projection view, the template with type projection is generated.

5.2.4.2.2.2 Defining the Behavior for the Approver BO Projection

When creating the behavior definition based on the projection, the template automatically creates the type projection and lists all available characteristics and operations of the underlying behavior definition. That means, if nothing is done explicitly the BO projection has exactly the same behavior as the underlying BO.

For the approver projection, only the following elements are used:

```
projection;
define behavior for /DMO/C_Travel_Approver_M alias Approver
use etag
{
  field ( readonly ) BeginDate, EndDate, TotalPrice, CustomerID;
  use update;
  use action acceptTravel;
  use action rejectTravel;
}
```

Explanation

Only the characteristics and operations that are relevant for the approver are used in the projection behavior definition. This is only a subset of the behavior that was defined in the underlying BO. See [Developing Business Logic \[page 175\]](#) to compare the projection BO to the underlying one.

The eTag that was defined in the underlying BO is used in the approver projection as well. The concurrency check is relevant for the approver. It must be ensured that the data has not been changed between checking the data and executing the action accept or reject travel.

The static field control that was defined for the underlying BO cannot be modified. However, new controls are added to correspond the approver role. The fields that are mandatory for the processor are set to read-only for the approver. In the approver application, one can only change the fields OverallStatus, AgencyID, BookingFee and Description.

The update operation is enabled for the approver as a modification on the travel entries must be available.

The actions accept and reject are enabled to change the status of the travel entry.

There is no behavior defined for the booking entity. All fields are read-only in this case.

The booking supplement entity is not part of the approver BO projection, so there is no behavior for this entity either.

5.2.5 Defining Business Services Based on Projections

With the help of the previous sections, you developed a business object and its projection for two complementary business roles. The next step is to build a business service for both projections in order to consume the business object. The business service defines the scope of the service and binds it to a specific OData protocol. For more information, see [Business Service \[page 98\]](#).

This scenario is designed to build a UI service for both business object projections. Follow the development steps to build an application for both BO projections. For a detailed step-by-step description, see [Creating an OData Service \[page 24\]](#).

1. Create a service definition for the processor service and one for the approver service.
2. Expose the relevant CDS views for each service.

Note

Only the projection CDS views of the business object projection must be exposed for the service. The delegation to the underlying BO is automatically done.

1. Service Definition for Processor Service:

```
@EndUserText.label: 'Service definition for managing travels'
define service /DMO/UI_TRAVEL_Processor_M {
    expose /DMO/C_Travel_Processor_M as Travel;
    expose /DMO/C_Booking_Processor_M as Booking;
    expose /DMO/C_BookSuppl_Processor_M as BookingSupplement;
    expose /DMO/I_Supplement as Supplement;
    expose /DMO/I_SupplementText as SupplementText;
    expose /DMO/I_Customer as Passenger;
    expose /DMO/I_Agency as TravelAgency;
    expose /DMO/I_Carrier as Airline;
    expose /DMO/I_Connection as FlightConnection;
    expose /DMO/I_Flight as Flight;
    expose I_Currency as Currency;
    expose I_Country as Country;
}
```

The complete composition hierarchy is exposed for the data processing service. In addition, the text and value help provider views are necessary components of the service scope to get the respective feature for the service.

2. Service Definition for Approver Service

```
@EndUserText.label: 'Service definition for managing travels'  
define service /DMO/UI_TRAVEL_Approver_M {  
    expose /DMO/C_Travel_Approver_M as Travel;  
    expose /DMO/C_Booking_Approver_M as Booking;  
    expose /DMO/I_Customer as Passenger;  
    expose /DMO/I_Agency as TravelAgency;  
    expose I_Currency as Currency;  
}
```

The approver service contains only two entities of the travel BO and a more limited number of text and value help provider views.

3. Create a service binding with binding type **ODATA V2 – UI** for both service definitions and activate the local service endpoints.

As soon as the service is activated, it is ready for consumption through an OData client such as an SAP Fiori app. You can use the preview function in the service binding to check how the UI of the Fiori application looks like.

5.3 Developing Unmanaged Transactional Apps

This section explains the main development tasks required for enabling transactional processing in a business objects provider that integrates existing business logic.

Based on an end-to-end example, you create and implement all requisite artifacts for providing OData services that combine CDS data model and business object semantics with transactional processing from legacy application logic.

Introduction

The scenario described below focuses on an [unmanaged \[page 821\]](#) implementation type of a business object provider in the context of the ABAP RESTful Application Programming Model. For the unmanaged implementation type, the application developer must implement essential components of the REST contract itself. In this case, all required operations (create, update, delete, or any application-specific actions) must be specified in the corresponding [behavior definition \[page 806\]](#) before they are manually implemented in ABAP.

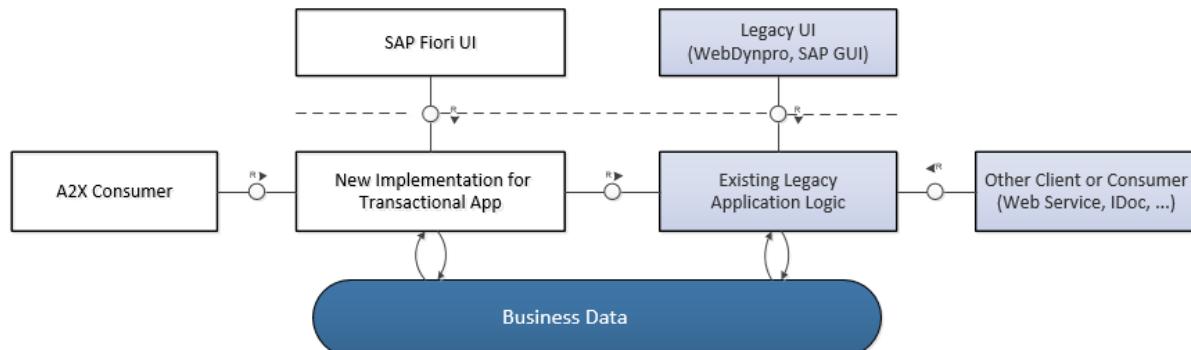
i Note

In a managed implementation type, on the other hand, a behavior definition would on its own be sufficient to obtain a ready-to-run business object.

Architecture Overview

The underlying scenario reuses the existing business application logic and the existing persistence, which manages business data.

If you are running ABAP developments on [SAP Cloud Platform](#), then you can introduce legacy business logic like this in the course of the custom code migration into [ABAP Environment](#).



Architecture Overview – Integration of Existing Application Logic

Prerequisites

Developing the scenario that is described in the subsequent chapters requires the following:

- You have access to and an account for [SAP Cloud Platform, ABAP environment](#).
- You have installed ABAP Development Tools (ADT).
SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.
- To recreate the demo scenario, the [ABAP Flight Reference Scenario](#) must be available in your ABAP system. You can download the complete reference scenario from [GitHub: Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

→ Remember

The namespace `/DMO/` is reserved for the demo content. Apart from the downloaded [ABAP Flight Scenario](#), do not use the namespace `/DMO/` and do not create any development objects in the downloaded packages.

You can access the development objects in `/DMO/` from your own namespace.

However, if you want to recreate all development objects of this demo content, make sure that you use different names from this documentation.

Development Process in Overview

The development of new business services by integrating the transactional behavior of an existing (legacy) application mainly requires developers to perform the following fundamental activities:

1. Defining a CDS Data Model and the Business Object Structure

The formal structure of a business object consists of a tree of entities (*Travel*, *Booking*, *Passenger*, and so on) where the entities are linked using associations. Each entity of this tree structure is an element that is modeled with a CDS entity. Entities of this kind are CDS views that are generally defined on top of the underlying persistence layer, which in turn is based on the corresponding database tables or public interface CDS views.

The root entity is of particular importance: this is indicated in the source code of the CDS data definition by the keyword `ROOT`. The root entity is a representation of the business object and defines the top node in a business object's structure.

The structure of the business object is projected in CDS projection views where you can define service-specifics for the data model. For the UI-related annotations, metadata extensions are used to separate data model from domain-specific semantics

More on this: [Providing CDS Data Model with Business Object Structure \[page 269\]](#)

2. Defining and Implementing the Transactional Behavior of Business Objects

Each node of a business object can offer the standard operations `create()`, `update()`, and `delete()` and specific operations with a dedicated input and output structure known as actions. All operations provided by a business object are defined in the behavior definition artifact that is created as an ABAP repository object.

The implementation of the transactional behavior is done in specific class pools, which refer to the behavior definition. The concrete implementation of the business object provider is based on the ABAP language (which has been expanded from the standard with a special syntax) and the corresponding [*API for Implementing the Unmanaged BO Contract*](#). The implementation tasks are roughly divided into an **interaction phase** and a **save sequence**.

The behavior that is relevant for the specific UI service is then projected in projection behavior definition.

More on this: [Defining and Implementing Behavior of the Business Object \[page 288\]](#)

3. Exposing the Relevant Application Artifacts for OData Service Enablement

For the service enablement, the relevant artifacts must be exposed to OData as a canonical OData service. This is implemented by data and behavior models, where the data model and the related behavior is projected in a service-specific way. This projection is separated into two different artifacts: the service definition and the service binding. The [service definition \[page 807\]](#) is a projection of the data model and the related behavior to be exposed, whereas the [service binding \[page 807\]](#) implements a specific protocol and the kind of service to be offered to a consumer.

More on this: [Defining Business Service for Fiori UI \[page 350\]](#)

4. Testing the OData (UI) Service

In ABAP Development Tools, you have the option of publishing the service to the local system repository. As soon as the service is published, it is ready for consumption through an OData client, such as an SAP Fiori app. The service binding editor offers a preview tool that you can use for testing the resulting app within your ABAP development environment.

i Note

Via ABAPGit You can import the service including the related development objects into your development environment for comparison and reuse. You find the service in the package /DMO/FLIGHT_UNMANAGED. The suffix for development objects in this development guide is _U.

For information about downloading the ABAP Flight Reference Scenario, see [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

5.3.1 Reference Business Scenario

The **Managing of Flight Travels** scenario used in this guide provides an example of an existing stateful business application whose business logic is reused in the new implementation for transactional apps. This application represents only a part of the full *ABAP Flight Reference Scenario* (in short: Flight Scenario) that is intended to be used for demonstration and learning purposes in the context of the ABAP RESTful programming model.

The application demo provided (which represents a legacy stateful application) allows a user to create and manipulate flight bookings. It involves different data sources and entities such as travel, travel agencies, customers (passengers), flights, and bookings. Some of these are editable (that is, they can be created or manipulated) and some are not.

Persistency and Data Model of an Existing Application

The following table gives an overview of the different travel entities involved in the current scenario, including a categorization into editable and non-editable entities.

i Note

All development objects referenced here are available in the package /DMO/FLIGHT_LEGACY. [More on this: ABAP Flight Reference Scenario \[page 774\]](#)

Flight Reference Scenario Objects Involved in the Business Scenario

Entity	Description	Editable
Travel	A Travel entity defines general travel data, such as the agency ID or customer ID, status of the travel booking, and the price of travel. The travel data is stored in the database table /DMO/TRAVEL.	Yes
Agency	An Agency entity defines travel agency data, such as the address and contact data. The corresponding data is stored in the database table /DMO/AGENCY. The flight data model defines a 1 : n cardinality between Agency and Travel.	No

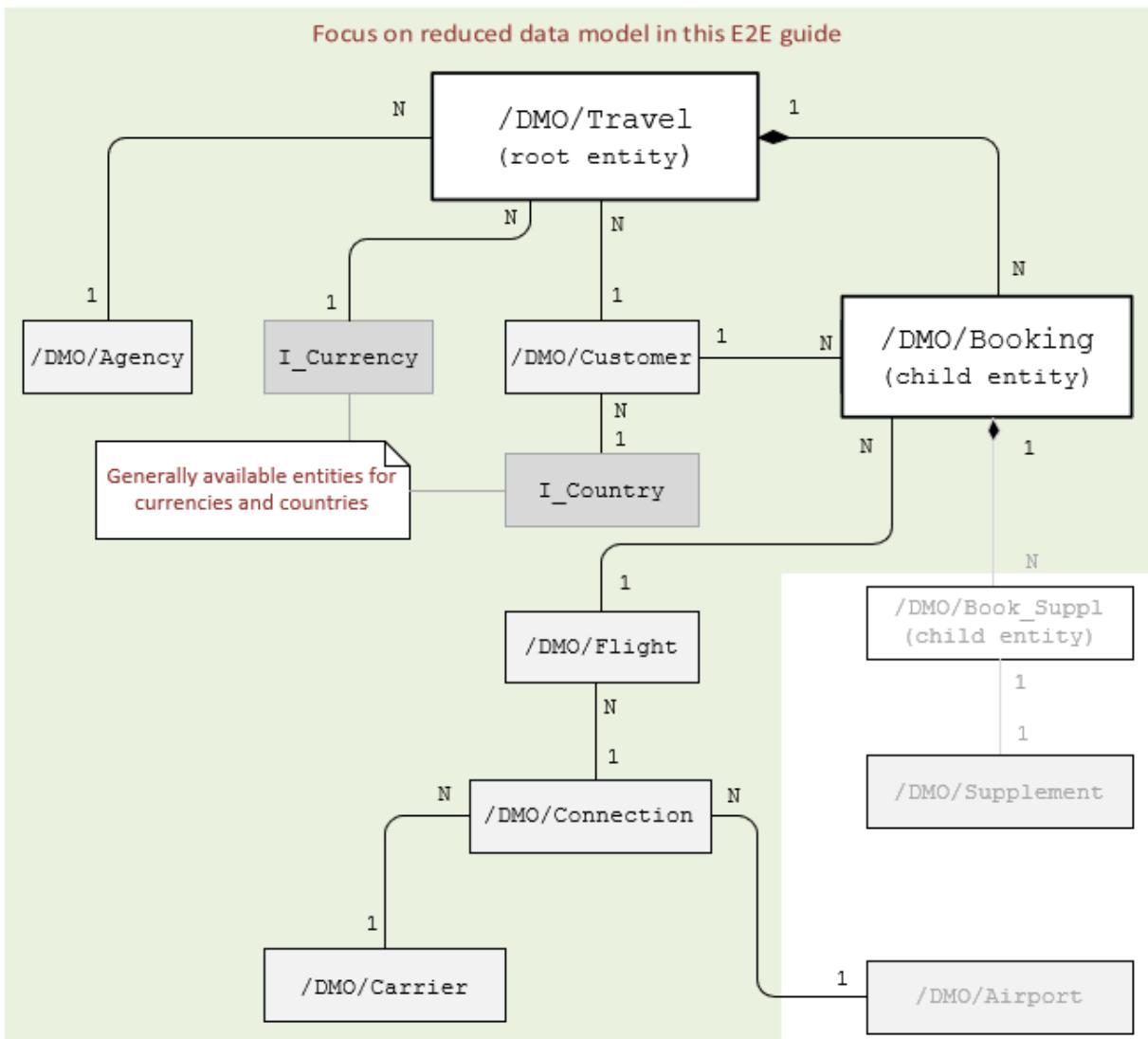
Entity	Description	Editable
Booking	The booking data is stored in the database table /DMO/BOOKING. The flight data model defines a 1 : n cardinality between a Travel and the Booking entity.	Yes
Flight	The specific flight data for each connection is stored in the database table /DMO/ FLIGHT. The flight data model defines a 1 : n cardinality between a Connection and the Flight entity.	No
Connection	The flight connections are stored in the database table /DMO/CONNECTION.	No
Carrier	The IDs and names of airlines are stored in the database table /DMO/CARRIER. Each airline has a number of flight connections. Therefore, the data model defines a 1 : n cardinality between a Carrier and the Connection entity.	No
Customer	A Customer entity provides a detailed description of a flight customer (passenger) such as the name, the address, and contact data. The corresponding data is stored in the database table /DMO/CUSTOMER. The flight data model defines a 1:n cardinality between Customer and Travel.	No
Booking Supplement	This entity is used to add additional products to a travel booking. The booking supplement data is stored in the database table /DMO/BOOK_SUPPL. The flight data model defines an n : 1 cardinality between a Booking Supplement entity and a Booking entity.	Yes
Supplement	A Supplement entity defines product data that the customer can book together with a flight, for example a drink or a meal. The supplement data is stored in the database table /DMO/SUPPLEMENT. The flight data model defines a 1:1 cardinality between a Supplement entity and the Booking Supplement entity.	No

Compositions and Associations

The figure below shows the relationships between the travel, agency, customer, and booking entities, where the travel entity represents the root of the data model. Additional entities for currencies (`I_Currency`) and countries (`I_Country`) are generally available in your system and are included in our data model using associations.

i Note

For didactic reasons, we have kept the data model as simple as possible. We have hence reduced the number of entities in our end-to-end guide (compared with the predefined ABAP flight model) to a minimum set of entities.

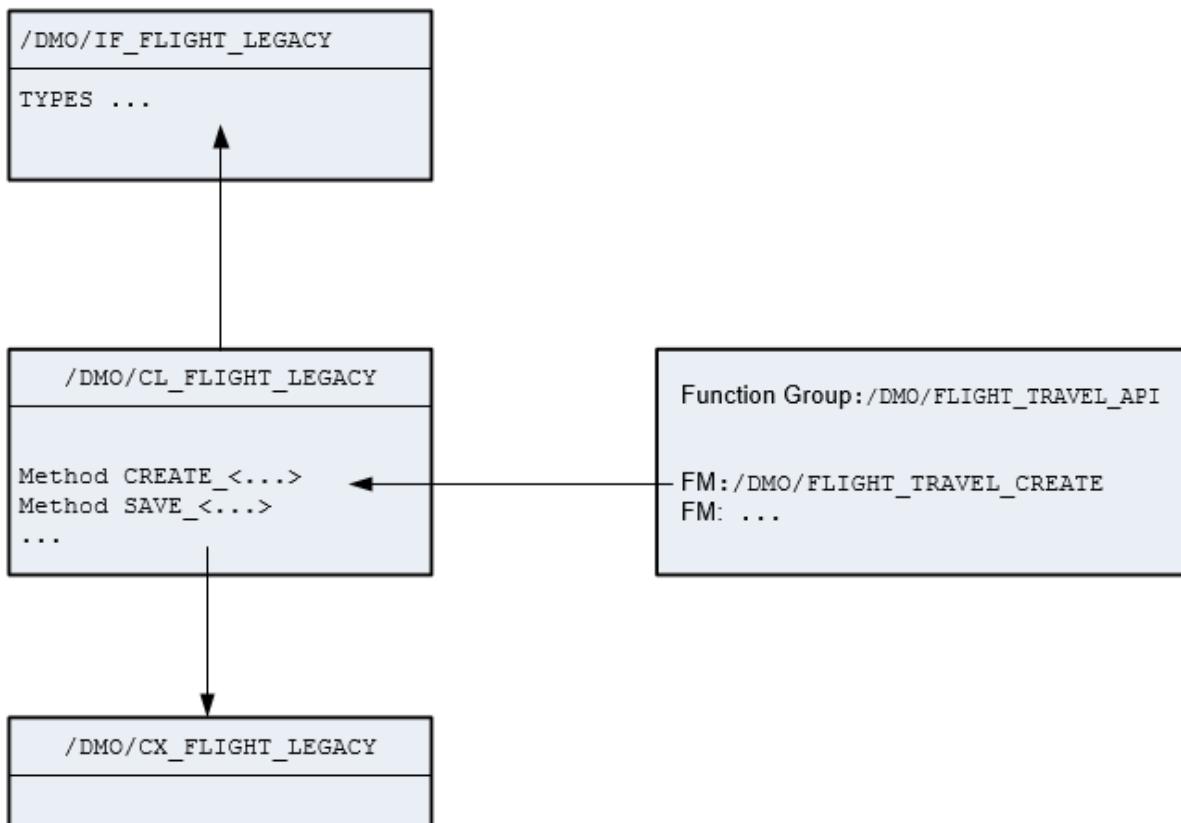


Travel Entities Involved in the Present Scenario and Their Relationships

Business Logic

The following figure summarizes the essential elements of the business logic:

- The function group `/DMO/FLIGHT_TRAVEL_API` is used to group all function modules that represent the application's legacy code.
- The class `/DMO/CL_FLIGHT_LEGACY` provides the actual implementation of the business logic in a more convenient (object-oriented) way.
- The interface `/DMO/IF_FLIGHT_LEGACY` defines global types and constants for reuse.
- Exception class `/DMO/CX_FLIGHT_LEGACY`.



Main Elements of the Legacy Business Logic

5.3.2 Providing CDS Data Model with Business Object Structure

From a structural point of view, a business object consists of a tree of entities that are linked by special associations known as compositions. A composition is a specialized association that defines a whole-part relationship. A composite part only exists together with its parent entity (whole).

i Note

For didactic reasons, we will demonstrate a one-level composition in our sample application, defining a root entity for the Travel BO and one child entity for bookings. This relationship also means that booking data can only be created to a given travel instance.

Every entity in this composition tree is an element that is modeled with a CDS entity. The root entity is of particular importance, since it defines the top node in a business object's structure and serves as a representation of the business object. This is considered in the source code of the CDS data definition with the keyword ROOT when defining the corresponding CDS entity.

Syntax for Defining a Root Entity

```
@AbapCatalog.sqlViewName: 'CDS_DB_VIEW'  
{@view_annotation_1}  
...  
{@view_annotation_n}]  
DEFINE ROOT VIEW root_entity  
[parameter_list]  
AS SELECT FROM data_source [AS alias]  
COMPOSITION [min..max] OF child_entity AS _comp_name  
[additional_composition_list]  
[association_list]  
{  
    element_list  
}
```

Effect:

Using this syntax, you define the `root_entity` as a root of the compositional hierarchy for the business object to be created.

With the keyword `COMPOSITION`, a `child_entity` is defined as a direct child entity to the business object's root. The `child_entity` is a CDS entity, which is the target of the composition. `_comp_name` defines the name of the composition and must be added to the `element_list` (like associations). The cardinality to the child entity is expressed in the composition definition with square brackets `[min .. max]`.

For `min` and `max`, positive integers (including 0) and asterisks (*) can be specified:

- `max` cannot be 0.
- An asterisk * for `max` means any number of rows.
- `min` cannot be *.

The meaning of the other elements of the syntax is identical to that of `DEFINE VIEW`.

Further information: [ABAP CDS - DEFINE VIEW \(ABAP Keyword Documentation\)](#)

Syntax for Defining a Child Entity

```
@AbapCatalog.sqlViewName: 'CDS_DB_VIEW'  
{@view_annotation_1}  
...  
{@view_annotation_n}]  
DEFINE VIEW child_entity  
[parameter_list]  
AS SELECT FROM data_source [AS alias]  
ASSOCIATION TO PARENT parent_entity AS _assoc_name ON condition_exp  
[additional_association_list]  
{  
    element_list  
}
```

Effect:

Using this syntax, you define a CDS entity `child_entity` that serves as a sub node in the compositional hierarchy of the business object structure. The sub node is a node in a business object's structure that is directly connected to another node when moving away from the root.

CDS entities that do not represent the root node of the hierarchy must have an association to their compositional parent entity `parent_entity` or `root_entity`. This relationship is expressed by the keyword `ASSOCIATION TO PARENT...`

The meaning of the other elements in the association syntax is identical to that of `ASSOCIATION` in the CDS `SELECT` statement.

Further information: [ABAP CDS - SELECT, ASSOCIATION \(ABAP Keyword Documentation\)](#))

Next Steps

[Creating Data Definitions for CDS Views \[page 271\]](#)

[Defining the Data Model in CDS Views \[page 273\]](#)

5.3.2.1 Creating Data Definitions for CDS Views

In this step you create a CDS views as the basis for the data model of our demo scenario. To do this, you create the appropriate data definitions as transportable ABAP repository objects, as specified in the table below.

Data Definitions and CDS Views to Create

i Note

Naming CDS views: Since CDS views are (public) interface views, they are prefixed with `I_` in accordance with the VDM (virtual data model) naming convention. In addition, we add the suffix `_U` to the view name in case it is specific for our unmanaged implementation type scenario. For detailed information, see: [Naming Conventions for Development Objects \[page 780\]](#)

→ Remember

The namespace `/DMO/` is reserved for the demo content. Therefore, do not use the namespace `/DMO/` when creating your own development objects and do not create any development objects in the downloaded packages.

Data Definitions Required for the Root Node (Travel):

Data Definition

CDS View Name

Database View Name	Data Source	Description
/DMO/I_TRAVEL_U	/DMO/ TRAVEL	This CDS view defines the root entity. The root entity is a representation of the travel business object and defines the top node in a business object's structure.
/DMO/I_Travel_U	(DB table)	
/DMO/I_TRAVEL_U		It is used for managing general travel data, such as the booking status of a travel or the total price of a travel.
/DMO/I_AGENCY	/DMO/ AGENCY	This CDS view represents the travel agency in the data model of our demo scenario.
/DMO/I_Agency	(DB table)	
/DMO/IAGENCY_RE		
/DMO/I_CUSTOMER	/DMO/ CUSTOMER	This CDS view defines the data model for managing flight travel customers (passengers).
/DMO/I_Customer	(DB table)	
/DMO/ICUSTOM_RE		

Data Definitions Required for the Sub Node (Booking):

Data Definition

CDS View Name

Database View Name	Data Source	Description
/DMO/I_BOOKING_U	/DMO/ BOOKING	This CDS view defines the flight booking entity. The booking entity is a sub node representation of the travel business object structure..
/DMO/I_Booking_U	(DB table)	
/DMO/IBOOKING_U		It is used for managing flight booking data, such as the customer, the flight connection, or the price and flight date.
/DMO/I_FLIGHT	/DMO/ FLIGHT	This CDS view represents the concrete flights in the travel data model. In our demo scenario, the CDS view is used for value help definition for specific elements in the booking view.
/DMO/I_Flight	(DB table)	
/DMO/IFLIGHT_RE		
/DMO/I_CONNECTION	/DMO/ CONNECTION	This CDS view defines the data model for managing flight connections.
/DMO/I_Connection	(DB table)	
/DMO/ICONNECT_RE		In our demo scenario, the connection view is used to retrieve the text information for the associated elements in the booking view.

Data Definition

CDS View Name

Database View Name	Data Source	Description
/DMO/I_CARRIER	/DMO/ CARRIER	This CDS view defines the data model for managing the airline data (ID and the name).
/DMO/I_CARRIER	(DB table)	In our demo scenario, the carrier view is used to retrieve the text information for the associated elements in the booking view.
/DMO/ICARRIER_RE		

Procedure: Creating a Data Definition

To launch the wizard tool for creating a data definition, do the following:

1. Launch [ABAP Development Tools](#).
2. In your ABAP project (or ABAP cloud project), select the relevant package node in [Project Explorer](#).
3. Open the context menu and choose ► [New](#) ► [Other ABAP Repository Object](#) ► [Core Data Services](#) ► [Data Definition](#) ▾

Further information: ([Tool Reference](#))

Results

This procedure creates a data definition as a transportable development object in the selected package. For each data definition, the related CDS view and the corresponding SQL view are created too.

5.3.2.2 Defining the Data Model in CDS Views

Travel Root View /DMO/I_Travel_U

The listing 1 (below) provides you with the implementation of the CDS data model for managing flights, where the database table /dmo/travel serves as the data source for the corresponding CDS view /DMO/I_Travel_U (note the camel case notation).

This CDS view defines the root entity of the data model and represents the root of the compositional hierarchy for the travel business object to be created.

From a structural point of view, a business object consists of a tree of nodes that are linked by special associations known as compositions. To define a composition relationship from the root to a child entity the

keyword COMPOSITION is used. In our example, you specify the /DMO/I_Booking_U as child entity of the composition _Booking. As a result, the booking node is defined as a direct sub node to the business object's root. With cardinality [0 ... *] you express that any number of booking instances can be assigned to each travel instance.

To be able to access business data from semantically related entities, a set of associations is defined in the CDS source code. These associations refer to CDS views that are part of our demo application scenario. Some of these views are used primarily as text views for retrieving text information and as value help provider views for specific UI fields, see [Projecting the Data Model in CDS Projection Views \[page 279\]](#).

Except for the administrative fields createdby, lastchangedby, and createdat, all fields of the data source table /dmo/travel have been added to the element list in the CDS view. The database table provides several administrative fields that are used for administrative data which usually includes the user who created or last changed an instance and the corresponding timestamps. In this example however, the element LastChangedAt plays a special part, as it is used for ETag checks to determine whether two representations of an entity are the same. If the representation of the entity ever changes, a new and different ETag value is assigned. ETAGs play a significant part in the lock lifetime when working with business objects.

For all elements, we can provide an alias to provide a readable name for the elements.

The price elements and the element CurrencyCode have a semantic relationship. This relationship is manifested in the business object view via the @semantics annotations. In a Fiori UI, the amount and the currency value can then be displayed together.

i [Expand the following listing to view the source code.](#)

Listing 1: Source Code of the CDS Root View /DMO/I_Travel_U

```
@AbapCatalog.sqlViewName: '/DMO/ITRAVEL_U'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT REQUIRED
@EndUserText.label: 'Travel view - CDS data model'
define root view /DMO/I_Travel_U
  as select from /dmo/travel as Travel -- the travel table is the data source
  for this view
    composition [0..*] of /DMO/I_Booking_U as _Booking
      association [0..1] to /DMO/I_Agency as _Agency on $projection.AgencyID =
      _Agency.AgencyID
      association [0..1] to /DMO/I_Customer as _Customer on $projection.CustomerID =
      _Customer.CustomerID
      association [0..1] to I_Currency as _Currency on
      $projection.CurrencyCode = _Currency.Currency
    {
      key Travel.travel_id as TravelID,
      Travel.agency_id as AgencyID,
      Travel.customer_id as CustomerID,
      Travel.begin_date as BeginDate,
      Travel.end_date as EndDate,
      @Semantics.amount.currencyCode: 'CurrencyCode'
      Travel.booking_fee as BookingFee,
      @Semantics.amount.currencyCode: 'CurrencyCode'
      Travel.total_price as TotalPrice,
      @Semantics.currencyCode: true
      Travel.currency_code as CurrencyCode,
      Travel.description as Memo,
      Travel.status as Status,
      Travel.lastchangedat as LastChangedAt,
      /* Associations */
      _Booking,
```

```

    _Agency,
    _Customer,
    _Currency
}
```

Booking View /DMO/I_Booking_U

Listing 2 (below) provides you with a data model implementation of the booking entity. In the data definition of the root entity /DMO/I_Travel_U, you specified the booking entity /DMO/I_Booking_U as a child entity. This composition relationship requires an association to their compositional parent entity for the booking child entity to be specified in the data model implementation. This relationship is expressed by the keyword ASSOCIATION TO PARENT. Using this syntax, you define the CDS entity /DMO/I_Booking_U as a direct sub node in the compositional hierarchy of the travel business object structure.

The SELECT list includes all elements of a booking entity that are relevant for building the BO structure. These elements are provided with an alias. Like in the Travel CDS view, the semantic relationship between Price and CurrencyCode is established with the @Semantics annotations.

To be able to access data from other entities, a set of additional associations (_Customer, _Carrier, and _Connection) is defined in the CDS source code. These views are primarily used as text views for retrieving text information and as value help provider views for specific booking fields on the UI, [Projecting the Data Model in CDS Projection Views \[page 279\]](#).

i *Expand the following listing to view the source code.*

Listing 2: CDS View /DMO/I_Booking_U

```

@AbapCatalog.sqlViewName: '/DMO/IBOOKING_U'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Booking view'
define view /DMO/I_Booking_U
  as select from /dmo/booking as Booking
    association to parent /DMO/I_Travel_U      as _Travel      on
    $projection.TravelID = _Travel.TravelID
    association [1..1] to /DMO/I_Customer       as _Customer    on
    $projection.CustomerID = _Customer.CustomerID
    association [1..1] to /DMO/I_Carrier         as _Carrier     on
    $projection.AirlineID = _Carrier.AirlineID
    association [1..1] to /DMO/I_Connection      as _Connection   on
    $projection.AirlineID      = _Connection.AirlineID
                                              and
    $projection.ConnectionID = _Connection.ConnectionID
{
  key Booking.travel_id      as TravelID,
  key Booking.booking_id     as BookingID,
  Booking.booking_date       as BookingDate,
  Booking.customer_id        as CustomerID,
  Booking.carrier_id         as AirlineID,
  Booking.connection_id      as ConnectionID,
  Booking.flight_date        as FlightDate,
  @Semantics.amount.currencyCode: 'CurrencyCode'
  Booking.flight_price       as FlightPrice,
  @Semantics.currencyCode: true
  Booking.currency_code      as CurrencyCode,
  /* Associations */
  _Travel,
```

```

    _Customer,
    _Carrier,
    _Connection
}
```

Travel Agency View /DMO/I_Agency

Listing 3 (below) provides you with a data definition for handling travel agency data. The database table /dmo/agency is the data source for the corresponding CDS view /DMO/I_Agency.

All fields in table /dmo/agency have been added to the element list in the CDS view.

Since the travel agency's data can vary from one country to another, the data model refers to the I_Country view using the association _Country.

This CDS entity also serves as a text provider view. For this purpose, the annotation @Semantics.text: true is used to identify the Name element as a text element, which - in this case - points to a textual description of agency names. In the travel and the booking projection views, the associated text element is added as a field to the referencing entity. At runtime, this field is read from the database and filtered by the logon language of the OData consumer automatically.

In addition, the data model enables search capabilities on the Name element: The annotation @Search.searchable: true marks the CDS view as searchable, whereas @Search.defaultSearchElement: true specifies that the annotated Name element is to be considered in a full-text search. For detailed information, see: [Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#)

i *Expand the following listing to view the source code.*

Listing 3: Agency CDS View /DMO/I_Agency

```

@AbapCatalog.sqlViewName: '/DMO/IAGENCY_RE'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Agency view - CDS data model'
@Search.searchable: true
define view /DMO/I_Agency
  as select from /dmo/agency as Agency -- the agency table serves as the data
  source for this view
  association [0..1] to I_Country as _Country on $projection.CountryCode =
  _Country.Country
{
  key Agency.agency_id           as AgencyID,
  @Semantics.text: true
  @Search.defaultSearchElement: true
  @Search.fuzzinessThreshold: 0.8
  Agency.name                   as Name,
  Agency.street                  as Street,
  Agency.postal_code             as PostalCode,
  Agency.city                    as City,
  Agency.country_code            as CountryCode,
  Agency.phone_number            as PhoneNumber,
  Agency.email_address           as EMailAddress,
  Agency.web_address              as WebAddress,
  /* Associations */
  _Country
}
```

Customer View /DMO/I_Customer

Listing 4 (below) is used as a data model implementation for managing passenger data. The database table /dmo/customer serves as the data source for the corresponding CDS view /DMO/I_Customer.

Except for the administrative fields (createdby, createdat, lastchangedat and lastchangedby), all fields of the table /dmo/customer have been added to the element list in the CDS view.

Since a passenger's data can vary from one country to another, the data model refers to the I_Country view using a corresponding association _Country.

The annotation @Semantics.text: true is added to the LastName element. This element serves as a text element, which - in this case - points to texts with customer names. This text annotation allows you to use this customer view as a text provider for the associated elements in the target views /DMO/I_Travel_U and /DMO/I_Booking_U.

i *Expand the following listing to view the source code.*

Listing 4: Customer CDS view /DMO/I_Customer

```
@AbapCatalog.sqlViewName: '/DMO/ICUSTOM_RE'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Customer view - CDS data model'
@Search.searchable: true
define view /DMO/I_Customer
  as select from /dmo/customer as Customer -- the customer table serves as the
  data source
  association [0..1] to I_Country as _Country on $projection.CountryCode =
    Country.Country
{
  key Customer.customer_id      as CustomerID,
  Customer.first_name          as FirstName,
  @Semantics.text: true
  @Search.defaultSearchElement: true
  @Search.fuzzinessThreshold: 0.8
  Customer.last_name           as LastName,
  Customer.title               as Title,
  Customer.street              as Street,
  Customer.postal_code         as PostalCode,
  Customer.city                as City,
  Customer.country_code        as CountryCode,
  Customer.phone_number        as PhoneNumber,
  Customer.email_address       as EMailAddress,
  /* Associations */
  _Country
}
```

Flight View /DMO/I_Flight

Listing 5 (below) provides you with a data definition for flights. The data of specific flights is stored in the database table /dmo/flight, which serves as the data source for the corresponding CDS view /DMO/I_Flight.

As demonstrated in listing 2 (above), you can implement the value help with additional binding. To define filter conditions for the value help based on the same value help provider, the flight view /DMO/I_Flight is used to filter the value help result list for the annotated elements.

i [Expand the following listing to view the source code.](#)

Listing 5: CDS View /DMO/I_Flight

```
@AbapCatalog.sqlViewName: '/DMO/IFLIGHT_RE'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Flight view'
define view /DMO/I_Flight as select from /dmo/flight as Flight
{

    key Flight.carrier_id           as AirlineID,
    key Flight.connection_id        as ConnectionID,
    key Flight.flight_date          as FlightDate,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    Flight.price                   as Price,
    @Semantics.currencyCode: true
    Flight.currency_code            as CurrencyCode,
    Flight.plane_type_id           as PlaneType,
    Flight.seats_max               as MaximumSeats,
    Flight.seats_occupied          as OccupiedSeats
}
```

Flight Connections View /DMO/I_Connection

Listing 6 (below) is used as a data definition for flight connections. The flight connections are stored in the database table /dmo/connection, which serves as the data source for the corresponding CDS view /DMO/I_Connection.

Except for the administrative fields, all fields in the table /dmo/connection have been added to the element list in the CDS view.

Using the annotation @Semantics.unitOfMeasure, the DistanceUnit element is tagged as an element containing a unit of measure for the value stored in the element Distance. The corresponding unit of measure is contained in the referenced Distance field.

i [Expand the following listing to view the source code.](#)

Listing 6: CDS View /DMO/I_Connection

```
@AbapCatalog.sqlViewName: '/DMO/ICONNECT_RE'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
```

```

@EndUserText.label: 'Connection view'
define view /DMO/I_Connection
  as select from /dmo/connection as Connection
{
  key Connection.carrier_id           as AirlineID,
  key Connection.connection_id        as ConnectionID,
  Connection.airport_from_id         as DepartureAirport,
  Connection.airport_to_id           as DestinationAirport,
  Connection.departure_time          as DepartureTime,
  Connection.arrival_time            as ArrivalTime,
  @Semantics.quantity.unitOfMeasure: 'DistanceUnit'
  Connection.distance                as Distance,
  @Semantics.unitOfMeasure: true
  Connection.distance_unit           as DistanceUnit
}

```

Carrier View /DMO/I_Carrier

The following data definition for the carrier CDS entity provides you with IDs and names of airlines that are stored in the database table /DMO/CARRIER.

This CDS entity mainly serves as a text provider view. It provides text data through text associations as defined in the travel and the booking views (Listing 1 and Listing 2).

For this purpose, a text annotation is required at element level in order to annotate the text elements from the view's element list: In this example, the Name element is identified as the text element.

i [Expand the following listing to view the source code.](#)

Listing 7: CDS View /DMO/I_Carrier

```

@AbapCatalog.sqlViewName: '/DMO/ICARRIER_RE'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Carrier view'
define view /DMO/I_Carrier as select from /dmo/carrier as Airline
{
  key Airline.carrier_id           as AirlineID,
  @Semantics.text: true
  Airline.name                     as Name,
  @Semantics.currencyCode: true
  Airline.currency_code             as CurrencyCode
}

```

5.3.2.3 Projecting the Data Model in CDS Projection Views

You use CDS projection views to expose the general data model for a specific business service.

Whereas the general business object defines all available entities and elements, the [Business Object Projection \[page 100\]](#) defines those entities and elements that are used for a specific service, in our case a service that is

exposed for UI consumption. That means, in the projection layer, we define the data model and its functionality that is needed for the use case of a UI service of the travel scenario. UI services include:

- value helps
- text elements
- search
- UI layout annotations

These UI enrichments are modeled via [CDS annotations \[page 809\]](#) in the [CDS projection view \[page 810\]](#). It is best practice, to outsource the annotations related to UI layout in [metadata extensions \[page 809\]](#), as these annotations can easily overload the projection view. A CDS metadata extension is always assigned to a layer such as industry, partner or customer to easily extend the scope of the metadata extension.

For more information, see [Business Object Projection \[page 100\]](#) and .

Creating CDS Projection Views

For the Travel UI service, create projection views for those CDS entities that are part of the compositional hierarchy of the business object. These are:

	root layer	1st child layer
CDS views for BO structure	/DMO/ I_TRAVEL_U	/DMO/ I_BOOKING_U
CDS views for BO projection	/DMO/ C_TRAVEL_U	/DMO/ C_BOOKING_U

i Note

The names are assigned according to the naming conventions for projection views: [Naming Conventions for Development Objects \[page 780\]](#).

For more information, see [Creating Projection Views \[page 773\]](#) (tool reference).

The resulting CDS projection views must have the following syntax:

```
define [root] view entity <projection_view> as projection on <projected_view>
```

For more information about the syntax in projection views, see [Syntax for CDS Projection Views \[page 239\]](#).

Data Modeling for the Business Object Projection

The unmanaged scenario provides just one projection use case that reuses all elements that are defined in the business object views. In the projection layer the data model is enriched with UI functionality that is implemented via CDS annotations or keywords in the projection view and manifested in the service metadata.

Root Projection View /DMO/C_Travel_U

The listing 1 (below) provides you with the implementation of the CDS view projection, where the CDS view /DMO/I_Travel_U serves as the projection source for the corresponding CDS projection view /DMO/C_Travel_U. It projects every element from the underlying view.

The Travel projection view also exposes all associations that are defined in the underlying travel view /DMO/I_Travel_U. However, the composition to the child /DMO/I_Booking_U must be redirected to the newly created projection counterpart. The syntax for defining redirected compositions is described in [CDS Projection View \[page 103\]](#).

To define a relationship between the elements `AgencyID` and `CustomerID` and their corresponding texts or descriptions, the text elements must be denormalized in the CDS projection view. Therefore the elements of the associated text provider views (`_Agency.Name` and `_Customer.LastName`) are included in the `select` list of the projection view. Their corresponding ID elements are annotated with `@ObjectModel.text.element`. At runtime, the referenced element is read from the database and filtered by the logon language of the OData consumer automatically. For detailed information, see: [Defining Text Elements \[page 422\]](#)

Value helps are defined in the source code of the CDS projection view /DMO/C_Travel_U by adding the annotation `@Consumption.valueHelpDefinition` to the elements `AgencyID`, `CustomerID` and `CurrencyCode`. In this annotation, you specify the elements for which the value help dialog should appear on the UI. The value help annotation allows you to reference the value help provider view without implementing an association. You simply assign a CDS entity as the value help provider and specify an element for the mapping in the annotation. All fields of the value help provider are displayed on the UI. When the end user chooses one of the entries of the value help provider, the value of the referenced element is transferred to the corresponding input field on the UI. For detailed information, see: [Simple Value Help \[page 430\]](#)

i Note

For the default implementation of value help, note that you can reuse any CDS entity that contains the required values of the element that corresponds to the input field on the UI. You do not need to explicitly define a CDS entity as the value help provider

In the projection view, you model the ability to search for specific values in the view. The entity annotation `@Search.searchable` is used in the travel projection to enable the general HANA search. This annotation also triggers the search bar in the Fiori elements UI. By using this annotation, you have to define elements that are primarily used as the search target for a free text search. These elements are annotated with `@Search.defaultSearchElement`. In addition, you can define a fuzziness threshold that defines the how exact the search values must be to be able to find element values. For more information on search, see [Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#).

The listing below does not include annotations that define the UI layout. They are outsourced to metadata extensions, see [Adding UI Metadata to the Data Model \[page 284\]](#).

i [Expand the following listing to view the source code.](#)

Listing 1: Source Code of the CDS Root Projection View /DMO/C_Travel_U

```
@EndUserText.label: 'Travel Projection View'  
@AccessControl.authorizationCheck: #NOT_REQUIRED  
@Metadata.allowExtensions: true  
@Search.searchable: true
```

```

define root view entity /DMO/C_Travel_U
  as projection on /DMO/I_Travel_U
{
  ///DMO/I_Travel_U
  key TravelID,
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/I_Agency',
                                                    element: 'AgencyID' } }]
    @ObjectModel.text.element: ['AgencyName']
    @Search.defaultSearchElement: true
  AgencyID,
    _Agency.Name           as AgencyName,
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/I_Customer',
                                                    element: 'CustomerID' } }]
    @ObjectModel.text.element: ['CustomerName']
    @Search.defaultSearchElement: true
  CustomerID,
    _Customer.LastName     as CustomerName,
  BeginDate,
  EndDate,
  BookingFee,
  TotalPrice,
  @Consumption.valueHelpDefinition: [{entity: { name: 'I_Currency',
                                                element: 'Currency' } }]
  CurrencyCode,
  Memo,
  Status,
  LastChangedAt,
  /* Associations */
  ///DMO/I_Travel_U
  _Booking : redirected to composition child /DMO/C_Booking_U,
  _Agency,
  _Currency,
  _Customer
}

```

Child Projection View /DMO/C_Booking_U

The listing 2 (below) provides you with the implementation of the CDS view projection, where the CDS view /DMO/I_Booking_U serves as the projection source for the corresponding CDS projection view /DMO/C_Booking_U. It projects every element from the underlying view.

The Booking projection view also exposes all associations that are defined in the underlying booking view /DMO/I_Travel_U. However, the composition to the parent /DMO/I_Travel_U must be redirected to the newly created projection counterparts. The syntax for defining redirected compositions is described in [CDS Projection View \[page 103\]](#).

To access the corresponding texts or descriptions, the relationship between the elements `AirlineID` and `ConnectionID` and the text elements in the associated text provider views /DMO/I_Carrier and /DMO/I_Connection are used in this example. Their corresponding ID elements are annotated with `@ObjectModel.text.element`. At runtime, the referenced element is read from the database and filtered by the logon language of the OData consumer automatically. For detailed information, see: [Defining Text Elements \[page 422\]](#)

Value helps are defined in the source code of the booking entity /DMO/C_Booking_U by adding the annotation `@Consumption.valueHelpDefinition` to the relevant elements. You simply assign a CDS entity as the value help provider to the elements `CustomerID`, `AirlineID`, and `CurrencyCode`, and specify an element for the mapping in the annotation. This simple value help approach is convenient if you only want to display values from the value help provider view for an input field. In this case, the annotation defines the binding to the value

help providing entity. You only have to specify the entity name and the element providing the possible values for the annotated element. For detailed information, see: [Simple Value Help \[page 430\]](#)

Listing 2 also demonstrates how you can implement the value help with additional binding, which defines a filter condition. Different filter conditions for the value help on the same value provider entity /DMO/I_Flight are defined for filtering the value help result list for the elements ConnectionID and FlightDate. For detailed information, see: [Value Help with Additional Binding \[page 435\]](#)

The elements in the booking projection view are also searchable in a UI service. The annotations related to search are therefore also maintained in the booking view on entity level and on certain elements to mark them as primary search element. For more information on search, see [Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#).

The listing below does not include annotations that define the UI layout. They are outsourced to metadata extensions, see [Adding UI Metadata to the Data Model \[page 284\]](#).

i [Expand the following listing to view the source code.](#)

Listing 2: Source Code of the CDS Projection View /DMO/C_Booking_U

```
@EndUserText.label: 'Booking Projection View'
@AccessControl.authorizationCheck: #NOT_REQUIRED
@Metadata.allowExtensions: true
@Search.searchable: true
define view entity /DMO/C_Booking_U
    as projection on /DMO/I_Booking_U
{
    // /DMO/I_Booking_U
    @Search.defaultSearchElement: true
    key TravelID,
        @Search.defaultSearchElement: true
    key BookingID,
        BookingDate,
        @Consumption.valueHelpDefinition: [ { entity: { name: '/DMO/I_Customer', element: 'CustomerID' } } ]
        @Search.defaultSearchElement: true
        @ObjectModel.text.element: ['CustomerName']
        CustomerID,
        _Customer.LastName as CustomerName,
        @Consumption.valueHelpDefinition: [ { entity: { name: '/DMO/I_Carrier', element: 'AirlineID' } } ]
        @ObjectModel.text.element: ['AirlineName']
        AirlineID,
        _Carrier.Name as AirlineName,
        @Consumption.valueHelpDefinition: [ { entity: { name: '/DMO/I_Flight', element: 'ConnectionID' }, additionalBinding: [ { localElement: 'FlightDate', element: 'FlightDate' }, { localElement: 'AirlineID', element: 'AirlineID' }, { localElement: 'FlightPrice', element: 'Price' }, { localElement: 'CurrencyCode', element: 'CurrencyCode' } ] } ]
        ConnectionID,
            @Consumption.valueHelpDefinition: [ { entity: { name: '/DMO/I_Flight', element: 'FlightDate' }, additionalBinding: [ { localElement: 'ConnectionID', element: 'ConnectionID' }, { localElement: 'AirlineID', element: 'AirlineID' }, { localElement: 'FlightPrice', element: 'Price' } ] } ]
```

```

    { localElement:
'CurrencyCode', element: 'CurrencyCode' } ] } ]
  FlightDate,
  FlightPrice,
  @Consumption.valueHelpDefinition: [ {entity: { name:      'I_Currency',
                                                 element: 'Currency' } } ]
  CurrencyCode,
  /* Associations */
  // /DMO/I_Booking_U
  _Travel: redirected to parent /DMO/C_Travel_U,
  _BookSupplement: redirected to composition child /DMO/
C_BookingSupplement_U,
  _Carrier,
  _Connection,
  _Customer
}

```

The other CDS views that are relevant for text provisioning and value helps to complete the data model structure do not have to be projected as they are not part of the business object.

Related Information

[Business Object Projection \[page 100\]](#)

5.3.2.3.1 Adding UI Metadata to the Data Model

Use @UI annotations to define the layout of the Fiori UI.

To enable the business service to be consumable by any UI client, UI relevant metadata are added to the backend service. These metadata are maintained with @UI annotations that are either added to the whole entity or, if it relates to a specific UI element, to the corresponding CDS element. Since these annotations can become excessive, it is recommended to maintain the UI annotations in metadata extensions.

Metadata extensions are used to define CDS annotations for a CDS view outside of the corresponding data definition. The use of metadata extensions allows the separation of concerns by separating the data model from domain-specific semantics, such as UI-related information for UI consumption. A CDS metadata extension is always assigned to a specific layer such as core, industry, partner or customer. These industries can extend the metadata for the data model.

Creating Metadata Extension for the Projection Views

For the Travel UI service, create metadata extensions for the CDS projection views that are part of the compositional hierarchy of the business object. These are:

root layer	1st child layer
------------	-----------------

CDS views for BO structure	/ DMO/ I_TRAVEL_U	/ DMO/ I_BOOKING_U
CDS views for BO projection	/ DMO/ C_TRAVEL_U	/ DMO/ C_BOOKING_U
Metadata extension for UI annotations	/ DMO/ C_TRAVEL_U	/ DMO/ C_BOOKING_U

i Note

The names of metadata extensions are the same as their related CDS entities, according to the naming conventions for metadata extensions: [Naming Conventions for Development Objects \[page 780\]](#).

To create a metadata extension object, open the context menu in the project explorer of the related CDS entity and choose [New Metadata Extension](#). Follow the steps in the wizard to get a metadata extension template that annotates the related projection view.

Syntax:

```
@Metadata.layer: layer
annotate view CDSProjectionView
    with
{
    element_name;
}
```

Define a metadata layer for your metadata extension and insert the elements that you want to annotate. The layering of metadata extension allows a stacking for each development layer, beginning with #CORE. Each layer can then add its own layer specific metadata to the service.

To enable the usage of metadata extensions for the projection views, add the annotation `@Metadata.allowExtensions:true` on entity level in your projection view. For more information, see .

Annotating CDS Elements in Metadata Extensions

Any CDS element that you want to annotate in metadata extensions must be inserted in the element list.

Metadata Extension for CDS Projection View /DMO/C_Travel_U

The listing below (listing 1) provides you with the implementation of the metadata extension `/DMO/C_Travel_U` for the projection view with the same name. It annotates the projection view itself and its elements.

To specify the header texts for Fiori UIs, the annotation `@UI.headerInfo` is used at entity level. It specifies the list title as well as the title for the object page. The main building blocks of the UI are specified as UI facets. The annotation `@UI.facet` also enables the navigation to the child entity, which is represented as list in the page body of the object page.

The annotations on element level are used to define the position of the element in the list report and on the object page. In addition, they specify the importance of the element for the list report. If elements are marked with low importance, they are not shown on a device with a narrow screen. The selection fields are also defined on the elements that require a filter bar in the UI.

For actions, you can define an action button on the Fiori Elements UI. This scenario contains the action to set the travel status to booked. To expose the action on the list report page, the information for the action button is added to the @UI.lineItem annotation on an element.

Syntax to expose an action button on the UI:

```
@UI: { lineItem: { type: #FOR_ACTION, dataAction: 'action_name', label: 'Button Label' } ] }
```

i [Expand the following listing to view the source code.](#)

Listing 1: Source Code of the Metadata Extension /DMO/C_Travel_U

```
@Metadata.layer: #CORE
@UI: { headerInfo: { typeName: 'Travel',
                     typeNamePlural: 'Travels',
                     title: { type: #STANDARD,
                               value: 'TravelID' } } }

annotate view /DMO/C_Travel_U with
{
  @UI.facet: [ { id: 'Travel',
                  purpose: #STANDARD,
                  type: #IDENTIFICATION_REFERENCE,
                  label: 'Travel',
                  position: 10 },
               { id: 'Booking',
                  purpose: #STANDARD,
                  type: #LINEITEM_REFERENCE,
                  label: 'Booking',
                  position: 20,
                  targetElement: '_Booking'}]

  @UI: { lineItem: [ { position: 10,
                      importance: #HIGH } ],
         identification: [ { position: 10 } ],
         selectionField: [ { position: 10 } ] }
  TravelID;
  @UI: { lineItem: [ { position: 20,
                      importance: #HIGH } ],
         identification: [ { position: 20 } ],
         selectionField: [ { position: 20 } ] }
  AgencyID;

  @UI: { lineItem: [ { position: 30,
                      importance: #HIGH } ],
         identification: [ { position: 30 } ],
         selectionField: [ { position: 30 } ] }
  CustomerID;

  @UI: { lineItem: [ { position: 40,
                      importance: #MEDIUM } ],
         identification: [ { position: 40 } ] }
  BeginDate;

  @UI: { lineItem: [ { position: 41,
                      importance: #MEDIUM } ],
         identification: [ { position: 41 } ] }
```

```

EndDate;
@UI: { identification: [ { position: 42 } ] }
BookingFee;
@UI: { identification: [ { position: 43 } ] }
TotalPrice;
@UI: { identification:[ { position: 45,
                           label: 'Comment' } ] }
Memo;
@UI: { lineItem: [ { position: 50,
                     importance: #HIGH },
                  { type: #FOR_ACTION,
                    dataAction: 'set_status_booked',
                    label: 'Set to Booked' } ] }
Status;
}

```

Metadata Extension for CDS Projection View /DMO/C_Booking_U

The listing below (listing 2) provides you with the implementation of the metadata extension /DMO/C_Booking_U for the projection view with the same name. It annotates the projection view itself and its elements.

To specify the header texts for Fiori UIs, the annotation @UI.headerInfo is used at entity level. It specifies the list title as well as the title for the object page. The main building blocks of the UI are specified as UI facets.

The annotations on element level are used to define the position of the element in the list report and on the object page. In addition, they specify the importance of the element for the list report. The selection fields are also defined on the elements that require a filter bar in the UI.

i [Expand the following listing to view the source code.](#)

Listing 2: Source Code of the Metadata Extension /DMO/C_Booking_U

```

@Metadata.layer: #CORE
@UI: {
  headerInfo: { typeName: 'Booking',
                typeNamePlural: 'Bookings',
                title: { type: #STANDARD,
                          value: 'BookingID' } } }
annotate view /DMO/C_Booking_U with
{
  @UI.facet: [ { id:                 'Booking',
                  purpose:          #STANDARD,
                  type:              #IDENTIFICATION_REFERENCE,
                  label:             'Booking',
                  position:          10 } ]
  @UI: { lineItem:      [ { position: 20,
                           importance: #HIGH } ],
         identification: [ { position: 20 } ] }
BookingID;
  @UI: { lineItem:      [ { position: 30,
                           importance: #HIGH } ],
         identification: [ { position: 30 } ] }
BookingDate;
  @UI: { lineItem:      [ { position: 40,
                           importance: #HIGH } ],
         identification: [ { position: 40 } ] }

```

```

        identification: [ { position: 40 } ] }
CustomerID;
@UI: { lineItem:      [ { position: 50,
                           importance: #HIGH } ],
       identification: [ { position: 50 } ] }
AirlineID;
@UI: { lineItem:      [ { position: 60,
                           importance: #HIGH } ],
       identification: [ { position: 60 } ] }
ConnectionID;
@UI: { lineItem:      [ { position: 70,
                           importance: #HIGH } ],
       identification: [ { position: 70 } ] }
FlightDate;
@UI: { lineItem:      [ { position: 80,
                           importance: #HIGH } ],
       identification: [ { position: 80 } ] }
FlightPrice;
}

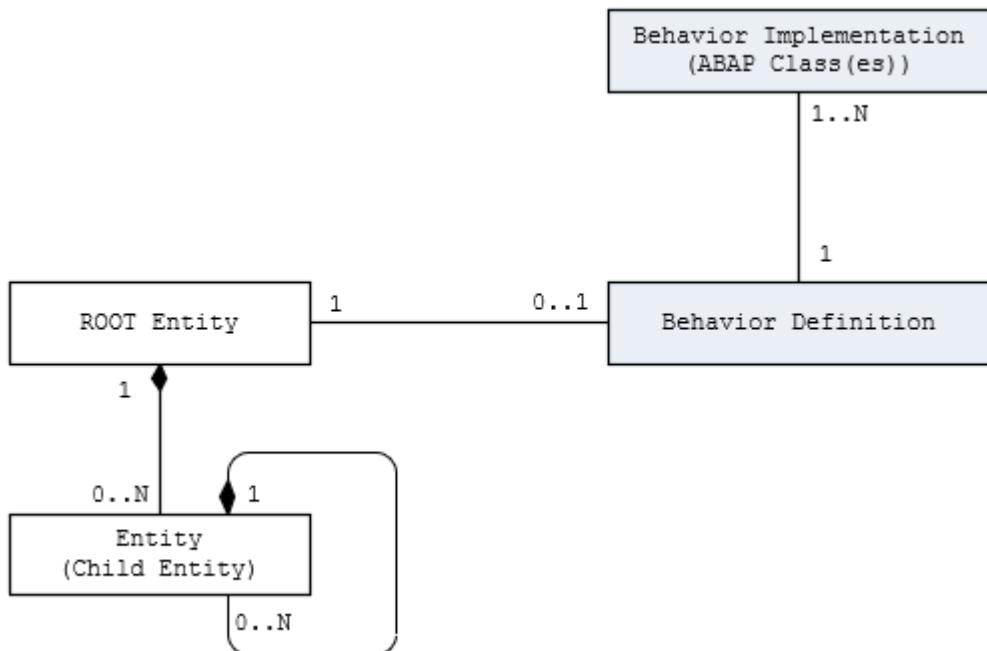
```

5.3.3 Defining and Implementing Behavior of the Business Object

Behavior of a Business Object

To specify the business object's behavior, the behavior definition as the corresponding development object is used. A business object behavior definition (behavior definition for short) is an ABAP Repository object that describes the behavior of a business object in the context of the ABAP RESTful application programming model. A behavior definition is defined using the Behavior Definition Language (BDL).

A behavior definition always refers to a CDS data model. As shown in the figure below, a behavior definition relies directly on the CDS root entity. One behavior definition refers exactly to one root entity and one CDS root entity has at most one behavior definition (a 0..1 cardinality), which also handles all included child entities that are included in the composition tree. The implementation of a behavior definition can be done in a single ABAP class (behavior pool) or can be split between an arbitrary set of ABAP classes (behavior pools). The application developer can assign any number of behavior pools to a behavior definition (1..N cardinality).



Relationship Between the CDS Entities and the Business Object Behavior

Overview of Steps

1. Create the Behavior Definition Object
2. Model the Behavior for Managing Travels
3. Create the Behavior Pool
4. Implement the Transactional Behavior of the Travel Business Object

Related Information

[Business Object \[page 55\]](#)

[Business Service \[page 98\]](#)

5.3.3.1 Adding Behavior to the Business Object

Procedure: Creating a Behavior Definition /DMO/I_TRAVEL_U

To launch the wizard tool for creating a behavior definition, do the following:

1. Launch ABAP Development Tools.
2. In the *Project Explorer* view of your ABAP project (or *ABAP Cloud Project*), select the node for the data definition that defines the root entity (/DMO/I_TRAVEL_U).
3. Open the context menu and choose *New Behavior Definition* to launch the creation wizard.
4. In the creation wizard, choose implementation type *unmanaged* to get the template for the unmanaged behavior definition.

Further information: ([Tool Reference](#)) [page 761]

→ Remember

By creating a behavior definition, the referenced root entity and its compositions (child entities) gain a transactional character. The behavior definition is hence the implementation of the BO concept within the context of the current programming model. All supported transactional operations of a concrete business object must be specified in the same behavior definition.

Parenthesis: Syntax for Defining Transactional Behavior

The syntax of the *Behavior Definition Language (BDL)* is oriented to the *Data Definition Language (DDL)* used to define CDS entities (camel-case notation). Technically, the respective artifacts differ substantially: behavior definitions are managed in the ABAP compiler and not in ABAP Dictionary.

You use the following syntax to define the transactional behavior for a CDSEntity.

```
/* Header of behavior definition */
[implementation] {unmanaged | managed | abstract};
/* Definition of entity behavior */
define behavior for CDSEntity [alias AliasName]
/* Entity properties */
[implementation in class CLASS_NAME unique]
[persistent table DB_TABLE]
[late numbering]
[etag {master Field | dependent by _Association}]
[lock {master Field | dependent by _Association}]
[authorization {master(instance) | dependent by _Association}]
{
  /* Static field control */
  [field (readonly | mandatory) field1[, field2, ..., fieldn];
  /* Standard operations */
  [internal] create;
  [internal] update;
  [internal] delete;
  /* Actions */
  [internal] [static] [factory] action ActionName;

  /* Associations */
  association AssociationName [abbreviation AbbreviationName] {[create;]}
  /* Mapping CDS view fields to db fields */
  mapping for DB_TABLE
  { CDSViewField1 = db_field1;
    CDSViewField2 = db_field2;
    ...
    CDSViewFieldn = db_fieldn; }
```

Explanation

A behavior definition consists of a header information and a set of definitions for entity behavior. Each entity of the composition tree can be referred in the behavior definition at most once.

→ Remember

Consider that if an entity does not occur in the behavior definition, then it would not be modifiable within the ABAP RESTful application programming model.

Within BDL source code, double slashes (//) introduce a comment that continues until the end of the line. Comments that span lines have the form: /* . . . */.

i Note

Keywords within the BDL source code are case-sensitive!

The header specifies the implementation type of the business object provider:

Implementation Type

Values	Effect
unmanaged	<p>For this implementation type, the application developer must implement essential components of the REST contract itself. In this case, all required operations (create, update, delete, or any application-specific actions) must be specified in the corresponding behavior definition before they are implemented manually in ABAP.</p> <p>Use this implementation type when developing transactional apps that are based on existing legacy business logic.</p>
managed	<p>When using this implementation type, the behavior definition is already sufficient to produce a ready-to-run business object.</p> <p>Use this implementation type if you want to develop transactional apps from scratch.</p>
abstract	You cannot use the <i>BO Provider API</i> to implement the behavior definition. An abstract behavior definition is only a metadata artifact for the representation of external services.

The behavior description is divided into a section with entity properties, followed by information on any operations enclosed in the brackets { ... }.

The `AliasName` defined in the behavior definition for `CDSEntity` gives you the option of introducing a more concise name than the entity name that is hence easier to read. The `AliasName` becomes visible in the implementation part BO provider (method syntax of the BO Provider API). The length of the `AliasName` is restricted to 20 characters.

The BDL allows you to add the following properties to a behavior definition:

Behavior Characteristic

Characteristic	Meaning
implementation in class ...unique	<p>In the behavior definition for an individual entity, you have the option of assigning a specific behavior pool that only implements the behavior for this entity. Behavior for the entity in question can only be implemented in a behavior pool with the specified name. Any other class that attempts this raises an ABAP compiler error.</p> <p>By including the restriction <code>implementation in class ... unique</code> in the behavior implementation, you can protect the application against multiple implementations so that each operation can only be implemented once for the relevant entity.</p>
persistent table ...	<p>In managed implementation type, this property specifies the database table for storing CDSEntity data changes that result from transactional behavior.</p>
late numbering	<p>Newly created entity instances are given a definitive (final) key just before they are persisted on the database (when saving the object's data). Until then, the business logic works with a temporary key (for example: \$00000001) which must be replaced upon saving the data.</p> <p>For providing late numbering, the <code>adjust_numbers()</code> method from the save sequence is used. If you redefine the respective method in the saver class, the runtime will call this method.</p> <p>Late numbering plays a role whenever it is of business-critical importance to draw gap-free numbers.</p>
i Note	<p>The current version of the ABAP RESTful Programming Model does not support late numbering.</p>
etag master / etag dependent by	<p>An ETag can be used for optimistic concurrency control in the OData protocol to help prevent simultaneous updates of a resource from overwriting each other.</p> <p>For more information, see Optimistic Concurrency Control [page 87].</p>
lock master / lock dependent by	<p>In the behavior definition, you can determine which entities support direct locking (<code>lock master</code>) and which entities depend on the locking status of a parent or root entity (<code>lock dependent</code>). For lock dependents it is required to specify which property in the child entity (<code>lock dependent</code>) refers to which property of the parent or root entity.</p>
i Note	<p>The definition of <code>lock master</code> is currently only supported for root nodes of business objects.</p>
For more information, see Pessimistic Concurrency Control (Locking) [page 91] .	

Field Properties

Property	Meaning
field (read only)	Defines that the specified fields must not be created or updated by the consumer. The BO runtime rejects modifying requests when creating or updating the specified fields.
field (mandatory)	Defines that the specified fields are mandatory. The specified fields must be filled by the consumer when executing modifying requests.

Standard Operations:

An important part of the transactional behavior of a business object are the **standard operations** `create`, `update` and `delete` (CUD). Whenever an entity can be created, updated, or deleted, these operations must be declared in the behavior definition.

To only implement an operation without exposing it to consumers, the option `internal` can be set before the operation, for example `internal update`.

Actions

Actions can be specified as non-standard operations in behavior definitions. For more information, see [Actions \[page 73\]](#).

Compositions

All **compositions** that form the business object's structure must also be declared in the behavior definition as associations. An abbreviation `AbbreviationName` needs to be defined if the composition name in the CDS view is longer than 11 characters. The keyword `{create;}` is used to declare that the association is create-enabled, which means that instances of the associated entity can be created by the source of the association .

Mapping

The keyword `mapping for` defines the mapping contract between database table fields and CDS view fields. This mapping contract solves the discrepancy between the names of database table fields and CDS view fields to facilitate writing records to the database table at runtime. Especially database tables that originate in the legacy application might contain quite short or cryptic field names. With the mapping specification in the behavior definition you can choose the names in the CDS data model independently from the names in the database tables.

Procedure: Defining the Transactional Behavior of the TRAVEL Business Object

As a quick glance shows you, the behavior definition looks quite easy in our case (see listing below).

It consists of a header information and two definitions for entity behavior: one for the root entity and one for the child entity – corresponding to the composition tree of the business object. Note that for each entity of the composition tree, the transactional behavior can be defined in the behavior definition at most once. All supported transactional operations of a concrete business object's node must be specified in the same behavior definition (that is introduced by the keyword `DEFINE BEHAVIOR FOR ...`).

As expected, the header specifies the **unmanaged** implementation type of our business object's contract provider since we are going to integrate the legacy business logic in the new app. For this implementation type,

you as application developer must implement the essential components of the business object's itself. In this case, you must specify all required operations (create, update, delete, or any application-specific actions) in the corresponding behavior definition and implement them manually in ABAP.

Our TRAVEL business object refers to the underlying CDS data model, which is represented by root entity /DMO/I_Travel_U. Behavior for the root entity can only be implemented in the specified behavior pool /DMO/BP_TRAVEL_U.

Static field control is defined in the behavior definition. In our scenario, the value for the key field TravelID is drawn by the function module for creating travel instances. Thus, the field must not be editable by the end user on the UI. Likewise, the field TotalPrice is set to read only. The total price is calculated by the price of the associated bookings and the booking fee in the function module. In this scenario the total price is not editable by the end user.

Mandatory fields are AgencyID, CustomerID, BeginDate and EndDate. These fields contain mandatory information for a travel instance.

The transactional handling of the business object's root node is determined by the standard operations create, update, and delete, and an instance-related action set_status_booked. Using this action, the end user is able to set the status of selected travel instances to booked. The action in our example affects the output instances with the same entity type and one input instance is related to exactly one output instance. Therefore, the output parameter is defined with the predefined type \$self and the cardinality [1]. The fact that in our scenario new instances of the booking sub node can only be created for a specific travel instance is considered by the addition of the _Booking association. The keyword {create;} declares that this association is create-enabled what exactly means that instances of the associated bookings can be created by a travel instance.

Dynamic operation control is defined for the create by association operation. That means, new bookings can only be created if the corresponding travel instance is not set to booked.

The names of the database table fields and the names of the CDS data model names differ. That is why, we specify the mapping contract for every field in the CDS data model. To map the control structure accordingly, the control structure /dmo/s_booking_intx is defined for the database table.

The sub node of TRAVEL business object refers to the corresponding data model for bookings that is represented by the child entity for /DMO/I_Booking_U. Behavior for the child entity can only be implemented in the specified behavior pool /DMO/BP_BOOKING_U. The transactional handling of the booking sub node of TRAVEL business object is determined by the standard operations update and delete. The creation of booking instances is handled by the create by association, that means bookings can only be created as a subnode of its travel parent.

i [Expand the following listing to view the source code.](#)

Listing: Behavior Definition /DMO/I_TRAVEL_U

```
implementation unmanaged;
// behavior definition for the TRAVEL root node
define behavior for /DMO/I_Travel_U alias travel
implementation in class /DMO/BP_TRAVEL_U unique
etag master LastChangedAt
lock master
{
  field ( read only ) TravelID, TotalPrice;
  field (mandatory) AgencyID, CustomerID, BeginDate, EndDate;
  create;
  update;
  delete;
```

```

action set_status_booked result [1] $self;
association Booking { create (features: instance); }
mapping for /dmo/travel control /dmo/s_travel_intx
{
    AgencyID      = agency_id;
    BeginDate     = begin_date;
    BookingFee    = booking_fee;
    CurrencyCode  = currency_code;
    CustomerID   = customer_id;
    EndDate       = end_date;
    LastChangedAt = lastchangedat;
    Memo          = description;
    Status        = status;
    TotalPrice    = total_price;
    TravelID      = travel_id;
}
}

// behavior defintion for the BOOKING sub node
define behavior for /DMO/I_Booking_U alias booking
implementation in class /DMO/BP_BOOKING_U unique
etag dependent by _Travel
lock dependent by _Travel
{
    field (read only) TravelID, BookingID;
    field (mandatory) BookingDate, CustomerID, AirlineID, ConnectionID, FlightDate;
    update;
    delete;
mapping for /dmo/booking control /dmo/s_booking_intx
{
    AirlineID      = carrier_id;
    BookingDate    = booking_date;
    BookingID      = booking_id;
    ConnectionID   = connection_id;
    CurrencyCode   = currency_code;
    CustomerID    = customer_id;
    FlightDate     = flight_date;
    FlightPrice    = flight_price;
    TravelID       = travel_id;
}
}
}

```

Related Information

[Business Object \[page 55\]](#)

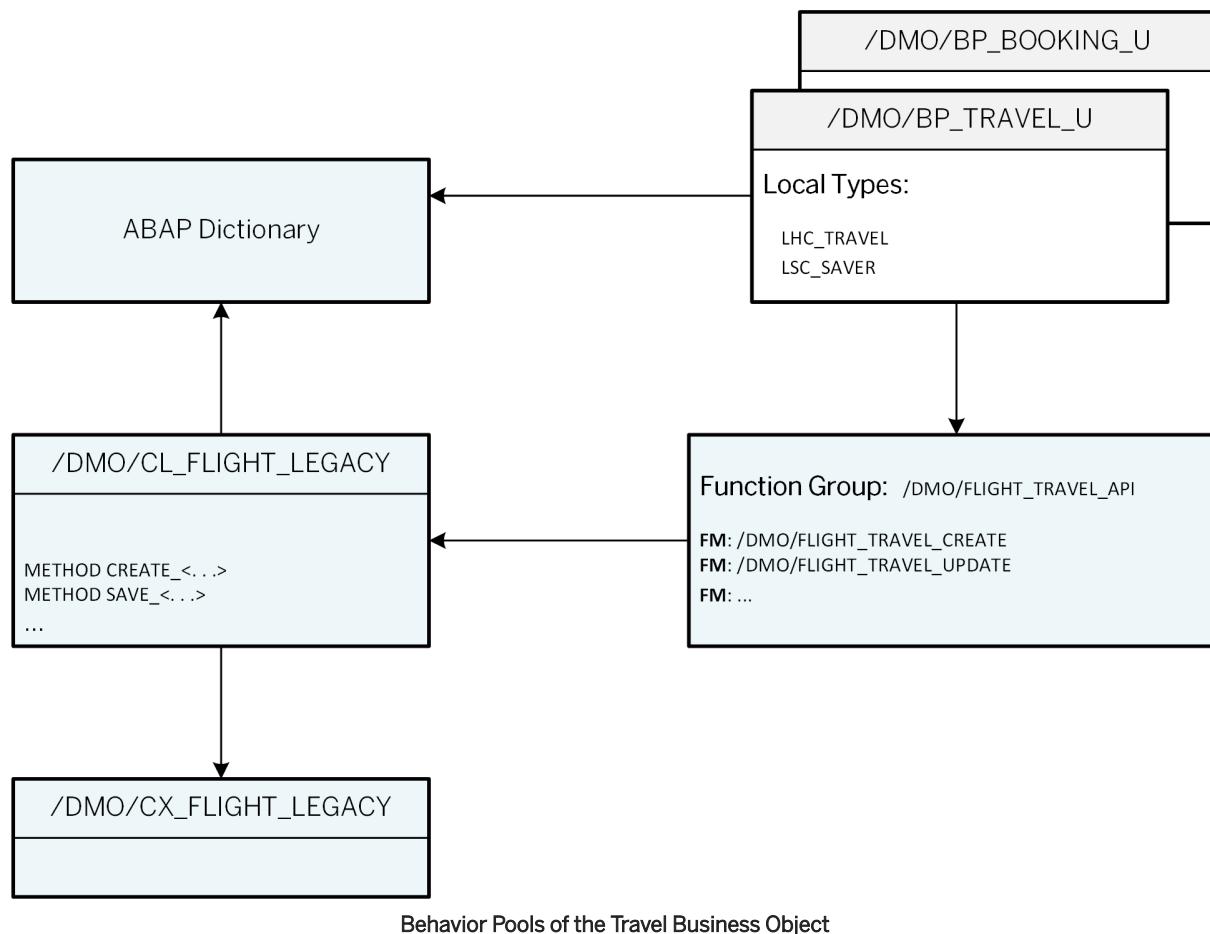
5.3.3.2 Implementing the Behavior of the Business Object

Behavior Pool

The transactional behavior of a business object in the context of the current programming model is implemented in one or more global ABAP classes. These special classes are dedicated only to implementing the business object's behavior and are called **behavior pools**. You can assign any number of behavior pools to a

behavior definition (a 1: n relationship). Within a single global class, you can define multiple local classes that handle the business object's behavior. The global class is just a container and is basically empty while the actual behavior logic is implemented in local classes.

Behavior Pool



Parenthesis: Syntax Extension for Defining a Behavior Pool

```

CLASS class_name DEFINITION PUBLIC
  ABSTRACT
  FINAL
  FOR BEHAVIOR OF MyRootBehavior.

  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS class_name IMPLEMENTATION.
ENDCLASS.

```

Effect

The above syntax defines a special ABAP class (a behavior pool) for the behavior specified in the behavior definition `MyRootBehavior` (which in turn has the same name as the CDS root entity). This special property

and relationship are persisted and transported using a corresponding system table. This specific information is assigned to the properties of the behavior pool and can no longer be changed. The behavior pool is dependent on the behavior definition, meaning the changes to the behavior definition cause it to be regenerated.

This global class is defined as abstract and final, which means there is no reason to instantiate or inherit this global behavior pool. In addition, such a ban prevents possible misuse.

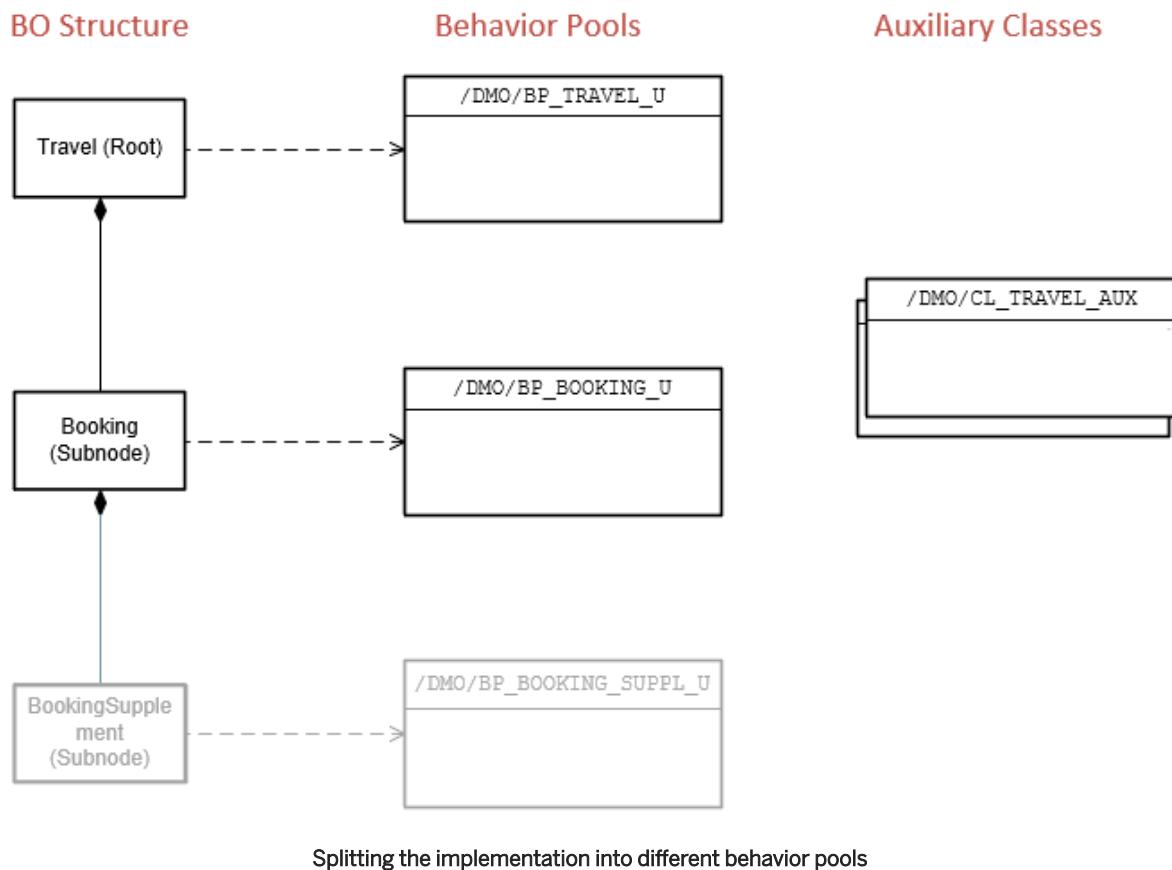
A behavior pool can have static methods, namely `CLASS-DATA`, `CONSTANTS`, and `TYPES`. The application may place common or even public aspects of its implementation in these methods.

Distributing Behavior Pool Implementation

You can assign any number of behavior pools to a behavior definition (1: n relationship). This allows the application developers to distribute their implementations between multiple units, for example one global class (behavior pool) for each business object's node, and one or more separate auxiliary classes for implementing helper methods. The figure below illustrates this distribution pattern for our sample application.

i Note

Best Practices: Splitting the implementation into different global classes allows developers to work in parallel (distributed work mode). If operations on each node have to be forwarded to different APIs (function module calls), then we recommend using a separate global class (behavior pool) for each node of the business object's compositional tree.



Related Information

- [Unmanaged BO Contract \[page 718\]](#)
- [Handler Classes \[page 719\]](#)
- [Saver Classes \[page 730\]](#)
- [Naming Conventions for Development Objects \[page 780\]](#)

5.3.3.2.1 Creating the Behavior Pool for the Root Entity

In this step, you create a behavior pool that is the implementation artifact of the corresponding behavior definition that you created earlier.

In doing so, we apply the contribution pattern and split the behavior implementation into two different behavior pools, one for the travel root entity and the other for the booking child entity. In addition, we create a separate auxiliary class for implementing helper methods (such as for mapping and message handling) that can be reused in both behavior implementation classes.

i Note

Best Practices for Modularization and Performance:

The granularity of the existing application code influences the granularity of handler implementation. If the existing legacy application logic has different APIs (function modules) for create, update, delete, and other transactional operations, then we recommend spreading the operations across different handler classes.

If, for example, the called API only implements one operation for `CREATE` and another API implements one for `UPDATE`, it is advisable to implement each operation in a different local handler class.

If, on the other hand, the called API of your application code is able to process **multiple changes in one call**, the handler should reflect this to achieve the best performance. In such a case, we combine all operations supported by this API in one common handler class.

Otherwise the application code is called multiple times with different input, which can result in bad performance.

If the application code supports a **deep create** (for example creation for root and child entity in one step), then this should be reflected in the design of handler classes to achieve best performance.

→ Remember

Beyond the performance aspects, it is beneficial to implement operations in different `FOR MODIFY` methods, since the orchestration is then passed to the BO runtime and the code of the behavior implementation is more **readable**.

i Note

Convention: The saver class that implements the save sequence for data persistence is either a separate global class or a part of the root implementation (behavior pool for the root entity).

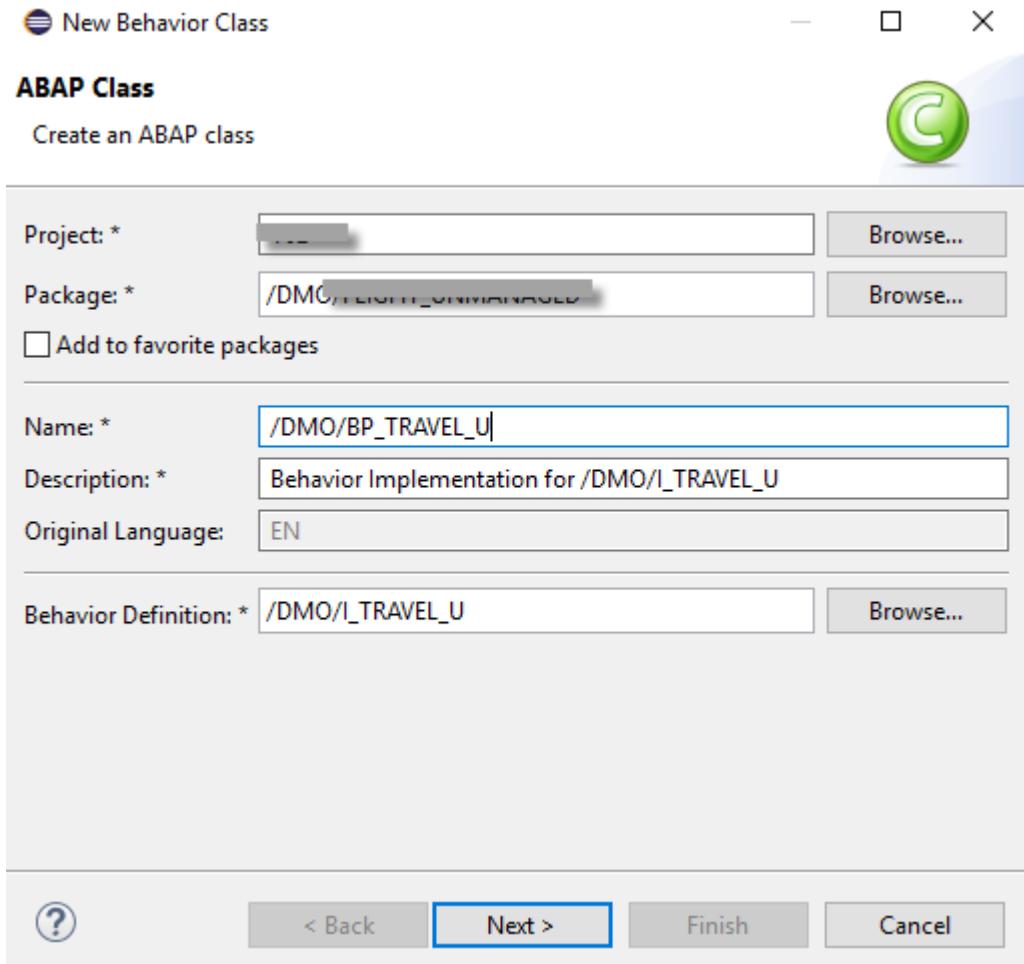
Procedure 1: Create a Behavior Pool /DMO/BP_TRAVEL_U

To launch the wizard tool for creating a behavior implementation, do the following:

1. Launch ABAP Development Tools.
2. In your ABAP project (or *ABAP Cloud Project*), select the relevant behavior definition node (`/DMO/I_TRAVEL_U`) in *Project Explorer*.
3. Open the context menu and choose *New Behavior Implementation* to launch the creation wizard.

i Note

The behavior definition must be active to get the behavior implementation template that matches the modeled behavior



Creating a Behavior Pool – Wizard

Further information:

- [Naming Conventions for Development Objects \[page 780\]](#)
- [\(Tool Reference\) \[page 764\]](#)

Results: Global Behavior Pool

The generated class pool (in our case /DMO/BP_TRAVEL_U) provides you with an extension FOR BEHAVIOR OF.

```

1@ CLASS /dmo/bp_travel_u DEFINITION
2   PUBLIC
3   ABSTRACT
4   FINAL
5   FOR BEHAVIOR OF /dmo/i_travel_u .
6
7   PUBLIC SECTION.
8   PROTECTED SECTION.
9   PRIVATE SECTION.
10  ENDCCLASS.
11
12
13
14@ CLASS /dmo/bp_travel_u IMPLEMENTATION.
15  ENDCCLASS.

```

New Global Behavior Pool

The real substance of a behavior pool is located in [Local Types](#). Here you can define two types of special local classes, namely handler classes for the operations within the **interaction phase** and saver classes for the operations within the **save sequence**. These classes can be instantiated or invoked only by the [ABAP runtime environment \(virtual machine\)](#) [page 804].

i Note

All local class source code within a single global class is stored within a single include, the CCIMP include.

Procedure 2: Define a Skeleton of Local Classes Corresponding to the Behavior Model

Based on the declarations in the behavior definition /DMO/I_TRAVEL_U, and taking best practices for modularization and performance into account, adapt the generated skeleton of the local classes for the root entity accordingly to the listing below:

⚠ Caution

In the current version of ADT tools, the skeleton with the code generated by the class pool creation wizard differs from the source code in the listing below.

[i Expand the following listing to view the source code.](#)

Listing: Template for local classes of /DMO/BP_TRAVEL_U

```

*****
*
* Handler class for managing travels

```

```

*
***** CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS:
    create_travel      FOR MODIFY
        IMPORTING it_travel_create FOR CREATE travel,
    update_travel      FOR MODIFY
        IMPORTING it_travel_update FOR UPDATE travel,
    delete_travel      FOR MODIFY
        IMPORTING it_travel_delete FOR DELETE travel,
    read_travel        FOR READ
        IMPORTING it_travel FOR READ travel
        RESULT      et_travel,
    create_booking_ba FOR MODIFY
        IMPORTING it_booking_create_ba FOR CREATE travel\booking,
    read_booking_ba   FOR READ
        IMPORTING it_travel FOR READ travel\Booking
            FULL iv_full_requested
        RESULT      et_booking
            LINK et_link_table,
    lock              FOR LOCK
        IMPORTING it_travel_lock FOR LOCK travel,
    set_travel_status FOR MODIFY
        IMPORTING it_travel_set_status_booked FOR ACTION travel~set_status_booked
        RESULT      et_travel_set_status_booked,
    get_features       FOR FEATURES
        IMPORTING keys     REQUEST requested_features FOR travel
        RESULT      result.
ENDCLASS.

CLASS lhc_travel IMPLEMENTATION.
METHOD create_travel.
ENDMETHOD.
METHOD update_travel.
ENDMETHOD.
METHOD delete_travel.
ENDMETHOD.
METHOD read_travel.
ENDMETHOD.
METHOD create_booking_ba.
ENDMETHOD.
METHOD read_booking_ba.
ENDMETHOD.
METHOD lock.
ENDMETHOD.
METHOD set_travel_status.
ENDMETHOD.
METHOD get_features.
ENDMETHOD.
ENDCLASS.
*****
*
* Saver class implements the save sequence for data persistence
*
***** CLASS lsc_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.
PROTECTED SECTION.
METHODS finalize          REDEFINITION.
METHODS check_before_save REDEFINITION.
METHODS save              REDEFINITION.
METHODS cleanup           REDEFINITION.
ENDCLASS.

CLASS lsc_saver IMPLEMENTATION.
METHOD finalize.
ENDMETHOD.
METHOD check_before_save.
ENDMETHOD.
METHOD save.
ENDMETHOD.

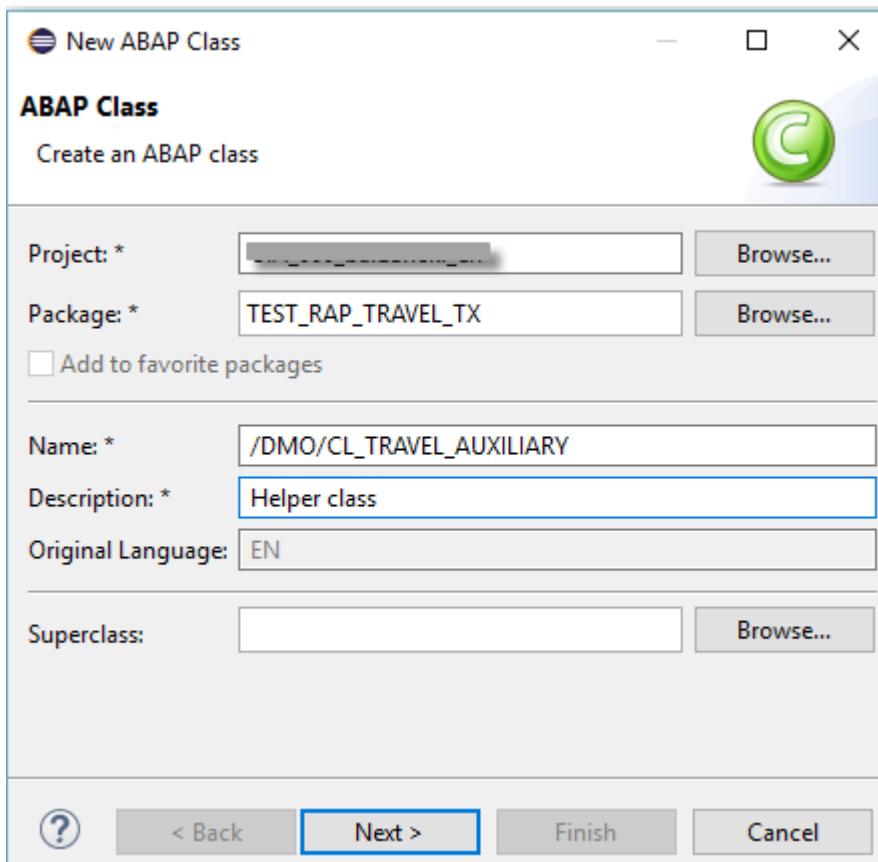
```

```
ENDMETHOD.  
METHOD cleanup.  
ENDMETHOD.  
ENDCLASS.
```

Procedure 3: Create an Auxiliary Class /DMO/CL_TRAVEL_AUXILIARY

Message handling can be reused by different behavior pools and is therefore outsourced in a separate helper class.

1. In your ABAP project (or *ABAP Cloud Project*), select the select the *Source Code Library* *Classes* node of the relevant package.
2. To launch the creation wizard, open the context menu and choose *New ABAP Class*.



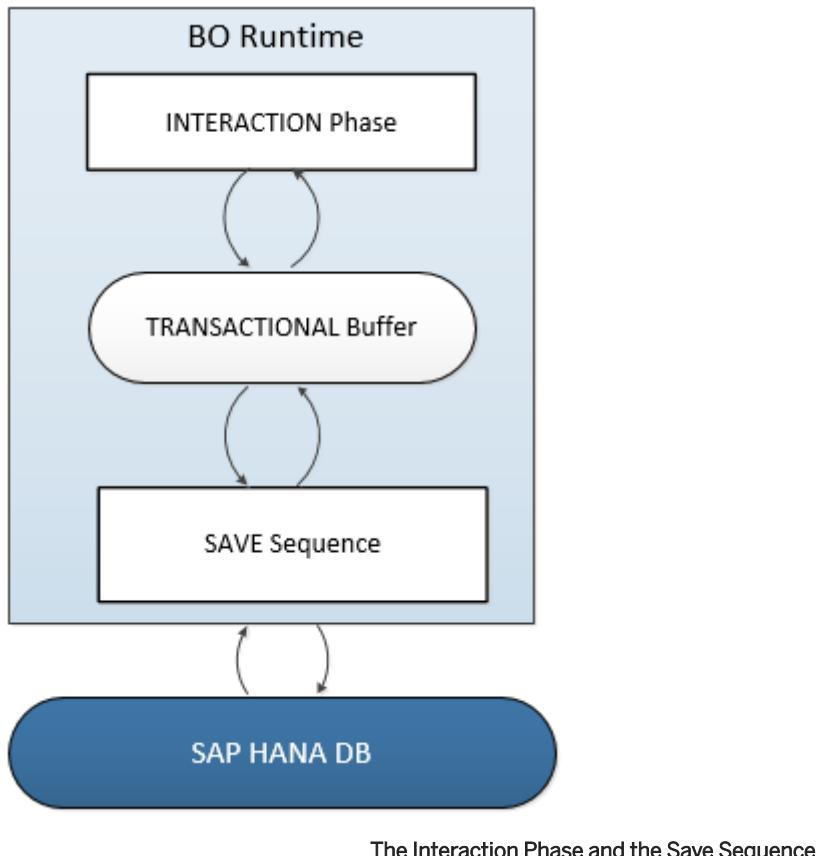
Creating ABAP Class – Wizard

5.3.3.2.2 Implementing the Interaction Phase and the Save Sequence

The business object runtime has two parts: The first part is the interaction phase where a consumer calls business object operations to change data and read instances with or without the transactional changes. The

business object keeps the changes in its internal transactional buffer, which represents the state. This transactional buffer is always required for a business object, regardless of how it is implemented. After all changes are performed, the data should be persisted. This is implemented within the save sequence.

Further information: [\(Reference\) \[page 718\]](#)



Implementation Steps

1. Implementing the CREATE Operation for Travel Instances [\[page 305\]](#)
2. Implementing the UPDATE Operation for Travel Data [\[page 312\]](#)
3. Implementing the READ Operation for Travel Data [\[page 316\]](#)
4. Implementing the DELETE Operation for Travel Instances [\[page 318\]](#)
5. Implementing the CREATE Operation for Associated Bookings [\[page 320\]](#)
6. Implementing the READ Operation for Associated Bookings [\[page 327\]](#)
7. Implementing the SET_STATUS_BOOKED Action [\[page 333\]](#)
8. Implementing Dynamic Feature Control [\[page 468\]](#)
9. Implementing the UPDATE, DELETE, and READ Operations for Booking Instances [\[page 339\]](#)

Related Information

[Handler Classes \[page 719\]](#)

5.3.3.2.2.1 Implementing the CREATE Operation for Travel Instances

In this topic, you will be guided through all implementation steps required for creation of new travel instances. We will motivate the steps for the implementation, starting from the UI.

Preview

If you run the UI service based on *SAP Fiori Elements Fiori Launchpad*, the resulting UI provides you with a list of existing travel items, including all fields exposed for UI consumption. in the

Travels (3,092)		Standard ▾	Set to Booked	Delete		
Travel ID	Agency ID	Customer ID	Starting Date	End Date		
1	70041	99	Apr 29, 2018	Apr 29, 2018 >		
	Travel Status: N					
2	70007	93	Apr 29, 2018	Apr 29, 2018 >		
	Travel Status: N					
3	70046	161	Apr 29, 2018	Apr 29, 2018 >		
List of Travel Items						

To create a travel item, the end user must click the + (*Create*) button and fill all required fields in the related object page to specify the required information for a new travel instance.

General Information

Agency ID:	Booking Fee:
70007	45 EUR
Customer ID:	Total Price:
99	1,777.00 EUR
Starting Date:	Comment:
Nov 17, 2018	Business trip to WDF
End Date:	
Nov 24, 2018	

Save **Cancel**

Object Page for Editing Individual Values

As soon as the user clicks the **Save** button on the object page, the data is persisted in the corresponding database table and a travel instance with a new travel ID is created.

Travel Status: N					
<input type="radio"/> 3092	70007	99	Jul 12, 2018	Jul 19, 2018	>
Travel Status: N					
<input type="radio"/> 3093	70007	99	Nov 17, 2018	Nov 24, 2018	>
Travel Status: N					

Saved Data Displayed on Fiori UI

Implementation Steps

1. Defining the Handler Class for Creation of Travel Instances

Corresponding to the [template \[page 301\]](#) for the root node behavior implementation, a local handler class `lhc_travel` is defined to implement each changing operation in one individual `FOR MODIFY` method. In this case, the `create_travel FOR MODIFY` method should only be used to implement the create operation for root instances. Therefore, the signature of this method includes only one import parameter `it_travel_create` for referring to the travel (root) instances to be created. To identify the root entity, the alias `travel` is used - according to the alias that is specified in the behavior definition.

i Note

The local handler class `lhc_travel` inherits from class `cl_abap_behavior_handler` and is automatically instantiated by the framework.

i Note

Note that import parameter `it_travel_create` does not have fixed data type at the design time. At runtime, the data type is assigned by the compiler with the types derived from behavior definition.

(x)= Variables		
Name	Value	Actual Type
IT_TRAVEL_CREATE	[1x13(144)]Stan...	CREATE
[1]	Structure: deep	CREATE
%CID	%SADL_CID_1	ABP_BEHV_CID
TRAVELID	00000000	/DMO/TRAVEL_ID
AGENCYID	070001	/DMO/AGENCY_ID
CUSTOMERID	000099	/DMO/CUSTOMER_ID
BEGINDATE	20181210	/DMO-BEGIN_DATE
ENDDATE	20181217	/DMO-END_DATE
BOOKINGFEE	11.00	/DMO/BOOKING_FEE
TOTALPRICE	111.00	/DMO/TOTAL_PRICE
CURRENCYCODE	EU	/DMO/CURRENCY_CODE
MEMO	AB	/DMO/DESCRIPTION
STATUS		/DMO/TRAVEL_STATUS
LASTCHANGEDAT	0.0000000	TIMESTAMPL
%CONTROL	Structure: flat, n...	\TYPE=%_T00004S0000...
TRAVELID	01	ABP_BEHV_FLAG
AGENCYID	01	ABP_BEHV_FLAG
CUSTOMERID	01	ABP_BEHV_FLAG

Derived Data Type for the Import Parameter `it_travel_create` in the ABAP Debugger

Further information: [<method> FOR MODIFY \[page 721\]](#)

i *Expand the following listing to view the source code.*

LISTING 1: Signature of the `create_travel FOR MODIFY` (excerpt from template)

```
*****
*
* Handler class for managing travels
*
*****
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS:
  create_travel FOR MODIFY
    IMPORTING    it_travel_create    FOR CREATE travel,
    ...
ENDCLASS.
CLASS lhc_travel IMPLEMENTATION.
METHOD create_travel.
ENDMETHOD.
...
ENDCLASS.
```

2. Implementing the <method> FOR MODIFY for Creation of New Travel Instances

As given in the listing below, the basic structure of the <method> FOR MODIFY implementation includes:

- A loop on all new travel instances to be created for the root node.
- Mapping the CDS view field names to the database table fields names by using the operator MAPPING FROM ENTITY USING CONTROL. This operator maps only the fields that are flagged in the control structure of the importing parameter.
- Call of the business logic function module /DMO/FLIGHT_TRAVEL_CREATE for creation of new travel instances.
- Message handling for processing messages in case of failure. See step 3 [Message Handling \[page 309\]](#).

Each create action call can produce failed keys (<fs_travel_create>-%cid) and messages (lt_messages). Any failed keys are stored in the table [FAILED \[page 738\]](#) whereas the [REPORTED \[page 739\]](#) table includes all instance-specific messages.

Besides an ID of the relevant BO instance and the [%FAIL \[page 741\]](#) component, the failed tables also include the predefined component [%CID \[page 740\]](#). It stands for the content ID and is used in an OData request to bind the result of an operation to a name so that it can be referenced in another operation later in the transactional processing.

→ Remember

In some use cases, it may happen that a consumer works with data that is not yet persisted and might not have a primary key yet. The primary key can be created in the <method> FOR MODIFY call or later in the save sequence (late numbering). In such cases, a temporary primary key, the content ID (%CID) for an instance, is used as long as no primary key was created by BO runtime. The content ID is consequently also used then as a foreign key.

Locals	
FAILED	Structure: deep
TRAVEL	[1x3(28)]Standard Table
[1]	Structure: deep
%CID	%SADL_CID_1
TRAVELEDID	00000000
%FAIL	Structure: flat, not charlike
CAUSE	UNSPECIFIC
BOOKING	[0x4(36)]Initial Standard Table
MAPPED	Structure: deep
REPORTED	Structure: deep
IT_TRAVEL_CREATE	[1x13(144)]Standard Table
LT_MESSAGES	[1x7(448)]Standard Table
[1]	E/DMO/CM_FLIGHT_LEGAC0011111...
MSGTY	E
MSGID	/DMO/CM_FLIGHT_LEGAC
MSGNO	001

FAILED-TRAVEL table in the Variables view of the ABAP Debugger

In case of success (lt_messages IS INITIAL), the two values with the content ID %cid and the new key travelid are written into the mapped-travel table.

→ Remember

The [MAPPED \[page 739\]](#) tables comprise the components %CID and %KEY [\[page 740\]](#). They include the information about which key values were created by the application for given content IDs.

i [Expand the following listing to view the source code.](#)

LISTING 2: Creating travel instances

```
CLASS lhc_travel IMPLEMENTATION.
METHOD create_travel.
  DATA lt_messages    TYPE /dmo/t_message.
  DATA ls_travel_in   TYPE /dmo/travel.
  DATA ls_travel_out  TYPE /dmo/travel.
  LOOP AT it_travel_create ASSIGNING FIELD-SYMBOL(<fs_travel_create>).
    ls_travel_in = CORRESPONDING #( <fs_travel_create> ) MAPPING FROM ENTITY
  USING CONTROL .
    CALL FUNCTION '/DMO/FLIGHT_TRAVEL_CREATE'
      EXPORTING
        is_travel     = CORRESPONDING /dmo/s_travel_in( ls_travel_in )
      IMPORTING
        es_travel     = ls_travel_out
        et_messages   = lt_messages.
    IF lt_messages IS INITIAL.
      INSERT VALUE #( %cid = <fs_travel_create>-%cid  travelid = ls_travel_out-
travel_id )
        INTO TABLE mapped-travel.
    ELSE.
      /dmo/cl_travel_auxiliary=>handle_travel_messages(
        EXPORTING
          iv_cid       = <fs_travel_create>-%cid
          it_messages  = lt_messages
        CHANGING
          failed       = failed-travel
          reported     = reported-travel
        ).
    ENDIF.
  ENDLOOP.
ENDMETHOD.
...
ENDCLASS.
```

3. Message Handling

When handling changing operations for travel instances, fault events may occur. For the processing of appropriate messages in such a case, the method `handle_travel_messages` is used. This method is defined in a separate auxiliary class `/dmo/cl_travel_auxiliary` so that it can be called in different `FOR MODIFY` methods of the class pools. In order to use the message object from behavior processing framework, the auxiliary class inherits from the framework class `cl_abap_behv`. The helper method `get_message_object` is defined to retrieve the message object `obj` in various message handler methods.

i [Expand the following listing to view the source code.](#)

LISTING 3: Declaration of the method `handle_travel_messages` at the beginning of the helper class `/dmo/cl_travel_auxiliary`

```
CLASS /dmo/cl_travel_auxiliary DEFINITION
  INHERITING FROM cl_abap_behv
  PUBLIC
  FINAL
```

```

CREATE PUBLIC .
PUBLIC SECTION.
  TYPES tt_travel_failed          TYPE TABLE FOR FAILED /dmo/i_travel_u.
  TYPES tt_travel_mapped           TYPE TABLE FOR MAPPED /dmo/i_travel_u.
  TYPES tt_travel_reported        TYPE TABLE FOR REPORTED /dmo/i_travel_u.

  CLASS-METHODS handle_travel_messages
    IMPORTING
      iv_cid      TYPE string   OPTIONAL
      iv_travel_id TYPE /dmo/travel_id OPTIONAL
      it_messages  TYPE /dmo/t_message
    CHANGING
      failed      TYPE tt_travel_failed
      reported     TYPE tt_travel_reported.
  PRIVATE SECTION.
    CLASS-DATA obj TYPE REF TO /dmo/cl_travel_auxiliary.
    CLASS-METHODS get_message_object
      RETURNING VALUE(r_result)          TYPE REF TO /dmo/cl_travel_auxiliary.
ENDCLASS.

```

4. Implementing the Message Handling for Travels

The following listing represents the implementation of the method `handle_travel_messages`.

To refer to the data set where an error (`msgty = 'E'`) or an abort (`msgty = 'A'`) occurred, the `failed` table is used, whereas the instance-specific messages are stored in the `reported` table.

However, messages that originate from the legacy code must be mapped to the messages of the class-based BO framework.

The method `new_message` is used in this implementation to map the T100 messages (that originate from the legacy code) to the messages of the class-based BO framework. It returns a message object and is implemented in the framework class `lcl_abap_behv`.

i [Expand the following listing to view the source code.](#)

LISTING 4: Implementation of the method `handle_travel_messages`

```

CLASS /dmo/cl_travel_auxiliary IMPLEMENTATION.
  METHOD handle_travel_messages.
    LOOP AT it_messages INTO DATA(ls_message) WHERE msgty = 'E' OR msgty = 'A'.
      APPEND VALUE #( %cid = iv_cid travelid = iv_travel_id )
      TO failed.
      APPEND VALUE #( %msg      = get_message_object( )->new_message( id      =
ls_message-msgid
                                         number   = ls_message-msgno
                                         severity = )
if_abap_behv_message=>severity-error
                                         v1      = ls_message-msgv1
                                         v2      = ls_message-msgv2
                                         v3      = ls_message-msgv3
                                         v4      = ls_message-msgv4 )
      %key-TravelID = iv_travel_id
      %cid          = iv_cid
      TravelID      = iv_travel_id )
      TO reported.
    ENDLOOP.
  ENDMETHOD.
  METHOD get_message_object.
    IF obj IS INITIAL.
      CREATE OBJECT obj.
    ENDIF.
    r_result = obj.
  ENDMETHOD.

```

```
ENDMETHOD.  
ENDCLASS.
```

5. Performing a Final Commit and Releasing Caches

When the `save` method is called, the final commit is executed on the database and the data entered by the user is persisted to the new travel instance. As depicted in the listing below, the `save` method only executes a call to the function module `/DMO/FLIGHT_TRAVEL_SAVE` from the legacy business logic.

Further information: [Method SAVE \[page 734\]](#)

To discard all changes after the last save, the `cleanup` method is used. This method delegates the call to the function module `/DMO/FLIGHT_TRAVEL_INITIALIZE` from legacy code.

Add the definition for the `cleanup` method to `lsc_saver` and call the function module in the implementation.

i [Expand the following listing to view the source code.](#)

LISTING 5: Implemented save sequence

```
CLASS lsc_saver IMPLEMENTATION.  
  ...  
  METHOD save.  
    CALL FUNCTION '/DMO/FLIGHT_TRAVEL_SAVE'.  
  ENDMETHOD.  
  METHOD cleanup.  
    CALL FUNCTION '/DMO/FLIGHT_TRAVEL_INITIALIZE'.  
  ENDMETHOD.  
ENDCLASS.
```

Checking Results

At this point, you have the opportunity to check how does the resulting app work, and especially the new implementation of the `CREATE` operation. For this to happen, however, a suitable business service for UI consumption must first be defined and published.

For more information, see: [Defining Business Service for Fiori UI \[page 350\]](#)

Related Information

[Handler Classes \[page 719\]](#)

[Saver Classes \[page 730\]](#)

[Declaration of Derived Data Types \[page 734\]](#)

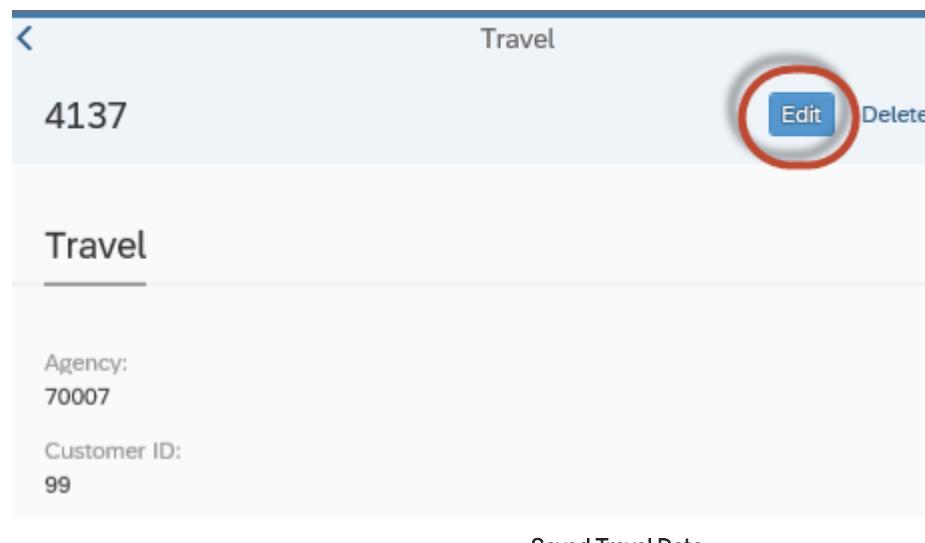
[Implicit Returning Parameters \[page 738\]](#)

5.3.3.2.2.2 Implementing the UPDATE Operation for Travel Data

This topic guides you through the implementation steps required for data updates to an existing travel instance. In this case, however, in addition to the `<method> FOR MODIFY`, the `<method> FOR READ` must also be implemented. It provides read access to the application buffer, which is necessary for **ETag** comparison.

Preview

In our travel application scenario, the appropriate business data should be modifiable for all required items of the travel instance when, for example, the user clicks the [Edit](#) button on the Fiori UI.



In change mode, the end user is able to change the relevant travel fields as shown in the figure below. As soon as the user chooses the [Save](#) button on the object page, the changed travel data is saved in the corresponding tables and a new version of the related travel instance is created.



Saved Travel Data

Implementation Steps

1. Defining the Method for Implementing Travel Data Update

Corresponding to the [template \[page 301\]](#) for the root node behavior implementation, a local handler class `lhc_travel` is defined to implement each changing operation. In this case, the `update_travel FOR MODIFY` method should only be used to implement the update operation for root instances. Therefore, the signature of this method includes only one import parameter `it_travel_update` for referring to the `travel` (root) instances to be updated.

LISTING 1: Signature of the `<method> FOR MODIFY` (excerpt from template)

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS:
  ...
  update_travel FOR MODIFY
    IMPORTING      it_travel_update      FOR UPDATE travel,
ENDCLASS.
```

2. Implementing the `<method> FOR MODIFY` for Travel Data Update

The basic structure of the `FOR MODIFY` method implementation is very similar to that of the handler class for creation of travel instances:

- A loop on all new travel instances to be updated for the root node.
- Mapping the CDS view field names to the database table fields names using the operator `MAPPING FROM ENTITY`. This operator maps every field that is declared in the mapping specification in the behavior definition.
- Mapping of the `%control`-structure of the importing parameter to the predefined flag structure `ls_travelx`. The control structure identifies which entity fields were changed by the client. It contains the key and the flag structure to the data fields and is used in BAPIs to flag each individual data field as changed.
- Call of the business logic function module `/DMO/FLIGHT_TRAVEL_UPDATE` to update travel instances.
- Message handling for processing messages in case of failure.

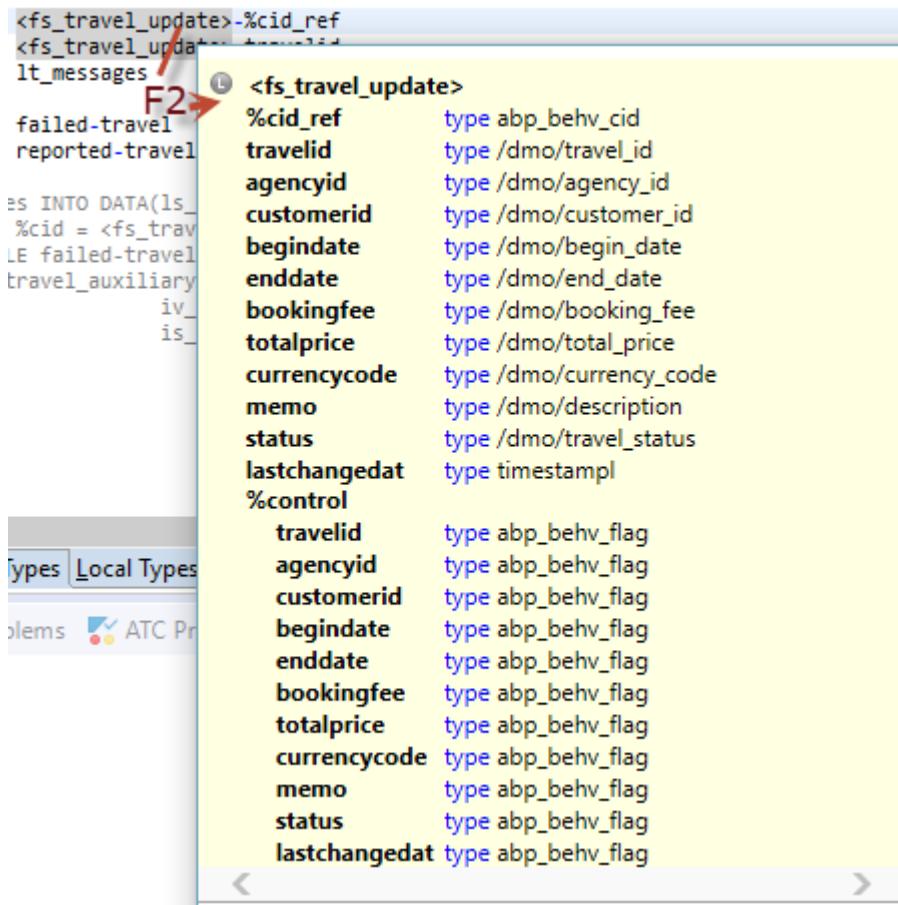
Each update call can produce failed keys and messages (`lt_messages`). Failed keys are addressed by the content ID (`<fs_travel_update>-%cid_ref`) and the value (`<fs_travel_update>-travelid`). In case of

failure, failed keys are saved in the failed-travel table, whereas the reported-travel table includes all instance-specific messages.

i [Expand the following listing to view the source code.](#)

LISTING 2: Updating data of travel instances

```
CLASS lhc_travel IMPLEMENTATION.  
  ...  
  METHOD update_travel.  
    DATA lt_messages TYPE /dmo/t_message.  
    DATA ls_travel TYPE /dmo/travel.  
    DATA ls_travelx TYPE /dmo/s_travel_inx. "refers to x structure (> BAPIs)  
    LOOP AT it_travel_update ASSIGNING FIELD-SYMBOL(<fs_travel_update>).  
      ls_travel = CORRESPONDING #( <fs_travel_update> MAPPING FROM ENTITY ).  
      ls_travelx-travel_id = <fs_travel_update>-TravelID.  
      ls_travelx-intx = CORRESPONDING #( <fs_travel_update> mapping from  
entity ).  
      CALL FUNCTION '/DMO/FLIGHT_TRAVEL_UPDATE'  
        EXPORTING  
          is_travel = CORRESPONDING /dmo/s_travel_in( ls_travel )  
          is_travelx = ls_travelx  
        IMPORTING  
          et_messages = lt_messages.  
      /dmo/cI_travel_auxiliary=>handle_travel_messages(  
        EXPORTING  
          iv_cid = <fs_travel_update>-%cid_ref  
          iv_travel_id = <fs_travel_update>-travelid  
          it_messages = lt_messages  
        CHANGING  
          failed = failed-travel  
          reported = reported-travel  
        ).  
    ENDLOOP.  
  ENDMETHOD.  
  ...  
ENDCLASS.
```



F2 Access for <fs_travel_update>

3. Implementing the <method> FOR READ for ETag Handling

In the context of data updates to an existing travel data set, it is important to retrieve current data from the application buffer. As you remember, we specified an ETag for the root entity in the behavior definition:

```

define behavior for /DMO/I_Travel_U alias travel
etag master LastChangedAt
{
  ...
}

```

An ETag [page 812] determines the changes to the requested data set to help prevent simultaneous updates of a data set from overwriting each other. This is precisely the reason why the ETag check requires data from the buffer. The <method> FOR READ is designed to return the data from the application buffer.

For more information on how to implement this method, see [Implementing the READ Operation for Travel Data \[page 316\]](#).

Related Information

[Handler Classes \[page 719\]](#)

[Declaration of Derived Data Types \[page 734\]](#)

5.3.3.2.2.3 Implementing the READ Operation for Travel Data

This topic guides you through the implementation steps required to read travel instances from the transactional buffer.

Context

In contrast to the CREATE, UPDATE, or DELETE operation, the READ does not have its direct trigger on a Fiori UI.

→ Remember

The GO button on the UI does not call the `<method> FOR READ` (the transactional READ) in the behavior pool. Instead, the GO button executes a query via the orchestration framework and reads data directly from the database.

The READ operation provides read access to the application buffer. It is used to retrieve data for further processing during the [interaction phase \[page 814\]](#). This is necessary, for example, for [ETag \[page 812\]](#) comparison when executing an UPDATE. Before the `<method> for UPDATE` is called, the ETag value in the application buffer must be compared to the value that is displayed on the UI. Only if these values correspond is the UPDATE triggered. This check ensures that the data the end user sees on the UI has not been changed by other consumers.

The READ is also necessary for [dynamic action control \[page 463\]](#) to check the conditions for actions and decide if they are enabled or disabled. In the travel scenario, the action `set_to_booked` is disabled when the status is booked. Hence, the data from the transactional buffer must be read to display the action button correspondingly.

Both use cases are relevant for our travel scenario. Therefore it is necessary to implement a `<method> FOR READ`.

The READ is also necessary to complete the business object and make it accessible by [EML \[page 812\]](#). In this case, you can consume the business object directly from ABAP.

Implementation Steps

The READ operation cannot be declared in the behavior definition as it is always implicitly assumed that a READ is implemented.

i Note

⌚ If you use groups in your behavior definition, you must explicitly specify the READ operation in one of the groups. For more information, see [Using Groups in Large Development Projects \[page 481\]](#).

1. Defining the signature of the <method> FOR READ

In the definition part of the handler class for travel, define the method `read_travel` for READ with an importing parameter and a result parameter.

LISTING 1: Signature of the <method> for READ

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS:  
    ...  
    read_travel FOR READ  
        IMPORTING it_travel FOR READ travel  
        RESULT      et_travel,  
    ... .  
ENDCLASS.
```

For more information, see [<method> FOR READ \[page 726\]](#).

2. Implementing the <method> FOR READ

To return the actual data from the application buffer, the travel ID that is imported is passed to the reading function module `/DMO/FLIGHT_TRAVEL_READ`. The function module returns the entity instance corresponding to the travel ID and messages if an error occurs. The result parameter `et_travel` is then filled with the keys and the values of the fields that the consumer has requested. These fields are flagged in the control structure of the importing parameter `it_travel`.

With the changing parameter `failed`, you specify the fail cause if something goes wrong. If, for example, the used function module returns the message number 16, the travel ID was not found. In this case, you can return the fail cause `not_found`. For all other errors, return fail cause `unspecific`.

i [Expand the following listing to view the source code.](#)

LISTING 2: Reading travel instances from the application buffer

```
*****  
*  
* Read travel data from buffer  
*  
*****  
METHOD read_travel.  
DATA: ls_travel_out TYPE /dmo/travel,  
      lt_message     TYPE /dmo/t_message.  
LOOP AT it_travel INTO DATA(ls_travel_to_read).  
  CALL FUNCTION '/DMO/FLIGHT_TRAVEL_READ'  
    EXPORTING  
      iv_travel_id = ls_travel_to_read-travelid  
    IMPORTING  
      es_travel      = ls_travel_out  
      et_messages   = lt_message.  
  IF lt_message IS INITIAL.  
    "fill result parameter with flagged fields  
    INSERT CORRESPONDING #( ls_travel_out MAPPING TO ENTITY ) INTO TABLE  
    et_travel.  
  ELSE.  
    "fill failed table in case of error  
    failed-travel = VALUE #(  
      BASE failed-travel  
      FOR msg IN lt_message (  
        %key = ls_travel_to_read-%key  
        %fail-cause = COND #(  
          WHEN msg-msgty = 'E' AND msg-msgno = '016'  
          THEN if_abap_behv=>cause-not_found
```

```

        ELSE if_abap_behv=>cause-unspecific
    )
)
).
ENDIF.
ENDLOOP.
ENDMETHOD.
```

Testing the READ

In contrast to the CREATE, UPDATE and DELETE operation, the READ cannot be easily tested using a Fiori UI, as there is no direct execution trigger for reading the data from the application buffer. Nevertheless, you can test your READ implementation by using EML.

• Example

```

READ ENTITY /DMO/I_Travel_U FIELDS ( TravelID
                                         AgencyID
                                         CustomerID
                                         BeginDate
                                         EndDate
                                         BookingFee
                                         TotalPrice
                                         CurrencyCode
                                         Memo
                                         Status
                                         LastChangedAt )
WITH VALUE #( ( %key-TravelID = lv_travel_id ) )
RESULT   DATA(lt_received_travel_data)
FAILED   DATA(ls_failed).
```

To retrieve the complete entity instance for the respective travel ID, you have to flag every element explicitly. For more information about EML, see [Consuming Business Objects with EML \[page 469\]](#).

5.3.3.2.2.4 Implementing the DELETE Operation for Travel Instances

This topic guides you through the implementation steps required to delete an existing travel instance.

Preview

In our scenario, the appropriate travel instance should be deleted when, for example, the user clicks the *Delete* button on the Fiori UI.

Travels (1) Standard ▾					Set to Booked	Delete	+	
Travel ID	Agency ID	Customer ID	Starting Date	End Date				
2	70007	93	Apr 28, 2018	Apr 28, 2018 >				
Travel Status: N								

Selecting a Travel Entry and Clicking the Delete Button

Travel ID	Agency ID	Customer ID	Starting Date
No items found.			
Object deleted			

Expected Behavior

Implementation Steps

1. Defining the Handler Class for Deletion of Travel Instances

Corresponding to the [template \[page 301\]](#) for the root node behavior implementation, a local handler class is defined to implement each changing operation. In this case, the `delete FOR MODIFY` is used to implement the delete operation for root instances. As given in the listing below, the signature of the `<method> FOR MODIFY` includes only one import parameter `it_travel_delete` for referring to the `travel` (root) instances to be deleted.

LISTING 1: Signature of the `delete_travel FOR MODIFY` (excerpt from template)

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
  METHODS:
    ...
    delete_travel FOR MODIFY
      IMPORTING     it_travel_delete      FOR DELETE travel,
    ...
  ENDCLASS.
```

2. Implementing the Deletion of Travel Instances

To delete travel instances, the function module `/DMO/FLIGHT_TRAVEL_DELETE` of the legacy business logic is called.

Each delete operation call can produce failed keys and messages (`lt_messages`). Failed keys are addressed by the content ID (`<fs_travel_delete>-%cid_ref [page 740]`) and the key value `<fs_travel_delete>-travel_id`). In case of failure, failed keys are saved in the `failed-travel` table, whereas the `reported-travel` table includes all instance-specific messages.

i [Expand the following listing to view the source code.](#)

LISTING 2: Deleting travel instances

```
CLASS lhc_travel IMPLEMENTATION.
  ...

```

```

METHOD delete_travel.
  DATA lt_messages TYPE /dmo/t_message.
  LOOP AT it_travel_delete ASSIGNING FIELD-SYMBOL(<fs_travel_delete>).
    CALL FUNCTION '7DMO/FLIGHT_TRAVEL_DELETE'
      EXPORTING
        iv_travel_id = <fs_travel_delete>-travelid
      IMPORTING
        et_messages = lt_messages.
    /dmo/cl_travel_auxiliary=>handle_travel_messages(
      EXPORTING
        iv_cid      = <fs_travel_delete>-%cid_ref
        iv_travel_id = <fs_travel_delete>-travelid
        it_messages = lt_messages
      CHANGING
        failed      = failed-travel
        reported    = reported-travel
    ).
  ENDLOOP.
ENDMETHOD.

...
ENDCLASS.

```

Related Information

[Handler Classes \[page 719\]](#)

[Saver Classes \[page 730\]](#)

[Declaration of Derived Data Types \[page 734\]](#)

[Implicit Returning Parameters \[page 738\]](#)

5.3.3.2.2.5 Implementing the CREATE Operation for Associated Bookings

In this topic, you will be guided through all implementation steps required for creation of new bookings.

In our demo application, we assume that new bookings cannot be created separately but can only in conjunction with a given travel instance.

The fact that new instances of the bookings can only be created for a specific travel instance is considered in the behavior definition by the `_Booking` association:

```

define behavior for /DMO/I_Travel_U alias travel
...
{
  ...
  association _Booking { create; }
}

```

The keyword `{ create; }` declares that this association is create-enabled what exactly means that instances of the associated bookings are created by a travel instance.

Preview

The figure below shows a list with booking items that belong to a travel instance.

The screenshot shows a Fiori object page for a travel instance. At the top left is the number '11'. To the right are buttons for 'Edit', 'Delete', and a refresh icon. Below this is a navigation bar with tabs: 'General Information' and 'Second Facet', with 'Second Facet' being the active tab. A search bar labeled 'Search' and a delete button are also in this header. The main area is titled 'Bookings (2)' and contains two rows of booking data. Each row has a radio button, a booking number, a booking date, a customer ID, and an airline ID. Row 1 (radio button selected) shows: Booking Number 1, Booking Date Aug 16, 2018, Customer ID 77, Airline ID UA. Below this row are details: Flight Number: 1537, Flight Date: Aug 18, 2018, Flight Price: 455.00 USD. Row 2 shows: Booking Number 2, Booking Date Nov 12, 2018, Customer ID 77, Airline ID AA. A red circle highlights the blue '+' icon in the top right corner of the booking list area. Below the list is a link labeled 'List of Bookings'.

To add a new booking to a selected travel instance, the end user must click the + icon and edit some fields to specify the required information for a new booking.

The screenshot shows an Fiori object page for editing individual booking values. It has a title 'General Information'. Below it are six input fields arranged in pairs: 'Booking Date:' with value 'Nov 29, 2018' and 'Flight Number:' with value '17'; 'Customer ID:' with value '99' and 'Flight Date:' with value 'Jun 12, 2019'; 'Airline ID:' with value 'AA' and 'Flight Price:' with value '462.00' and a currency dropdown set to 'USD'. At the bottom is a dark footer bar with 'Save' and 'Cancel' buttons. The entire page is titled 'Object Page for Editing Individual Booking Values'.

As soon as the user clicks the **Save** button on the Fiori object page, a booking data set with a new booking number is created.

Bookings (3)					
	Booking Number	Booking Date	Customer ID	Airline ID	Flight Number
1		Aug 16, 2018	77	UA	1537
	Flight Date: Aug 18, 2018				
	Flight Price: 455.00 USD				
2		Nov 12, 2018	77	AA	17
	Flight Date: Jun 12, 2019				
	Flight Price: 462.00 USD				
3		Nov 29, 2018	99	AA	17
	Flight Date: Jun 12, 2019				
	Flight Price: 462.00 USD				

New Booking Data Set

Implementation Steps

1. Defining Method for Creation of Associated Bookings

Corresponding to the [template \[page 301\]](#) for the root entity, the local handler class `lhc_travel` is defined to implement each changing operation in one individual `<method> FOR MODIFY`. In this case, the `create_booking_ba FOR MODIFY` method should only be used to implement the create operation for booking instances by means of an association. The signature of the `cba_travel FOR MODIFY` includes only one import parameter `it_booking_create_ba` to refer to the associated booking instances to be created.

To identify the associated bookings, the aliases for the root entity and the child entity are used - according to the aliases specified in the behavior definition. The association is expressed in the form:

```
... FOR CREATE root\_\_child\_entity.
```

LISTING 1: Signature of the `create_booking_ba FOR MODIFY` method (excerpt from template)

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS:
  ...
    cba_booking FOR MODIFY
      IMPORTING     it_booking_create_ba
      FOR CREATE
travel\_\_booking.
ENDCLASS.
```

2. Implementing a `FOR MODIFY` Method for Creation of New Booking Instances

As reproduced in the listing below, the implementation of the `FOR MODIFY` method is initiated by a loop across all selected travel instances for which associated bookings are to be created. Each selected travel (root) instance is represented by the travel ID as a primary key (`lv_travelid`).

Even in a case like this, it can happen that a consumer works with data that is not yet persisted and might not have a primary key yet (for example, if the primary key is going to be created later in the save sequence (late numbering)). In such cases, a temporary primary key, the content ID (`%CID`) for the travel instance is used as long as no primary key was created by BO runtime. The content ID is then written to the mapped-travel table.

Before a new booking data set is created, we first need to retrieve all bookings (`lt_booking_old`) that already exist for the selected travel instance. This is done by calling the `/DMO/FLIGHT_TRAVEL_READ` function module.

In case of failure, the message is handled by means of the `handle_travel_messages` method, as we know from the implementation of `FOR MODIFY` in the previous topics.

In case of success (`lt_messages IS INITIAL`), the maximum booking number has to be determined. This is done by the condition `COND # (WHEN ...)` where the last given booking number `lv_last_booking_id` is compared with the maximum booking ID `lv_max_booking_id`.

The creation of new bookings for a given travel instance takes place in a further loop across the booking instances to be created, which are addressed by the association `<fs_booking_create_ba>-%target`. This association includes the predefined component `%target`, which is used to address the target of composition.

```
<fs_booking_create_ba>-%target ASSIGNING FIELD-SYMBOL(<fs_booking_create>).
!king = CORRESPONDING FIELDS OF %target.
!_booking_crea
!ooking-bookin
!
!FUNCTION '/DMO
!RTING
:_travel    = V
:_travelx   = V
:_booking   = V
:_bookingx  = V
!
!RTING
:_messages = 1
!
_messages IS I
!RT VALUE #( %
!
message_help
!TING
:_cid      = <
:_travel_id =
:_messages = 1
!ING
:_led      = f
:_sorted   = r
!
! AT lt_message
F2
① <fs_booking_create_ba>
%cid_ref           type abp_behv_cid
travelid          type /dmo/travel_id
%target            type standard table of
%cid               type abp_behv_cid
travelid          type /dmo/travel_id
bookingid         type /dmo/booking_id
bookingdate       type /dmo/booking_date
customerid        type /dmo/customer_id
airlineid          type /dmo/carrier_id
connectionid      type /dmo/connection_id
flightdate         type /dmo/flight_date
flightprice        type /dmo/flight_price
currencycode       type /dmo/currency_code
lastchangedat     type timestamp
%control
travelid          type abp_behv_flag
bookingid         type abp_behv_flag
bookingdate       type abp_behv_flag
customerid        type abp_behv_flag
airlineid          type abp_behv_flag
connectionid      type abp_behv_flag
flightdate         type abp_behv_flag
flightprice        type abp_behv_flag
currencycode       type abp_behv_flag
lastchangedat     type abp_behv_flag
```

F2 on `<fs_booking_create_ba>` with the predefined component `%target`.

To provide the incoming structure for bookings with data, the mapping between the element in CDS views and the original table fields is required. This mapping is implemented by the operator `MAPPING FROM ENTITY USING CONTROL`, which maps the CDS view fields to the database table fields based on the mapping specification in the behavior definition and the control structure.

Before the new booking data sets are created by calling the function module /DMO/FLIGHT_TRAVEL_UPDATE, the booking ID for the booking instance to be created is easily determined by the statement ls_booking_id = lv_last_booking_id + 1.

In case of success, the values with the content ID %CID and the key values travelid, and bookingid are written to the mapped-booking table.

The function call can produce failed keys (<fs_booking_create>-%cid) and messages (lt_messages). Any failed keys are stored in the table failed-booking whereas the reported-booking table includes all messages that are specific for the failed booking instance

i [Expand the following listing to view the source code.](#)

LISTING 2: Creating booking instances by using association

```
CLASS lhc_travel IMPLEMENTATION.
  ...
  METHOD create_booking_ba.
    DATA lt_messages      TYPE /dmo/t_message.
    DATA lt_booking_old   TYPE /dmo/t_booking.
    DATA ls_booking        TYPE /dmo/booking.
    DATA lv_last_booking_id TYPE /dmo/booking_id VALUE '0'.
    LOOP AT it_booking_create_ba ASSIGNING FIELD-SYMBOL(<fs_booking_create_ba>).
      DATA(lv_travelid) = <fs_booking_create_ba>-travelid.
      CALL FUNCTION '/DMO/FLIGHT_TRAVEL_READ'
        EXPORTING
          iv_travel_id = lv_travelid
        IMPORTING
          et_booking    = lt_booking_old
          et_messages   = lt_messages.
      IF lt_messages IS INITIAL.
        IF lt_booking_old IS NOT INITIAL.
          lv_last_booking_id = lt_booking_old[ lines( lt_booking_old ) ]-
        booking_id.
        ENDIF.
        LOOP AT <fs_booking_create_ba>-%target ASSIGNING FIELD-
SYMBOL(<fs_booking_create>).
          ls_booking = CORRESPONDING #( <fs_booking_create> MAPPING FROM ENTITY
USING CONTROL ) .
          lv_last_booking_id += 1.
          ls_booking-booking_id = lv_last_booking_id.
          CALL FUNCTION '/DMO/FLIGHT_TRAVEL_UPDATE'
            EXPORTING
              is_travel     = VALUE /dmo/s_travel_in( travel_id = lv_travelid )
              is_travelx    = VALUE /dmo/s_travel_inx( travel_id = lv_travelid )
              it_booking    = VALUE /dmo/t_booking_in( ( CORRESPONDING
#( ls_booking ) ) )
              it_bookingx   = VALUE /dmo/t_booking_inx(
                (
                  booking_id = ls_booking-booking_id
                  action_code = /dmo/if_flight_legacy=>action_code-create
                )
              )
            IMPORTING
              et_messages = lt_messages.
          IF lt_messages IS INITIAL.
            INSERT
              VALUE #(
                %cid = <fs_booking_create>-%cid
                travelid = lv_travelid
                bookingid = ls_booking-booking_id
              )
              INTO TABLE mapped-booking.
          ELSE.
```

```

LOOP AT lt_messages INTO DATA(ls_message) WHERE msgty = 'E' OR msgty
= 'A'.
  INSERT VALUE #( %cid = <fs_booking_create>-%cid ) INTO TABLE
failed-booking.
  INSERT
    VALUE #( 
      %cid      = <fs_booking_create>-%cid
      travelid = <fs_booking_create>-TravelID
      %msg     = new_message(
        id       = ls_message-msgid
        number   = ls_message-msgno
        severity = if_abap_behv_message=>severity-error
        v1       = ls_message-msgv1
        v2       = ls_message-msgv2
        v3       = ls_message-msgv3
        v4       = ls_message-msgv4
      )
    )
  INTO TABLE reported-booking.
ENDLOOP.
ENDIF.
ENDLOOP.
ELSE.
  /dmo/cl_travel_auxiliary=>handle_travel_messages(
    EXPORTING
      iv_cid      = <fs_booking_create_ba>-%cid_ref
      iv_travel_id = lv_travelid
      it_messages  = lt_messages
    CHANGING
      failed      = failed-travel
      reported    = reported-travel
    ).
ENDIF.
ENDLOOP.
ENDMETHOD.

```

Related Information

[Handler Classes \[page 719\]](#)

[Declaration of Derived Data Types \[page 734\]](#)

[Implicit Returning Parameters \[page 738\]](#)

5.3.3.2.2.5.1 Implementing Dynamic Operation Control for Create By Association

This topic guides you through the required implementation steps to dynamically enable and disable the create by association operation for bookings.

Context

Depending on the value of the `Status` field of the travel instance, the create by association operation is enabled or disabled on the corresponding travel instance.

The Fiori Elements app preview displays the create button for the associated bookings on the object page only if the operation is enabled.

Implementation Steps

1. Defining dynamic feature control in the behavior definition

In the behavior definition, the feature control for the create by association is defined as follows:

```
implementation unmanaged;
define behavior for /DMO/I_Travel_U alias travel
implementation in class /DMO/BP_TRAVEL_U unique
...
{
  ...
  association _Booking { create (features: instance); }
  ...
}
```

2. Implementing dynamic feature control in the behavior pool

The `<method> FOR FEATURES` must be declared. For more information, see [<method> FOR FEATURES \[page 729\]](#).

The travel instance is read and, depending on the value of the `Status` field, the create by association is enabled or disabled.

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
  METHODS:
    ...
    get_features      FOR FEATURES
      IMPORTING keys      REQUEST requested_features FOR travel
      RESULT          result.
  ENDCLASS.
  CLASS lhc_travel IMPLEMENTATION.
  ...
  ****
  *
  * Implements the dynamic action handling for travel instances
  *
```

```

*****
METHOD get_features.
  READ ENTITY /dmo/i_travel_u
  FIELDS ( TravelID Status )
  WITH VALUE #(
    FOR keyval IN keys (
      %key          = keyval-%key
    )
  )
  RESULT DATA(lt_travel_result).
result = VALUE #(
  FOR ls_travel IN lt_travel_result (
    %key          = ls_travel-%key
    %assoc_Booking = COND #( WHEN ls_travel-Status = 'B'
                           THEN if_abap_behv=>fc-o-
                           disabled ELSE if_abap_behv=>fc-o-enabled )
  )
ENDMETHOD.
ENDCLASS.

```

For more information, see [Adding Feature Control \[page 459\]](#).

5.3.3.2.2.6 Implementing the READ Operation for Associated Bookings

This topic guides you through the implementation steps required to read booking instances associated to a travel instance from the buffer.

Context

Just like the `READ` for travel data, the `READ by association` does not have an explicit trigger on the UI. Nevertheless, the BO must be fully consumable by [EML \[page 812\]](#) and therefore it is necessary to implement a method that reads the bookings that are associated to a specific travel ID in our travel scenario.

The `READ by association` operation provides read access to the application buffer by reading child entity instances of a parent entity. It is used to retrieve data for further processing during the [interaction phase \[page 814\]](#).

Implementation Steps

The `READ by association` operation does not have to be declared in the behavior definition as it always implicitly assumed that `READ by association` is implemented. However, you can specify it explicitly:

```

define behavior for /DMO/I_Travel_U alias travel
{
  ...
  association _Booking;
}

```

i Note

Cloud If you use groups in your behavior definition, you must explicitly specify the READ by association operation in one of the groups. If CREATE by association is also enabled for the BO, the READ and the CREATE by association must be assigned to the same group due to its syntactical relation. For more information, see [Using Groups in Large Development Projects \[page 481\]](#).

1. Defining the signature of the <method> FOR READ by association

In the definition part of the handler class for travel, define the method `read_booking_ba` for `READ travel_Booking` with the entity input parameters `IMPORTING` and `FULL`, and the output parameters `,RESULT` and `LINK`.

The boolean parameter `FULL` indicates whether the consumer requests complete entity instances or only the keys which are returned with the parameter `LINK`.

The parameter `RESULT` returns the complete set of associated entity instances based on the control structure of the importing parameter.

The parameter `LINK` returns the key of the source entity and of the associated target entities.

LISTING 1: Signature of the <method> FOR READ by association

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS:  
    ...  
    read_booking_ba    FOR READ  
        IMPORTING it_travel    FOR READ travel\Booking  
            FULL iv_full_requested  
        RESULT      et_booking  
            LINK et_link_table,  
    ... .  
ENDCLASS.
```

2. Implementing the <method> FOR READ by association

To return the actual data from the application buffer, the imported travel ID is passed to the reading function module /DMO/FLIGHT_TRAVEL_READ. The function module returns the entity instances of the associated bookings corresponding to the travel ID and possible messages if an error occurs. If no message is returned, for each associated booking, fill the output parameter `et_link_table` with the corresponding source key for the travel instance and the target key for the booking instance. The result parameter `et_booking` can then be filled with the matching values for the fields that are flagged in the control structure if the full parameter `iv_full_requested` is set.

In the changing parameter `failed`, you specify the fail cause if something goes wrong. If, for example, the used function module returns the message number 16, the travel ID was not found. In this case, you can return the fail cause `not_found`. For all other error, return fail cause `unspecified`.

i [Expand the following listing to view the source code.](#)

LISTING 2: Reading associated booking instances from the application buffer

```
*****  
*  
* Read booking data by association from buffer  
*
```

```

*****METHOD read_booking_ba.
  DATA: ls_travel_out    TYPE /dmo/travel,
        lt_booking_out  TYPE /dmo/t_booking,
        ls_booking       LIKE LINE OF et_booking,
        lt_message       TYPE /dmo/t_message.
LOOP AT it_travel ASSIGNING FIELD-SYMBOL(<fs_travel_rba>).
  CALL FUNCTION '/DMO/FLIGHT_TRAVEL_READ'
    EXPORTING
      iv_travel_id = <fs_travel_rba>-travelid
    IMPORTING
      es_travel      = ls_travel_out
      et_booking     = lt_booking_out
      et_messages    = lt_message.
  IF lt_message IS INITIAL.
    LOOP AT lt_booking_out ASSIGNING FIELD-SYMBOL(<fs_booking>).
      "fill link table with key fields
      INSERT
        VALUE #(
          source-%key = <fs_travel_rba>-%key
          target-%key = VALUE #(
            TravelID   = <fs_booking>-travel_id
            BookingID = <fs_booking>-booking_id
          )
        )
      INTO TABLE et_link_table.
      "fill result parameter with flagged fields
    IF iv_full_requested = abap_true.
      ls_booking = CORRESPONDING #( <fs_booking> MAPPING TO ENTITY ).
      ls_booking-lastchangedat = ls_travel_out-lastchangedat.
      INSERT ls_booking INTO TABLE et_booking.
    ENDIF.
  ENDLOOP.
ELSE.
  "fill failed table in case of error
  failed-travel = VALUE #(
    BASE failed-travel
    FOR msg IN lt_message (
      %key = <fs_travel_rba>-TravelID
      %fail-cause = COND #(
        WHEN msg-msgty = 'E' AND msg-msgno = '016'
        THEN if_abap_behv=>cause-not_found
        ELSE if_abap_behv=>cause-unspecific
      )
    )
  .
ENDIF.
ENDLOOP.
ENDMETHOD.
ENDCLASS.
```

Testing

Just like the READ operation, the READ by association cannot be easily tested using a Fiori UI, as there is no direct execution trigger for reading the associated booking from the application buffer. Nevertheless, you can test your READ by association implementation by using EML.

Example

```

READ ENTITY /dmo/i_travel_u
BY \_Booking FIELDS ( AirlineID
                      BookingDate
```

```

        BookingID
        CustomerID
        CurrencyCode
        LastChangedAt )
WITH VALUE #( ( %key-TravelID = travelID ) )
RESULT DATA(lt_read_bookings_ba)
LINK DATA(lt_link_table)
FAILED DATA(ls_failed_rba).

```

If you request the `RESULT` parameter, the importing parameter `FULL` is set and you retrieve the values based on the control flags. The parameter `LINK` is independent on the control flags, it always retrieves the keys of the source and target entity.

5.3.3.2.2.7 Implementing the LOCK Operation

This topic guides you through the implementation steps required to lock instances on the database.

Context

Locking prevents simultaneous changes on the database.

The lock operation is executed before modify operations. It locks the entity instance during the time of the modify operation and disables other clients to modify the specific instance and all its locking related instances. As soon as the modify operation is finished, the lock is released.

For more information, see [Pessimistic Concurrency Control \(Locking\) \[page 91\]](#).

In our unmanaged scenario, the travel entity was defined as lock master and the booking entity as lock dependent. That means, the locking mechanism always locks the lock master instances and all its dependents. For example, if the lock operation is executed because a booking instance is updated, the parent travel instance and all its booking instances are also being locked.

To enable locking, the method `FOR LOCK` must be implemented. The legacy code of the unmanaged scenario includes a lock object, which must be called in the method `FOR LOCK` to lock the relevant entity instances.

Preview

In our scenario, it should not be possible to save changes on the database, while another entity instance of the same lock master entity instance is being changed at the same moment. In this case, the end user on the UI gets an error message:

Concurrent Changes of Entity Instances of the Same Lock Master Instance

Implementation Steps

The legacy code of the unmanaged scenario includes a lock object, which must be called in the `method FOR LOCK` to lock the relevant entity instances.

1. Defining the Handler Class for Locking Travel Instances

Corresponding to the [template \[page 301\]](#) for the root node behavior implementation, a local handler class is defined to implement each operation. In this case, the method `lock FOR LOCK` is used to implement the lock operation for the travel business object. As you can see in the listing below, the signature of `<method> FOR LOCK` includes only one importing parameter `it_travel_lock` for referring to the lock master's key.

LISTING 1: Signature of the `lock FOR LOCK` (excerpt from template)

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
  METHODS:
    ...
    lock          FOR LOCK
      IMPORTING it_travel_lock FOR LOCK travel,
    ...
  ENDCCLASS.
```

2. Implementing Locking of Travel Instances

To lock a travel instance and its lock dependents, instantiate the lock object `/DMO/ETRAVEL` of the legacy business logic. This is done by calling the factory method `get_instance` of `cl_abap_lock_object_factory`. This instantiation must not fail, except for technical errors. Handle the exception adequately, otherwise there will be a warning.

Call the `enqueue` method of the lock object and pass the imported travel ID to execute locking. If the instance is already locked, you will get an exception. This exception can be handled by the `handle_travel_messages` of the auxiliary class to fill `failed` and `reported` table.

i [Expand the following listing to view the source code.](#)

LISTING 2: Locking travel instances

```
CLASS lhc_travel IMPLEMENTATION.
  ...
  METHOD lock
    "Instantiate lock object
    DATA(lock) = cl_abap_lock_object_factory->get_instance( iv_name = '/DMO/
ETRAVEL' ).  

    LOOP AT it_travel_lock ASSIGNING FIELD-SYMBOL(<fs_travel>).
      TRY.
        "enqueue travel instance
        lock->enqueue(
          it_parameter = VALUE #( ( name = 'TRAVEL_ID' value = REF
#( <fs_travel>-travelid ) ) )
        ).  

        "exception is raised if foreign lock exists
        CATCH cx_abap_foreign_lock INTO DATA(lx_foreign_lock).
          /dmo/cl_travel_auxiliary->handle_travel_messages(
            EXPORTING
              iv_travel_id = <fs_travel>-TravelID
              it_messages = VALUE #( (
                msgid = '/DMO/CM_FLIGHT_LEGAC'
                msgty = 'E'
                msgno = '032'
                msgv1 = <fs_travel>-travelid
                msgv2 = lx_foreign_lock-
              >user_name )
            )
            CHANGING
              failed = failed-travel
              reported = reported-travel
            ).
        ENDTRY.
      ENDLOOP.
    ENDMETHOD.
  ...
ENDCLASS.
```

Testing

To test if the locking mechanism works properly, open the preview twice and set a breakpoint after the method enqueue.

In the first preview application, delete or update any entity instance. The ADT debugger stops at the breakpoint. At this point, the enqueue lock is already set on the chosen entity instance's lock master and its dependents. Try to do a modify operation with the other preview application on the same entity instance or any of its lock master's dependents. Do not stop at the breakpoint this time. You will see that the modify operation is rejected because the entity instance is locked by the client of the first preview application.

Related Information

[Handler Classes \[page 719\]](#)

[Saver Classes \[page 730\]](#)

[Declaration of Derived Data Types \[page 734\]](#)

5.3.3.2.2.8 Implementing the SET_STATUS_BOOKED Action

This topic describes the implementation of an action related to the travel instances. Using this action, the end user should be able to change the status of travel processing.

Preview

Again, we use the option of running the resulting app based on *Fiori Elements* to check the action execution. When we run the app, the UI screen provides the button *Set to Booked* for the action as shown in the figure below.

Travels (1) Standard ▾					Set to Booked	Delete	+	⚙️
Travel ID	Agency ID	Customer ID	Starting Date	End Date				
→ 4137	70007	99	Nov 17, 2018	Nov 24, 2018 >				
Travel Status: P								
Original Status is P (= In Process)								
Travel ID	Agency ID	Customer ID	Starting Date	End Date				
4137	70007	99	Nov 17, 2018	Nov 24, 2018 >				
Travel Status: B								
Changed Status After Action Execution (B = Booked)								

Implementation Steps

Once an action is defined in the behavior definition, it must be implemented in the behavior pool of the business object.

1. Defining the Method for Action

As described in [Action Implementation \[page 79\]](#) the method `FOR ACTION` must be declared in the private section of the behavior pool.

The beginning of the source code excerpt in the following listing shows the declaration of two table types, one for `ACTION IMPORT` (the importing parameter) and the other for `ACTION RESULT` (the exporting parameter).

The importing parameter is a freely selected name (in our case: `it_travel_set_status_booked`). The action name `set_status_booked` refers to the name of the action defined in the behavior definition and the

entity travel, on which the action is assigned. The result parameter is also a freely selected name (in our case: et_travel_set_status_booked).

LISTING 1: Defining the signature of set_travel_status FOR MODIFY

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
METHODS:  
  ...  
    set_travel_status FOR MODIFY  
      IMPORTING it_travel_set_status_booked FOR ACTION  
      travel~set_status_booked RESULT et_travel_set_status_booked.  
  ...  
ENDCLASS.
```

2. Implementing set_travel_status FOR MODIFY

To execute the actual action, the function module /DMO/FLIGHT_TRAVEL_SET_BOOKING of the legacy business logic is called.

The main implementation steps are the same as in the implementation of the CUD operations.

To fill the action result parameter accordingly to the action definition, a read must be executed.

i [Expand the following listing to view the source code.](#)

LISTING 2: Implementing the action set_travel_status

```
CLASS lhc_travel IMPLEMENTATION.  
  ...  
  
  METHOD set_travel_status.  
    DATA lt_messages TYPE /dmo/t_message.  
    DATA ls_travel_out TYPE /dmo/travel.  
    DATA ls_travel_set_status_booked LIKE LINE OF et_travel_set_status_booked.  
    CLEAR et_travel_set_status_booked.  
    LOOP AT it_travel_set_status_booked ASSIGNING FIELD-  
      SYMBOL(<fs_travel_set_status_booked>).  
      DATA(lv_travelid) = <fs_travel_set_status_booked>-travelid.  
      CALL FUNCTION '/DMO/FLIGHT_TRAVEL_SET_BOOKING'  
        EXPORTING  
          iv_travel_id = lv_travelid  
        IMPORTING  
          et_messages = lt_messages.  
      IF lt_messages IS INITIAL.  
  
        CALL FUNCTION '/DMO/FLIGHT_TRAVEL_READ'  
          EXPORTING  
            iv_travel_id = lv_travelid  
          IMPORTING  
            es_travel = ls_travel_out.  
            ls_travel_set_status_booked-travelid = lv_travelid.  
            ls_travel_set_status_booked-%param = CORRESPONDING  
      #( ls_travel_out MAPPING TO ENTITY ).  
      ls_travel_set_status_booked-%param-travelid = lv_travelid.  
      APPEND ls_travel_set_status_booked TO et_travel_set_status_booked.  
    ELSE.  
      /dmo/cl_travel_auxiliary=>handle_travel_messages(  
        EXPORTING  
          iv_cid = <fs_travel_set_status_booked>-%cid_ref  
          iv_travel_id = lv_travelid  
          it_messages = lt_messages  
        CHANGING
```

```

        failed      = failed-travel
        reported   = reported-travel
    ).
ENDIF.
ENDLOOP.
ENDMETHOD.
```

3. Implementing Dynamic Action Control

For dynamic control of actions acting on individual entity instances, the option (`features: instance`) must be added to the action `set_status_booked` in the behavior definition. The required implementation must be provided in the referenced class pool. In the implementation handler of the class pool you can specify the condition on which the action is enabled or disabled.

More on this: [Implementing Dynamic Feature Control \[page 468\]](#)

Related Information

[Adding Feature Control \[page 459\]](#)

[Handler Classes \[page 719\]](#)

[Saver Classes \[page 730\]](#)

[Declaration of Derived Data Types \[page 734\]](#)

[Implicit Returning Parameters \[page 738\]](#)

5.3.3.2.2.8.1 Implementing Dynamic Action Control

This topic guides you through the required implementation steps to dynamically enable and disable actions.

Preview

The following figure shows the effect of dynamic control on the action button `Set to Booked`: Since the selected travel instance has a status of `B` (Booked), the action button is disabled.



Travels (5,324) Standard					Set to Booked	Delete	Create	
Travel ID	Agency ID	Customer ID	Starting Date	End Date				
15	Aussie Travel (70028)	Miguel (466)	May 4, 2019	Feb 28, 2020 >				
16	Not Only By Bike (70050)	Kirk (640)	May 4, 2019	Mar 1, 2020 >				
Dynamic Feature Control for an Action								

Implementation Steps

1. Defining dynamic feature control in the behavior definition

In the behavior definition, the feature control for the action `set_status_booked` is defined as follows:

```
implementation unmanaged;
define behavior for /DMO/I_Travel_U alias travel
implementation in class /DMO/BP_TRAVEL_U unique
...
{
    ...
    action ( features : instance ) set_status_booked result [1] $self;
    ...
}
```

2. Adding dynamic feature control for actions to the method `get_features` in the behavior pool

The feature control for actions must be added to the method `get_features` in the behavior pool. You can implement feature control for different actions or operations in the same method.

Depending on the value of `Status` field, the action `set_status_booked` is enabled or disabled.

```
CLASS lhc_travel DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS:
    ...
    get_features      FOR FEATURES
                      IMPORTING keys REQUEST requested_features FOR
travel
                      RESULT result.
ENDCLASS.
CLASS lhc_travel IMPLEMENTATION.
...
*****  
*  
* Implements the dynamic action handling for travel instances  
*  
*****  
METHOD get_features.
READ ENTITY /dmo/i_travel_u
FIELDS ( TravelID Status )
WITH VALUE #(
    FOR keyval IN keys (
        %key = keyval-%key
    )
)
RESULT DATA(lt_travel_result).
result = VALUE #(
    FOR ls_travel IN lt_travel_result (
        %key = ls_travel-%key
        %features-%action-set_status_booked = COND #( WHEN ls_travel-Status = 'B'
            THEN if_abap_behv=>fc-o-
        disabled ELSE if_abap_behv=>fc-o-enabled )
        %assoc_Booking = COND #( WHEN ls_travel-Status = 'B'
            THEN if_abap_behv=>fc-o-
        disabled ELSE if_abap_behv=>fc-o-enabled )
    )
).
ENDMETHOD.
ENDCLASS.
```

For more information, see [Adding Feature Control \[page 459\]](#).

5.3.3.2.3 Creating the Behavior Pool for the Booking Child Entity

Procedure 1: Create a Behavior Pool /DMO/BP_BOOKING_U

To launch the wizard tool for creating a behavior implementation, do the following:

1. Launch ABAP Development Tools.
2. In your ABAP project (or *ABAP Cloud Project*), select the relevant behavior definition node (/DMO/I_TRAVEL_U) in *Project Explorer*.
3. Open the context menu and choose *New Behavior Implementation* to launch the creation wizard.

Further information:

- [Naming Conventions for Development Objects \[page 780\]](#)
- [\(Tool Reference\) \[page 764\]](#)

Results: Behavior Pool for the Booking Child Entity

The generated behavior pool (in our case /DMO/CL_BOOKING_U) provides you with an extension FOR BEHAVIOR OF.

```
► G /DMO/BP_BOOKING_U ►
1④ CLASS /dmo/bp_booking_u DEFINITION
2   PUBLIC
3   ABSTRACT
4   FINAL
5   FOR BEHAVIOR OF /dmo/i_travel_u .
6
7   PUBLIC SECTION.
8   PROTECTED SECTION.
9   PRIVATE SECTION.
10  ENDCCLASS.
11
12
13
14④ CLASS /dmo/bp_booking_u IMPLEMENTATION.
15  ENDCCLASS.

Global Class Class-relevant Local T... Local Types (highlighted in red) Test Classes (non exist...) Macros
New Behavior Pool
```

Procedure 2: Define a Skeleton of Local Classes Corresponding to the Behavior Model

Based on the declarations in the behavior definition /DMO/I_TRAVEL_U, and taking [best practices \[page 298\]](#) for modularization and performance into account, adapt the generated skeleton of the local classes for the child entity in accordance with the listing below:

⚠ Caution

In the current version of ADT tools, the skeleton with the code generated by the class pool creation wizard differs from the source code in the listing below.

i [Expand the following listing to view the source code.](#)

Listing: Template for local classes of /DMO/BP_BOOKING_U

```
*****
*
* Handler class implements UPDATE for booking instances
*
*****
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
  TYPES tt_booking_update      TYPE TABLE FOR UPDATE      /dmo/i_booking_u.
  METHODS:
    update_booking FOR MODIFY
      IMPORTING it_booking_update FOR UPDATE booking,
    delete_booking FOR MODIFY
      IMPORTING it_booking_delete FOR DELETE booking,
    read_booking FOR READ
      IMPORTING it_booking_read FOR READ booking
      RESULT      et_booking,
    cba_supplement FOR MODIFY
      IMPORTING it_supplement_create_ba FOR CREATE booking\_booksupplement,
    read_travel_ba FOR READ
      IMPORTING it_booking FOR READ booking\travel
        FULL iv_full_requested
      RESULT      et_travel
        LINK et_link_table.
  ENDCLASS.
  CLASS lhc_booking IMPLEMENTATION.
    METHOD update_booking.
    ENDMETHOD
    METHOD delete_booking.
    ENDMETHOD.
    METHOD read_booking.
    ENDMETHOD.
    METHOD read_travel_ba.
    ENDMETHOD.
  ENDCLASS.
```

5.3.3.2.3.1 Implementing the UPDATE, DELETE, and READ Operations for Booking Instances

This topic guides you through all implementation steps required for data updates and deletion of booking data sets.

The behavior definition in our demo application scenario requires the standard operations `update`, and `delete` for the booking child entity. Note that the `read` operation is not explicitly declared in the behavior definition, but implicitly expected. The `create` is handled in the travel behavior by a `create_by_association`. In addition, since we use different CDS view field names than in the database table, we need a mapping specification from CDS names to database fields names.

```
define behavior for /DMO/I_Booking_U alias booking
implementation in class /DMO/BP_BOOKING_U unique
{
    field ( read only ) TravelID, BookingID;
    field ( mandatory ) BookingDate, CustomerID, AirlineID, ConnectionID,
FlightDate;
    update;
    delete;
    mapping for /dmo/booking
    {
        AirlineID      = carrier_id;
        BookingDate   = booking_date;
        BookingID     = booking_id;
        ConnectionID  = connection_id;
        CurrencyCode  = currency_code;
        CustomerID    = customer_id;
        FlightDate    = flight_date;
        FlightPrice   = flight_price;
        TravelID      = travel_id;
    }
}
```

The `create` operation is already implemented by using the association relation between the `travel` root entity and the `booking` child entity. **Further information:** [Implementing the CREATE Operation for Associated Bookings \[page 320\]](#)

Preview (Update of Booking Data Sets)

In our application scenario, the appropriate booking data sets should be modifiable for all required fields of the booking instance when, for example, the user clicks the `Edit` button on the object page of the Fiori UI.



General Information

General Information

Booking Date:	Flight Number:
Jul 29, 2018	1537
Customer ID:	Flight Date:
93	Aug 18, 2018
Airline ID:	Flight Price:
UA	438.00 USD

Edit and Delete Buttons are Available for Each Booking Data Set

In change mode, the end user is able to change the relevant travel fields as shown in the figure below. As soon as the user chooses the **Save** button on the object page, the changed booking data is persisted on the database and a new version of the related booking instance is created.

Booking Date:	Flight Number:
Jul 29, 2018	1537
Customer ID:	Flight Date:
93	Aug 18, 2018
Airline ID:	Flight Price:
UA	438.00 USD

Object Page for Editing Individual Booking Values

Implementation Steps

1. Defining and Implementing UPDATE for Booking Data

Corresponding to the [template \[page 338\]](#) for behavior implementation of the booking child entity, one local handler class `lhc_booking` is defined to implement each changing operation in one individual `<method> FOR MODIFY`, one for updating booking data sets and another one for deleting bookings.

The `update_booking FOR MODIFY` method of the handler `lhc_booking` implements the update operation for bookings. The signature of this method includes only one import table parameter `it_booking_update` for referring to the booking instances to be updated.

To update data of bookings, the function module `/DMO/FLIGHT_TRAVEL_UPDATE` is called. In addition to the incoming parameters `is_travel` and `it_booking`, the corresponding flag structure `is_travelx` as well as the flag table type `it_bookingx` are used.

When updating booking data sets, we must first check which individual data was changed by the end user. This check is done by mapping the control structure of the importing parameter on a local structure that can be passed to the update function module.

Message handling for processing instance-specific messages in case of failure is implemented by the `handle_booking_messages` method. Failed keys are addressed by the booking content ID (`<fs_booking_update>-%cid_ref`) and the values for the travel ID (`<fs_booking_update>-travelid`) and the booking ID (`<fs_booking_update>-bookingid`). In case of failure, failed keys are saved in the failed-booking table, whereas the reported-booking table contains all instance-specific messages.

i [Expand the following listing to view the source code.](#)

LISTING 1: Handler class `lhc_booking`

```
*****
*
*   Handler class for managing bookings
*
*****
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
    TYPES:
      tt_booking_update          TYPE TABLE FOR UPDATE /dmo/i_booking_u,
      tt_bookingsupplement_create TYPE TABLE FOR CREATE /dmo/
i_bookingsupplement_u.
    METHODS:
      update_booking FOR MODIFY
        IMPORTING it_booking_update FOR UPDATE booking,
        ...
ENDCLASS.

CLASS lhc_booking IMPLEMENTATION.
  ****
  *
  * Implements the UPDATE operation for a set of booking instances
  *
  ****
  METHOD update_booking.
    DATA lt_messages TYPE /dmo/t_message.
    DATA ls_booking TYPE /dmo/booking.
    DATA ls_bookingx TYPE /dmo/s_booking_inx.
    LOOP AT it_booking_update ASSIGNING FIELD-SYMBOL(<fs_booking_update>).
      ls_booking = CORRESPONDING #( <fs_booking_update> MAPPING FROM ENTITY ).
      ls_bookingx-booking_id = <fs_booking_update>-BookingID.
      ls_bookingx-_intx      = CORRESPONDING #( <fs_booking_update> MAPPING FROM
ENTITY ).
```

```

ls_bookingx-action_code = /dmo/if_flight_legacy=>action_code-update.
CALL FUNCTION '/DMO/FLIGHT_TRAVEL_UPDATE'
  EXPORTING
    is_travel    = VALUE /dmo/s_travel_in( travel_id = <fs_booking_update>-
travelid )
    is_travelx   = VALUE /dmo/s_travel_inx( travel_id = <fs_booking_update>-
travelid )
    it_booking   = VALUE /dmo/t_booking_in( ( CORRESPONDING
#( ls_booking ) ) )
    it_bookingx  = VALUE /dmo/t_booking_inx( ( ls_bookingx ) )
  IMPORTING
    et_messages = lt_messages.
/dmo/cl_travel_auxiliary=>handle_booking_messages(
  EXPORTING
    iv_cid        = <fs_booking_update>-%cid_ref
    iv_travel_id  = <fs_booking_update>-travelid
    iv_booking_id = <fs_booking_update>-bookingid
    it_messages   = lt_messages
  CHANGING
    failed      = failed-booking
    reported    = reported-booking .
ENDLOOP.
ENDMETHOD.
...
ENDCLASS.

```

2. Adding Data Type Declarations for the Booking Entity in the Auxiliary Class

To use the required import, export, or changing parameters in the signature of methods to be defined for handling booking operations (later on, in the booking behavior pool's FOR MODIFY methods), we must first add the appropriate type declarations to the PUBLIC section in the definition part of our auxiliary class.

i [Expand the following listing to view the source code.](#)

LISTING 2: Data type declarations in the auxiliary class /dmo/cl_travel_auxiliary

```

CLASS /dmo/cl_travel_auxiliary DEFINITION
  INHERITING FROM cl_abap_behv
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.

  *  Type definition for import parameters -----
  TYPES tt_booking_failed    TYPE TABLE FOR FAILED      /dmo/i_booking_u.
  TYPES tt_booking_mapped     TYPE TABLE FOR MAPPED     /dmo/i_booking_u.
  TYPES tt_booking_reported  TYPE TABLE FOR REPORTED  /dmo/i_booking_u.

  ...
ENDCLASS.

```

3. Implementing Message Handling

When handling changing operations for booking instances, fault events may occur. For the processing of instance-specific messages in such a case, the method handle_booking_messages is used.

To refer to an individual data set where an error (msgty = 'E') or an abort (msgty = 'A') occurred, the FAILED table is used, whereas the instance-specific messages are stored in the REPORTED table.

However, messages that originate from the legacy code must be mapped to the messages of the class-based BO framework. To be reused in different behavior pools, the corresponding message handler method is defined and implemented in the helper class /dmo/cl_travel_auxiliary.

i [Expand the following listing to view the source code.](#)

LISTING 3: Declaration and implementation of the method handle_booking_messages

```

CLASS /dmo/cl_travel_auxiliary DEFINITION
  INHERITING FROM cl_abap_behv
  PUBLIC
  FINAL
  CREATE PUBLIC .
PUBLIC SECTION.

...
  CLASS-METHODS handle_booking_messages
    IMPORTING
      iv_cid          TYPE string OPTIONAL
      iv_travel_id   TYPE /dmo/travel_id OPTIONAL
      iv_booking_id  TYPE /dmo/booking_id OPTIONAL
      it_messages     TYPE /dmo/t_message
    CHANGING
      failed         TYPE tt_booking_failed
      reported       TYPE tt_booking_reported.
  ENDCLASS.

CLASS /dmo/cl_travel_auxiliary IMPLEMENTATION.

...
  METHOD handle_booking_messages.
    LOOP AT it_messages INTO DATA(ls_message) WHERE msgty = 'E' OR msgty = 'A'.
      APPEND VALUE #( %cid      = iv_cid
                      travelid  = iv_travel_id
                      bookingid = iv_booking_id ) TO failed.
      APPEND VALUE #( %msg = get_message_object( )->new_message(
                      id        = ls_message-msgid
                      number    = ls_message-msgno
                      severity  =
                      if_abap_behv_message=>severity-error
                      v1        = ls_message-msgv1
                      v2        = ls_message-msgv2
                      v3        = ls_message-msgv3
                      v4        = ls_message-msgv4 )
                      %key-TravelID = iv_travel_id
                      %cid          = iv_cid
                      TravelID      = iv_travel_id
                      BookingID     = iv_booking_id ) TO reported.
    ENDLOOP.
  ENDMETHOD.

...
ENDCLASS.

```

4. Defining and Implementing the DELETE operation for Booking Instances

In this case, the FOR MODIFY method is used to implement the delete operation for booking data sets. Therefore, the signature of the <method> FOR MODIFY includes only one import parameter `it_booking_delete` for referring to the booking instance to be deleted. To identify the child entity for bookings, the alias `booking` is used - according to the alias specified in the behavior definition.

The basic steps within the method implementation are similar to those we got to know in the previous method `update_booking`. The function module /DMO/FLIGHT_TRAVEL_UPDATE is also called for the delete operation. Here, as in the previous case, the same incoming parameters are used, including the flag structure `is_travelx` as well as the flag table type `it_bookingx`). The appropriate action code for deleting bookings is defined by the statement `action_code = /dmo/if_flight_legacy=>action_code-delete`.

i Expand the following listing to view the source code.

LISTING 4: Method delete_booking

```
*****  
*  
* Handler class for managing bookings  
*  
*****  
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
    ...  
    METHODS delete_booking FOR MODIFY  
        IMPORTING     it_booking_delete      FOR DELETE booking.  
ENDCLASS.  
*****  
* Implements the DELETE operation for a set of booking instances  
*****  
CLASS lhc_booking IMPLEMENTATION.  
    ...  
  
METHOD delete_booking.  
    DATA lt_messages TYPE /dmo/t_message.  
    LOOP AT it_booking_delete INTO DATA(ls_booking_delete).  
        CALL FUNCTION '/DMO/FLIGHT_TRAVEL_UPDATE'  
            EXPORTING  
                is_travel    = VALUE /dmo/s_travel_in( travel_id = ls_booking_delete-travelid )  
                is_travelx   = VALUE /dmo/s_travel_inx( travel_id = ls_booking_delete-travelid )  
                it_booking   = VALUE /dmo/t_booking_in( ( booking_id =  
ls_booking_delete-bookingid ) )  
                it_bookingx = VALUE /dmo/t_booking_inx( ( booking_id =  
ls_booking_delete-bookingid  
                                action_code = /dmo/  
if_flight_legacy=>action_code-delete ) )  
            IMPORTING  
                et_messages = lt_messages.  
            IF lt_messages IS NOT INITIAL.  
                /dmo/cl_travel_auxiliary=>handle_booking_messages(  
                    EXPORTING  
                        iv_cid          = ls_booking_delete-%cid_ref  
                        iv_travel_id    = ls_booking_delete-travelid  
                        iv_booking_id   = ls_booking_delete-bookingid  
                        it_messages     = lt_messages  
                    CHANGING  
                        failed         = failed-booking  
                        reported       = reported-booking ).  
            ENDIF.  
        ENDLOOP.  
    ENDMETHOD.  
ENDCLASS.
```

5. Defining and Implementing the READ operation for Booking Instances

The READ operation is not specified in the behavior definition. To complete the BO, it is expected that the READ for all entities is available to consume the BO via EML.

i Note

☞ If you use groups in your behavior definition, you must explicitly specify the READ operation in one of the groups. For more information, see [Using Groups in Large Development Projects \[page 481\]](#).

In the handler class for booking entities add the declaration of the method `read_booking FOR READ` with importing parameter `it_booking_read` and the result parameter `et_booking`.

For more information, see [<method> FOR READ \[page 726\]](#).

The implementation steps for reading booking entities are similar to those for reading travel entities, see [Implementing the READ Operation for Travel Data \[page 316\]](#). The same function module is used. It reads the booking instances based on the travel ID that is passed to the function module. For performance reasons, we only call the function module once for all bookings with the same travel ID. That is why, the loop at the importing parameter is grouped by the travel ID. As the function module retrieves all available bookings for the travel ID, the output table must be read for the bookings that match the importing parameter `it_booking_read`. The read results are then be passed to the result parameter `et_booking` for those fields that are requested by the consumer. The key values are always passed to the result.

The error handling is separated for booking IDs that are not found for the respective travel IDs, for travel IDs that are not found by the function module, and for other fail causes that cannot be specified.

i *Expand the following listing to view the source code.*

LISTING 5: Method `read_booking`

```
*****
*
* Handler class for managing bookings
*
*****
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
  ...
  METHODS read_booking FOR READ
    IMPORTING it_booking_read FOR READ booking
    RESULT      et_booking.
ENDCLASS.
*****
* Implements the READ operation for a set of booking instances
*****
CLASS lhc_booking IMPLEMENTATION.
  ...
  METHOD read_booking.
    DATA: ls_travel_out  TYPE /dmo/travel,
          lt_booking_out TYPE /dmo/t_booking,
          lt_message      TYPE /dmo/t_message.
    "Only one function call for each requested travelid
    LOOP AT it_booking_read ASSIGNING FIELD-SYMBOL(<fs_travel_read>)
      GROUP BY <fs_travel_read>-travelid .
    CALL FUNCTION '/DMO/FLIGHT_TRAVEL_READ'
      EXPORTING
        iv_travel_id = <fs_travel_read>-travelid
      IMPORTING
        es_travel     = ls_travel_out
        et_booking    = lt_booking_out
        et_messages   = lt_message.
    IF lt_message IS INITIAL.
      "For each travelID find the requested bookings
      LOOP AT GROUP <fs_travel_read> ASSIGNING FIELD-SYMBOL(<fs_booking_read>).
        READ TABLE lt_booking_out INTO DATA(ls_booking) WITH KEY travel_id =
<fs_booking_read>-%key-TravelID
                                         booking_id =
<fs_booking_read>-%key-BookingID .
      "if read was successfull
      IF sy-subrc = 0.
        "fill result parameter with flagged fields
```

```

      INSERT
      VALUE #( travelid      = ls_booking-travel_id
              bookingid     = ls_booking-booking_id
              bookingdate   = COND #( WHEN <fs_booking_read>-%control-
BookingDate      = cl_abap_behv=>flag_changed THEN ls_booking-booking_date )
                          customerid    = COND #( WHEN <fs_booking_read>-%control-
CustomerID       = cl_abap_behv=>flag_changed THEN ls_booking-customer_id )
                          airlineid     = COND #( WHEN <fs_booking_read>-%control-
AirlineID         = cl_abap_behv=>flag_changed THEN ls_booking-carrier_id )
                          connectionid  = COND #( WHEN <fs_booking_read>-%control-
ConnectionID     = cl_abap_behv=>flag_changed THEN ls_booking-connection_id )
                          flightdate    = COND #( WHEN <fs_booking_read>-%control-
FlightDate        = cl_abap_behv=>flag_changed THEN ls_booking-flight_date )
                          flightprice   = COND #( WHEN <fs_booking_read>-%control-
FlightPrice       = cl_abap_behv=>flag_changed THEN ls_booking-flight_price )
                          currencycode  = COND #( WHEN <fs_booking_read>-%control-
CurrencyCode      = cl_abap_behv=>flag_changed THEN ls_booking-currency_code )
*                      lastchangedat = COND #( WHEN <fs_booking_read>-
%control-LastChangedAt = cl_abap_behv=>flag_changed THEN ls_travel_out-
lastchangedat    )
                           ) INTO TABLE et_booking.
      ELSE.
        "BookingID not found
      INSERT
      VALUE #( travelid      = <fs_booking_read>-TravelID
              bookingid     = <fs_booking_read>-BookingID
              %fail-cause   = if_abap_behv=>cause-not_found )
                           INTO TABLE failed-booking.
      ENDIF.
      ENDLOOP.
      ELSE.
        "TravelID not found or other fail cause
        LOOP AT GROUP <fs_travel_read> ASSIGNING <fs_booking_read>.
          failed-booking = VALUE #(  BASE failed-booking
                                      FOR msg IN lt_message ( %key-TravelID      =
<fs_booking_read>-TravelID
                                      %key-BookingID      =
<fs_booking_read>-BookingID
                                      %fail-cause        =
COND #( WHEN msg-msgty = 'E' AND msg-msgno = '016'
        THEN if_abap_behv=>cause-not_found
        ELSE if_abap_behv=>cause-unspecific ) ) .
        ENDLOOP.
      ENDIF.
      ENDLOOP.
ENDMETHOD.
```

6. Defining and Implementing the READ Operation for the Associated Travel Instance

The read by association from booking instances to their parent travel instance is used to read the ETag master of the booking instances. That is why, it is essential to support complete ETag handling for the travel BO. In addition, the read by association must be implemented to complete the transactional handling for the BO, so that it is consumable via EML.

In order to use the association transactionally, in this case, via a transactional read, the association must be specified in the behavior definition for the booking instance:

```

define behavior for /DMO/I_Booking_U alias booking
{
  ...
  association _Travel;
  ...
}
```

In the handler class for booking entities, add the declaration of the method `read_travel_ba` FOR READ `booking\travel` with importing parameter `it_booking`, FULL `iv_full_requested`, RESULT `et_travel` and LINK `et_link_table`.

For more information, see [<method> FOR READ By Association \[page 728\]](#).

In the implementation for reading travel instances by association from booking entities, loop at the incoming booking and group by the incoming `TravelID`. This grouping optimizes the performance as the function call is only executed once for each travel ID. If the function call to read the associated travel instance is successful, group at the group `<fs_travel>` to fill the link table for each requested booking. If all fields of travel are requested, fill the result parameter `et_travel` with the values that are read by the function module. To ensure that only those fields are filled that were requested (indicated by the %control structure of the importing parameter), the corresponding operator with the addition MAPPING TO ENTITY is used.

If the function modules returns messages, fill the `failed` table with the corresponding entries. The error handling is separated for booking IDs that are not found for the respective travel IDs, for travel IDs that are not found by the function module, and for other fail causes that cannot be specified.

i *Expand the following listing to view the source code.*

LISTING 6: Method `read_travel_ba`

```
*****
*
* Handler class for managing bookings
*
*****
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
...
METHODS read_travel_ba FOR READ
    IMPORTING it_booking FOR READ booking\travel FULL
    iv_full_requested
    RESULT      et_travel LINK et_link_table.
ENDCLASS.
*****
* Implements the READ operation for the travel instance
*****
CLASS lhc_booking IMPLEMENTATION.
...
METHOD read_travel_ba.
  DATA: ls_travel_out  TYPE /dmo/travel,
        lt_booking_out TYPE /dmo/t_booking,
        ls_travel       LIKE LINE OF et_travel,
        lt_message      TYPE /dmo/t_message.
  "Only one function call for each requested travelid
  LOOP AT it_booking ASSIGNING FIELD-SYMBOL(<fs_travel>)
    GROUP BY <fs_travel>-TravelID.
    CALL FUNCTION '/DMO/FLIGHT_TRAVEL_READ'
      EXPORTING
        iv_travel_id = <fs_travel>-%key-TravelID
      IMPORTING
        es_travel     = ls_travel_out
        et_messages   = lt_message.
    IF lt_message IS INITIAL.
      LOOP AT GROUP <fs_travel> ASSIGNING FIELD-SYMBOL(<fs_booking>).
        "fill link table with key fields
        INSERT VALUE #(
          source-%key = <fs_booking>-%key
          target-%key = ls_travel_out-travel_id )
        INTO TABLE et_link_table.
    IF iv_full_requested = abap_true.
      "fill result parameter with flagged fields
```

```

        ls_travel = CORRESPONDING #( ls_travel_out MAPPING TO ENTITY ).
        INSERT ls_travel INTO TABLE et_travel.
ENDIF.
ENDLOOP.
ELSE. "fill failed table in case of error
failed-booking = VALUE #( BASE failed-booking
                           FOR msg IN lt_message ( %key-TravelID      =
<fs_travel>-%key-TravelID
                           %key-BookingID      =
<fs_travel>-%key-BookingID
                           %fail-cause          = COND
#( WHEN msg-msgty = 'E' AND msg-msgno = '016'
THEN if_abap_behv=>cause-not_found
ELSE if_abap_behv=>cause-unspecific ) ) .
ENDIF.
ENDLOOP.
ENDMETHOD.
ENDCLASS.
```

Checking Results

At this point, you have the opportunity to check how the resulting app works for the CREATE and the UPDATE. For this to happen, however, a suitable business service for UI consumption must first be defined and published.

For more information, see: [Defining Business Service for Fiori UI \[page 350\]](#)

The READ cannot be easily tested using a Fiori UI, as there is no direct execution trigger for reading the data from the application buffer. Nevertheless, you can test your READ implementation by using EML.

❖ Example

```

READ ENTITY /dmo/i_booking_U
FIELDS ( TravelID
          BookingID
          CustomerID
          BookingDate )
WITH VALUE #( ( %key = VALUE #( TravelID = travelid
                                BookingID = bookingid ) ) )
RESULT DATA(et_bookings)
FAILED DATA(et_failed_booking).
```

To retrieve the complete entity instance for the respective travel ID, you have to set every element explicitly. For more information about EML, see [Consuming Business Objects with EML \[page 469\]](#).

5.3.3.3 Projecting the Behavior

You project the behavior to define the behavior for the specific business service.

Whereas the general business object defines and implements the behavior of what can be done in general with the data provided by the data model, the BO projection defines only the behavior that is relevant for the

specific service. In this travel scenario, the BO projection does not define a specific role of the end user, but projects the entire behavior for the UI service.

In the behavior projection, you only define the behavior that is relevant for the specific service. The implementation of the individual characteristics is only done in the general BO. The projection behavior definition delegates to the underlying layer for the behavior that is defined in the projection layer.

For more information, see [Business Object Projection \[page 100\]](#).

Creating the Projection Behavior Definition for the Travel BO

To create a projection behavior definition, do the following:

1. Open the context menu of the root projection view /DMO/C_Travel_U.
2. Choose *New Behavior Definition*.
3. In the wizard to create a new behavior definition, choose the implementation type *Projection* to get the template for projection.

A new projection behavior definition is created that defines the behavior for the specific UI service. The behavior definition uses the same name as the root projection view.

For the UI service, the entire behavior that is defined in the underlying BO is exposed. This includes:

- the etag to prevent overwriting on simultaneous editing
- the standard operations create, update, delete for the root entity,
- the standard operations update and delete for the child entity,
- the operation create by association for the child entity,
- the action set_status_booked on the root entity.

The following listing displays the source code of the projection behavior definition.

i [Expand the following listing to view the source code.](#)

Listing 1: Source Code of the Projection Behavior Definition /DMO/C_Travel_U

```
projection;
define behavior for /DMO/C_Travel_U alias travel
use etag
{
  use create;
  use update;
  use delete;
  use action set_status_booked;
  use association _BOOKING { create; }
}
define behavior for /DMO/C_Booking_U alias booking
{
  use update;
  use delete;
}
```

For more information, see [Projection Behavior Definition \[page 107\]](#).

5.3.4 Defining Business Service for Fiori UI

This section explains how you can model an OData service based on the data model and the related behavior model. A service like this consists of two artifacts, a service definition and a service binding.

The **service definition** is a projection of the data model and the related behavior to be exposed, whereas the **service binding** implements a specific protocol and the kind of service offered to a consumer.

Further information: [Business Service \[page 98\]](#)

Steps Relevant to Developers

1. Create the service definition
2. Specify which CDS entities are exposed as a UI service
3. Create a service binding
4. Publish the UI service locally
5. [Optional] Run the resulting app

5.3.4.1 Exposing the Relevant CDS Views as a Service

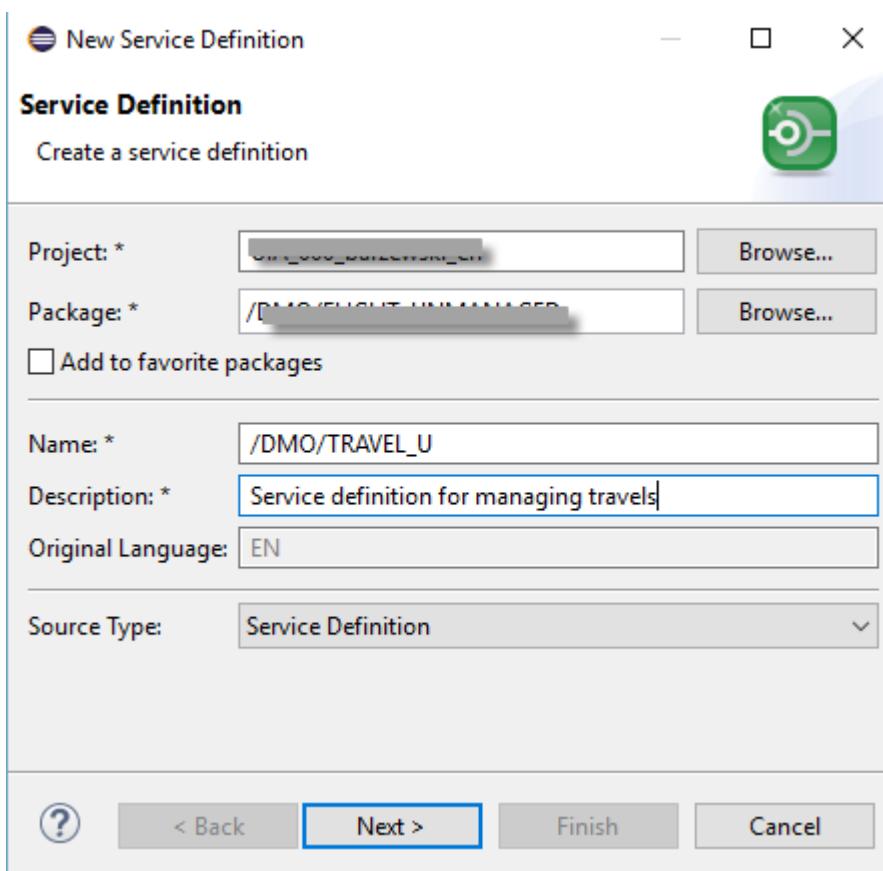
To describe the consumer-specific perspective as a data model, you need to create a business service definition (service definition for short) as an ABAP Repository object. A service definition represents the service model that is derived from the underlying CDS-based data model.

Further information: [Service Definition \[page 108\]](#)

Procedure: Creating a Service Definition

To launch the wizard tool for creating a service definition, do the following:

1. Launch [ABAP Development Tools](#).
2. In your ABAP project (or [ABAP Cloud Project](#)), select the relevant package node in [Project Explorer](#).
3. Open the context menu and choose  [New Other ABAP Repository Object](#)  [Business Services](#)  [Service Definition](#)  to launch the creation wizard.



Creating a Service Definition

Further information: [Creating Service Definitions \[page 767\]](#) (Tool Reference)

Procedure: Define Which CDS Entities Are Exposed as a UI Service

As in the entries in the listing below, add the following entities for to expose as a service:

LISTING 1: Service Definition /DMO/TRAVEL_U

```
@EndUserText.label: 'Service definition for managing travels'
define service /DMO/TRAVEL_U {
    expose /DMO/C_Travel_U as Travel;
    expose /DMO/C_Booking_U as Booking;
    expose /DMO/I_Agency_U as TravelAgency;
    expose /DMO/I_Customer_U as Passenger;
    expose I_Currency as Currency;
    expose I_Country as Country;
    expose /DMO/I_Carrier_U as Airline;
    expose /DMO/I_Connection_U as FlightConnection;
    expose /DMO/I_Flight_U as Flight;
}
```

The entire source code of a service definition for managing travels is included within the single bracket { ... }. It groups all the related CDS entities which are to be exposed as part of the UI service - including

their compositions and associations with the relevant entities. Note that the value help provider or text provider views must also be exposed for the OData service to make use of the value help and text associations.

Further information: [Syntax for Defining a Service \[page 109\]](#)

5.3.4.2 Creating a Service Binding

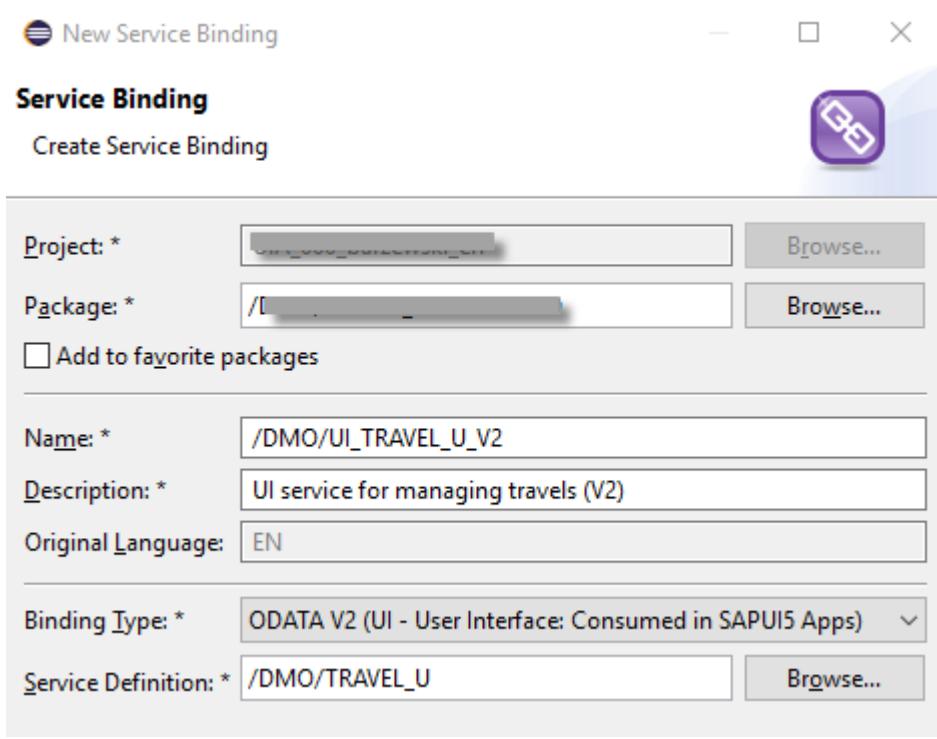
Using the business service binding (service binding for short), you can bind a service definition to a client-server communication protocol.

Further information: [Service Binding \[page 111\]](#) (Concept Information)

Procedure: Creating a Service Binding

To launch the wizard tool for creating a service binding, do the following:

1. Launch [ABAP Development Tools](#).
2. In your ABAP project (or [ABAP Cloud Project](#)), select the relevant package node in [Project Explorer](#).
3. Open the context menu and choose [New Service Binding](#) to launch the creation wizard.



Further information: [Creating Service Binding \[page 769\]](#) (Tool Reference)

Procedure: Activating the New Service

After successful activation, the editor provides additional information about the entire entity set as well as about the navigation path of the respective entity.

The screenshot shows the SAP Fiori Launchpad Service Binding Editor. At the top, it displays the title "Service Binding: /DMO/UI_TRAVEL_U_V2". Below this is a "General Information" section with the sub-section "This section describes general information about this service binding". Under "Binding Type", it says "ODATA V2 (UI - User Interface: Consumed in SAPUI5 Apps)". On the left, there's a "Service Versions" table with one row selected: "Vers... Service Defi... 000 /DMO/TRAVEL". To the right of the table are buttons for "Add..." and "Remove". In the center, there's a "View information on selected service version" section. It shows "Local service endpoint: Active" and "Local Service Endpoint Information" with the "Service URL" set to "/sap/opu/odata/DMO/UI_TRAVEL_U_V2". Below this is a "Type filter text" input field. On the right, under "Entity Set and Association", there's a tree view of entities: TravelAgency, to_Country, BookingSupplement, Booking, Airline, FlightConnection, Passenger, Flight, Supplement, SupplementText, Travel, to_Agency, to_Booking, to_Customer, and to_Customer. The "to_Customer" node is highlighted with a blue border, and a tooltip "Open Fiori Elements App Preview" is shown over it. The bottom of the editor is labeled "Service Binding Editor".

[Optional] Procedure: Running the Resulting UI Service

As soon as the service is activated, it is ready for consumption through an OData client such as an SAP Fiori app.

In the course of the UI development in the SAP Web IDE, you have the option of testing the resulting app within the SAP Fiori launchpad environment.

→ Tip

Alternatively, you can use the preview function in the service binding to check how the UI of a Fiori application looks like. **More on this:** [Previewing the Resulting UI Service \[page 33\]](#)

5.3.5 Adding Another Layer to the Transactional Data Model

In this section, you get a brief overview on how you can proceed when going to add a further layer to the flight demo scenario. Specifically, a booking supplement entity is now to be added to the previous 2-tier layer of the travel business object.

Entities for Transactional Data

As depicted in the figure below, the 3-tier entity hierarchy for managing transactional data in our scenario consists of the following editable entities, with a 1: n cardinality on each level:

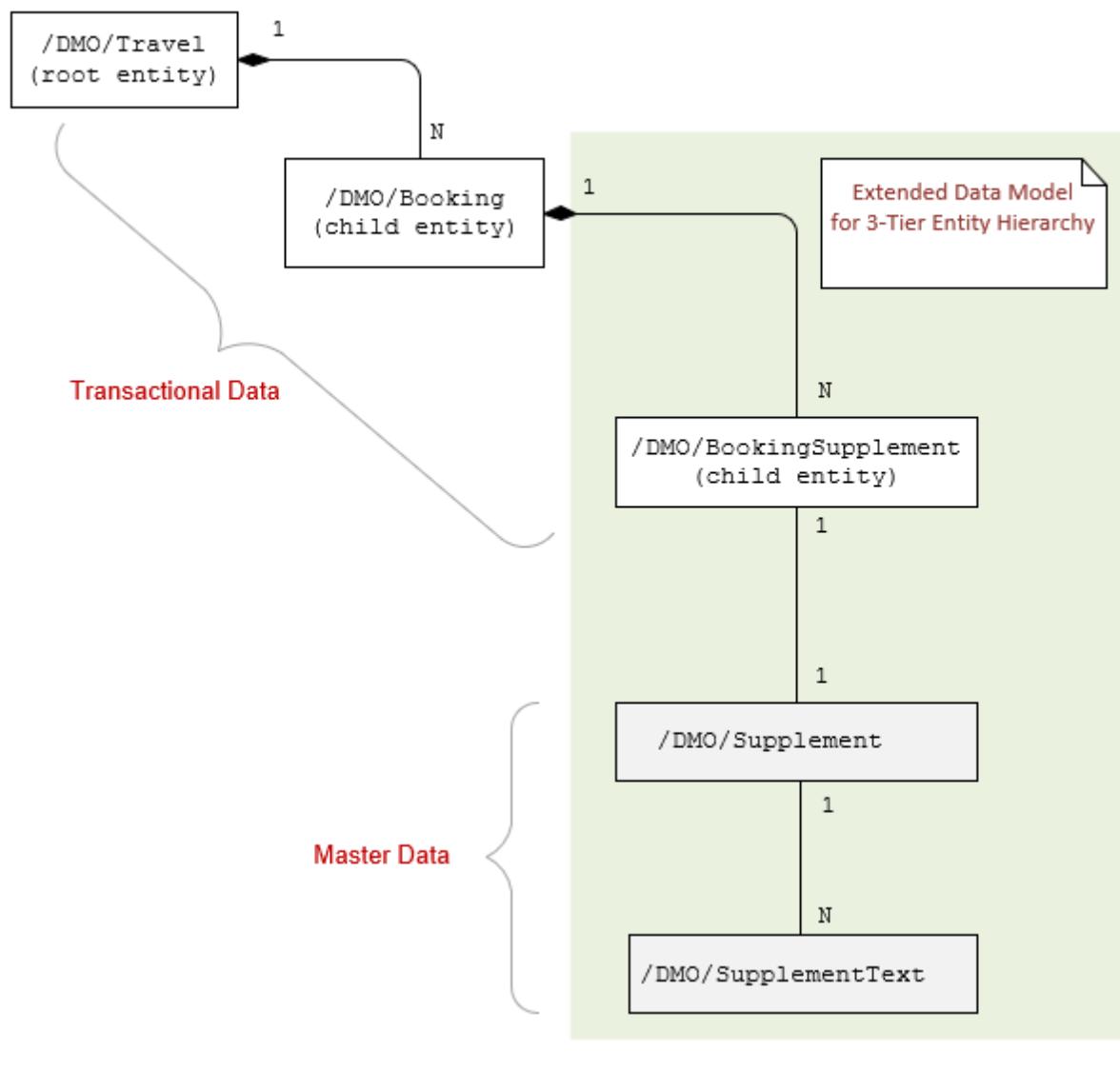
- Travel
- Booking
- **BookingSupplement**

That is, each travel instance has 0..n bookings and each booking has 0..n booking supplements.

Additional Entities for Master Data

To populate the booking supplement instances with business data, you also need to extend the data model for master data by the following entities:

- **Supplement**
- **SupplementText**



Data Model for 3-tier Entity Hierarchy

i Note

With the knowledge so far, you can easily reproduce the concrete steps of this extension. Therefore, in this topic, we will only outline the implementation steps in a nutshell and refer to the full implementation as it is available in the demo ABAP package `/DMO/FLIGHT/UNMANAGED`.

1. Extending the Data Model

Adding the CDS Entities to Extend the Data Model

Extended Data Model

Entity	Description	Editable
BookingSupplement	This entity is used to add additional products to a travel booking. The booking supplement data is stored in the database table / DMO/BOOK_SUPPL. The flight data model defines an n:1 cardinality between a Booking_Supplement entity and a Booking entity.	Yes
Supplement	A Supplement entity defines product data that the customer can book together with a flight, for example a drink or a meal. The supplement data is stored in the database table / DMO/SUPPLEMENT. The flight data model defines a 1:1 cardinality between a Supplement entity and the Booking_Supplement entity.	No
SupplementText	This entity mainly serves a text provider for the associated elements in the target entity BookingSupplement. By using a text association, you define the relationship between an element of the target entity and its corresponding texts or descriptions.	NO

More on this: [Getting Text Through Text Associations \[page 424\]](#)

The BookingSupplement entity is part of the compositional hierarchy of the travel business object. This composition relationship requires an association to their compositional parent entity. This relationship is expressed by the keyword ASSOCIATION TO PARENT. Using this syntax, you define the CDS entity BookingSupplement as a sub node in the compositional hierarchy of the travel business object structure. To access master data from other entities, additional associations _Product and _SupplementText are defined in the CDS source code. The associated views are primarily used as a text view for retrieving text information and as value help provider for specific product fields.

i *Expand the following listing to view the source code.*

Listing 1: Source Code of the CDS View /DMO/I_BookingSupplement_U

```
@AbapCatalog.sqlViewName: '/DMO/IBOOKSUPP_U'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Booking Supplement view - CDS data model'
define view /DMO/I_BookingSupplement_U
  as select from /dmo/book_suppl as BookingSupplement
    association to parent /DMO/I_Booking_U as _Booking
      on $projection.TravelID = _Booking.TravelID
      and
      $projection.BookingID = _Booking.BookingID
      association [1..1] to /DMO/I_Supplement as _Product
        on $projection.SupplementID = _Product.SupplementID
        association [1..*] to /DMO/I_SupplementText as _SupplementText
          on $projection.SupplementID = _SupplementText.SupplementID
{
  key BookingSupplement.travel_id as TravelID,
  key BookingSupplement.booking_id as BookingID,
  key BookingSupplement.booking_supplement_id as BookingSupplementID,
```

```

        BookingSupplement.supplement_id      as SupplementID,
        @Semantics.amount.currencyCode: 'CurrencyCode'
        BookingSupplement.price            as Price,
        @Semantics.currencyCode: true
        BookingSupplement.currency_code    as CurrencyCode,
        /* Associations */
        _Booking,
        _Product,
        _SupplementText
    }
}

```

Since the `BookingSupplement` entity is part of the compositional hierarchy of the business object, it is projected in a projection view. In the projection view, the text relationships, the search and value helps are defined via CDS annotations. In addition, the association to parent must be redirected to `/DMO/C_Booking_U`.

i [Expand the following listing to view the source code.](#)

Listing 2: Source Code of the CDS Projection View `/DMO/C_BookingSupplement_U`

```

@EndUserText.label: 'Booking Supplement Projection View'
@AccessControl.authorizationCheck: #NOT_REQUIRED
@Metadata.allowExtensions: true
@Search.searchable: true
define view entity /DMO/C_BookingSupplement_U
    as projection on /DMO/I_BookingSupplement_U
{
    ///DMO/I_BookingSupplement_U
    @Search.defaultSearchElement: true
    key TravelID,
        @Search.defaultSearchElement: true
    key BookingID,
    key BookingSupplementID,
        @Consumption.valueHelpDefinition: [ {entity: { name:      '/DMO/
I_SUPPLEMENT',
                                                element: 'SupplementID' },
                                             additionalBinding: [ { localElement:
'Price',           element: 'Price'},
                                         { localElement:
'CurrencyCode', element: 'CurrencyCode' } ] } ]
        @ObjectModel.text.element: ['SupplementText']
    SupplementID,
    _SupplementText.Description as SupplementText : localized,
    Price,
    @Consumption.valueHelpDefinition: [ { entity: { name:      'I_Currency',
                                                element: 'Currency' } } ]
    CurrencyCode,
    /* Associations */
    ///DMO/I_BookingSupplement_U
    _Booking : redirected to parent /DMO/C_Booking_U,
    _Product,
    _SupplementText
}

```

The `BookingSupplement` Entity also has a representation in the UI. It must therefore be equipped with UI annotations, which are outsourced to metadata extension as done with the other BO entities.

i [Expand the following listing to view the source code.](#)

Listing 3: Source Code of the Metadata Extension `/DMO/C_BookingSupplement_U`

```

@Metadata.layer: #CORE
@UI: { headerInfo: { typeName: 'Booking Supplement',
                     typeNamePlural: 'Booking Supplements',
                     title: { type: #STANDARD, label: 'Booking Supplement',
                           value: 'BookingSupplementID' } } }

```

```

annotate view /DMO/C_BookingSupplement_U with
{
  @UI.facet: [ { id: 'BookingSupplement',
    purpose: '#STANDARD',
    type: '#IDENTIFICATION_REFERENCE',
    label: 'Booking Supplement',
    position: 10 },
    { id: 'PriceHeader',
      type: '#DATAPPOINT_REFERENCE',
      purpose: '#HEADER',
      targetQualifier: 'Price',
      label: 'Price',
      position: 20 } ]

  @UI: { lineItem: [ { position: 10, importance: #HIGH } ],
    identification: [ { position: 10 } ] }
  BookingSupplementID;

  @UI: { lineItem: [ { position: 20, importance: #HIGH } ],
    identification: [ { position: 20 } ] }
  SupplementID;
  @UI: { lineItem: [ { position: 30, importance: #HIGH } ],
    identification: [ { position: 30 } ],
    dataPoint: { title: 'Price' } }
  }
  Price;
}

```

i Note

To complete the compositional hierarchy of the business object, you need to add the composition to child in the `Booking` entity `/DMO/I_Booking_U`, as well as the redirected composition in the projection view `/DMO/C_Booking_U`.

In order to display and change booking supplement data on Fiori UI, you must add the corresponding UI facet in the booking metadata extension `/DMO/C_Booking_U`.

Adding a CDS View for Supplements

Listing 4 (below) provides you with a data definition for additional products that a customer can book together with a flight booking.

i [Expand the following listing to view the source code.](#)

Listing 4: CDS View `/DMO/I_Supplement`

```

@AbapCatalog.sqlViewName: '/DMO/ISUPPL'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Supplement View - CDS Data Model'
@Search.searchable: true
define view /DMO/I_Supplement
  as select from /dmo/supplement as Supplement
  association [0..*] to /DMO/I_SupplementText as _SupplText on
  $projection.SupplementID = _SupplText.SupplementID
  association [0..1] to I_Currency as _Currency on
  $projection.CurrencyCode = _Currency.Currency
{
  @ObjectModel.text.association: '_SupplText'
  key Supplement.supplement_id as SupplementID,
  @Semantics.amount.currencyCode: 'CurrencyCode'
}

```

```

        Supplement.price      as Price,
        @Semantics.currencyCode: true
        Supplement.currency_code as CurrencyCode,
        /* Associations */
        @Search.defaultSearchElement: true
        _SupplText,
        _Currency
    }
}

```

Adding a Text Provider View for Supplements

Listing 5 (below) provides you with a data definition that serves as text provider. It provides text data through text associations as defined in the booking supplement view .

i [Expand the following listing to view the source code.](#)

Listing 5: CDS View /DMO/I_SupplementText

```

@AbapCatalog.sqlViewName: '/DMO/ISUPPTXT'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Supplement Text View - CDS Data Model'
@Search.searchable: true
define view /DMO/I_SupplementText
    as select from /dmo/suppl_text as SupplementText
{
    @ObjectModel.text.element: ['Description']
    key SupplementText.supplement_id as SupplementID,
        @Semantics.language: true
    key SupplementText.language_code as LanguageCode,
        @Search.defaultSearchElement: true
        @Search.fuzzinessThreshold: 0.8
        @Semantics.text: true
        SupplementText.description as Description
}

```

2. Extending the Behavior Definition for the BookingSupplement Entity

The fact that in our scenario new instances of the booking supplement entity can only be created for a specific travel and booking instance is considered by the addition of the `_BookSupplement` association. The keyword `{create;}` declares that this association is create-enabled what exactly means that instances of the associated booking supplements can be created by an individual booking instance.

The sub node of travel business object refers to the corresponding data model for booking supplements that is represented by the child entity for `/DMO/I_BookingSupplement_U`. The transactional behavior of the booking supplement sub node is determined by the standard operations `create`, `update`, and `delete`. In addition, since we use different CDS view field names than in the database table, we need a mapping specification from CDS names to database fields names.

Listing 6: Extended Behavior Definition /DMO/I_TRAVEL_U

```

implementation unmanaged;
define behavior for /DMO/I_Travel_U alias travel
{...
}
define behavior for /DMO/I_Booking_U alias booking

```

```

{
  ...
  association _BookSupplement { create; }
}
define behavior for /DMO/I_BOOKINGSUPPLEMENT_U alias bookingsupplement
implementation in class /DMO/BP_BOOKINGSUPPLEMENT_U unique
etag dependent by _Travel
lock dependent by _Travel
{
  field (read only) TravelID, BookingID, BookingSupplementID;
  field (mandatory) SupplementID, Price;
  create;
  update;
  delete;
  association _Travel;
  mapping for /dmo/book_suppl control /dmo/s_booking_supplement_intx
  {
    BookingID          = booking_id;
    BookingSupplementID = booking_supplement_id;
    CurrencyCode       = currency_code;
    Price              = price;
    SupplementID       = supplement_id;
    TravelID           = travel_id;
  }
}

```

3. Creating and Implementing the Behavior Pool for BookingSupplement

Creating a Behavior Pool for BookingSupplement Entity

Listing 7: Behavior Pool for BookingSupplement Child Entity

```

CLASS /dmo/bp_bookingsupplement_u DEFINITION
  PUBLIC
  ABSTRACT
  FINAL
  FOR BEHAVIOR OF /dmo/i_travel_u.
  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
CLASS /dmo/bp_bookingsupplement_u IMPLEMENTATION.
ENDCLASS.

```

Implementing the Handler for UPDATE, DELETE, READ and READ BY ASSOCIATION

You will find the same basic structure when implementing the handler methods for update and delete:

- A loop on booking supplement instances to be updated or deleted.
- Call of the business logic function module.
- Error handling for processing messages in case of failure.

Listing 8: Updating data of booking supplements in lhc_supplement

```

CLASS lhc_supplement DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
    TYPES tt_booking_update TYPE TABLE FOR UPDATE      /dmo/i_booking_u.
    METHODS update_bookingsupplement FOR MODIFY

```

```

    IMPORTING it_bookingsupplement_update FOR UPDATE bookingsupplement.
    ...
ENDCLASS.
CLASS lhc_supplement IMPLEMENTATION.
METHOD update_bookingsupplement.
    ...
ENDMETHOD.
ENDCLASS.

```

Listing 9: Deleting booking supplements in lhc_supplement

```

CLASS lhc_supplement DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
...
METHODS delete_bookingsupplement FOR MODIFY
    IMPORTING it_bookingsupplement_delete FOR DELETE bookingsupplement.
ENDCLASS.
CLASS lhc_supplement IMPLEMENTATION.
...
METHOD delete_bookingsupplement.
    ...
ENDMETHOD.
ENDCLASS.

```

Listing 10: Reading booking supplements in lhc_supplement

```

CLASS lhc_supplement DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
...
METHODS read_bookingsupplement FOR READ
    IMPORTING it_bookingsupplement_read FOR READ bookingsupplement
    RESULT      et_bookingsupplement.
ENDCLASS.
CLASS lhc_supplement IMPLEMENTATION.
...
METHOD read_bookingsupplement.
    ...
ENDMETHOD.
ENDCLASS.

```

Listing 11: Reading travel by association in lhc_supplement

```

CLASS lhc_supplement DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
...
METHODS read_travel_ba FOR READ
    IMPORTING it_bookingsupplmement FOR READ bookingsupplement\_Travel
    FULL iv_full_requested
    RESULT      et_travel LINK et_link_table.
ENDCLASS.
CLASS lhc_supplement IMPLEMENTATION.
...
METHOD read_travel_by.
    ...
ENDMETHOD.
ENDCLASS.

```

Implementing the CREATE Handler for Associated Booking Supplements

In our demo scenario, we assume that new booking supplements cannot be created separately but only in conjunction with a given booking that, in turn, belongs to an individual travel instance.

To identify the associated booking supplements, the aliases for the parent entity and the child entity are used - according to the aliases specified in the behavior definition. The association is expressed in the signature of the implementing method `cba_supplement` of the booking behavior pool `/DMO/BP_BOOKING_U` in the form: ...
FOR CREATE parent_child_entity.

Listing 12: Creating booking supplements in the `cba_supplement` method

```
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
  TYPES tt_bookingsupplement_create TYPE TABLE FOR CREATE /dmo/  
i_bookingsupplement_u.  
  METHODS cba_supplement FOR MODIFY  
    IMPORTING it_supplement_create_ba FOR CREATE booking  
  \_booksupplement.  
  ...  
ENDCLASS.  
CLASS lhc_booking IMPLEMENTATION.  
  ...  
  METHOD cba_supplement  
  ...  
ENDMETHOD.  
ENDCLASS.
```

Booking Supplement

Book. Supp. Number:

*Product ID:
ML-0032

*Product Price:
4.50 EUR

Save Cancel

Preview – Object Page when Creating Booking Supplements

Book. Supp. Number	Product ID	Product Price
1	Mango Juice (BV-0007)	4.51 EUR >
2	Vanilla Ice Cream with Hot Raspberries (ML-0032)	4.50 EUR >

Preview - Saved Data Displayed in the List of Items

Implementing the READ for Associated Booking Supplements

To complete the BO, you must implement a READ operation for associated bookings. The READ by association is implicitly defined in the behavior definition in the behavior for bookings by the definition of CREATE by association:

```
define behavior for /DMO/I_Booking_U alias booking
{
  ...
  association _BookSupplement { create; }
}
```

The declaration and the implementation of the method to read booking supplements by association is done in the handler class for bookings /DMO/BP_BOOKING_U in the same manner as in [Implementing the READ Operation for Associated Bookings \[page 327\]](#):

- Group the imported booking keys by travel ID
- Call the function module /DMO/FLIGHT_TRAVEL_READ
- Filter for the requested booking keys
- Fill the link table with the respective source and target keys
- Fill the result parameter, if the full parameter is set
- Handle errors

Listing 13: Read booking supplements in the read_supplement_ba method

```
CLASS lhc_booking DEFINITION INHERITING FROM cl_abap_behavior_handler.
  PRIVATE SECTION.
    TYPES tt_booksupplement_create TYPE TABLE FOR CREATE /dmo/
i_booksupplement_u.
    METHODS read_supplement_ba FOR READ
      IMPORTING it_booking FOR READ booking\booksupplement FULL
iv_full_requested
      RESULT et_booksuppl LINK et_link_table,
      ...
  ENDCLASS.
  CLASS lhc_booking IMPLEMENTATION.
  ...
  METHOD read_supplement_ba
  ...
ENDMETHOD.
```

```
ENDCLASS.
```

The READ by association can be tested by using EML.

4. Projecting the Behavior for the Booking Supplement Entity

As described in [Projecting the Behavior \[page 348\]](#), the behavior of the travel BO must be projected in the projection behavior definition /DMO/C_Travel_U.

The behavior that is relevant for the booking supplement entity is

- the standard operation update and delete on the behavior node /DMO/C_BookingSupplement_U and
- the operation create_by_association for booking supplements on the behavior node /DMO/C_Booking_U.

i [Expand the following listing to view the source code.](#)

Listing 11: Projection Behavior Definition /DMO/C_Travel_U

```
projection;
define behavior for /DMO/C_Travel_U alias travel
use etag
{
  use create;
  use update;
  use delete;
  use action set_status_booked;
  use association _BOOKING { create; }
}
define behavior for /DMO/C_Booking_U alias booking
use etag
{
  use update;
  use delete;
  use association _BOOKSUPPLEMENT { create; }
}
define behavior for /DMO/C_BookingSupplement_U alias bookingsupplement
use etag
{
  use update;
  use delete;
}
```

5. Exposing New Entities in the Service Definition

Corresponding to the listing below, the following CDS entities are to be exposed with the UI service:

Listing 12: Added Entities in the Service Definition

```
@EndUserText.label: 'Service definition for managing travels'
define service /DMO/TRAVEL_U {
  ...
  expose /DMO/C_BookingSupplement_U as BookingSupplement;
  expose /DMO/I_Supplement as Supplement;
  expose /DMO/I_SupplementText as SupplementText;
```

```
...  
}
```

5.4 Developing a Web API

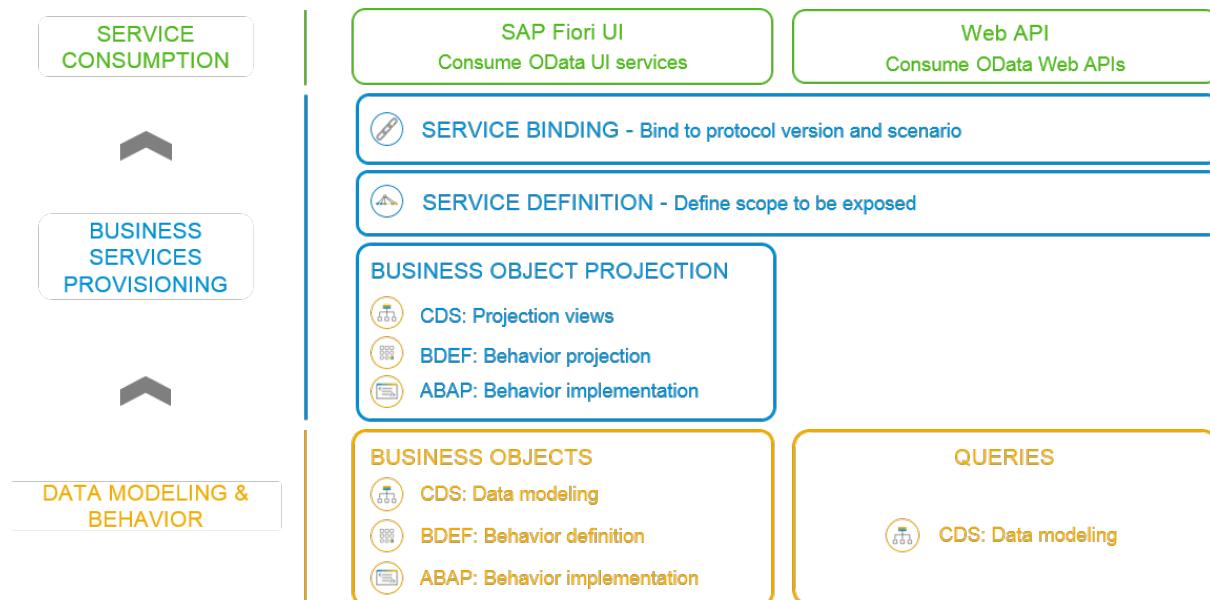
An OData service can be published as a web API. This enables the consumption of the service by any client, independent from a particular UI.

Introduction

A [Web API \[page 822\]](#) is an OData service whose metadata does not entail any UI-specific annotations that are defined for the data model. It is published for the purpose of providing an API to access the service by an unspecified client. A Web API facilitates the exchange of business information between an application and any client, including from a different system or server.

In this development guide you will reuse the service that you created in [Developing Unmanaged Transactional Apps \[page 263\]](#) and publish a Web API for it. Since a Web API is not being used directly in a UI context, the consumer of a service of this type only requires a reduced set of metadata. The metadata lacks any kind of UI-relevant information.

The basis for the service remains identical to an OData service exposed for a UI. It is just the binding type in the service binding that differs, which can be seen in the subsequent figure. It is even possible to expose a service that was created originally as a UI service for API consumption. The metadata are automatically reduced to the relevant information for Web API, which means without UI-related annotations or value helps.



Cloud icon: The procedure of consuming a remote Web API service like this from a foreign system is described in the following development guide [Developing a UI Service with Access to a Remote Service \[page 369\]](#).

Prerequisites

Developing the scenario that is described in the subsequent chapters requires the following:

- ☐ You have access to and an account for **SAP Cloud Platform, ABAP environment**.
- You have installed ABAP Development Tools (ADT). SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.
- To recreate the demo scenario, the *ABAP Flight Reference Scenario* must be available in your ABAP system. You can download the complete reference scenario from [GitHub: Downloading the ABAP Flight Reference Scenario \[page 13\]](#).
- You have understood the development steps to create a transactional OData service for a UI as described in [Developing Unmanaged Transactional Apps \[page 263\]](#).

In particular, you are able to reuse the data model including the behavior of the existing OData service /DMO/TRAVEL_U to expose it for a Web API.

Via ABAPGit You can import the service /DMO/TRAVEL_U including the related development objects into your development environment. So you do not have to build the service to test the publishing as Web API. You find the service in the package /DMO/FLIGHT_UNMANAGED.

For information about downloading the ABAP Flight Reference Scenario, see [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

Example Scenario

As described above, the following guide reuses the data model and behavior of the service that was created in the transactional guide. This means that the following artifacts must be available in your system to follow the steps of this guide.

Artefact	Description
CDS Entities	
/DMO/I_TRAVEL_U	The Travel entity defines general travel data, such as the agency ID or customer ID, status of the travel booking, and the price of travel. The entity represents the root node of the business object
/DMO/I_Booking_U	The Booking entity manages data for a booked flight for a certain travel instance. It is a composition of the Travel entity and is therefore dependent on its root.
/DMO/I_Agency	The Agency defines general data about the responsible agency for a travel. It is associated with the Travel entity.
/DMO/I_Customer	The Customer defines personal data about the customers involved in travel and booking. It is associated with the Travel entity and the Booking entity.
Behavior Artifacts	

Artefact	Description
/DMO/I_TRAVEL_U	The behavior definition defines the capabilities of the business object involved.
/DMO/CL_TRAVEL_U	The behavior is implemented in the special ABAP classes.
Business Service Artifact	
/DMO/TRAVEL_U	The service definition defines all the entities that are exposed for the service.

5.4.1 Publishing a Web API

Prerequisites

You have an existing service definition for which you want to create a Web API service. In our example scenario we reuse the service definition [/DMO/TRAVEL_U](#), which was already exposed as a UI service in the transactional scenario.

If no service definition is available, choose the entities that you want to expose as an API and create a service definition for these entities. For a description on how to create a service definition, refer to [Creating a Service Definition \[page 24\]](#).

Context

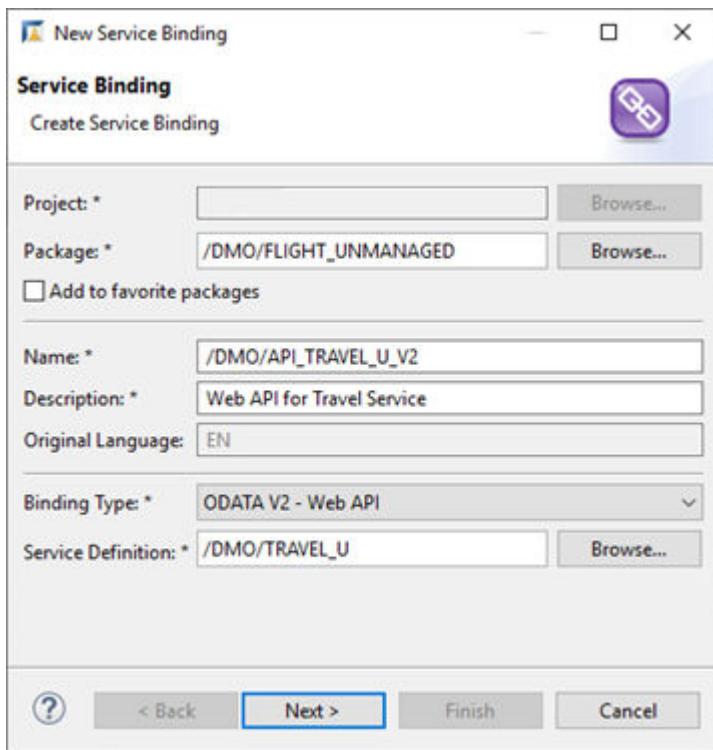
You have defined the scope of the service that you want to expose in a service definition. The service must now be bound to an OData protocol and published as a Web API.

Procedure

1. In your ABAP project, open the context menu for the existing service definition [/DMO/TRAVEL_U](#) and choose [New Service Binding](#) to launch the creation wizard.
2. In addition to the [Project](#) and [Package](#), enter the [Name](#) and the [Description](#) for the service binding you want to create.

i Note

The maximum length for the name of a service binding is 26 characters.



3. Select the **Binding Type ODATA V2 – Web API** to bind the service to a V2 protocol and expose it as a Web API.
4. Verify that the correct **Service Definition** is preset in the wizard to be used as a base for your service binding.
5. Choose **Next**.
6. Assign a transport request and choose **Finish**.

The ABAP back end creates a service binding and stores it in the ABAP Repository.

In the **Project Explorer**, the new service binding is added to the **Business Services** folder of the corresponding package node. As a result, the service binding form editor is opened and you can verify the information of the service.

Service Binding: /DMO/API_TRAVEL_U_V2

General Information																				
This section describes general information about this service binding																				
Binding Type:	ODATA V2 - Web API																			
Service Versions		Service Version Details																		
Define service versions associated with the service binding		View information on selected service version																		
type filter text <table border="1"> <thead> <tr> <th>Version</th> <th>API State</th> <th>Service Definition</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>Not Released</td> <td>/DMO/TRAVEL_U</td> </tr> <tr> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>		Version	API State	Service Definition	0001	Not Released	/DMO/TRAVEL_U													Default Authorization Values: 468B31104FB1CA24482EE0992275BAHT Local Service Endpoint: Inactive Activate Local Service Endpoint: Local service endpoint is not active. To view service URL, entity sets and associations, <a>activate local service endpoint.
Version	API State	Service Definition																		
0001	Not Released	/DMO/TRAVEL_U																		

Service Binding Artifact Form Editor for a Web API

7. To expose the service, choose the button **Activate**.

The metadata document can be accessed by following the link [Service URL](#). The UI preview is naturally not available.

Results

Except for UI features, the service including all implemented features is now exposed as an API. In particular, this means that the implemented behavior is also exposed in this API.

The difference to a service that is published for a UI client can best be seen in the service metadata. Whereas the metadata of a UI service carries information about the UI representation of the service, the Web API service does not contain any such information.

• Example

In our example scenario this difference can be seen when comparing the annotation section that refers to the implemented value help.

The UI service lists every field of a value help provider view as `ValueListParameter` in the annotation section. The Web API on the other hand lacks any UI specific information, although the value help annotation is defined in the respective CDS views.

5.5 Developing a UI Service with Access to a Remote Service

Based on a remote OData service, you create a new service that enhances the remote service with additional information.

Introduction

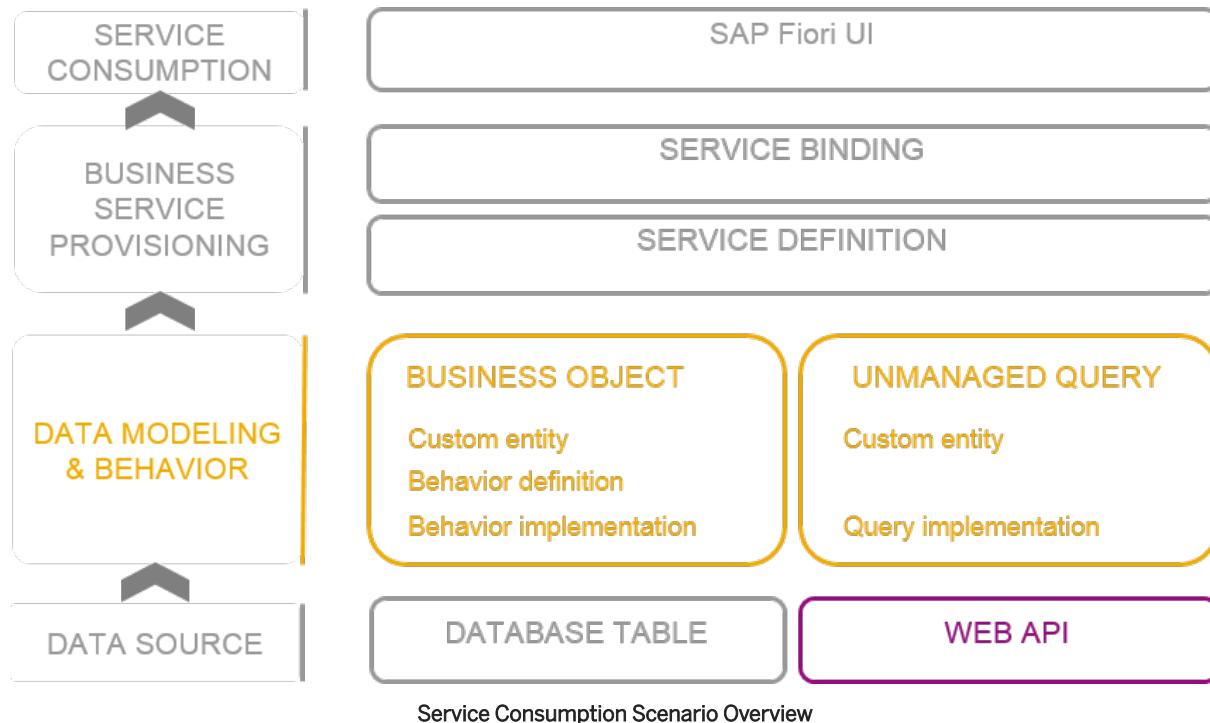
In this scenario, you develop an OData service that consumes a [Web API \[page 822\]](#). This Web API represents a remote service from which you retrieve data. This data is then enhanced with additional data. That means, the new OData service has two data sources. One is the remote service, the other source is a database table in the consuming development system. You build a new SAP Fiori UI application that consumes the remote service from the provisioning tenant, and enhances the remote service with persistent data of the consuming tenant.

The data model for the new service is defined in a [CDS custom entity \[page 809\]](#), which enables you to retrieve data from different sources. A CDS custom entity has an [unmanaged query \[page 821\]](#), which has to be implemented manually in a related ABAP class, the [query implementation class \[page 818\]](#). In this ABAP class, data from different sources, including from another system, can be retrieved.

With the help of the [service consumption model \[page 819\]](#), you can import the OData metadata of the remote service into the consuming system and create proxy artifacts of the remote service. This gives you a local representation of the remote data model in your tenant, which helps you to define a data model for the new service.

This guide also describes how you implement transactional behavior to modify the additional data on the local database table.

The following image gives an overview of the architecture of the service consumption scenario.



To be able to get data from a remote service, you build an [OData client proxy \[page 816\]](#) in your implementation to forward and transform the requests for the remote service. The client proxy can only consume the remote service if a connection to the provisioning tenant is established. The configuration of such a destination is a precondition for developing this scenario.

Development Steps to Create a New Service to Consume a Remote Service

- Save the XML file of the remote service locally.
- Create proxy artifacts with the service consumption model wizard.
- Create a CDS custom entity as a data model for the new service.
- Implement the query in an ABAP class.
- Implement the transactional behavior for the local fields.
- Create an OData service.

Prerequisites

Developing the scenario that is described in the subsequent chapters requires the following:

- You have access to and an account for **SAP Cloud Platform, ABAP environment**.

- You have installed ABAP Development Tools (ADT).
SAP recommends to use the latest version of the client installation. The ADT download is available on the update site <https://tools.hana.ondemand.com/>.
- To recreate the demo scenario, the *ABAP Flight Reference Scenario* must be available in your ABAP system. You can download the complete reference scenario from [GitHub: Downloading the ABAP Flight Reference Scenario \[page 13\]](#).
- You know the URL of the remote service that you want to consume or have access to the CSDL XML file of the service.
In this demo scenario, the service /DMO/API_TRAVEL_U_V2 is used as remote service. You can download the service as part of the ABAP Flight Reference Scenario, see [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

Prerequisites for the Consumption of a Remote Service

- A cloud destination is configured in the Cloud Cockpit to connect to the provisioning system.
For more information, see [Configure a Destination for the Sample Apps](#).
- A communication arrangement for the destination service instance in the ABAP environment has been created in your consuming system (SAP_COM_0276).
For more information, see [Creating a Communication Arrangement for the Destination Service Instance in the ABAP Environment](#)

Prerequisites to Consume a Remote Service from an S/4 HANA Cloud System

This scenario consumes a remote service from an S/4HANA Cloud System. You can consume any OData service, but different prerequisites might apply.

The following prerequisites apply for this consumption scenario:

- The remote OData service has been published in a communication scenario.
For more information, see [Consuming Services in the Context of API with Technical Users](#).
- A communication arrangement exists in the provisioning system.
For more information, see [Implementation Steps in the SAP S/4HANA Cloud System](#).

5.5.1 Scenario Description

Starting Point: Remote OData Service

In this development guide, we reuse the service that was built in [Developing a Web API \[page 365\]](#). We only use the root node from the remote service for our consumption and the transactional behavior is ignored. The new service in the consuming system does not implement operations to create, update, or delete data from the provisioning system. Instead, it retrieves the data from there and enriches the service with additional fields. These additional fields, however, can be modified. Therefore, the new UI service contains transactional capabilities for these additional fields.

A Web API service is the prototypical example for this consumption scenario. It contains all the required information, but does not contain any metadata that is superfluous in the consumption scenario (for example, UI-relevant annotations). However, it is also possible to consume an OData service that was published as a UI service.

The available entity types of the remote service can be viewed in the metadata document when extended. We only consume the root node *Travel*.

i Expand the following item to view the metadata of the remote service.

```

<Schema xml:lang="en" xmlns="http://schemas.microsoft.com/ado/2008/09/edmx" Namespace="cds_xdmoxi_travel_u_sd" sap:schema-version="1">
  + <EntityType sap:content-version="1" sap:label="Agency view" Name="TravelAgencyType">
  + <EntityType sap:content-version="1" sap:label="Booking view" Name="BookingType">
  + <EntityType sap:content-version="1" sap:label="Carrier view" Name="AirlineType">
  + <EntityType sap:content-version="1" sap:label="Connection view" Name="FlightConnectionType">
  + <EntityType sap:content-version="1" sap:label="Customer view" CDS data model="PassengerType" sap:value-list="true">
  + <EntityType sap:content-version="1" sap:label="Flight view" Name="FlightType" sap:value-list="true">
  + <EntityType sap:content-version="1" sap:label="Travel view" CDS data model="TravelType">
    - <Key>
      <PropertyRef Name="TravelID"/>
    />
    + <Property sap:label="Travel ID" Name="TravelID" sap:quickinfo="Flight Reference Scenario: Travel ID" sap:display-format="NonNegative" MaxLength="8" Nullable="false" Type="Edm.String"/>
    + <Property sap:label="Agency ID" Name="AgencyID" sap:quickinfo="Flight Reference Scenario: Agency ID" sap:display-format="NonNegative" MaxLength="6" Type="Edm.String" sap:value-list="standard"/>
    + <Property sap:label="Customer ID" Name="CustomerID" sap:quickinfo="Flight Reference Scenario: Customer ID" sap:display-format="NonNegative" MaxLength="6" Type="Edm.String" sap:value-list="standard"/>
    + <Property sap:label="Start Date" Name="StartDate" sap:quickinfo="Flight Reference Scenario: Start Date" sap:display-format="Date" Type="Edm.DateTime" Precision="0"/>
    + <Property sap:label="End Date" Name="EndDate" sap:quickinfo="Flight Reference Scenario: End Date" sap:display-format="Date" Type="Edm.DateTime" Precision="0"/>
    + <Property sap:label="Booking Fee" Name="BookingFee" sap:quickinfo="Flight Reference Scenario: Booking Fee" Type="Edm.Decimal" Precision="17" sapunit="CurrencyCode" Scale="3"/>
    + <Property sap:label="Currency Code" Name="CurrencyCode" sap:quickinfo="Flight Reference Scenario: Currency Code" MaxLength="3" Type="Edm.String" sap:value-list="standard" sap:semantics="currency-code"/>
    + <Property sap:label="Description" Name="Description" sap:quickinfo="Flight Reference Scenario: Description" MaxLength="104" Type="Edm.String"/>
    + <Property sap:label="Status" Name="Status" sap:quickinfo="Flight Reference Scenario: Status" sap:display-format="UpperCase" MaxLength="1" Type="Edm.String"/>
    + <Property sap:label="Time Stamp" Name="LastChangedAt" sap:quickinfo="UTC Time Stamp in Long Form (YYYYMMDDhhmmssmmmmuuun)" Type="Edm.DateTimeOffset" Precision="7"/>
  + <NavigationProperty Name="to_Agency" ToRole="assoc_840EE5D2273403037381414EF708445D4" FromRole="FromRole_assoc_840EE5D2273403037381414EF708445D4"/>
  Relationship=><cds_xdmoxi_travel_u_sd>assoc_840EE5D2273403037381414EF708445D4</Relationship>
  + <NavigationProperty Name="to_Currency" ToRole="ToRole_assoc_77303B8674894889563756E7DD0F6870" FromRole="FromRole_assoc_77303B8674894889563756E7DD0F6870"/>
  Relationship=><cds_xdmoxi_travel_u_sd>assoc_77303B8674894889563756E7DD0F6870</Relationship>
  + <NavigationProperty Name="to_Customer" ToRole="ToRole_assoc_83A36B55BA535DD7C4776FC7F0E77DF4" FromRole="FromRole_assoc_83A36B55BA535DD7C4776FC7F0E77DF4"/>
  Relationship=><cds_xdmoxi_travel_u_sd>assoc_83A36B55BA535DD7C4776FC7F0E77DF4</Relationship>
  + <EntityType sap:content-version="1" sap:label="Country" Name="CountryType">
    + <Property sap:label="Name" Name="Name" sap:quickinfo="Country Name" sap:value-list="true"/>
  + <EntityType sap:content-version="1" sap:label="Currency" Name="CurrencyType">
    + <Property sap:label="Name" Name="Name" sap:quickinfo="Currency Name" sap:value-list="true"/>
  + <Association sap:content-version="1" Name="assoc_3ADAO6257630544A74085A3B04FDE16D">
    + <Association sap:content-version="1" Name="assoc_163F4B8468310f4915A6E1199A2E8B11">
    + <Association sap:content-version="1" Name="assoc_9B1A07E66D2A72CFAEC262689ED505B">
    + <Association sap:content-version="1" Name="assoc_840EE5D2273403037381414EF708445D4">
    + <Association sap:content-version="1" Name="assoc_3BCC647AC8A2E2FA80E9ABC453AB5">
    + <Association sap:content-version="1" Name="assoc_77303B8674894889563756E7DD0F6870">
    + <Association sap:content-version="1" Name="assoc_83A36B55BA535DD7C4776FC7F0E77DF4">
    + <Association sap:content-version="1" Name="assoc_788A0A23F3148555D/C81C4537928E">
  + <EntityType sap:content-version="1" sap:label="Travel" Name="TravelType" sap:isDefaultEntityContainer="true">
    + <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm" Target="cds_xdmoxi_travel_u_sd.cds_xdmoxi_travel_u_sd_Entities">
    + <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm" Target="cds_xdmoxi_travel_u_sd.cds_xdmoxi_travel_u_sd_Entities/Passenger">
    + <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm" Target="cds_xdmoxi_travel_u_sd.cds_xdmoxi_travel_u_sd_Entities/Travel">
    + <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm" Target="cds_xdmoxi_travel_u_sd.cds_xdmoxi_travel_u_sd_Entities/TravelAgency">
    + <Annotations href="https://sapui5.hana.ondemand.com/sap/opu/odata/DMO/TRAVEL_U_A2XX/$metadata" xmlns:atom="http://www.w3.org/2005/Atom" rel="self"/>
    + <Annotations href="https://sapui5.hana.ondemand.com/sap/opu/odata/DMO/TRAVEL_U_A2XX/$metadata" xmlns:atom="http://www.w3.org/2005/Atom" rel="latest-version"/>
  />
</Schema>

```

Root View

Metadata of the Remote Service

End Point: New OData Service with Additional Fields

We want to consume the remote service in the local system and extend it with additional fields for discounts. The end user will be able to maintain possible discounts for each trip in an SAP Fiori application. These discount fields are persisted on a local database table and retrieved together with the data of the remote service. The end user UI does not show any difference between the persisted fields and the fields that are retrieved remotely.

The discount fields are:

- *Discount Percentage*: The end user can maintain a proportional discount for the total price of travel.
- *Discount Absolute*: The end user can maintain an absolute value as discount for the total price of the travel

In our scenario, these fields are persisted in the database table */dmo/traveladd*, which consists of the two discount fields, the travel ID as the key, and administrative data to track changes.

In addition to the persistent fields, there is also a calculated field (*Total Price with Discount*) that displays the new total price including the possible discount. This field, however, is not persisted in the database, but is transient.

The following figure shows the end user UI with fields that are retrieved remotely from the provisioning system, persistent fields, and the transient field.

Travel ID:		Agency ID:		Customer ID:		Status:				
Trips (125)		Standard * ▾								
Travel ID	Agency ID	Customer ID	Begin Date	End Date	Total Price	Discount %	Discount Absolute	Total Price with Discount	Description	Status
1	70041	594	Aug 4, 2018	Jun 2, 2019	1,881.30 USD	0	0.00 USD	1,881.30 USD	Vacation 2	B >
3	70046	93	Aug 4, 2018	Jun 2, 2019	3,645.14 USD	0	0.00 USD	3,645.14 USD	Vacation	B >
4	70042	665	Aug 4, 2018	Jun 2, 2019	1,871.00 USD	0	0.00 USD	1,871.00 USD	Vacation	P >
5	70007	225	Aug 4, 2018	Aug 4, 2018	992.00 USD	0	0.00 USD	992.00 USD	Business Trip for Kurt, Ida	B >
6	70040	72	Aug 4, 2018	Jun 2, 2019	1,881.30 USD	0	0.00 USD	1,881.30 USD	Vacation	P >

Fiori Elements App for Managing Travel and Discounts

Procedure

To merge the locally and remotely retrieved data, the application developer must import the metadata of the remote service into the consuming system. Abstract entities that mirror the original data model of the remote service are generated by the service consumption wizard. It is then possible to access the service within the local ABAP code. To build a new OData service based on the remote service including additional database fields, a CDS custom entity is used to represent the data model, including both local and remote fields.

The program logic of a custom entity is not managed by ABAP runtime frameworks, but has to be implemented manually. The implementation includes the query itself and every query option as well as the transactional behavior for the local fields. The query is implemented in a query implementation class and the transactional runtime behavior is defined via a behavior definition and implemented in the related behavior pool. To access the data from the remote service, a destination to the provisioning system must be instantiated in the implementation classes.

To create the OData service, we expose the custom entity in a service definition and create a service binding to bind the service against a protocol. In our example case, this is a V2 UI service.

As a result, the existing remote read-only service and database fields with transactional capabilities are merged within the same SAP Fiori Elements app.

Related Information

[Developing Unmanaged Transactional Apps \[page 263\]](#)

[Developing a Web API \[page 365\]](#)

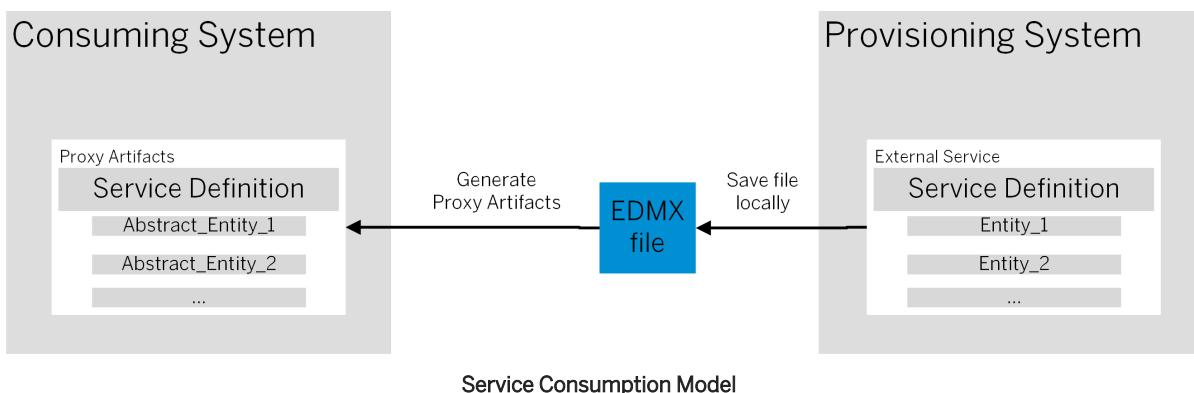
5.5.2 Preparing Access to the Remote OData Service

In order to consume the remote service in your local system, you need a local representation of the remote service.

Context

To be able to access the remote OData service, you need to generate proxy artifacts for it. These abstract entities mirror the data model of the remote service.

The wizard that creates the proxy artifacts requires a [CSDL XML \[page 810\]](#) document of the external service as a basis for the abstract entities.



5.5.2.1 Getting the OData Service Metadata

Context

A CSDL XML file can be retrieved from any service if the service URL is known. The following procedure describes the steps for saving a CSDL XML document on your local machine.

Procedure

1. Open your browser.
2. Call the external OData service metadata document.

Note

You get the metadata document by adding `/$metadata` to the service URL.

3. Save the metadata as an [XML Document](#) on your local machine.



Next Steps

You can now use this CSDL XML document in the service consumption wizard.

5.5.2.2 Generating Proxy Artifacts

Prerequisites

The metadata CSDL XML file of the service you want to consume is stored on your local machine.

Context

The proxy artifacts that represent the remote service in your system are required to access the remote service in your ABAP code. A wizard generates these proxy artifacts.

Procedure

1. In your ABAP package, open the context menu and choose to launch the creation wizard.
2. In addition to the **Project** and **Package**, enter a **Name** for the new service consumption model and a **Description**.

We use the name `/DMO/TRAVEL_C_A` in our example scenario. The suffix `_C` for consumption scenario and `_A` for abstract.

i Note

This name is also used for the service definition that will be generated.

3. Browse your local machine for the *Service Metadata File*.
4. Choose *Next*.
5. On the *Mapping Names* screen, deselect all service entity sets, except for *Travel* and edit the ABAP artifact name for this entity set to prevent name clashes with other artifacts.

	Service Entity Set	ABAP Artifact Name	ETag ...	Issue
<input type="checkbox"/>	Airline	ZAIRLINE366B6BD685	<input type="checkbox"/>	
<input type="checkbox"/>	Booking	ZBOOKING094C81E31B	<input type="checkbox"/>	
<input type="checkbox"/>	BookingSupplement	ZBOOKINGSUPPLEMENT	<input type="checkbox"/>	
<input type="checkbox"/>	Country	ZCOUNTRY6D82E6ADC3	<input type="checkbox"/>	
<input type="checkbox"/>	Currency	ZCURRENCY7CA15FDA43	<input type="checkbox"/>	
<input type="checkbox"/>	FlightConnection	ZFLIGHTCONNECTIONDA00C...	<input type="checkbox"/>	
<input type="checkbox"/>	Passenger	ZPASSENGER	<input type="checkbox"/>	
<input type="checkbox"/>	Supplement	ZSUPPLEMENT	<input type="checkbox"/>	
<input type="checkbox"/>	SupplementText	ZSUPPLEMENTTEXT	<input type="checkbox"/>	
<input checked="" type="checkbox"/>	Travel	/DMO/TRAVEL_C_A	<input checked="" type="checkbox"/>	
<input type="checkbox"/>	TravelAgency	ZTRAVELAGENCY	<input type="checkbox"/>	
<input type="checkbox"/>	Flight	ZFLIGHT601290F8A0	<input type="checkbox"/>	Type Edi...

Select All Deselect All

?

< Back Next > Finish Cancel

In this example scenario, we only use the root node of the business object (*Travel*).

6. Choose *Next*.

On the *ABAP Artifact Generation List* screen, in addition to the selected entity set, the service definition that is generated is listed.

7. Choose *Next* and assign a transport request.
8. Choose *Finish*.

The service consumption model editor is opened, in which you can see and verify the generated artifacts. The wizard creates an [abstract entity \[page 808\]](#) in your local system for each entity from the external service that you have chosen. If the source service contains creatable, updatable, or deletable entities, a behavior definition is created with the same name as the related abstract entity. These entities are found in the Core Data Services folder in ADT. Additionally, the generated [service definition](#) is created in the Business Services folder. It carries the same name as the service consumption model.

→ Remember

We only use the root node of the external service, which is now represented as [/DMO/TRAVEL_C_A](#).

You can access any of the development objects directly from this editor.

The service consumption model also generated code samples with placeholders for CRUD operations, which facilitate your entity set consumption.

Results

The following codeblock displays the source code of the abstract entity [/DMO/TRAVEL_C_A](#):

```
***** Generation Administration Data*****
@OData.entitySet.name: 'Travel'
@OData.entityType.name: 'TravelType'
define root abstract entity /DMO/TRAVEL_C_A
{
  key TravelID      : abap.numc( 8 );
  AgencyID        : abap.numc( 6 );
  AgencyID_Text   : abap.char( 80 );
  CustomerID     : abap.numc( 6 );
  CustomerID_Text : abap.char( 40 );
  BeginDate       : abap.dats;
  EndDate         : abap.dats;
  @Semantics.amount.currencyCode: 'CurrencyCode'
  BookingFee      : abap.dec( 17, 3 );
  @Semantics.amount.currencyCode: 'CurrencyCode'
  TotalPrice      : abap.dec( 17, 3 );
  @Semantics.currencyCode: true
  CurrencyCode    : abap.cuky( 5 );
  Memo            : abap.char( 1024 );
  Status          : abap.char( 1 );
  LastChangedAt   : tzntstmp;
  ETAG__ETAG      : abap.string( 0 );
}
```

i Note

Element types, the semantics annotations, and the OData entity set name, which is referenced in the annotation, are taken from the service metadata document of the remote service. If an ETAG is maintained in the original service, an element for the ETAG is added. The text element for the elements that have a text association in the original service are also added to the abstract entity. We will not use them in the service consumption scenario.

This abstract entity serves as a template for the data model of the new service.

5.5.3 Creating a Database Table for the Persistent Fields

Context

The remote service is enriched with discount fields (`discount_pct` and `discount_abs`). These enable the end user to maintain discounts for trips. The discounts can be either absolute or relative to the total price of the trip. These fields are used to calculate the new price (`TotalPriceWithDiscount`) in a transient field.

To create the database table for the discount fields, follow the steps.

Procedure

1. In the *Project Explorer*, select the relevant *Package* node, open the context menu, and choose  *New*  *Other ABAP Repository Object* .
2. Select *Database Table* to launch the creation wizard.
The *Creation wizard* opens.
3. Enter the necessary information for the wizard to create the database table `/dmo/traveladd`.
For a detailed description of how to use the wizard, see .
Once you have completed the wizard, the initially generated source code is displayed and ready for editing.
4. Define the fields for the database table.

```
@EndUserText.label : 'Travel Discount Information'  
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE  
@AbapCatalog.tableCategory : #TRANSPARENT  
@AbapCatalog.deliveryClass : #A  
@AbapCatalog.dataMaintenance : #LIMITED  
define table /dmo/traveladd {  
    key client : abap.cln1 not null;  
    key travel_id : abap.numc(8) not null;  
    discount_pct : abap.dec(3,1);  
    discount_abs : abap.dec(16,2);  
    lastchangedat : timestamp1;  
}
```

5. Save and activate.

5.5.4 Using a CDS Custom Entity for Data Modeling

Custom entities are used for data models whose runtime is implemented manually.

Context

This service consumption scenario retrieves its data partly from a remote service and partly from a local database table. In [custom entities \[page 809\]](#), you define the elements and type them to be used in the local

OData service. Custom entities do not come with a `select` on the data source. The implementation of the logic to retrieve the data is implemented in an ABAP class that is referenced in an entity annotation.

For more information, see [Unmanaged Query \[page 51\]](#).

5.5.4.1 Creating a CDS Custom Entity

Procedure

1. In your ABAP project, select the relevant package node in the *Project Explorer*. Open the context menu and choose     to launch the creation wizard.
2. Enter the necessary information for the wizard to create the CDS custom entity. Choose *Define Custom Entity with Parameters* as the template.

For a detailed description of how to use the wizard, see [.](#)

A new data definition for a CDS custom entity is created and added to the *Core Data Services* folder of the corresponding package node. In our scenario, the name of the custom entity is `/DMO/I_TRAVEL_C_C` (first suffix for consumption scenario, the second for custom entity).

3. Delete the `with parameters` statement. Parameters are not needed in the current scenario.

Results

Unlike CDS views, custom entities do not have a `select` statement to retrieve the data from a data source. The runtime of a custom entity must be implemented in a related ABAP class, which must be referenced using the annotation `@ObjectModel.query.implementedBy`.

Next Steps

The elements can now be defined in the editor. At least one element must be `key`. It is also necessary to determine a class for the query implementation in the annotation `@ObjectModel.query.implementedBy`.

5.5.4.2 Defining the Data Model in a CDS Custom Entity

Context

As described in [Scenario Description \[page 371\]](#), the data for the new OData service is partly retrieved from a foreign S/4HANA system and partly taken from a local database. As we have already generated the proxy artifacts, we can copy the data structure and the data types from the abstract entity [/DMO/TRAVEL_C_A](#). In addition, the additional discount elements are included, which have the same data types as declared in the database table [/dmo/traveladd](#).

Procedure

1. If you have not already done so, open the new data definition for the CDS custom entity in the editor.
2. Define the data model for the new travel service. You can copy the elements from the abstract entity [/DMO/TRAVEL_C_A](#).

You do not have to use the same names as in the abstract entity, but if you do use a different name, you have to map this manually in the query implementation. For example, the element `Memo` is renamed to `Description` in the custom entity.

3. Delete `ETAG_ETAG` element and the text elements `AgencyID_Text` and `CustomerID_Text`.

Note

You do not need the element `ETAG_ETAG` from the remote service. We use a calculated eTag in this scenario, which contains the timestamp from the local database table and the time timestamp from the remote service to ensure that no data has changed, neither on the local database table nor in the remote service.

4. Include the additional discount fields from [/dmo/traveladd](#).

```
DiscountPct      : abap.dec( 3, 1 );
DiscountAbs      : abap.dec( 16, 3 );
```

5. Add the element for the transient field that calculates the total price with discount.

```
TotalPriceWithDiscount: abap.dec(17,3)
```

6. Add an element for the calculated eTag.

```
CalculatedEtag    : abap.string( 0 );
```

7. Define the semantic relations between the amount and currency fields with the relevant annotations..

```
@Semantics.amount.currencyCode: 'CurrencyCode'
@Semantics.currencyCode: true
```

Results

The following codeblock displays the custom entity elements with the right types and the semantic annotations.

```
@EndUserText.label: 'CE for Service Consumption Scenario'
define custom entity /DMO/I_TRAVEL_C_C
{
    key TravelID          : abap.numc( 8 );
    AgencyID              : abap.numc( 6 );
    CustomerID            : abap.numc( 6 );
    BeginDate              : abap.dats;
    EndDate                : abap.dats;
    @Semantics.amount.currencyCode: 'CurrencyCode'
    BookingFee             : abap.dec( 17, 3 );
    @Semantics.amount.currencyCode: 'CurrencyCode'
    TotalPrice              : abap.dec( 17, 3 );
    @Semantics.currencyCode: true
    CurrencyCode           : abap.cuky( 5 );
    Description             : abap.char( 1024 ); //renamed element
    Status                  : abap.char( 1 );
    LastChangedAt           : tzntstmp;
    DiscountPct             : abap.dec( 3, 1 );
    @Semantics.amount.currencyCode: 'CurrencyCode'
    DiscountAbs              : abap.dec( 16, 3 );
    @Semantics.amount.currencyCode: 'CurrencyCode'
    TotalPriceWithDiscount : abap.dec(17,3);
    CalculatedEtag           : abap.string( 0 );
}
```

Next Steps

Before you can activate the CDS custom entity, you need to create an ABAP class that implements the interface `IF_RAP_QUERY_PROVIDER`. This class needs to be referenced in the entity annotation `@ObjectModel.query.implementedBy`.

5.5.4.3 Creating the Query Implementation Class

Context

The runtime of a CDS custom entity must be implemented manually. Therefore, it requires an ABAP class that implements the `select` method of the interface `IF_RAP_QUERY_PROVIDER` to handle query requests by the client. This class is referenced by the custom entity and is dedicated to implementing the OData client data requests.

Procedure

1. In the *Project Explorer*, select the relevant package node, open the context menu, and choose ► **New** ► **ABAP Class** to launch the creation wizard.
2. Follow the steps of the creation wizard and enter the necessary information. In the example scenario, we choose the name **/DMO/CL_TRAVEL_C_Q** for the query implementation class.

For a detailed description of how to use the wizard, see .

Once you have completed the wizard, the initially generated source code is displayed and ready for editing.

3. Implement the `select` method of the query provider interface `IF_RAP_QUERY_PROVIDER`.

The custom entity can only be activated once the `select` method of the interface `IF_RAP_QUERY_PROVIDER` is implemented in the query implementation class.

```
CLASS /dmo/cl_travel_c_q DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_rap_query_provider.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
CLASS /dmo/cl_travel_c_q IMPLEMENTATION.
  METHOD if_rap_query_provider~select.
  ENDMETHOD.
ENDCLASS.
```

4. In the custom entity `/DMO/I_TRAVEL_C_C`, add the entity annotation `@ObjectModel.query.implementedBy` and reference the newly created query implementation class `/DMO/CL_TRAVEL_C_Q`.

The reference to the ABAP class must start with `ABAP:`.

```
@ObjectModel.query.implementedBy: 'ABAP:/dmo/cl_travel_c_q'
define custom entity /DMO/I_TRAVEL_C_C
```

5. Activate the custom entity `/DMO/I_TRAVEL_C_C`.

5.5.5 Consuming the Remote OData Service

Address a remote service in your ABAP code and implement the query contract and the transactional behavior for the business object to build a new OData service.

Our service consumption scenario includes the creation of a new OData service in the consuming tenant. You have defined a data model for this service in a CDS custom entity, and now, the unmanaged query must be implemented in a related ABAP class as the query is not managed by a framework.

Additionally, this scenario also illustrates how transactional behavior can be included for the new OData service. To update the discount data, the business object (consisting only of the custom entity in this case) must be equipped with the implementation of the transactional runtime in a behavior pool. This includes the implementation of the interaction phase and save sequence.

Both, the query and the behavior implementation need to retrieve data from the remote service. With the help of a remote client proxy, the connection to the provisioning system is established and read requests are delegated to consume the remote service. The client proxy implementation is done in an auxiliary class for reuse reasons.

→ Remember

The consumption of a Web API requires the configuration of the involved systems (provisioning and consuming system) to allow the communication between them.

See [Prerequisites for the Consumption of a Remote Service \[page 371\]](#).

The following sections guide you through the implementation steps to instantiate a remote client proxy to reach the provisioning system, to implement the query for read access to the remote service and to implement transactional access to the discount persistence layer.

- [Creating a Remote Client Proxy \[page 383\]](#)
- [Implementing the Query for Service Consumption \[page 386\]](#)
- [Adding Transactional Behavior to the Business Object \[page 401\]](#)

5.5.5.1 Creating a Remote Client Proxy

To enable data retrieval from a remote service, you must establish a connection to the provisioning system and instantiate a client proxy that passes the OData requests to the remote service.

Prerequisites

To be able to address a remote service, make sure that you meet the [Prerequisites for the Consumption of a Remote Service \[page 371\]](#).

Context

A remote client proxy is used when an OData service is consumed remotely using an HTTP request. For reuse reasons, it is useful to instantiate the client proxy in a separate helper class.

Exception and Message Handling

In the service consumption scenario, several types of errors might occur. The execution of query or transactional requests is quite complex, as one request must pass not only the consuming service but also the provisioning service. In different parts of your coding, errors can happen. For example, while trying to connect to the provisioning system, while instantiating the client proxy, while executing the unmanaged query, during the execution of the application logic or during the modification of the discount values. Every kind of error must be handled with a suitable exception and message. Therefore, it is convenient to unite all exceptions and their related message in one exception class and one message class.

Before starting with the implementation of the consumption of the remote service, create an exception and a message class. For more information about exception and message handling, see [, , and](#).

The exception class must inherit from the superclass **CX_RAP_QUERY_PROVIDER**.

In the example scenario, the name of the exception class is **CX_TRAVEL_SERV_CONS**. The name of the message class is **CM_SERV_CONS**.

Implementation Steps

1. Create an Auxiliary Class for the Client Proxy Implementation

- In the *Project Explorer*, select the relevant package node, open the context menu, and choose **New > ABAP Class** to launch the creation wizard.
- Follow the steps of the creation wizard and enter the necessary information. In the example scenario, we choose the name **/DMO/CL_TRAVEL_C_CP_AUX**. Once you have completed the wizard, the initially generated source code is displayed and ready for editing.
For a detailed description of how to use the wizard, see [.](#)
- Declare a static method in the public section of the auxiliary class for the creation of a client proxy: **create_client_proxy** with a returning parameter **ro_client_proxy** and an exception to be raised if there are errors..

```
CLASS /dmo/cl_travel_cp_aux DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    CLASS-METHODS:
      get_client_proxy RETURNING VALUE(ro_client_proxy) TYPE REF TO /iwbeb/
if_cp_client_proxy
                                RAISING /dmo/cx_travel_serv_cons.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_travel_cp_aux IMPLEMENTATION.
  METHOD get_client_proxy.
  ENDMETHOD.
ENDCLASS.
```

2. Get the destination to the provisioning system and create an HTTP client

To be able to access the provisioning tenant, you must maintain a cloud destination for the foreign system and create an HTTP client. Use the code samples of the service consumption model for this.

- Call the method **create_by_http_destination** of the class **CL_WEB_HTTP_MANAGER** to create an HTTP client.
- Call the public static method **create_by_cloud_destination** of the class **CL_HTTP_DESTINATION_PROVIDER** to create a client proxy. Provide the name of the destination and the name of the service instance name. In our scenario, the destination is set up under the name **TRAVEL_BASIC** and the service instance name is **OutboundCommunication**.
The method has another input parameter, which relates to the authentication mode. If you do not fill this input parameter, the default authentication is BasicAuthentication with the user of the destination. This can be made explicit by using the parameter with the value **if_a4c_cp_service=>service_specific**.

- Include error handling. Use the exception class /DMO/CX_TRAVEL_SERV_CONS to raise adequate exception message, for example Access to remote system failed. .
For information about how to work with exception classes and message classes, see .

```

    " Getting the destination of foreign system
TRY.
    " Getting the destination of foreign system
    " Create http client
    " Details depend on your connection settings
    DATA(lo_http_client) =
cl_web_http_client_manager=>create_by_http_destination(
    cl_http_destination_provider=>create_by_cloud_destination(
        i_name = 'Travel_Basic'
        i_service_instance_name = 'OutboundCommunication' ) .
    " Error handling
    CATCH cx_http_dest_provider_error INTO DATA(lx_http_dest_provider_error).
    RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
        EXPORTING
            textid = /dmo/cx_travel_serv_cons=>remote_access_failed
            previous = lx_http_dest_provider_error.
    CATCH cx_web_http_client_error INTO DATA(lx_web_http_client_error).
    RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
        EXPORTING
            textid = /dmo/cx_travel_serv_cons=>remote_access_failed
            previous = lx_web_http_client_error.
ENDTRY..

```

3. Instantiate the Client Proxy

To consume the remote service from the provisioning system, we need a client proxy that delegates the requests to the remote service. Since our scenario works with OData V2, we need a V2 client proxy.

The service consumption model provides code templates for this.

- Create a client proxy by calling the method `create_v2_remote_proxy` of the class `CL_WEB_ODATA_CLIENT_FACTORY`. The client proxy is instantiated with the service definition generated by the service consumption model, the previously created HTTP client, and the URL path to the Web API.
- Use the exception class /DMO/CX_TRAVEL_SERV_CONS to raise adequate exception message, for example Client Proxy could not be instantiated.

For information about how to work with exception classes and message classes, see .

```

    " Instantiation of client proxy
TRY.
    ro_client_proxy = cl_web_odata_client_factory=>create_v2_remote_proxy(
        EXPORTING
            iv_service_definition_name = '/DMO/TRAVEL_C_A'
            io_http_client = lo_http_client
            iv_relative_service_root = '/sap/opu/odata/DMO/API_TRAVEL_U_V2' ) .
    CATCH cx_web_http_client_error INTO DATA(lx_web_http_client_error).
    RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
        EXPORTING
            textid = /dmo/cx_travel_serv_cons=>client_proxy_failed
            previous = lx_web_http_client_error.
    CATCH /iwbep/cx_gateway INTO DATA(lx_gateway).
    RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
        EXPORTING
            textid = /dmo/cx_travel_serv_cons=>client_proxy_failed
            previous = lx_gateway.
ENDTRY.

```

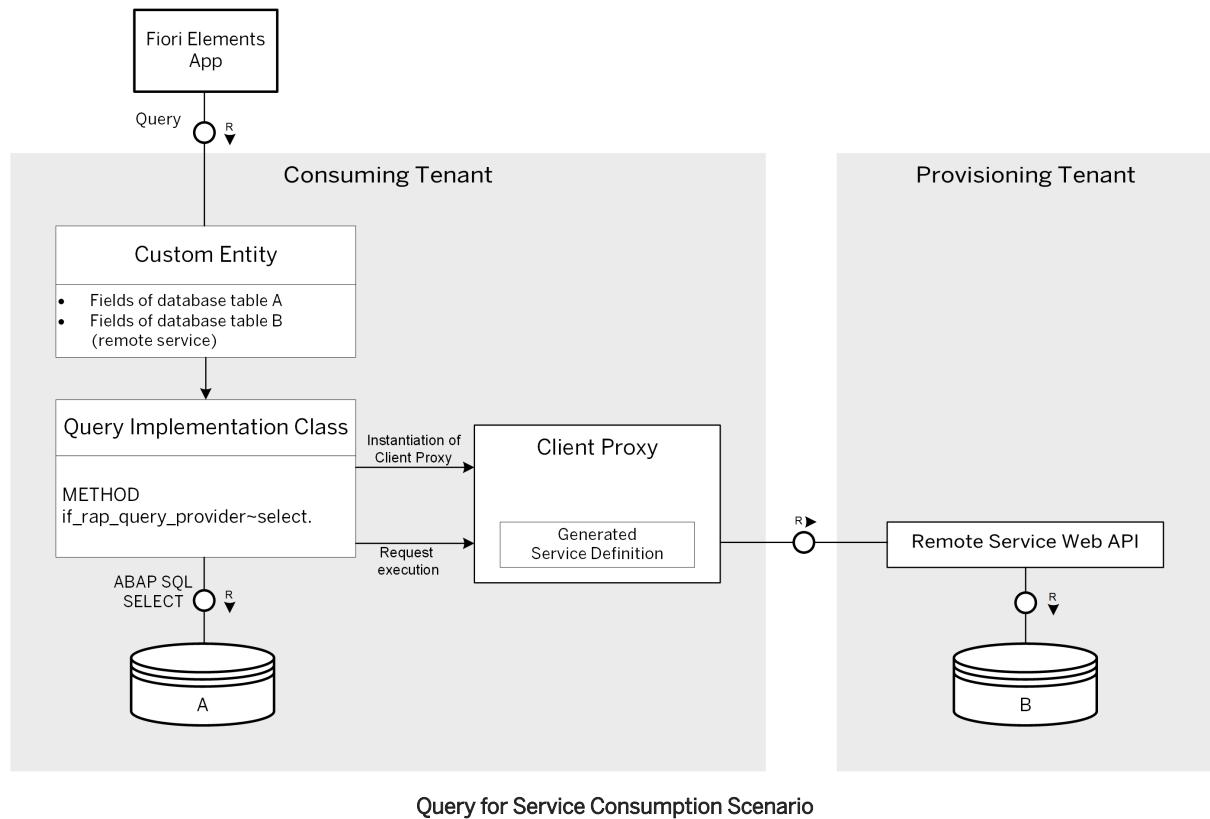
5.5.5.2 Implementing the Query for Service Consumption

The following topics guide you through the implementation steps required to execute a query request in a service consumption scenario.

Runtime Logic for the Query in the Service Consumption Scenario

Our scenario has two different data sources. One is a remote service from a possibly different system. To retrieve data from this service, you must have an appropriate connection to the provisioning tenant (see [Prerequisites for the Consumption of a Remote Service \[page 371\]](#)) and the `select` method in the query implementation class must instantiate a client proxy that executes requests for the remote service. Additionally, it must include the implementation of every query option you want to support. The other data source is a database table in the consuming system that stores the additional discount fields. To retrieve the data from this table, the `select` method includes an ABAP SQL SELECT.

The following figure illustrates how the query works in the service consumption scenario.



Apart from the simple data retrieval, the scenario also includes the query options [filtering](#), [selecting](#), [sorting](#), and [paging](#). All of these capabilities must be implemented in the query implementation class.

The client proxy is a powerful means to pass on OData requests to the provisioning tenant. It must be instantiated in the query implementation class. All the possible query options of the OData request in the consuming system must then be added to the request for the client proxy. This needs to be done for every

query option separately. After the actual execution of the request and the data retrieval, the data set from the remote service is extended with its matching discount data.

i Note

The query options can only be delegated to the client proxy if the query options only affect the elements from the remote service. If the query options are operated on local database fields, they must be implemented manually after the retrieval of the data set from the provisioning system. As this might cause serious performance issues, it is not demonstrated in this example scenario. Consider performance aspects for your use case, if you decide to implement this.

Structure of the Query

For every request that is sent to the remote service you need to instantiate the client proxy. The method to create a client proxy. With this client proxy, you create a read request (as you only want to execute a query on the remote service). Before sending this read request to the remote service, it must be equipped with all the necessary query options. For example, whenever the count is requested by the UI, the query option for count must be added to the read request. The interface `IF_RAP_QUERY_PROVIDER` provides methods to find out which query options are requested by the UI and hence must be delegated to the remote service.

i Note

When choosing the `GO` button on the UI to retrieve data, it always requests the count and paging. Therefore, the implementation for data retrieval must include at least counting and paging. Otherwise you will get a runtime error.

When all necessary query options are added to the request, it can be executed and the result must be given back to the UI.

The procedure for implementing the query implementation is described in the following in detail:

1. [Implementing Data and Count Retrieval \[page 388\]](#)
2. [Implementing Filtering \[page 394\]](#)
3. [Implementing Column Selections \[page 397\]](#)
4. [Implementing Sorting \[page 399\]](#)

Related Information

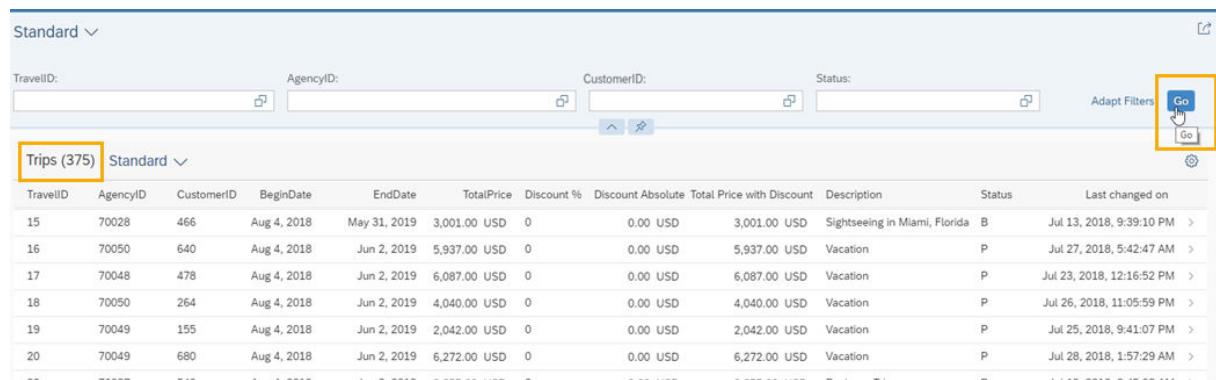
[Query Runtime Implementation \[page 50\]](#)

5.5.5.2.1 Implementing Data and Count Retrieval

To enable data retrieval from a remote service, you must establish a connection to the provisioning tenant and instantiate a client proxy that passes the OData requests to the remote service. The data sets from the remote service are then imported to the consuming tenant and can be merged with additional data.

Preview

Choosing the **Go** button on an SAP Fiori UI triggers the OData request for data and count retrieval. The UI automatically includes a paging query option with usually 25 data records per page.



Trips (375) Standard ▾											
TravelID	AgencyID	CustomerID	BeginDate	EndDate	TotalPrice	Discount %	Discount Absolute	Total Price with Discount	Description	Status	Last changed on
15	70028	466	Aug 4, 2018	May 31, 2019	3,001.00 USD	0	0.00 USD	3,001.00 USD	Sightseeing in Miami, Florida	B	Jul 13, 2018, 9:39:10 PM >
16	70050	640	Aug 4, 2018	Jun 2, 2019	5,937.00 USD	0	0.00 USD	5,937.00 USD	Vacation	P	Jul 27, 2018, 5:42:47 AM >
17	70048	478	Aug 4, 2018	Jun 2, 2019	6,087.00 USD	0	0.00 USD	6,087.00 USD	Vacation	P	Jul 23, 2018, 12:16:52 PM >
18	70050	264	Aug 4, 2018	Jun 2, 2019	4,040.00 USD	0	0.00 USD	4,040.00 USD	Vacation	P	Jul 26, 2018, 11:05:59 PM >
19	70049	155	Aug 4, 2018	Jun 2, 2019	2,042.00 USD	0	0.00 USD	2,042.00 USD	Vacation	P	Jul 25, 2018, 9:41:07 PM >
20	70049	680	Aug 4, 2018	Jun 2, 2019	6,272.00 USD	0	0.00 USD	6,272.00 USD	Vacation	P	Jul 28, 2018, 1:57:29 AM >

Implementation Steps

1. Instantiate the Client Proxy

To send requests to the provisioning system, you need a remote client proxy.

- In the query implementation class, call the method `get_client_proxy` of the auxiliary class `/DMO/CL_TRAVEL_CP_AUX` in the `select` method.

```
METHOD if_rap_query_provider~select.  
    """Instantiate Client Proxy  
    DATA(lo_client_proxy) = /dmo/cl_travel_cp_aux=>get_client_proxy( ).  
    ...  
ENDMETHOD.
```

2. Create a Read Request

To execute a query request on the remote service, a read request for the relevant entity set must be created. This read request will then be enhanced with query options.

- On the client proxy object, call the method `create_resource_for_entity_set` with the `TRAVEL` entity set name as importing parameter.

i Note

The OData entity set name can be found in the relevant abstract entity:

```

@OData.entitySet.name: 'Travel'
@OData.entityType.name: 'TravelType'
define root abstract entity /DMO/TRAVEL_C_A

{
  key TravelID      : abap.numc( 8 );
  AgencyID         : abap.numc( 6 );
}

```

In our scenario, the alias *Travel* is used.

i Note

Use uppercase for the OData entity set name of the entity for this method, since the internal ABAP representation of entity sets is in uppercase.

This method returns an object for the entity set resource.

- Create a read request by calling the method `create_request_for_read` on the entity set resource.
- Surround the method call with a `TRY-CATCH` block and include error handling with a suitable error message.

```

TRY.
  """Create Read Request
  DATA(lo_read_request) = lo_client_proxy-
>create_resource_for_entity_set( 'TRAVEL' )->create_request_for_read( ).
  CATCH /iwbep/cx_gateway INTO DATA(lx_gateway).
    RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
      EXPORTING
        textid   = /dmo/cx_travel_serv_cons=>query_failed
        previous = lx_gateway.
  ENDTRY.

```

3. Request the Inline Count of the Remote Service

As a Fiori UI always requests the inline count of the service, this request must be delegated to the remote service to get the total number of records from there.

- Within the try-catch Block, check whether the total number of records is requested by the UI by calling the method `is_total_numb_of_rec_requested` of the interface `IF_RAP_QUERY_REQUEST`.
For more information about `is_total_numb_of_rec_requested`, see [Method `is_total_numb_of_rec_requested` \[page 745\]](#).

i Note

The same exception as previously defined can be used for all methods called on the read request.

- Call the method `request_count` on the read request.

```

  """Request Count
  IF io_request->is_total_numb_of_rec_requested( ).
    lo_read_request->request_count( ).
  ENDIF.

```

4. Implement Paging if Data is Requested

Whenever the UI requests paging, the query framework checks that the query does not return more data records than requested. Therefore, you need to set the paging information for the read request the client proxy sends to the remote service.

- Within the try-catch block, check whether data is requested.
- Get the paging information of the UI request into a local variable (`ls_paging`) by calling the method `get_paging` of the interface `IF_RAP_QUERY_REQUEST`.
For more information about `get_paging`, see [Method `get_paging` \[page 746\]](#).
- If the offset is ≥ 0 , pass it to the read-request.
- Pass the page size to the read request if it is not unlimited.

i Note

If all data records are requested, the method returns the constant `page_size_unlimited`, which has the value `-1`. If this constant is returned, the query option `$skip` must not be passed to the read request.

```
"""Request Paging
DATA(ls_paging) = io_request->get_paging( ).
IF ls_paging->get_offset( ) >= 0.
    lo_read_request->set_skip( ls_paging->get_offset( ) ).
ENDIF.
IF ls_paging->get_page_size( ) <>
if_rap_query_paging->page_size_unlimited.
    lo_read_request->set_top( ls_paging->get_page_size( ) ).
ENDIF.
```

5. Execute the Request

The read request for the remote OData service is now equipped with the query options for count `$inlinecount=allpages` (if total number of records is requested) and for paging `$skip` and `$top` (if data is requested). At least, these two query options should be implemented in the query implementation class if the consuming service is a UI service.

→ Remember

By default, the UI sends the query options for counting and paging.

Now, the request can be sent. In other words, it is executed on the remote service.

- Execute the request and write it into a local variable `lo_response`.

```
"""Execute the Request
DATA(lo_response) = lo_read_request->execute( ).
```

6. Provide the Count Result for the Response of the Unmanaged Query

The response parameter `io_response` of `IF_RAP_QUERY_PROVIDER~select` must be filled with the total number of records that the count request to the remote service returns.

- Check if the total number of records is requested.
- Get the count from the response of the remote service by calling the method `get_count` on the remote service response object.
- Set the returned number for the response to the unmanaged query by calling the method `set_total_number_of_records` on the response object of the unmanaged query.

```
"""Set Count
IF io_request->is_total numb of rec requested( ).
    io_response->set_total number of records( lo_response->get_count( ) ).
ENDIF.
```

7. Provide the Result Data for the Response of the Unmanaged Query

The response parameter `io_response` of `IF_RAP_QUERY_PROVIDER~select` must be filled with the requested data, which are returned by the remote service and enhanced with the local discount data.

- Check whether data is requested.
- Declare `lt_travel` with the same type as the generated abstract entity. The data from the remote service will be written into that table.
- Declare `lt_travel_ce` with the type of the custom entity. This table is used to fill the output parameter of the select method.
- Declare `lt_traveladd` with the type of the local database table. The data from the database table `/dmo/traveladd` is read into this table.
- Get the data from the response object of remote service request by calling the method `get_business_data` and write it into `lt_travel`.
- If `lt_travel` returns entries, provide a mapping for those elements in the custom entity that are not identical with the names of the abstract entity elements. In our scenario, this concerns the element `Description`.
- Select the local data (discounts) based on the entity sets that are retrieved from the remote service.
- Select the latest time stamp for the discount data.
- Calculate the eTag from the values of the `lastChangedAt` fields of the remote and local service.

i Note

The eTag is used to ensure that discount data has not changed. That is why, the retrieved value is overwritten with the initial timestamp for the field.

- Calculate the amount for the field `TotalPriceWithDiscount` if a discount exists.
- If no discount is maintained, fill the field `TotalPriceWithDiscount` with the value of `TotalPrice` without discount and set an initial value for the field `lastchangedat`.
- Set the returned data for the response to the unmanaged query by calling the method `set_data` on the response object of the unmanaged query.

```
"""/Set Data
IF io_request->is_data_requested( ).
  DATA: lt_travel      TYPE STANDARD TABLE OF /dmo/travel_c_a,
        lt_travel_ce   TYPE STANDARD TABLE OF /dmo/i_travel_c_c,
        lt_traveladd    TYPE STANDARD TABLE OF /dmo/traveladd.
  lo_response->get_business_data( IMPORTING et_business_data =
lt_travel ).

  IF lt_travel IS NOT INITIAL.
    lt_travel_ce = CORRESPONDING #( lt_travel MAPPING description =
memo ).

    SELECT * FROM /dmo/traveladd FOR ALL ENTRIES IN @lt_travel_ce WHERE
travel_id = @lt_travel_ce-travelid INTO TABLE @lt_traveladd.
    LOOP AT lt_travel_ce ASSIGNING FIELD-SYMBOL(<fs_travel_ce>).
      IF line_exists(`lt_traveladd[ travel_id = <fs_travel_ce>-
travelid ]`).
        <fs_travel_ce>-discountpct          = lt_traveladd[ travel_id =
<fs_travel_ce>-travelid ]-discount_pct.
        <fs_travel_ce>-discountabs         = lt_traveladd[ travel_id =
<fs_travel_ce>-travelid ]-discount_abs.
        <fs_travel_ce>-totalpricewithdiscount = <fs_travel_ce>-
totalprice * ( 1 - <fs_travel_ce>-discountpct / 100 ) - <fs_travel_ce>-
discountabs.
        <fs_travel_ce>-lastchangedat       = lt_traveladd[ travel_id =
<fs_travel_ce>-travelid ]-lastchangedat.
      ENDIF.
    ENDLOOP.
  ENDIF.
ENDIF.
```

```

        <fs_travel_ce>-calculatedetag      = <fs_travel_ce>-
calculatedetag && '-' && lt_traveladd[ travel_id = <fs_travel_ce>-travelid ]-
lastchangedat.
        ELSE.
            <fs_travel_ce>-totalpricewithdiscount = <fs_travel_ce>-
totalprice.
            <fs_travel_ce>-lastchangedat          = '20000101120000' .
"initial value Jan 1, 2000, 12:00:00 AM
        ENDIF.
    ENDLOOP.
ENDIF.
io_response->set_data( lt_travel_ce ).
ENDIF.

```

Results

i Expand the following listing to view the source code of the query implementation for data retrieval and count:

Query Implementation Class

```

CLASS /dmo/cl_travel_c_q DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_rap_query_provider.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_travel_c_q IMPLEMENTATION.
  METHOD if_rap_query_provider~select.
    """Instantiate Client Proxy
    DATA(lo_client_proxy) = /dmo/cl_travel_cp_aux=>get_client_proxy( ).
    TRY.
      """Create Read Request
      DATA(lo_read_request) = lo_client_proxy->create_resource_for_entity_set( 'TRAVEL' )->create_request_for_read( ).
      """Request Count
      IF io_request->is_total numb_of_rec_requested( ).
        lo_read_request->request_count( ).
      ENDIF.
      """Request Data
      IF io_request->is_data_requested( ).
        """Request Paging
        DATA(ls.paging) = io_request->get_paging( ).
        IF ls.paging->get_offset( ) >= 0.
          lo_read_request->set_skip( ls.paging->get_offset( ) ).
        ENDIF.
        IF ls.paging->get_page_size( ) <>
if_rap_query_paging=>page_size_unlimited.
          lo_read_request->set_top( ls.paging->get_page_size( ) ).
        ENDIF.
      """Execute the Request
      DATA(lo_response) = lo_read_request->execute( ).
      """Set Count
      IF io_request->is_total numb_of_rec_requested( ).
        io_response->set_total_number_of_records( lo_response->get_count( ) ).
      ENDIF.
      """Set Data
      IF io_request->is_data_requested( ).
        DATA: lt_travel TYPE STANDARD TABLE OF /dmo/travel_c_a,
              lt_travel_ce TYPE STANDARD TABLE OF /dmo/i_travel_c_c,

```

```

        lt_traveladd TYPE STANDARD TABLE OF /dmo/traveladd.
        lo_response->get_business_data( IMPORTING et_business_data =
lt_travel )..
        IF lt_travel IS NOT INITIAL.
            lt_travel_ce = CORRESPONDING #( lt_travel MAPPING description =
memo )..
            SELECT * FROM /dmo/traveladd FOR ALL ENTRIES IN @lt_travel_ce WHERE
travel_id = @lt_travel_ce-travelid INTO TABLE @lt_traveladd.
            LOOP AT lt_travel_ce ASSIGNING FIELD-SYMBOL(<fs_travel_ce>).
                IF line_exists('lt_traveladd[ travel_id = <fs_travel_ce>-
travelid ]').
                    <fs_travel_ce>-discountpct           = lt_traveladd[ travel_id =
<fs_travel_ce>-travelid ]-discount_pct.
                    <fs_travel_ce>-discountabs          = lt_traveladd[ travel_id =
<fs_travel_ce>-travelid ]-discount_abs.
                    <fs_travel_ce>-totalpricewithdiscount = <fs_travel_ce>-
totalprice * ( 1 - <fs_travel_ce>-discountpct / 100 ) - <fs_travel_ce>-
discountabs.
                    <fs_travel_ce>-lastchangedat       = lt_traveladd[ travel_id =
<fs_travel_ce>-travelid ]-lastchangedat.
                ELSE.
                    <fs_travel_ce>-totalpricewithdiscount = <fs_travel_ce>-
totalprice.
                    <fs_travel_ce>-lastchangedat       = '20000101120000' .
                ENDIF.
            ENDLOOP.
        ENDIF.
        io_response->set_data( lt_travel_ce )..
    ENDIF.
    CATCH /iwbep/cx_gateway INTO DATA(lx_gateway).
        RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
        EXPORTING
            textid   = /dmo/cx_travel_serv_cons->query_failed
            previous = lx_gateway.
    ENDTRY.
ENDMETHOD.
ENDCLASS.
```

If you have defined a UI service for the custom entity (see [Defining an OData Service \[page 417\]](#)), you can test your implementation with the SAP Fiori Elements Preview. When you open the app and choose the [Go](#) button, you receive the data and the count from the remote service together with the matching discount data.

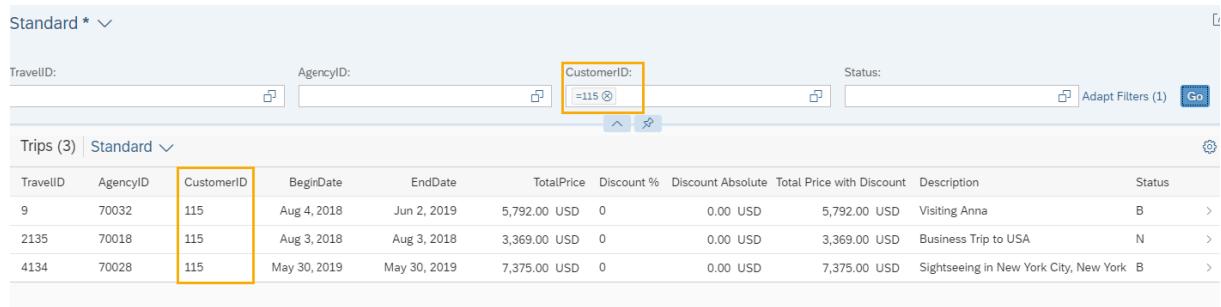
Related Information

- [Implementing an Unmanaged Query \[page 494\]](#)
- [Implementing Paging in an Unmanaged Query \[page 507\]](#)
- [Interface IF_RAP_QUERY_REQUEST \[page 744\]](#)

5.5.5.2.2 Implementing Filtering

Filter options can be delegated to the remote service. Fill the request that is passed to the OData client proxy with the query option `$filter`.

Preview



The screenshot shows a Fiori application interface. At the top, there is a search bar with fields for TravelID, AgencyID, CustomerID, and Status. The CustomerID field contains the value '=115' and has a delete icon next to it. A yellow box highlights this input field. Below the search bar is a table titled 'Trips (3) | Standard'. The table has columns: TravelID, AgencyID, CustomerID, BeginDate, EndDate, TotalPrice, Discount %, Discount Absolute, Total Price with Discount, Description, and Status. Three rows of data are listed:

TravelID	AgencyID	CustomerID	BeginDate	EndDate	TotalPrice	Discount %	Discount Absolute	Total Price with Discount	Description	Status
9	70032	115	Aug 4, 2018	Jun 2, 2019	5,792.00 USD	0	0.00	5,792.00 USD	Visiting Anna	B >
2135	70018	115	Aug 3, 2018	Aug 3, 2018	3,369.00 USD	0	0.00	3,369.00 USD	Business Trip to USA	N >
4134	70028	115	May 30, 2019	May 30, 2019	7,375.00 USD	0	0.00	7,375.00 USD	Sightseeing in New York City, New York	B >

To retrieve the appropriate entries for the filter, you must implement a filter factory to get the filtered results from the remote service.

Implementation Steps

This scenario only implements filtering by fields of the remote service. The service proxy API provides a filter factory to delegate the filter to the remote service. Filtering by local fields is not supported in this scenario.

Note

To filter by local fields, you would have to retrieve all available data sets of the remote service and merge the local fields to these first, to be able to filter by discount values then. In this case, paging could not be delegated to the remote service either and would have to be implemented after data retrieval manually. Hence filtering for discount values might cause serious performance issues and is therefore not exemplified in this scenario.

Note

Filtering on an amount field only works if the end user also provides a filter on the unit field, in our case a currency code. In our scenario, the fields `TotalPrice` and `BookingFee` are amount fields. If a filter is entered for these fields, the currency code must be in the filter condition as well. Hence, the implementation must also include the currency every time a filter for the field `TotalPrice` is supplied.

The filter information is relevant for data and count retrieval. Therefore the implementation for the filter must be independent of the checks for data or count requests. The filter information must be added to the read request for the remote service. So the filter implementation must be called before executing the read request.

- The unmanaged query API provides a method to get the filter conditions from the UI OData request into a local variable. Use `io_request->get_filter()->get_as_ranges()` to get the filter conditions as ranges into a local variable `lt_filter`. Handle the exception if the filter cannot be converted into a ranges table.
For more information about the unmanaged query filter API, see [Interface IF_RAP_QUERY_FILTER \[page 749\]](#).
- Loop over `lt_filter`.

→ Remember

Some filter requests require special handling before sending them to the remote service:

- Filtering on local fields is not allowed.
- For filtering on an amount field, the unit must be provided in the filter.

- Raise an exception with an adequate message, for example `Filtering on &1 is not allowed` if filter is requested for a local field.
For information about how to work with exception classes and message classes, see [and](#).
- Find out the currency code for filter requests on amount fields.
- Raise an exception if currency code is not provided, for example `Currency code is not supplied for &1`.
- Create the filter property to be passed to the remote service. Map the element `Description` to `Memo` for the remote service to match the data model of the remote service.
- Create a filter factory to provide the filter request for the client proxy.
- Create the filter ranges for the remote service by calling the method `create_by_range` of the filter factory and export the filter property, the filter range and, if necessary, the currency code.
- If a filter is defined for more than one element, concatenate the filter condition.
- Set the filter for the request.

```
"""
Request Filtering
TRY.
  DATA(lt_filter) = io_request->get_filter( )->get_as_ranges( ).
  CATCH cx_rap_query_filter_no_range INTO DATA(lx_no_range).
    RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
      EXPORTING
        textid   = /dmo/cx_travel_serv_cons=>no_ranges
        previous = lx_no_range.
```

```

ENDTRY.
LOOP AT lt_filter ASSIGNING FIELD-SYMBOL(<fs_filter>).
  IF <fs_filter>-name = 'DISCOUNTPCT' OR
    <fs_filter>-name = 'DISCOUNTABS' OR
    <fs_filter>-name = 'TOTALPRICEWITHDISCOUNT' OR
    <fs_filter>-name = 'CALCULATEDETAG'.
    RAISE EXCEPTION TYPE /dmo/cx_travel_query
      EXPORTING
        textid = /dmo/cx_travel_query=>filtering_failed
        element = <fs_filter>-name.
  ENDIF.
  "provide currency code, if filtering on amount field
  IF <fs_filter>-name = 'TOTALPRICE' OR
    <fs_filter>-name = 'BOOKINGFEE'.
    IF line_exists( lt_filter[ name = 'CURRENCYCODE' ] ).
      DATA(lv_currencycode) = VALUE waers_curc( lt_filter[ name =
'CURRENCYCODE' ]-range[ option = 'EQ' ]-low OPTIONAL ).
    ELSE.
      RAISE EXCEPTION TYPE /dmo/cx_travel_serv_cons
        EXPORTING
          textid = /dmo/cx_travel_serv_cons=>no_currencycode
          element = <fs_filter>-name.
   ENDIF.
  ENDIF.
  "map element names
  DATA(lv_filter_property) = COND /iwbep/
if_cp_runtime_types=>ty_property_path( WHEN <fs_filter>-name ='DESCRIPTION'
THEN 'MEMO'

ELSE <fs_filter>-name .
  "create filter factory for read request
  DATA(lo_filter_factory) = lo_read_request->create_filter_factory( ).
"
  DATA(lo_filter_for_current_field) = lo_filter_factory-
>create_by_range( iv_property_path = lv_filter_property

it_range = <fs_filter>-range

iv_currency_code = lv_currencycode .
  "Concatenate filter if more than one filter element
  DATA: lo_filter           TYPE REF TO /iwbep/if_cp_filter_node.
  IF lo_filter IS INITIAL.
    lo_filter = lo_filter_for_current_field.
  ELSE.
    lo_filter = lo_filter->and( lo_filter_for_current_field ).
 ENDIF.
ENDLOOP.
"set filter
IF lo_filter IS NOT INITIAL.
  lo_read_request->set_filter( lo_filter ).
ENDIF.

```

Results

If you have defined a UI service for the custom entity (see [Defining an OData Service \[page 417\]](#)), you can test your implementation with the SAP Fiori Elements Preview. You can define filters and check that the UI only retrieves data that matches the filter request.

Related Information

[Implementing Filtering in an Unmanaged Query \[page 501\]](#)

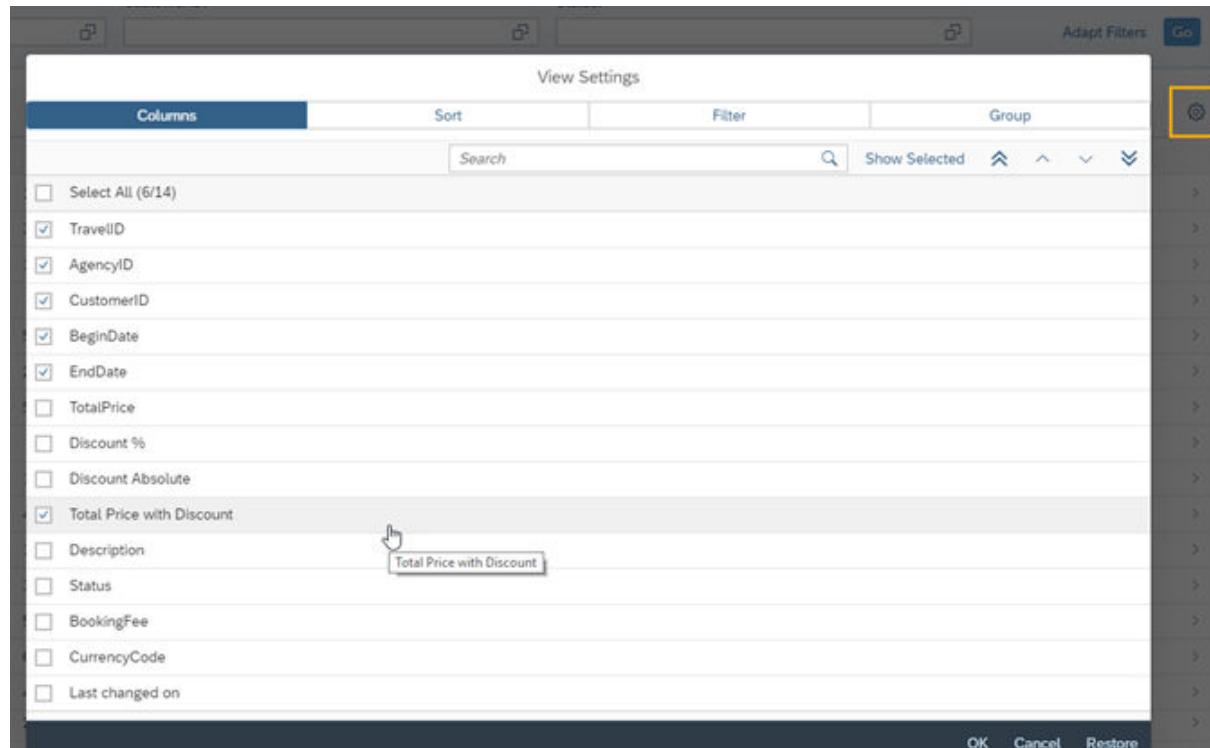
[Interface IF_RAP_QUERY_FILTER \[page 749\]](#)

5.5.5.2.3 Implementing Column Selections

Select options can be delegated to the remote service. Fill the request that is passed to the OData client proxy with the query option \$select.

Preview

The SAP Fiori UI allows you to select columns.



To retrieve the appropriate data, you must transfer the selected properties to the read request for the remote service.

Implementation Steps

You can optimize the performance if you only select those elements from the remote service that are selected on the UI. The local fields are selected separately. Therefore, you must exclude the local fields from the requested elements as they are not delegated to the remote service.

Considering the requested elements from the UI when delegating the request to the remote service is only necessary when data is requested. Therefore the implementation must only be called if data is requested before the execution of the read request.

- The unmanaged query API provides a method to get those elements that are requested by the UI. Use `DATA(lt_req_elements) = io_request->get_requested_elements().` to get the elements into a local variable.
For more information about the unmanaged query API, see [Interface IF_RAP_QUERY_REQUEST \[page 744\]](#).
- Delete the local elements from the properties that are selected.

→ Remember

Local fields are not delegated to the remote service. They are selected manually.

- Loop over `lt_req_elements` and map the element `Description` to `Memo` for the remote service to match the data model of the remote service.
- Declare `lt_select_properties` and type it as a table of property paths. This table contains the properties that are given to the client proxy for creating the select query option.
- Fill the table `lt_select_properties` that is given to the client proxy with the elements to be selected from the remote service.
- Set the select properties for the OData request of the remote service.

```
"""
Request Elements
DATA(lt_req_elements) = io_request->get_requested_elements( ).
"delete local fields out of the fields to be selected via OData Client
Proxy
    DELETE lt_req_elements WHERE table_line = 'DISCOUNTPCT' OR
                                table_line = 'DISCOUNTABS' OR
                                table_line = 'TOTALPRICEWITHDISCOUNT' OR
                                table_line = 'CALCULATEDETAG'.
    "map differing names
    LOOP AT lt_req_elements ASSIGNING FIELD-SYMBOL(<fs_req_elements>).
        DATA(lv_select_property) = COND /iwbeb/
        if_cp_runtime_types=>ty_property_path( WHEN <fs_req_elements> ='DESCRIPTION'
        THEN 'MEMO'

        ELSE <fs_req_elements> .
        DATA: lt_select_properties TYPE /iwbeb/
if_cp_runtime_types=>ty_t_property_path.
        APPEND lv_select_property TO lt_select_properties.
    ENDLOOP.
    "set select properties
    IF lt_select_properties IS NOT INITIAL.
        lo_read_request->set_select_properties( lt_select_properties ) .
    ENDIF.
```

Results

If you have defined a UI service for the custom entity (see [Defining an OData Service \[page 417\]](#)), you can test your implementation with the SAP Fiori Elements Preview. Trigger the selection by choosing columns on the SAP Fiori UI and check whether the request works.

Related Information

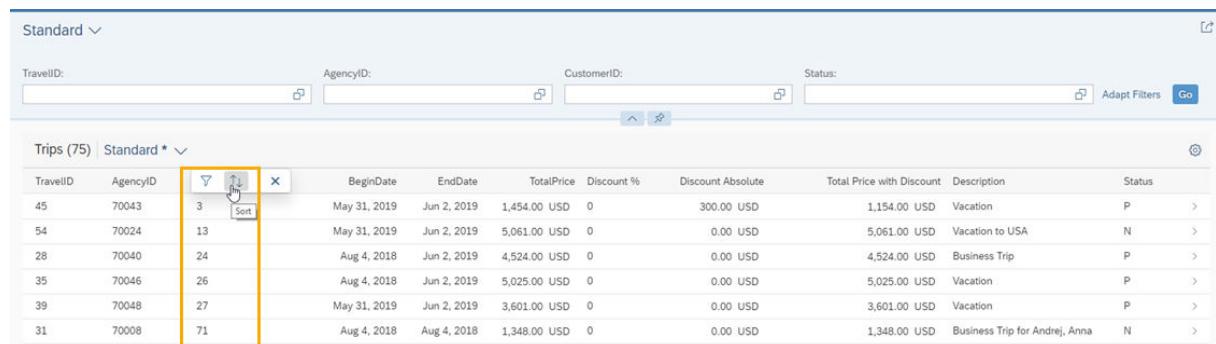
[Interface IF_RAP_QUERY_REQUEST \[page 744\]](#)

[Considering Requested Elements in an Unmanaged Query \[page 510\]](#)

5.5.5.2.4 Implementing Sorting

Sorting options can be delegated to the remote service. Fill the request that is passed to the OData client proxy with the relevant query option `$orderby`.

Preview



TravellID	AgencyID	BeginDate	EndDate	TotalPrice	Discount %	Discount Absolute	Total Price with Discount	Description	Status
45	70043	May 31, 2019	Jun 2, 2019	1,454.00 USD	0	300.00 USD	1,154.00 USD	Vacation	P >
54	70024	May 31, 2019	Jun 2, 2019	5,061.00 USD	0	0.00 USD	5,061.00 USD	Vacation to USA	N >
28	70040	Aug 4, 2018	Jun 2, 2019	4,524.00 USD	0	0.00 USD	4,524.00 USD	Business Trip	P >
35	70046	Aug 4, 2018	Jun 2, 2019	5,025.00 USD	0	0.00 USD	5,025.00 USD	Vacation	P >
39	70048	May 31, 2019	Jun 2, 2019	3,601.00 USD	0	0.00 USD	3,601.00 USD	Vacation	P >
31	70008	Aug 4, 2018	Aug 4, 2018	1,348.00 USD	0	0.00 USD	1,348.00 USD	Business Trip for Andrej, Anna	N >

To retrieve the data in the appropriate order, you must add sorting information to the request for the OData client proxy.

Implementation Steps

This scenario only implements sorting by fields of the remote service. The service proxy API provides the method `set_order_by` to delegate the sorting order to the remote service.

i Note

To sort by local fields, you would have to retrieve all available data sets of the remote service and merge the local fields to these first, to be able to sort by discount values then. In this case, paging could not be

delegated to the remote service either and would have to be implemented after data retrieval manually. Hence, sorting by discount values might cause serious performance issues and is therefore not exemplified in this scenario.

The sorting information is only relevant if you retrieve data. Therefore the implementation must only be called if data is requested to equip the read request for the remote service with this information.

- The unmanaged query API provides a method to get the sorting information. Use `DATA(lt_sort) = io_request->get_sort_elements()` to get the sorting information into a local variable. For more information about the unmanaged query API, see [Interface IF_RAP_QUERY_REQUEST \[page 744\]](#).
- Loop over `lt_sort`.

→ Remember

Sorting on local elements is not supported due to performance reasons.

- Raise an exception with an adequate message, for example `Sorting on &1 is not allowed.` For information about how to work with exception classes and message classes, see [and](#).
- Declare `lt_sort_properties`. This table is passed to the client proxy with the elements to be sorted.
- Map the element `Description` to `Memo` for the remote service to match the data model of the remote service.
- Fill the table `lt_sort_properties` that is given to the client proxy with the elements to be selected from the remote service and map the columns.
- Set the sorting properties for the request by using the `set orderby`.

```
"""Request Sorting
DATA(lt_sort) = io_request->get_sort_elements().
LOOP AT lt_sort ASSIGNING FIELD-SYMBOL(<fs_sort>).
  IF <fs_sort>-element_name = 'DISCOUNTPCT' OR
    <fs_sort>-element_name = 'DISCOUNTABS' OR
    <fs_sort>-element_name = 'TOTALPRICEWITHDISCOUNT' OR
    <fs_sort>-element_name = 'CALCULATEDETAG'.
    RAISE EXCEPTION TYPE /dmo/cx_travel_query
      EXPORTING
        textid = /dmo/cx_travel_query->sorting_failed
        element = <fs_sort>-element_name.
  ENDIF.
  "map differing names
  DATA: lt_sort_properties TYPE /iwbepl/
if_cp_runtime_types=>ty_t_sort_order.
  APPEND VALUE #(
    property_path = COND #( WHEN <fs_sort>-element_name
= 'DESCRIPTION' THEN 'MEMO'
                                ELSE <fs_sort>-element_name )
    descending = <fs_sort>-descending )
    TO lt_sort_properties.
ENDLOOP.
"set sorting properties
IF lt_sort_properties IS NOT INITIAL.
  lo_read_request->set orderby( lt_sort_properties ).
ENDIF.
ENDIF.
```

Results

If you have defined a UI service for the custom entity (see [Defining an OData Service \[page 417\]](#)), you can test your implementation with the SAP Fiori Elements Preview. Try out the sorting on different elements and check whether the results are sorted or an exception is thrown.

Related Information

[Interface IF_RAP_QUERY_REQUEST \[page 744\]](#)

[Implementing Sorting in an Unmanaged Query \[page 509\]](#)

5.5.5.3 Adding Transactional Behavior to the Business Object

The following steps guide you through the implementation steps to update a business object when a remote service is merged with local data.

Like in [Developing Unmanaged Transactional Apps \[page 263\]](#), creating, updating, and deleting data requires a transactional runtime implementation. The service consumption scenario is also a use case for the unmanaged implementation type as the transactional logic is defined and implemented by the application developer. That means, the transactional behavior must be defined in a [behavior definition \[page 805\]](#) and has to be implemented in ABAP classes of a [behavior pool \[page 805\]](#).

Note

This scenario only supports transactional activities on the data that is persisted on a database table in the consuming tenant. Transactional operations on the remote service are not supported.

In this example scenario, only the discount data can be manipulated by the end user. The fields on the database table `/dmo/traveladd` that can be changed are `discount_pct` and `discount_abs`. The travel instance for which the discount is maintained is identified by the travel ID, which is retrieved from the remote service. Travel ID is a key field and therefore cannot be changed by the end user. For the eTag check, a distinct field `CalculatedeTag` is used. Its value is concatenated with the timestamp of the remote service and the timestamp from the local database table.

The following tasks are relevant to include transactional behavior for the business object:

- [Defining a Behavior for the Business Object \[page 402\]](#)
- [Implementing the Behavior Pool for the Business Object \[page 404\]](#)

5.5.5.3.1 Defining a Behavior for the Business Object

The transactional behavior of the travel business object must be defined in a behavior definition.

To get transactional access to the local data, the data model must be turned into a business object. The custom entity `/DMO/I_TRAVEL_C_C` must be defined as root entity.

1. Open the custom entity `/DMO/I_TRAVEL_C_C` and insert the keyword root. A behavior definition can only be created for a root entity.

```
define root custom entity /DMO/I_TRAVEL_C_C
```

2. Activate the root custom entity.

Creating a Behavior Definition /DMO/I_TRAVEL_C_C

By creating a [behavior definition \[page 806\]](#), the referenced root entity gains a transactional character. In the behavior definition, you define the transactional capabilities of your business object. It is directly related to the root custom entity and must therefore have the same name `/DMO/I_TRAVEL_C_C`.

1. In the [Project Explorer](#), select the relevant node for the data definition that contains the CDS root entity `/DMO/I_Travel_C_C`, for which you want to create a behavior definition.
2. Open the context menu and select [New Behavior Definition](#) to launch the creation wizard.
3. Follow the steps of the creation wizard to create the behavior definition

Note

The implementation type of a custom entity can only be `unmanaged`. That is why, the wizard only allows this option.

Result

The created behavior definition object represents the root node of a new business object in ABAP RESTful programming model.

In the Project Explorer, the new behavior definition is added to the Core Data Services folder.

[For more information, see Creating Behavior Definitions \[page 761\].](#)

Defining the Behavior for the Travel Business Object with Discount Data

For the service consumption scenario, only the implementation type [unmanaged \[page 291\]](#) is supported. That means, you as an application developer must implement the essential components of the business object yourself.

Remember

Our scenario is limited to modifying the data that is stored on the database table `/dmo/traveladd`.

Looking at the data model of the custom entity and its data sources, that means that only the discount data can be modified.

```
{
  key TravelID      : abap.numc( 8 );
  AgencyID       : abap.numc( 6 );
  AgencyID_Text  : abap.char( 80 );
  CustomerID     : abap.numc( 6 );
  CustomerID_Text: abap.char( 40 );
  BeginDate      : abap.dat;
  EndDate        : abap.dat;
  @Semantics.amount.currencyCode: 'CurrencyCode'
  BookingFee      : abap.dec( 17, 3 );
  @Semantics.amount.currencyCode: 'CurrencyCode'
  TotalPrice      : abap.dec( 17, 3 );
  @Semantics.currencyCode: true
  CurrencyCode    : abap.cuky( 5 );
  Description      : abap.char( 1024 ); //renamed element
  Status          : abap.char( 1 );

  data from remote service

  DiscountPct      : abap.dec( 3, 1 );
  @Semantics.amount.currencyCode: 'CurrencyCode'
  DiscountAbs      : abap.dec( 16, 3 );
  @Semantics.amount.currencyCode: 'CurrencyCode'
  TotalPriceWithDiscount : abap.dec(17,3);
  LastChangedAt    : timestamppl;

  data from local database table

}
}

  DiscountPct      : abap.dec( 3, 1 );                         >>> modifiable
  @Semantics.amount.currencyCode: 'CurrencyCode'
  DiscountAbs      : abap.dec( 16, 3 );                         >>> calculated
  @Semantics.amount.currencyCode: 'CurrencyCode'
  TotalPriceWithDiscount : abap.dec(17,3);                      >>> used for eTag
  LastChangedAt    : timestamppl;
```

The description of modifying the data that is exposed by the remote service is out of scope of this document. Nevertheless, each manipulation of the persistent data requires the retrieval of the remote data, as the discount data can only be stored in conjunction with the travel ID to match the discount to the relevant travel entry of the remote service. That is why, the creation of new discounts for a travel entry is considered as an update on a certain travel instance of the business object. In the case of creating a data set of discount data, you change the discount from the initial value but the key entry (TRAVEL_ID) is already there.

Compare the UI when retrieving the travel data merged with the discount data. The discount fields are displayed with the initial values if no discount data is maintained for the relevant travel instances.

TravelID	AgencyID	CustomerID	BeginDate	EndDate	TotalPrice	Discount %	Discount Absolute	Total Price with Discount	Last changed on
22	70037	540	Aug 4, 2018	Jun 2, 2019	2,244.51 USD	0	0.00 USD	2,244.51 USD	Dec 4, 2018, 3:53:22 PM >
23	70049	542	Aug 4, 2018	Jun 2, 2019	2,365.00 USD	0	0.00 USD	2,365.00 USD	Jul 22, 2018, 10:11:24 PM >
24	70050	346	Aug 4, 2018	Jun 2, 2019	6,072.64 USD	20	0.00 USD	4,858.11 USD	Dec 7, 2018, 1:02:59 PM >
25	70048	478	Aug 4, 2018	Jun 2, 2019	2,213.00 USD	0	0.00 USD	2,213.00 USD	Aug 7, 2018, 6:42:55 PM >
26	70006	220	Aug 4, 2018	Aug 4, 2018	1,245.00 USD	0	0.00 USD	1,245.00 USD	Nov 12, 2018, 12:50:47 PM >
27	70025	515	Aug 4, 2018	May 31, 2019	3,535.27 USD	0	0.00 USD	3,535.27 USD	Dec 5, 2018, 9:14:57 AM >

The modification of discount data is an **UPDATE**. Hence, we do not need to implement neither the **CREATE** nor the **DELETE** operation to maintain discount data.

A distinct field must be indicated as [ETag \[page 292\]](#) in the behavior definition. The eTag checks that the data the end user sees that on the UI is consistent with the data on the database and has not been changed in the meantime. The element `CalculatedEtag` is used for this purpose and calculated with data from the database table `/dmo/traveladd` and the remote service. This ensure that neither the remote data nor the local data can be changed without the end user knowing the change.

The newly created behavior definition requires

- defining an update (delete the definitions of create and delete if they are generated)
- defining the field for the eTag

The following code block displays the behavior definition with the definitions that are relevant for the transactional behavior of the business object.

```
implementation unmanaged;
define behavior for /DMOI_/TRAVEL_C_C alias Travel_CE
etag calculatedetag
{
  update;
}
```

Note

For better usability in the behavior implementation, you can define an alias for the business object in the behavior definition.

Related Information

[Adding Behavior to the Business Object \[page 289\]](#)

5.5.5.3.2 Implementing the Behavior Pool for the Business Object

The implementation of behavior is done in the local types of an ABAP class.

Creating a Behavior Pool /DMO/BP_TRAVEL_C_C

A [behavior pool \[page 805\]](#) for a behavior implementation is needed to implement the behavior that was defined in a behavior definition. A behavior pool is a special ABAP class that is equipped with the extension FOR BEHAVIOR OF and references the root entity of the relevant business object.

1. In the [Project Explorer](#), select the relevant behavior definition for which you want to create a behavior implementation class.
2. Open the context menu and select [New Behavior Implementation](#) to launch the creation wizard.
3. Follow the steps of the creation wizard and check if the suggested entries are correct.

Result

The generated global class pool provides you with the extension FOR BEHAVIOR OF with reference to the behavior definition. This statement determines the connection to the business object. One behavior implementation can only define the behavior for one business object.

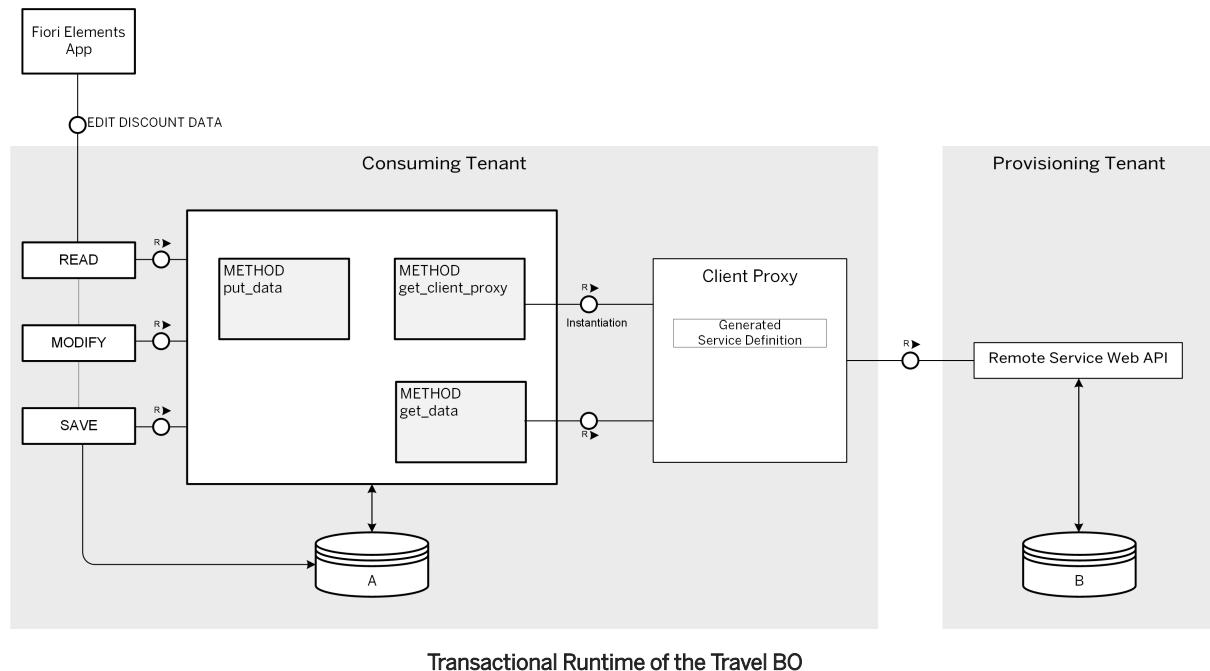
In the local types, a handler class with an [UPDATE method for modify \[page 721\]](#) is generated. The generated saver class contains the methods FINALIZE, CHECK_BEFORE_SAVE, and SAVE, the predefined standard methods for the save sequence.

For more information, see [Creating Behavior Implementations \[page 764\]](#).

Runtime Logic for Transactional Behavior in the Service Consumption Scenario

In a service consumption scenario, the transactional behavior for the business object (the updating of discount data) must be implemented in the local types of a behavior pool. The implementation includes the data retrieval from the remote service, merging the remote data with the local discount data and finally updating the local discount data. Hence, the logic of how to access a foreign system also comes into play when updating local data. To keep performance expenses on a low level, it is indispensable to work with a transactional buffer and retrieve the remote data only once in one unit of work. Therefore we use a local buffer class to work actively with a transactional buffer.

The runtime of an update operation of the discount data is displayed in the following diagram. It illustrates how the methods for READ, MODIFY, and SAVE and the local buffer class interact with each other and how the OData client proxy is used to request the necessary data from the remote service.



BUFFER

Due to performance reasons, it is useful to work with the transactional buffer in the service consumption scenario. Retrieving data from the remote service is very costly. Therefore, the data from the remote service should only be read once in one logical unit of work. We use the buffer as intermediate storage for data retrieval and processing.

The **READ**, **MODIFY**, and **SAVE** methods call dedicated methods from the buffer to process data. These buffer methods are:

- `get_data`
To receive data the Client Proxy is used. To instantiate the client proxy object, the method `get_client_proxy` is called. With this request object, you can define the resource object of the entity set and define the HTTP operation. After the execution, the data can be read from the response object returned by the execution.
In case of an error, the `FAILED` table is filled and can be used to raise an application error.
- `get_client_proxy`
This method establishes a cloud destination to the provisioning tenant and instantiates the client proxy. The destination is used at runtime for retrieving data from the remote service.
- `put_data`
This method calls `get_data` to retrieve the remote data. Additionally, it updates the local discount data for the retrieved entity and calculates the total price with discount. Finally, the method merges the remote and local data.
If there is an error, the `FAILED` table is filled and can be used to raise an application error.

A detailed step-by-step description for the implementation of the local buffer class is given in [Implementing the Buffer Class \[page 407\]](#).

`read_travel FOR READ`

For the ETag check (which is necessary for the `UPDATE` operation), a method for the `READ` operation must be implemented.

The latest timestamp of the field `lastchangedat` from `/dmo/traveladd` is used to compare with the data in `Last changed on` that is displayed on the UI. Only if these timestamps are equal is the `UPDATE` triggered.

→ Remember

The field `lastchangedat` is filled with the time stamp of the last modification of the discount data. In our example scenario, only the timestamp from the database table `/dmo/traveladd` is relevant as only these fields can be updated.

The eTag field is filled with the initial value if no discount is maintained for a certain travel instance.

i Note

The example implementation in this guide does not only retrieve the data from the relevant ETag fields, but reads the whole data set when the `READ` operation is executed, including the merge with the persistent data. This is done for performance reasons. To calculate the total price with discount, the value of the field `TotalPrice` from the remote instance is needed. Therefore it is more convenient to read the whole data set once, store it in the buffer and use this data set for the calculation on the update.

A detailed step-by-step description for the implementation of the `READ` is given in [Implementing the READ \[page 412\]](#).

For general information about the `READ` method, see [`<method> FOR READ \[page 726\]`](#).

`update_discount FOR MODIFY`

The real update takes place during the `MODIFY` operation. The method `update_discount` calls the buffer method `put_data`, which uses the method `get_data` to retrieve the data from the buffer (if available). The discount data is updated and the entry for the transient field `TotalPriceWithDiscount` is calculated.

The method `update_discount` also calls the method `_map_messages`, which wraps the messages to write them in the `REPORTED` table.

A detailed step-by-step description for the implementation of the `MODIFY` is given in [Implementing the MODIFY \[page 413\]](#).

For general information about the `MODIFY` method, see [<method> FOR MODIFY \[page 721\]](#).

SAVE

The `save` method writes the updated data to the database. It uses the method `get_data` to retrieve the updated data from the buffer. It generates a new time stamp to fill the field `lastchangedat` on the database table `/dmo/traveladd`. At last, it persists the updated data on the database table.

A detailed step-by-step description is given in [Implementing the SAVE \[page 416\]](#).

For general information about the `SAVE` method, see [Method SAVE \[page 734\]](#).

Implementation Steps

In this service consumption scenario, the `READ` and the `MODIFY` method, including a `_map_messages` method are treated in one handler class, as the retrieved data from the `READ` is reused in the `MODIFY` method. This relies to the [Best Practices for Modularization and Performance \[page 298\]](#).

This guide starts with the implementation of the buffer and consequently describes calls of the buffer methods from the predefined methods `READ`, `MODIFY`, and `SAVE`.

- [Implementing the Buffer Class \[page 407\]](#)
- [Implementing the READ \[page 412\]](#)
- [Implementing the MODIFY \[page 413\]](#)
- [Implementing the SAVE \[page 416\]](#)

5.5.5.3.2.1 Implementing the Buffer Class

The local buffer class serves as intermediate storage for the data processing in the service consumption scenario. It is a separate local class in the behavior pool.

Creating a Buffer Class

The buffer must be accessed from the predefined methods `READ`, `MODIFY`, and `SAVE`.

1. Create a local buffer class `lcl_buffer` in the local types of the behavior implementation pool.
2. Set the local class to `PRIVATE`.

Listing: Local Buffer Class

```
CLASS lcl_buffer DEFINITION CREATE PRIVATE.  
ENDCLASS.  
  
CLASS lcl_buffer IMPLEMENTATION.
```

```
ENDCLASS.
```

Creating a Buffer Instance

When working with a transactional buffer to store and process data, an instance of the buffer must be created at first. The singleton pattern is used to ensure that there is only one existing buffer instance of the application buffer.

1. Declare the static method `get_instance` which returns a buffer instance in the public section of the buffer class
2. Implement `get_instance` to receive a buffer instance if no instance exists

Listing: Declaration and Implementation of `get_instance`

```
CLASS lcl_buffer DEFINITION CREATE PRIVATE.  
  PUBLIC SECTION.  
    CLASS-METHODS get_instance  
      RETURNING VALUE(ro_instance) TYPE REF TO lcl_buffer.  
  PRIVATE SECTION.  
    CLASS-DATA: go_instance TYPE REF TO lcl_buffer.  
ENDCLASS.  
  
CLASS lcl_buffer IMPLEMENTATION.  
  METHOD get_instance.  
    IF go_instance IS NOT BOUND.  
      go_instance = NEW #().  
    ENDIF.  
    ro_instance = go_instance.  
  ENDMETHOD.  
ENDCLASS.
```

Retrieving Data from the Remote Service

The following listing represents the signature and implementation of the method `get_data` that retrieves data from the remote service, if it is not already in the buffer. The implementation of the method includes

1. checking whether the travel instance for which discount data is to be updated is already in the buffer and writing it to the exporting parameter `et_travel`, if available in the buffer. If not in the buffer, the `travel_id` has to be collected to retrieve the relevant data from the persistence.
2. the instantiation of a client proxy, by calling the method `get_client_proxy` of the auxiliary class `/DMO/CL_TARVEL_CP_AUX`, see [Creating a Remote Client Proxy \[page 383\]](#).
3. a reading request for the client proxy with a filter for only requesting the data set for the specific travel ID
4. a select to get and calculate the matching discount data from the database table `/dmo/traveladd` and to process the time stamp for the ETag field. A concatenated eTag from the fields `lastchangedat` of both, the remote and the local service, is used.
5. calculating the `TotalPriceWithDiscount` and setting an initial time stamp if no discount data has changed..

→ Remember

The eTag is used to ensure that discount data has not changed. That is why, the retrieved value is not used for the eTag check but the concatenated value is passed to the UI.

6. filling the `FAILED` table if the travel id is not found.
and
7. error handling if the access to the remote system fails.

i Note

If you want to display a suitable message, you need to create a message class. For example, here, the message Accessss to remote system cannot be established is convenient. It is retrieved from the message class /DMO/CM_SERV_CONS.

For information about how to create a message class, see .

Listing: Declaration and Implementation of get_data

```
CLASS lcl_buffer DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    CLASS-METHODS get_instance
      RETURNING VALUE(ro_instance) TYPE REF TO lcl_buffer.
    "types used in get_data
    TYPES: BEGIN OF ts_message,
      travelid TYPE /dmo/i_travel_c_c-travelid,
      symsg   TYPE symsg,
      fields   TYPE string_table,
    END OF ts_message,
      tt_travel      TYPE STANDARD TABLE OF /dmo/i_travel_c_c,
      tt_travel_in   TYPE TABLE FOR READ IMPORT /dmo/i_travel_c_c,
      tt_travel_out   TYPE TABLE FOR READ RESULT /dmo/i_travel_c_c,
      tt_travel_failed TYPE TABLE FOR FAILED /dmo/i_travel_c_c,
      tt_message     TYPE STANDARD TABLE OF ts_message.
    METHODS get_data
      IMPORTING it_travel      TYPE tt_travel_in OPTIONAL
      EXPORTING et_travel      TYPE tt_travel_out
        et_travel_failed TYPE tt_travel_failed
        et_message       TYPE tt_message
      RAISING   /dmo/cx_travel_query.
    PRIVATE SECTION.
      CLASS-DATA: go_instance TYPE REF TO lcl_buffer.
      DATA: mt_travel      TYPE tt_travel.
    ENDCLASS..
```



```
CLASS lcl_buffer IMPLEMENTATION.
  ...
  METHOD get_data.
    DATA: lt_travel      TYPE STANDARD TABLE OF /dmo/travel_c_a.
    DATA: ls_result      LIKE LINE OF et_travel.
    DATA: lt_travel_id   TYPE STANDARD TABLE OF /dmo/i_travel_c_c-travelid.
    DATA: lt_filter      TYPE RANGE OF /dmo/i_travel_c_c-travelid.
    DATA: ls_filter      LIKE LINE OF lt_filter.
    DATA: lt_travel_ce   TYPE STANDARD TABLE OF /dmo/i_travel_c_c.
    DATA: lt_traveladd   TYPE STANDARD TABLE OF /dmo/traveladd.
    FIELD-SYMBOLS: <fs_travel_ce> LIKE LINE OF lt_travel_ce.
    IF it_travel IS SUPPLIED.
      LOOP AT it_travel ASSIGNING FIELD-SYMBOL(<fs_travel>).
        IF line_exists( mt_travel[ travelid = <fs_travel>-travelid ] ).
          ls_result = CORRESPONDING #( mt_travel[ travelid = <fs_travel>-travelid ] ).
          " collect from buffer for result
          APPEND ls_result TO et_travel.
        ELSE.
          " collect to retrieve from persistence
          APPEND <fs_travel>-travelid TO lt_travel_id.
        ENDIF.
      ENDLOOP.
    IF lt_travel_id IS NOT INITIAL.
      TRY.
        DATA(lo_client_proxy) = /dmo/cl_travel_cp_aux=>get_client_proxy( ).
        DATA(lo_request) = lo_client_proxy-
      >create_resource_for_entity_set('TRAVEL')->create_request_for_read( ).
```

```

        lt_filter = VALUE #( FOR travel_id IN lt_travel_id ( sign = 'I'
option = 'EQ' low = travel_id ) ).  

        DATA(lo_filter) = lo_request->create_filter_factory( )-  

>create_by_range( iv_property_path = 'TRAVELID'  

        it_range      = lt_filter ).  

        lo_request->set_filter( lo_filter ).  

        DATA(lo_response) = lo_request->execute( ).  

        " get relevant data sets  

        lo_response->get_business_data( IMPORTING et_business_data =  

lt_travel ).  

        " add local data  

IF lt_travel IS NOT INITIAL.  

    " map OData service to custom entity  

    lt_travel_ce = CORRESPONDING #( lt_travel MAPPING description =  

memo ).  

    SELECT * FROM /dmo/traveladd FOR ALL ENTRIES IN @lt_travel_ce  

WHERE travel_id = @lt_travel_ce-travelid INTO TABLE @lt_traveladd.  

    LOOP AT lt_travel_id ASSIGNING FIELD-SYMBOL(<fs_travel_id>).  

        IF line_exists( lt_travel_ce[ travelid = <fs_travel_id> ] ).  

            ASSIGN lt_travel_ce[ travelid = <fs_travel_id> ] TO  

<fs_travel_ce>.  

            IF line_exists( lt_traveladd[ travel_id = <fs_travel_ce>-  

travelid ] ).  

                <fs_travel_ce>-discountpct      = lt_traveladd[ travel_id  

= <fs_travel_ce>-travelid ]-discount_pct.  

                <fs_travel_ce>-discountabs     = lt_traveladd[ travel_id  

= <fs_travel_ce>-travelid ]-discount_abs.  

                <fs_travel_ce>-totalpricewithdiscount = <fs_travel_ce>-  

totalprice * ( 1 - <fs_travel_ce>-discountpct / 100 ) - <fs_travel_ce>-  

discountabs.  

                <fs_travel_ce>-lastchangedat      = lt_traveladd[ travel_id  

= <fs_travel_ce>-travelid ]-lastchangedat.  

                <fs_travel_ce>-calculatedettag   = <fs_travel_ce>-  

calculatedettag && '-' && lt_traveladd[ travel_id = <fs_travel_ce>-travelid ]-  

lastchangedat.  

            ELSE.  

                <fs_travel_ce>-totalpricewithdiscount = <fs_travel_ce>-  

totalprice.  

                <fs_travel_ce>-lastchangedat = '20000101120000' . "initial  

value Jan 1, 2000, 12:00:00 AM  

            ENDIF.  

            ls_result = CORRESPONDING #( <fs_travel_ce> ).  

APPEND <fs_travel_ce> TO mt_travel.  

APPEND ls_result           TO et_travel.  

        ELSE.  

APPEND VALUE #( travelid = <fs_travel_id> ) TO  

et_travel_failed.  

APPEND VALUE #( travelid      = <fs_travel_id>  

symsg-msgty = 'E'  

symsg-msgid = '/DMO/CM_SERV_CONS'  

symsg-msgno = '004'  

symsg-msgv1 = <fs_travel_id> )  

TO et_message.  

        ENDIF.  

    ENDLOOP.  

ENDIF.  

CATCH /iwbepl/cx_gateway.  

    et_travel_failed = CORRESPONDING #( lt_travel_id MAPPING travelid =  

table_line ).  

    et_message = CORRESPONDING #( lt_travel_id MAPPING travelid =  

table_line ).  

    LOOP AT et_message ASSIGNING FIELD-SYMBOL(<fs_message>).  

        <fs_message>-symsg-msgty = 'E'.  

        <fs_message>-symsg-msgid = '/DMO/CM_SERV_CONS'.  

        <fs_message>-symsg-msgno = '001'.  

    ENDLOOP.  

ENDTRY.

```

```

ENDIF.
ELSE.
    et_travel = CORRESPONDING #( mt_travel ).
ENDIF.
ENDMETHOD.
```

Updating the Discount Data

With the preceding method, the data is retrieved from the remote service into the buffer. The essence of the UPDATE operation, however, is done with the method `put_data`. It uses the data in the buffer, which is already enriched with the additional discount data from the database table `/dmo/traveladd`, and writes the new discount data in the buffer. This is done by

1. calling the method `get_data` to retrieve the data from the buffer
2. looping at `it_travel_upd` to identify the changed fields
3. overwriting the changed fields with the new discount data
4. calculating the data for `TotalPriceWithDiscount`
5. filling the `FAILED` table in case the total price with discount is negative after discount calculation.

i Note

If you want to display a suitable message, you need to add a message to the message class `/DMO/CM_SERV_CONS`. In this case, an exception needs to be thrown if the Total Price with Discount is negative. For example, here, the message `Total Price with Discount` must be greater than 0 is convenient.

For information about how to create a message class, see .

6. writing the updated data set to the buffer.

Listing: Declaration and Implementation of `put_data`

```

CLASS lcl_buffer DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    "types used in put_data
    TYPES:
      tt_travel_upd    TYPE TABLE FOR UPDATE /dmo/i_travel_c_c,
      tt_travel_mapped TYPE TABLE FOR MAPPED /dmo/i_travel_c_c.
    METHODS put_data
      IMPORTING it_travel_upd    TYPE tt_travel_upd
      EXPORTING et_travel_failed TYPE tt_travel_failed
          et_message        TYPE tt_message.
    ...
  ENDCLASS.

CLASS lcl_buffer IMPLEMENTATION.
  ...
  METHOD put_data.
    get_data(
      EXPORTING it_travel           = CORRESPONDING #( it_travel_upd mapping %key =
%key except * )
      IMPORTING et_travel            = data(lt_travel)
              et_travel_failed     = DATA(lt_travel_failed)
              et_message           = DATA(lt_message)
    ).
    LOOP AT it_travel_upd ASSIGNING FIELD-SYMBOL(<fs_travel_upd>).
      CHECK line_exists( lt_travel[ KEY entity COMPONENTS travelid =
<fs_travel_upd>-travelid ] ).
      ASSIGN lt_travel[ KEY entity COMPONENTS travelid = <fs_travel_upd>-
travelid ] TO FIELD-SYMBOL(<fs_travel>).
```

```

        IF <fs_travel_upd>-%control-&DiscountAbs = if_abap_behv=>mk-on.
          <fs_travel>-DiscountAbs = <fs_travel_upd>-DiscountAbs.
        ENDIF.
        IF <fs_travel_upd>-%control-&DiscountPct = if_abap_behv=>mk-on.
          <fs_travel>-DiscountPct = <fs_travel_upd>-DiscountPct.
        ENDIF.
      ENDLOOP.
      " Postprocessing
      LOOP AT lt_travel ASSIGNING FIELD-SYMBOL(<fs_traveladdinfo>).
        DATA(lv_totalpricediscount) = <fs_traveladdinfo>-totalprice * ( 1 -
<fs_traveladdinfo>-discountpct / 100 ) - <fs_traveladdinfo>-discountabs.
        IF lv_totalpricediscount >= 0.
          <fs_traveladdinfo>-totalpricewithdiscount = lv_totalpricediscount.
        ELSE.
          APPEND VALUE #( travelid      = <fs_traveladdinfo>-travelid ) TO
et_travel_failed.
          APPEND VALUE #( travelid      = <fs_traveladdinfo>-travelid
                        symsg-msgty = 'E'
                        symsg-msgid = '/DMO/CM_SERV_CONS'
                        symsg-msgno = '009'
                        symsg-msgv1 = <fs_traveladdinfo>-travelid
                        symsg-msgv2 = |{ lv_totalpricediscount NUMBER = USER }|
                        fields       = VALUE #( ( |discountpct| )
                                              ( |discountabs| )
                                         )
                     )
        )
      TO et_message.
    ENDIF.
  ENDLOOP.
  "save data in buffer
  mt_travel = CORRESPONDING #( lt_travel ) .
ENDMETHOD.
ENDCLASS.
```

5.5.5.3.2.2 Implementing the READ

The READ is necessary for ETag check.

The READ method processes reading requests. Hence, it is necessary to retrieve the data from the field `lastchangedat`.

This implementation does not only retrieve the time stamp, which is used for the eTag check, but the whole data set to store it in the buffer and use it in the actual UPDATE phase without retrieving the data again. The implementation must ensure that this data is retrieved and handed over to the framework, which handles the comparison with the time stamp of the UI. It is also the framework that prevents the update if the time stamps do not match.

See [<method> FOR READ \[page 726\]](#) for information about parameters of the READ.

Declaring a method for READ

A code template for the READ method is generated automatically if an update is declared in the behavior definition.

1. Rename the method for read in the definition part of the handler class with the importing parameter `it_travel_read` and the result parameter `et_traveladdinfo`, which is used for the ETag check. The implementation.

```
CLASS lhc_travel_update DEFINITION INHERITING FROM cl_abap_behavior_handler.
```

```

PRIVATE SECTION.
METHODS read_travel FOR READ
    IMPORTING it_travel_read FOR READ travel_ce
    RESULT et_traveladdinfo.
...
ENDCLASS.

```

Implementing the method for READ

The actual READ is done by the method `get_data` in the buffer class. It either retrieves the data from the remote system into the buffer or reads the data from the buffer. The data is provided with the result parameter `et_traveladdinfo`. The method `read_travel` instantiates the buffer and calls the method `get_data`.

1. Instantiate the buffer with the method `get_instance`.
 2. Call the `get_data` method of the buffer class to get the data into `et_traveladdinfo`.
- If the READ does not return the requested instance, the key of the instance to be read must be written in the `failed` table.

→ Remember

The `failed` table is an implicit changing parameter of the READ method.

```

METHOD read_travel.
DATA(lo_buffer) = lcl_buffer->get_instance( ).
lo_buffer->get_data(
    EXPORTING
        it_travel           = it_travel_read
    IMPORTING
        et_traveladdinfo   = et_traveladdinfo
        et_travel_failed   = failed-travel_ce
).
ENDMETHOD.

```

5.5.5.3.2.3 ☁ Implementing the MODIFY

The MODIFY updates data.

The MODIFY method is called by the framework when an UPDATE is defined in the behavior definition. It modifies the data in the application buffer. In this consumption scenario, the buffer method `put_data` is called to do this. To return adequate messages, the MODIFY method also includes a mapping method for messages.

Declaring a method for MODIFY

The declaration of the MODIFY method is generated by the template for the behavior implementation as the UPDATE was declared in the behavior definition.

1. Change the name of the MODIFY method to the more transparent name `update_discount`.
Change the name of the importing parameter into `it_travel_update`.

```

CLASS lhc_travel_update DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
...
METHODS update_discount FOR MODIFY
    IMPORTING it_travel_update FOR UPDATE travel_ce.
...

```

```
ENDCLASS.
```

Implementing the method for MODIFY

The real update is done by the method `put_data` in the buffer class. This method firstly calls the method `get_data` to retrieve the data from the buffer or get it from the remote service. Then it updates the discount data for the selected travel instance and calculates the amount for `TotalPriceDiscount`.

For the processing of the appropriate messages if there is a fault event, the method `_map_messages` is called within the method `update_discount`.

1. Instantiate the buffer with the method `get_instance`.
2. Call the `put_data` method of the buffer class to update the discount data for the selected travel ID.
If the `UPDATE` does not work, the key of the failed travel instance must be written in the `failed` table.

→ Remember

The `failed` table is an implicit changing parameter of the method for `MODIFY`.

Write the messages of `put_data` into a local table `lt_message`.

3. Call the `_map_messages` method to map `lt_message` to the `reported` table.

→ Remember

The `reported` table is an implicit changing parameter of the method for `MODIFY`.

```
METHOD update_discount.  
  DATA(lo_buffer) = lcl_buffer=>get_instance( ).  
  lo_buffer->put_data  
    EXPORTING  
      it_travel_upd      = it_travel_update  
    IMPORTING  
      et_travel_failed   = failed-travel_ce  
      et_message         = DATA(lt_message)  
    ).  
  _map_messages(  
    EXPORTING  
      it_message        = lt_message  
    IMPORTING  
      et_travel_reported = reported-travel_ce  
  ).  
ENDMETHOD.
```

Message Handling

In case of failure, the issue has to be transferred to the `REPORTED` table, which includes all instance-specific messages. For the processing of such messages, the method `_map_messages` is used. In the consumption scenario, it is defined in the local handler class, as it is only used in the `MODIFY` method.

Declare _map_messages

1. Declare the method `_map_messages` with an importing parameter `it_message` that is compatible to the type the method `put_data` returns for messages; and an exporting parameter `et_travel_reported` that is compatible to the `reported` table of the travel entity.

```
CLASS lhc_travel_update DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
...  
    TYPES: tt_travel_reported TYPE TABLE FOR REPORTED /dmo/  
i_travel_ce.  
    METHODS: _map_messages  
        IMPORTING it_message TYPE lcl_buffer=>tt_message  
        EXPORTING et_travel_reported TYPE tt_travel_reported.  
...  
ENDCLASS.
```

Implement _map_messages

To write messages to the REPORTED table, they have to align with a fixed structure. Messages for this structure are created via the method `new_message`, which is inherited from the interface `IF_ABAP_BEHV_MESSAGE`.

1. For each message in `it_message`, create a new message for the REPORTED table by calling the method `new_message`.
2. Fill the relevant fields in `ls_travel_reported` with the `travel_id` of the message and change the flag for the element which causes the message.

```
METHOD _map_messages.  
    DATA: ls_travel_reported LIKE LINE OF et_travel_reported.  
    FIELD-SYMBOLS: <fs_element> TYPE data.  
    LOOP AT it_message ASSIGNING FIELD-SYMBOL(<fs_message>).  
        CLEAR ls_travel_reported.  
        ls_travel_reported-%msg = new_message( id = <fs_message>-symsg-  
msgid  
                    number = <fs_message>-symsg-  
msgno  
                    severity =  
if_abap_behv_message=>severity-error  
                    v1 = <fs_message>-symsg-  
msgv1  
                    v2 = <fs_message>-symsg-  
msgv2  
                    v3 = <fs_message>-symsg-  
msgv3  
                    v4 = <fs_message>-symsg-  
msgv4 ).  
        IF <fs_message>-travelid IS NOT INITIAL.  
            ls_travel_reported-%key-travelid = <fs_message>-travelid.  
            ls_travel_reported-travelid = <fs_message>-travelid.  
            LOOP AT <fs_message>-fields ASSIGNING FIELD-SYMBOL(<fs_field>).  
                ASSIGN COMPONENT <fs_field> OF STRUCTURE ls_travel_reported-  
%element TO <fs_element>.  
                CHECK sy-subrc = 0.  
                <fs_element> = if_abap_behv=>mk-on.  
            ENDLOOP.  
            APPEND ls_travel_reported TO et_travel_reported.  
        ENDIF.  
    ENDLOOP.  
ENDMETHOD.
```

5.5.5.3.2.4 Implementing the SAVE

The template for the behavior implementation provides you with the method declarations for redefinitions of the save sequence.

```
CLASS lsc_travel DEFINITION INHERITING FROM cl_abap_behavior_saver.  
  PROTECTED SECTION.  
    METHODS finalize          REDEFINITION.  
    METHODS check_before_save REDEFINITION.  
    METHODS save              REDEFINITION.  
  ENDCLASS.
```

Note

In this consumption scenario, we do not exemplify the implementation for `finalize` and `check_before_save` as both of the implementations are optional and not required in our sample scenario.

In the method `SAVE`, the transactional buffer is saved to the database. The following listing illustrates the procedure for the `SAVE` implementation. The saving action includes

1. instantiating of the buffer instance
2. calling the method `get_data` to get the updated data set
3. retrieving the latest time stamp
4. modifying the database table `/dmo/traveladd`

```
CLASS lsc_travel_update IMPLEMENTATION.  
  METHOD finalize.  
  ENDMETHOD.  
  METHOD check_before_save.  
  ENDMETHOD.  
  METHOD save.  
    DATA: ls_traveladd TYPE /dmo/traveladd.  
    DATA(lo_buffer) = lcl_buffer=>get_instance( ).  
    lo_buffer->get_data(  
      IMPORTING  
        et_travel = DATA(lt_travel)  
    ).  
    LOOP AT lt_travel ASSSIGNING FIELD-SYMBOL(<fs_traveladdinfo>).  
      ls_traveladd = CORRESPONDING #( <fs_traveladdinfo> MAPPING travel_id      =  
      travelid  
                                discount_pct =  
      discountpct  
                                discount_abs =  
      discountabs ).  
      GET TIME STAMP FIELD ls_traveladd-lastchangedat.  
      MODIFY /dmo/traveladd FROM @ls_traveladd.  
    ENDOLOOP.  
  ENDMETHOD..  
ENDCLASS.
```

5.5.6 Defining an OData Service

In order to be able to consume the service with an SAP Fiori UI, you need to create an OData service.

1. Follow the development steps as described in [Creating an OData Service \[page 24\]](#).

Create a *service definition* and a *service binding* to expose the CDS custom entity including its query implementation for a service. You only need to include the custom entity in the service definition, its implementation is included implicitly.

You can then test the resulting UI service by using the *Fiori Elements Preview* in the service binding.

2. Follow the steps described in [Designing the User Interface for a Fiori Elements App \[page 35\]](#)

Use @UI annotations to define the UI of the SAP Fiori app. UI annotations are maintained in the CDS custom entity in exactly the same way as in CDS views.

For this consumption scenario, it is convenient to label the element for line items and identification annotations as the data element information is not retrieved from the remote service.

Note

For selection fields, it is not possible to maintain a label in the @UI annotations. Use the @EndUser.label annotation instead for elements with selection fields.

 Expand the following listing to view the source code of the travel CDS view.

```
@EndUserText.label: 'CE for Service Consumption Scenario'  
@QueryImplementedBy: '/DMO/CL_TRAVEL_C_Q'          //reference to query  
implementation class  
@UI.headerInfo:{ typeName: 'Trip',  
                 typeNamePlural: 'Trips'}  
  
define root custom entity /DMO/TRAVEL_C_C  
{  
    @UI.facet           : [  
        { id             : 'Travel',  
         purpose        : #STANDARD,  
         type           : #IDENTIFICATION_REFERENCE,  
         label          : 'Travel',  
         position       : 10 } ]  
    @UI                : {lineItem: [ { position: 10, label: 'Travel  
ID', importance: #HIGH } ],  
                         selectionField: [ { position: 10 } ],  
                         identification: [ { position: 10, label:  
'Travel ID' } ]}  
    @EndUserText.label : 'Travel ID'  
    key TravelID      : abap.numc( 8 );  
    @UI                : {lineItem: [ { position: 20, label: 'Agency  
ID', importance: #HIGH } ],  
                         selectionField: [ { position: 20 } ],  
                         identification: [ { position: 20, label:  
'Agency ID' } ]}  
    @EndUserText.label : 'Agency ID'  
    AgencyID          : abap.numc( 6 );  
    @UI                : {lineItem: [ { position: 30, label: 'Customer  
ID', importance: #HIGH } ],  
                         selectionField: [ { position: 30 } ],  
                         identification: [ { position: 30, label:  
'Customer ID' } ]}  
    @EndUserText.label : 'Customer ID'  
    CustomerID        : abap.numc( 6 );  
    @UI                : {lineItem: [ { position: 40, label: 'Starting  
Date', importance: #MEDIUM } ],
```

```

        identification:[ { position: 40, label:
'Starting Date' } ] }
    BeginDate : abap.dats;
    @UI : {lineItem: [ { position: 45, label: 'End
Date', importance: #MEDIUM } ],
identification:[ { position: 45, label:
'End Date' } ] }
    EndDate : abap.dats;
    @UI : {identification:[ { position: 60, label:
'Booking Fee' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    BookingFee : abap.dec( 17, 3 );
    @UI : {lineItem: [ { position: 70, label: 'Total
Price', importance: #MEDIUM } ],
identification:[ { position: 70, label:
'Total Price' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    TotalPrice : abap.dec( 17, 3 );
    @UI : {lineItem: [ { position: 80, label: 'Discount
%', importance: #MEDIUM } ],
identification:[ { position: 80, label:
'Discount %' } ] }
    DiscountPct :
abap.int1;
    @UI : {lineItem: [ { position: 75, label: 'Discount
Absolute', importance: #MEDIUM } ],
identification:[ { position: 75, label:
'Discount Absolute' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    DiscountAbs : abap.dec( 17,
3 ); //modifiable data
    @UI : {lineItem: [ { position: 72, label: 'Total
Price with Discount', importance: #MEDIUM } ],
identification:[ { position: 72, label:
'Total Price with Discount' } ] }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    TotalPriceWithDiscount : abap.dec( 17,
3 ); //calculated data
    @Semantics.currencyCode: true
    CurrencyCode : abap.cuky;
    @UI : {lineItem: [ { position: 50, label: 'Comment',
importance: #MEDIUM } ],
identification:[ { position: 50, label:
'Comment' } ] }
    Description :
abap.char( 1024 );
    @UI : {lineItem: [ { position: 90, label: 'Status',
importance: #HIGH } ],
identification:[ { position: 90, label:
>Status' } ] }
    Status : abap.char( 1 );
    @EndUserText.label : 'Last changed on'
    LastChangedAt : timestamppl;
}

```

3. Add static feature control for attributes to set them read-only or mandatory.

The only attributes that are modifiable are `DiscountPct` and `DiscountAbs`. They need to be filled with values. Set all other attributes as mandatory. For further information, see [Adding Feature Control \[page 459\]](#).

i Expand the following listing to view the source code of the travel CDS view with the relevant feature control additions.

```

implementation unmanaged;
define behavior for /DMO/TRAVEL_C_C alias Travel_CE
etag LASTCHANGEDAT
{ update;

```

```
    field (read only) TRAVELID, AGENCYID, CUSTOMERID,  
          BEGINDATE, ENDDATE, DESCRIPTION,  
          BOOKINGFEE, TOTALPRICE, TOTALPRICEWITHDISCOUNT,  
          STATUS, CURRENCYCODE;  
    field (mandatory) DISCOUNTABS, DISCOUNTPCT;  
}
```

Related Information

[Creating an OData Service \[page 24\]](#)

[Designing the User Interface for a Fiori Elements App \[page 35\]](#)

[Adding Feature Control \[page 459\]](#)

6 Extend

SAP provides you with a solution to extend delivered applications with additional features. You can extend an application without modifying the delivered code.

Two approaches are available:

Key User Extensibility

The extensibility option by [S/4 HANA](#) enables you to customize an app that was delivered by SAP or an SAP partner with additional features. In that case you do not have to add or modify any delivered code, but you can apply the configuration directly in the app. This is only possible if the app is enabled for extensibility.

More on this: [General Functions for the Key User](#)

SAP recommends to use this option whenever possible and sufficient.

7 Common Tasks

In the common tasks section you find self-contained development tasks that you can apply in any development scenario.

Data Model

[Defining Text Elements \[page 422\]](#)

[Providing Value Help \[page 428\]](#)

[Enabling Text and Fuzzy Searches in SAP Fiori Apps \[page 437\]](#)

[Using Aggregate Data in SAP Fiori Apps \[page 446\]](#)

Business Objects

 [Automatically Drawing Primary Key Values in Managed BOs \[page 455\]](#)

[Adding Feature Control \[page 459\]](#)

 [Using Type and Control Mapping \[page 477\]](#)

 [Using Groups in Large Development Projects \[page 481\]](#)

[Consuming Business Objects with EML \[page 469\]](#)

 [Adding Authorization Control to Managed Business Objects \[page 487\]](#)

Query

[Implementing an Unmanaged Query \[page 494\]](#)

UI Semantics

[Adding Field Labels and Descriptions \[page 513\]](#)

[Defining CDS Annotations for Metadata-Driven UIs \[page 515\]](#)

7.1 Defining Text Elements

This section describes how to determine and provide related text for a CDS view element within the context of the ABAP RESTful Programming Model.

Example

Contents

You have different options to provide text for CDS view elements:

- [Providing Text by Text Elements in the Same Entity \[page 422\]](#)

Use this option if the identifier element and the text element are part of the same entity. You can establish a direct link between the two elements.

- [Getting Text Through Text Associations \[page 424\]](#)

Use this option if the identifier element and the text element are not part of the same entity, or if the text element is language-dependent and shall be displayed in the system log-on language in scenarios without projection layer.

- [Getting Language-Dependent Text in Projection Views \[page 426\]](#)

Use this option if you are working with a projection layer.

i Note

You cannot combine text associations with projection scenarios.

7.1.1 Providing Text by Text Elements in the Same Entity

Language independent text elements can be maintained in the same entity as the identifier element.

Within the context of CDS views, the text elements establish the link between identifier elements (code values) of the view and its descriptive language-independent texts. For example, you can define a link between a company code and the (descriptive) company name, or between currency code and the currency name if these elements are part of the one CDS view. These kinds of descriptive texts are **language-independent**. That means, they do not contain text that is to be translated.

Relevant Annotation

Annotation	Effect
<code>@ObjectModel.text.element[]</code>	Establishes the link between the annotated element (that defines an identifier element) and its descriptive language-independent texts

i Note

The usage of this annotation excludes the usage of `@ObjectModel.text.association` in the same CDS entity. .

More on this: [ObjectModel Annotations \[page 565\]](#)

i Note

Runtime Behavior: In scenarios with exposure via OData, only the **first text element** listed in the annotation array will be handled as a descriptive text of the annotated field.

Example

In the listing below, the CDS view `/DMO/I_SupplementText` defines the fields `Description` that serves as language-independent descriptions for the view field `SupplementID`.

```
define view /DMO/I_SupplementText
  as select from /dmo/suppl_text as SupplementText
{
  @ObjectModel.text.element: ['Description']
  key SupplementText.supplement_id as SupplementID,
  ...
  SupplementText.description as Description
  ...
}
```

Related Information

[Getting Text Through Text Associations \[page 424\]](#)

[Getting Language-Dependent Text in Projection Views \[page 426\]](#)

7.1.2 Getting Text Through Text Associations

Context

Using the CDS text association, you can define the relationship between an element (field) and its corresponding texts or descriptions. The texts are usually language-dependent and are displayed in the end user's language. If you annotate an element with a text association (as described below), the associated text or description field is added as a field to the referencing entity. At runtime, this field is read from the database and filtered by the logon language of the OData consumer automatically. It is not necessary to use session properties or view parameters in your CDS view.

i Note

In scenarios, in which you use projection views, getting text through text associations is not supported.

To retrieve texts by direct use of text associations, proceed as follows:

Procedure

1. Create a data definition with a CDS view that serves as text provider

The following annotations are required at **element level** in the text provider view to annotate the language key element and the text elements of the view's element list:

Annotation and Values	Effect
@Semantics.language: true	The annotated element identifies the language field.
@Semantics.text: true	Identifies view elements as text elements (fields pointing to a textual description)

i Note

In general, you can annotate more than one view field as a text field. However, only the first annotated field will be considered in the text consumer view for OData exposure.

More on this: [Semantics Annotations \[page 606\]](#)

2. Create a text association from your consumer CDS view to the text provider view .

The following CDS annotation is relevant when creating text associations:

Annotation and Values	Effect
@ObjectModel.text.association: '<_AssocToTextProvider>' <_AssocToTextProvider>.	Name of an association with a text view that provides descriptive texts for the annotated element. In other words: the annotation indicates that the description for the annotated element is available using the text association <_AssocToTextProvider>.

More on this: [ObjectModel Annotations \[page 565\]](#)

The view /DMO/I_BookSuppl_T serves as a consumer for the text provider view /DMO/I_SupplText_T. For this purpose, the association _SupplementText with the text provider view as target is defined. To indicate the field for which a text should be made available through the association _SupplementText, the annotation @ObjectModel.text.association is added. Note that only the first text element (Description) from the text provider, which is annotated with @Semantics.text: true, will be considered for OData exposure. In Fiori Elements apps, the supplement ID will then be displayed together with the long text in description of the text provider view.

Text Provider View /DMO/I_SupplText_T

```
define view /DMO/I_SupplText_T as select from /dmo/suppl_text as
SupplementText
{
    key SupplementText.supplement_id as SupplementID,
        @Semantics.language: true
    key SupplementText.language_code as LanguageCode,
        @Semantics.text: true
    SupplementText.description as Description
        @Semantics.text: true
    SupplementText.alt_Description as AlternativeDescription
}
```

Text Consuming View /DMO/I_BookSuppl_T

```
define root view /DMO/I_BookSuppl_T as select from /dmo/book_suppl as
BookingSupplement
    association [1..*] to /DMO/I_SupplText_T as _SupplementText on
$projection.supplement_id = _SupplementText.SupplementID
{
    key travel_id,
    key booking_id,
    key booking_supplement_id,
        @ObjectModel.text.association: '_SupplementText'
        supplement_id,
        /* Associations */
        _SupplementText
}
```

Related Information

[Providing Text by Text Elements in the Same Entity \[page 422\]](#)

[Getting Language-Dependent Text in Projection Views \[page 426\]](#)

7.1.3 Getting Language-Dependent Text in Projection Views

The denormalization of language-dependent text in projection views is done via the keyword `localized` for the text elements, which are included in the projection view and referenced with the annotation `@ObjectModel.text.element: '<text_element>'`.

Context

You have a text provider view, in which text for unreadable elements is maintained. To get the text from there you do not need any preparation in the text provider view.

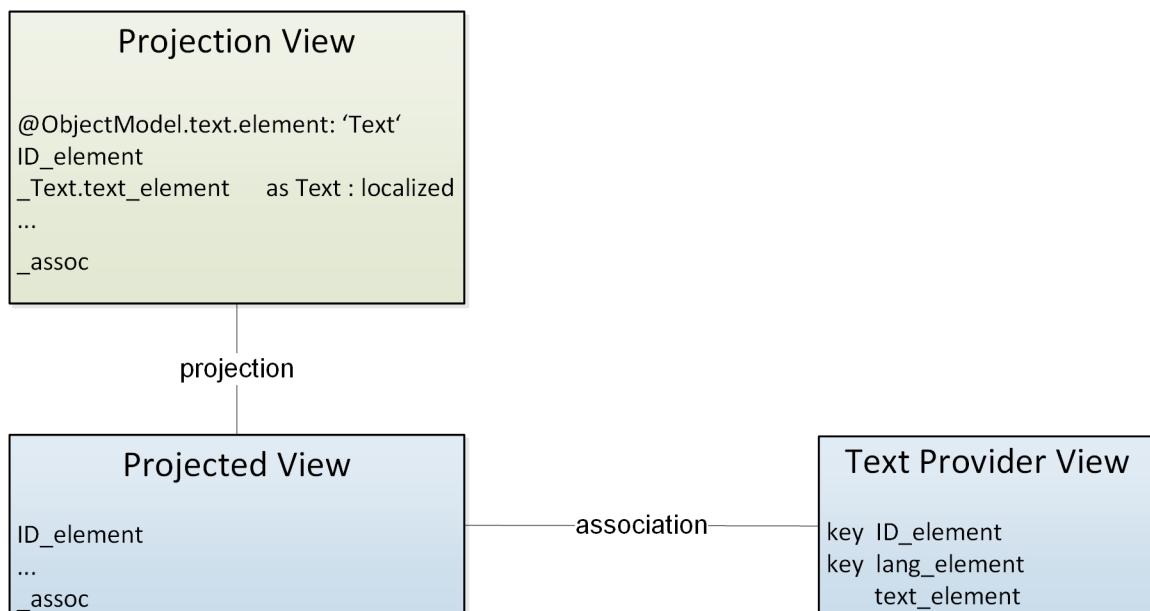
To establish a connection to this text provider view, the projected view must have an association to the text provider view.

i Note

To use the denormalization in projection views, you must not use the indicator for text associations `@ObjectModel.text.association: '<text_association>'` in the projected view.

You can then include the text element via the association in the projection view. The relationship between identifier elements and the respective text is defined in the projection view via the annotation `@ObjectModel.text.element: '<text_element>'`, see [CDS Projection View \[page 103\]](#). If the keyword `localized` is used, the text in the system log-on language is drawn.

The following diagram illustrates the modeling of text denormalization in projection views.



Prerequisites

- The text provider view has a key element indicating the language of the text.
- The projected CDS view has an association to the text provider view, but does not use the annotation `ObjectModel.text.association`.

Procedure

1. Include the text element from the text provider view into the projection view.
2. Annotate the element in the projection view, for which you want to provide the text with `@ObjectModel.text.element: <text_element>` and reference the text element.
3. To get language-dependent texts use the keyword `localized` on the text element.

• Example

Text Provider View

The text provider view selects from a database table, in which the description texts in different languages are stored.

```
define view /DMO/I_SupplText_T as select from /dmo/suppl_text as
SupplementText
{
    key SupplementText.supplement_id as SupplementID,
    key SupplementText.language_code as LanguageCode,
    SupplementText.description as Description
}
```

Projected View

```
define root view /DMO/I_BookSuppl_T as select from /dmo/book_suppl as
BookingSupplement
    association [1..*] to /DMO/I_SupplText_T as _SupplementText on
$projection.supplement_id = _SupplementText.SupplementID
{
    key travel_id,
    key booking_id,
    key booking_supplement_id,
    supplement_id,
    /* Associations */
    _SupplementText
}
```

Projection View

The projection view denormalizes the supplement description and filters the relevant values based on the requested language.

i Note

It depends on the language configuration of your cloud system, which language is allowed.

```
define root view entity /DMO/C_BookSuppl_T as projection on /DMO/I_BookSuppl_T
{
```

```

key travel_id           as TravelID,
key booking_id          as BookingID,
key booking_supplement_id as BookingSupplementID,
@ObjectModel.text.element: ['SupplementDescription']
supplement_id            as SupplementID,
_SupplementText.Description      as SupplementDescription: localized,
_SupplementText

```

If you create a UI service for this example, you can check the results when sending the request in different languages.

Product ID	Product Price	Description
Apple Juice (BV-0005)	3.50 EUR	Apple Juice
Lemon Limonade (BV-0008)	3.50 EUR	Lemon Limonade
Pear Juice (BV-0006)	3.50 EUR	Pear Juice
Mango Juice (BV-0007)	3.50 EUR	Mango Juice
Hot Chocolate (BV-0001)	2.30 EUR	Hot Chocolate
Alcohol free Champagne (BV-0002)	7.50 EUR	Alcohol free Champagne

Product ID	Product Price	Description
Apfelsaft (BV-0005)	3,50 EUR	Apfelsaft
Zitronenlimonade (BV-0008)	3,50 EUR	Zitronenlimonade
Birensaft (BV-0006)	3,50 EUR	Birensaft
Mangosaft (BV-0007)	3,50 EUR	Mangosaft
Heiße Schoki (BV-0001)	2,30 EUR	Heiße Schoki
Alkoholfreier Champagner (BV-0002)	7,50 EUR	Alkoholfreier Champagner

Supplement Descriptions in English and German

Related Information

[Getting Text Through Text Associations \[page 424\]](#)

[Providing Text by Text Elements in the Same Entity \[page 422\]](#)

7.2 Providing Value Help

The implementation of a value help in CDS enables the end user to choose values from a predefined list for input fields on a user interface.

Why and When to Use Value Help

You can define value helps in the CDS data layer for any input field on the UI (for example, selection fields, parameters, or writable fields in transactional scenarios). A value help is useful when the values for the input field have to be taken from a predefined set of values. When you call the value help, it displays all the available values from the value help provider. It appears on the UI when you choose the button in the input field or

press the **F4** key. The end user can filter by related information to identify the correct value. This makes it easier to find the desired value, especially if the value itself contains little or no identifying information, for example, an ID number.

• Example

To help the end user to enter the right customer ID to create a new booking, the application developer defines a value help that enables the user to enter the name or any other element from the value help provider to help find the correct number. The value help provider view in this case is a CDS view that manages customer information. As shown below, the end user is searching for a particular customer ID. The value help offers to filter by the customer last name, so that the end user can choose from the available entries. The value of the customer ID field is then transferred to the respective input field.

Expand the following figure to view the procedure of calling the value help on a Fiori Elements UI.

The screenshot shows a Fiori Elements UI for a travel booking application. The header bar includes the SAP logo and the text "Travel". The main area is titled "<Unnamed Object>" and contains a section for "General Information". It has five input fields: "Agency ID" (value: 70006), "Customer ID" (empty), "Starting Date" (placeholder: "MMM d, y"), "End Date" (placeholder: "MMM d, y"), and "Booking Fee" (two empty input fields). Below these fields is a dark blue button labeled "Total Price". At the bottom right are "Save" and "Cancel" buttons. A caption at the bottom of the screenshot reads "Value Help on a Fiori Elements UI".

How are Value Helps Implemented?

Value helps are defined in CDS with an annotation on the element or parameter for which the value help dialog is to appear on the UI. The annotation `@Consumption.valueHelpDefinition` allows you to reference the value help provider without implementing an association. You simply assign a CDS entity as the value help provider and provide an element for the mapping in the annotation. All fields of the value help provider are displayed on the UI. When you choose one of the entries of the value help provider, the value of the referenced element is transferred to the input field.

For the default implementation of the value help, you can use any CDS entity that contains the desired values of the element for the input field. You do not need to explicitly define a CDS entity as the value help provider. However, the value help provider must be exposed for the OData service to make use of the value help.

i Note

Any annotation that is maintained in the value help provider is evaluated. This means that associated entities, value helps, and text associations of the value help provider view are also displayed and can be used in the value help. This means that you can have a value help in the value help.

The following value help options are available within the programming model:

[Simple Value Help \[page 430\]](#)

[Multiple Value Helps on One Element \[page 433\]](#)

[Value Help with Additional Binding \[page 435\]](#)

Related Information

[Providing Value Help for the Selection Fields \[page 138\]](#)

[Consumption Annotations \[page 562\]](#)

7.2.1 Simple Value Help

A simple value help is convenient if you want to display values from the value help provider for an input field.

Context

You want to provide a value help for an input field on the UI.

The following steps implement a value help for a customer ID field, using a booking CDS view as an example.

Procedure

1. Create a data definition for a CDS view that serves as a value help provider. It must contain a field with the available values for the input field in the source view.

• Example

The value help provider view contains the customer ID and fields to identify the customer ID, such as the customer's name or address. The end user can then filter by these fields to find the right customer ID.

```
define view /DMO/I_Customer_VH as select from /dmo/customer
{
  key customer_id,
  first_name,
```

```
last_name,  
title,  
street,  
postal_code,  
city,  
country_code,  
phone_number,  
email_address  
}
```

2. In your CDS source view, add the following annotation to the element for which you want to provide the value help on the UI.

```
@Consumption.valueHelpDefinition: [{ entity: { name: 'entityRef' ,  
element:  
'elementRef' } } ]
```

The annotation defines the binding for the value help to the value help providing entity. You must specify the entity name and the element providing the possible values for the annotated element.

Example

The following code example shows how an annotation is used on the element `CustomerID` in `/DMO/I_Booking`. It references the value help provider view (`/DMO/I_CUSTOMER_VH`) for the customer ID and the element providing the possible values (`customer_id`) in the value help provider view.

```
define view /DMO/I_Booking as select from /dmo/booking  
{...  
    @Consumption.valueHelpDefinition: [{entity: { name: '/DMO/  
I_CUSTOMER_VH',  
element:  
'customer_id' }}]  
    customer_id as CustomerID,  
    ... }
```

Results

Choosing **[F4]** in the selection field opens a search mask and the end user can filter by any field in the value help provider view. Selecting an entry transfers the value of the element that is referenced in the annotation to the annotated element in the source view.

The metadata of the OData service displays the value help implementation for the following properties:

- The property in the CDS source view for which a value help is implemented (`sap:value-list="standard"`)
- The value help provider entity type (`sap:value-list="true"`)
- The value help provider entity is marked as `Target` in the `Annotations` property. The value list enumerates every property in the value help provider that is exposed for the value help (`Annotation Term="Common.ValueList"`).
- The element that is defined in the mapping condition is marked as an inbound and outbound parameter `Record Type="Common.ValueListParameterInOut"`.

i Expand to view the extracts of the metadata document of a service exposing a booking CDS view and the value help provider for the element `CustomerID`.

```

- <EntityType sap:content-version="1" sap:label="Consumer CDS View - Booking" Name="BookingType">
  + <Key>
    <Property sap:label="Travel ID" Name="TravelID" sap:quickinfo="Flight Reference Scenario: Travel ID" sap:display-format="NonNegative" MaxLength="8" Nullable="false" Type="Edm.String"/>
    <Property sap:label="Booking Number" Name="BookingID" sap:quickinfo="Flight Reference Scenario: Booking ID" sap:display-format="NonNegative" MaxLength="4" Nullable="false" Type="Edm.String"/>
    <Property sap:label="Booking Date" Name="BookingDate" sap:quickinfo="Flight Reference Scenario: Booking Date" sap:display-format="Date" Type="Edm.DateTime" Precision="0"/>
    <Property sap:label="Customer ID" Name="CustomerID" sap:quickinfo="Flight Reference Scenario: Customer ID" sap:display-format="NonNegative" MaxLength="6" Type="Edm.String" sap:value-list="standard"/>
    <Property sap:label="Airline ID" Name="CarrierID" sap:quickinfo="Flight Reference Scenario: Carrier ID" sap:display-format="UpperCase" MaxLength="3" Type="Edm.String"/>
    <Property sap:label="Flight Number" Name="ConnectionID" sap:quickinfo="Flight Reference Scenario: Connection ID" sap:display-format="NonNegative" MaxLength="4" Type="Edm.String"/>
    <Property sap:label="Flight Date" Name="FlightDate" sap:quickinfo="Flight Reference Scenario: Flight Date" sap:display-format="Date" Type="Edm.DateTime" Precision="0"/>
    <Property sap:label="Flight Price" Name="FlightPrice" sap:quickinfo="Flight Reference Scenario: Flight Price" Type="Edm.Decimal" Precision="17" sap:unit="CurrencyCode" Scale="3"/>
    <Property sap:label="Currency Code" Name="CurrencyCode" sap:quickinfo="Flight Reference Scenario: Currency Code" MaxLength="5" Type="Edm.String" sap:semantics="currency-code"/>
  </EntityType>
- <EntityType sap:content-version="1" sap:label="CDS View Customer - Value Help Provider" Name="CustomerType" sap:value-list="true">
  + <Key>
    <Property sap:label="Customer ID" Name="customer_id" sap:quickinfo="Flight Reference Scenario: Customer ID" sap:display-format="NonNegative" MaxLength="6" Nullable="false" Type="Edm.String"/>
    <Property sap:label="First Name" Name="first_name" sap:quickinfo="Flight Reference Scenario: First Name" MaxLength="40" Type="Edm.String"/>
    <Property sap:label="Last Name" Name="last_name" sap:quickinfo="Flight Reference Scenario: Last Name" MaxLength="40" Type="Edm.String"/>
  </EntityType>
- <Annotations xmlns="http://docs.oasis-open.org/odata/ns/edm" Target="cds_xdmobooking_vh.BookingType/CustomerID">
  - <Annotation Term="Common.ValueList">
    - <Record>
      <PropertyValue String="CDS View Customer - Value Help Provider" Property="Label"/>
      <PropertyValue String="Customer" Property="CollectionPath"/>
      <PropertyValue Property="SearchSupported" Bool="true"/>
    - <PropertyValue Property="Parameters">
      - <Collection>
        - <Record Type="Common.ValueListParameterInOut">
          <PropertyValue Property="LocalDataProperty" PropertyPath="CustomerID"/>
          <PropertyValue String="customer_id" Property="ValueListProperty"/>
        </Record>
        - <Record Type="Common.ValueListParameterDisplayOnly">
          <PropertyValue String="first_name" Property="ValueListProperty"/>
        </Record>
        - <Record Type="Common.ValueListParameterDisplayOnly">
          <PropertyValue String="last_name" Property="ValueListProperty"/>
        </Record>
        + <Record Type="Common.ValueListParameterDisplayOnly">
          <Record Type="Common.ValueListParameterDisplayOnly">
            <PropertyValue String="street" Property="ValueListProperty"/>
          </Record>
        + <Record Type="Common.ValueListParameterDisplayOnly">
          <Record Type="Common.ValueListParameterDisplayOnly">
            <PropertyValue String="city" Property="ValueListProperty"/>
          </Record>
        - <Record Type="Common.ValueListParameterDisplayOnly">
          <Record Type="Common.ValueListParameterDisplayOnly">
            <PropertyValue String="country_code" Property="ValueListProperty"/>
          </Record>
        - <Record Type="Common.ValueListParameterDisplayOnly">
          <Record Type="Common.ValueListParameterDisplayOnly">
            <PropertyValue String="phone_number" Property="ValueListProperty"/>
          </Record>
        + <Record Type="Common.ValueListParameterDisplayOnly">
          <Record Type="Common.ValueListParameterDisplayOnly">
            <PropertyValue String="email_address" Property="ValueListProperty"/>
          </Record>
        </Collection>
      </PropertyValue>
    </Record>
  </Annotation>

```

This value help provider is also search supported.

Metadata Document

Postrequisites

If the source view is exposed for an OData service, you need to include the value help provider view in the service definition to be able to retrieve the values from the value help.

Other Value Help Options

Other Capabilities for the Value Help Provider

Any annotation that is maintained in the value help provider is evaluated. This means that the following capabilities are possible for the value help:

- Associations:** If the target of the association is included in the OData service, the elements of entities associated with the value help provider are also displayed as additional input fields.
- Search Capabilities:** Including search capabilities in your value help provider enables the end user to search for any detail in an input field.

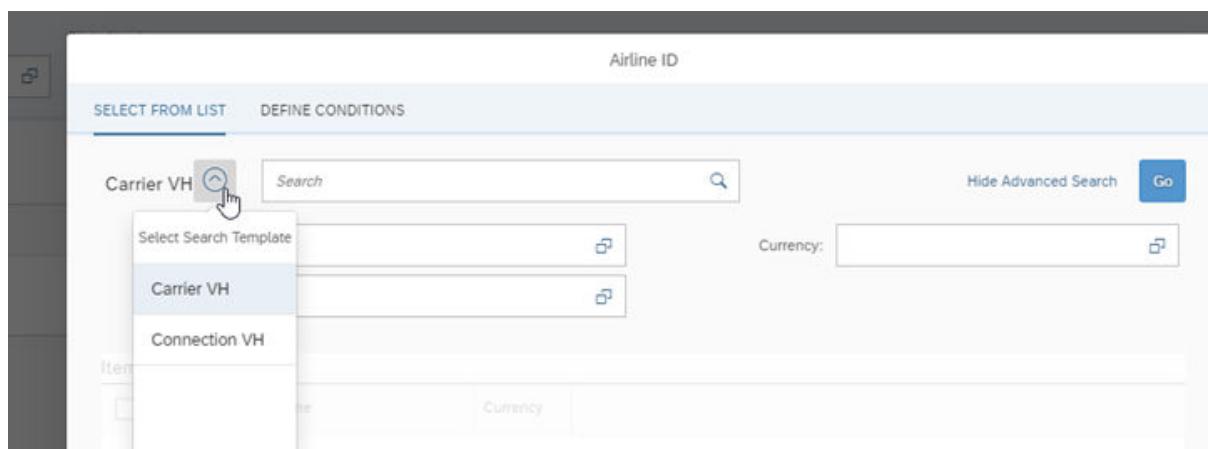
- **Value Helps:** The value help can itself contain value helps.
- **Text:** Text that is provided for the value help provider is also displayed.

7.2.1.1 Multiple Value Helps on One Element

Context

It is possible to provide more than one value help on one element. The end user selects which value help to use to find the correct value.

The following image displays the value helps for the carrier ID element in the booking CDS view. One value help provider is defined by a carrier CDS view and one by a connection CDS view that also contains the carrier ID field.



Two Value Helps on one Element

Procedure

1. To implement two value helps on one element, proceed as described in [Simple Value Help \[page 430\]](#) and add another entity as a value help provider.

```
define view /DMO/I_Booking as select from /dmo/booking
{...
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/
I_Carrier_VH' ,
                                              element:
'carrier_id' } }
                                         { entity: { name: '/DMO/
I_Connection_VH',
                                              element:
'carrier_id' } ]
    carrier_id as CarrierID,
... }
```

You can define more than two value helps on one element.

- Assign labels to the different value helps to differentiate them on the UI.

```
define view /DMO/I_Booking as select from /dmo/booking
{...
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/
I_Carrier_VH' ,
                                              element:
'carrier_id' },
                                         label: 'Carrier VH' },
                                         { entity: { name: '/DMO/
I_Connection_VH',
                                              element:
'carrier_id' },
                                         label: 'Connection VH'}]
    carrier_id as CarrierID,
... }
```

- Equip one value help with a qualifier.

```
define view /DMO/I_Booking as select from /dmo/booking
{...
    @Consumption.valueHelpDefinition: [{ entity: { name: '/DMO/
I_Carrier_R' ,
                                              element:
'carrier_id' },
                                         label: 'Carrier VH' },
                                         { entity: { name: '/DMO/
I_Connection_VH',
                                              element:
'carrier_id' },
                                         label: 'Connection VH',
                                         qualifier: 'Secondary Value
Help'}]
    carrier_id as CarrierID,
... }
```

If you have more than one value help, it is important that all except one are equipped with a qualifier. The default value help is the one without a qualifier. The qualifier marks the value helps that are less important. If all value helps are annotated with the qualifier argument, then none of them are displayed as there is no default.

Results

Choosing **F4** in the input field opens a search mask with the fields of the default value help. The default value help is the one without a qualifier. The end user can select which value help to use from a dropdown menu.

7.2.2 Value Help with Additional Binding

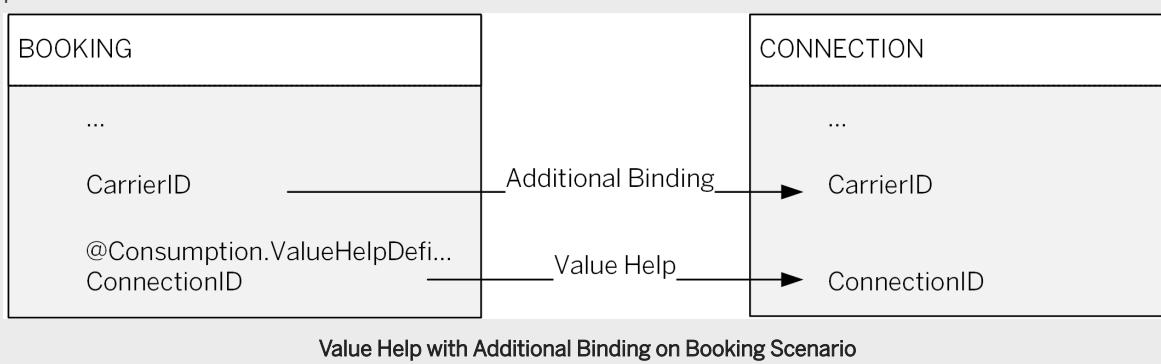
A preset filter condition can be established by an additional binding for the value help.

Context

You use an additional binding to define more than one binding condition between the source view and the value help provider. The value help is then automatically filtered by the additional binding. It proposes only entries that match the additional binding. This additional binding can either be another element or a parameter. These need to be present in the source view and in the value help provider view. When an entry is selected in the value help provider, the values of both binding elements are transferred to the input fields of the source CDS view.

Example

In our booking scenario, we can apply the value help with an additional binding on the field `ConnectionID`. The value help provider is a view that manages connections. This value help provider contains not only the field for the connection IDs, but also a field for carrier IDs, which is also in the consumer view. We can establish a second binding condition so that the value help provider only displays connections with the prechosen carrier ID.



Procedure

1. Create a data definition for a CDS view that serves as a value help provider. It must contain a field with the available values for the input field in the source view. In addition, it must contain the field for which the additional binding is established.

The value help provider view contains the connection ID and the carrier ID, for which a mapping condition is defined.

```
define view /DMO/I_Connection_VH as select from /dmo/connection
{
  key carrier_id,
  key connection_id,
  airport_from_id,
  airport_to_id,
  departure_time,
```

```

    arrival_time,
    distance,
    distance_unit
}

```

2. In your CDS source view, add the following annotation to the element for which you want to provide the value help on the UI.

```

        @Consumption.valueHelpDefinition: [{ entity: {name:
'entityRef', element: 'elementRef'},
additionalBinding: [{ element: 'elementRef',
localElement: 'elementRef' }]}]

```

The additional binding defines a second condition for the value help on the same target value help provider entity for filtering the value help result list and/or returning values from the selected value help record. The additional binding can be defined for an additional element or parameter.

The annotation requires an array as value.

The following code example shows the usage of the annotation on the element `ConnectionID` in `/DMO/I_Booking`. It references the value help provider view (`/DMO/I_Connection_VH`) and the element providing the possible values (`connection_id`) in the value help provider view, as well as the matching condition on the elements `CarrierID` and `carrier_id` in the consumer view and the value help provider view, respectively.

```

define view /DMO/I_Booking_VH as select from /dmo/booking
{...
  @Consumption.valueHelpDefinition: [{ entity:
    {name: '/DMO/I_Connection_VH', element: 'connection_id' },
    additionalBinding: [{ localElement: 'CarrierID', element:
      'carrier_id' }]}
  connection_id as ConnectionID,
  ...
}

```

Results

Choosing **F4** in the selection field opens a search mask and the end user can display all available entries in the value help provider or filter by a field, for example, the destination airport. If the carrier ID is already filled in the consumer view, the value help provider is prefetched by that value. Selecting an entry in the value help transfers the connection ID as well the carrier ID to the CDS consumer view.

The metadata of the OData service displays the value help implementation on the following properties:

- The property in the CDS source view for which a value help is implemented (`sap:value-list="standard"`)
- The value help provider entity type (`sap:value-list="true"`)
- The value help provider entity is marked as `Target` in the `Annotations` property. The value list enumerates every property in the value help provider that is exposed for the value help (`Annotation Term="Common.ValueList"`).
- The elements that are defined in the mapping conditions are marked as inbound and outbound parameters `Record Type="Common.ValueListParameterInOut"`.

Next Steps

If the consumer view is exposed for an OData service, you need to include the value help provider view in the service definition to be able to retrieve the value help.

7.3 Enabling Text and Fuzzy Searches in SAP Fiori Apps

The descriptions in this topic refer to the range of functions for text and fuzzy searches that are provided in the context of SAP HANA.

Text and Fuzzy Searches

The full text searching (or just text search) provides you with the capability to identify natural language terms that satisfy a query and, optionally, to sort them by relevance (ranking) to the query. The most common type of search is to find texts that contain the term specified and return them in the order of their similarity to these terms.

Fuzzy search is a fast and fault-tolerant search feature of SAP HANA. The basic concept behind the **fault-tolerant search** is that a database query returns records even if the search term (user input) contains additional or missing characters, or even spelling errors. Fuzzy search can be used in various applications -- for example, to trigger a fault-tolerant search in a structured database content, like a search for a product called 'coffe krisp biscuit' and you find 'Toffee Crisp Biscuits'.

Providing Freestyle Search Capabilities in SAP Fiori UI screen

Within the context of the ABAP RESTful programming model, you only need to enable the text and fuzzy search functionality in your data model definitions. For this purpose, you implement it in designated CDS views using appropriate text and fuzzy search annotations (listed below).

i Note

As an application developer however, you must ensure that your CDS views are suitable for text and fuzzy search enabling. For more information take a look at the corresponding topics in the [SAP HANA Search Developer Guide](#).

Annotations for Text- and Fuzzy Search

As the name suggests, search annotations enable the search feature on the CDS view elements.

First of all, you need the following CDS annotation at the **view level**:

Annotation and Value	Effect
<code>@Search.searchable: true/false</code>	Defines whether a CDS view is generally relevant for search scenarios. This annotation provides a general switch and a means to quickly detect whether a view is search-relevant or not. Set to value true in order to enable search support by means of @Search annotations. Here, at least one view field must be defined as <code>@defaultSearchElement</code> at element level.

The annotations (required) at the **element level** are:

Annotation and Values	Effect
<code>@Search.defaultSearchElement: true/false</code>	<p>Specifies that the annotated element is to be considered in a full-text search</p> <div style="background-color: #e0f2f1; padding: 10px;"> <p>i Note</p> <p>At least one element has to be defined for the default full-text search. Searching in views without default full-text search elements is not supported!</p> </div>
	<p>All view elements that are annotated for the default search define the search scope. (The search will be performed on all elements that have this annotation.).</p> <div style="background-color: #e0f2f1; padding: 10px;"> <p>⚠ Caution</p> <p>Such a search must not operate on all elements – for performance reasons and because not all elements qualify for this kind of access.</p> </div>
<code>@Search.fuzzinessThreshold: <value></code>	<p>This annotation specifies the least level of fuzziness the element has to have in order to be considered in a fuzzy search at all.</p> <p>The <code><value></code> defines the threshold for a fuzzy search (how fuzzy scores are calculated when comparing two strings or two terms).</p> <p>Possible values are: 0 .. 1 The default value is 1. The fuzzy search algorithm calculates a fuzzy score for each string comparison. The higher the score, the more similar the strings are. A score of 1 . 0 means the strings are identical. A score of 0 . 0 means the strings have nothing in common.</p>
<code>@Search.ranking: <value></code>	<p>This annotation specifies how relevant the values of an element (view field) are for ranking, should the freestyle search terms match the element's value.</p> <p>The ranking can have the following values:</p> <ul style="list-style-type: none"> • HIGH - The element is of high relevance; typically, this is useful for IDs and their descriptions. • MEDIUM - The element is of medium relevance; designated usually for important elements. • LOW - Although the element is relevant for a freestyle search, a hit for this element has no real significance for the ranking of a result item.

→ Tip

For the fuzzy search threshold, we recommend using the default value `0.7` to start with. Later on, you can fine-tune the value based on your experiences with the search. You can also fine-tune the search using feedback collected from your users.

Example

The listing below implements a search model for searching products. The model definition results from the data source `db_pd` that already specifies the persistence layer for searching. The data source provides product data and text reference fields.

The annotation `@Search.searchable: true` marks the view as searchable. In addition, the elements `Name` and `Category` are annotated with `@Search.defaultSearchElement: true`. This means that a freestyle search is enabled on the search UI where it is possible to search for the annotated elements. The annotation `@Search.fuzzinessThreshold: 0.7 (0.8)` defines that the text search should be applied to the element `Category` with a similarity value of 70% and to the element `Name` with a similarity value of 80%.

```
...
@Search.searchable : true

define view SearchForPuducts as select from db_pd

key pd_id as ID,
    @Search.defaultSearchElement : true
    @Search.fuzzinessThreshold : 0.8
    @Search.ranking : #HIGH
    pd_text as Name,
    @Search.defaultSearchElement : true
    @Search.fuzzinessThreshold : 0.7
    @Search.ranking : #LOW
    pd_category as Category
}
```

Results

If you expose a CDS view with search annotations for an OData service, the OData entity set receives the annotation `sap:searchable: true`.

The following image display a Fiori UI that consumes an OData service with the example CDS view.

The screenshot shows a SAP Fiori application interface. At the top, there is a filter bar with two standard filters: 'Standard *' and 'Notebooks HT-'. A yellow box highlights the 'Notebooks HT-' filter. To its right is a button with a close icon and the text 'Hide Filter Bar'. Below the filter bar, a callout bubble points to the 'Notebooks HT-' filter with the text: 'To show filters here, add them to the filter bar in Filters'. The main area displays a table with three columns: Product Category, Product ID, and a third column which is partially visible. The table contains four rows, all of which have 'Notebooks' in the Product Category column and a value starting with 'HT-' in the Product ID column.

Product Category	Product ID	
Notebooks	HT-1000	
Notebooks	HT-1001	
Notebooks	HT-1002	

Standard filter allows to search for product category and product name

Related Information

[Search Annotations \[page 602\]](#)

7.4 Using Virtual Elements in CDS Projection Views

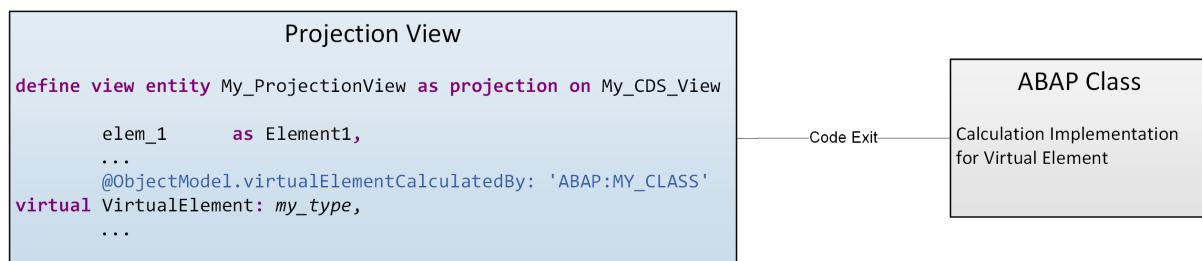
With virtual elements, you define additional CDS elements that are not persisted on the database, but calculated during runtime using ABAP classes that implement the virtual element interface.

What Are Virtual Elements, and Why Do I Need Them?

For some business use cases, it may be necessary to add new elements to the data model of an application whose values cannot be fetched from the database. Virtual elements are used if these elements are not provided as part of the original persistence data model but can be calculated using ABAP resources during runtime.

Virtual elements represent transient fields in business applications. They are defined at the level of CDS projection views as additional elements within the `SELECT` list. However, the calculation of their values is carried out by means of ABAP classes that implement the specific virtual element interface provided for this purpose. The ABAP implementation class is referenced by annotating the virtual element in the CDS projection view with `@ObjectModel.virtualElementCalculatedBy: 'ABAP:<CLASS_NAME>'`. With data retrieval via OData, the query framework calls the ABAP implementation class of the virtual element to retrieve the value for it.

The OData service metadata do not differentiate between regular CDS elements with database persistence and virtual elements. Consequently, the virtual element appears in an application UI equal to any other element.



Definition of Virtual Element in CDS Projection View and Implementation in Referenced ABAP Class

i Note

- You can use virtual elements only in CDS projection views.
- In the current version of the ABAP RESTful Programming Model, you cannot add implementations for filtering or sorting by virtual elements.
- Virtual elements cannot be keys of the CDS projection view.
- Virtual elements cannot be used together with the grouping or the aggregation function.
- Data from virtual elements can only be retrieved via the query framework. In particular this means that the following options to retrieve data from CDS are **not** possible for virtual elements:
 - ABAP SQL SELECT on CDS views return initial values for the virtual element,
 - EML READ on BO entities is not possible as EML does not know virtual elements.

Example

In an application that processes flight bookings, we want to include a field that calculates how many days are left until the flight date, or how many days have passed since the flight date. The following booking app UI provides the booking information with an additional field `Days to Flight` that calculates the days that have passed since the flight or the days that are left until the flight. The value is calculated by comparing the flight date with the system date.

Travel ID	Booking Number	Booking Date	Customer ID	Airline ID	Flight Number	Flight Date	Flight Price	Days to Flight
1	1	Nov 27, 2019	Ryan (594)	United Airlines, Inc. (UA)	1537	Dec 14, 2019	438.00 USD	-79 >
1	2	Dec 14, 2019	Ryan (594)	American Airlines Inc. (AA)	322	Dec 16, 2019	438.00 USD	-77 >
1	3	Sep 20, 2020	Ryan (594)	United Airlines, Inc. (UA)	1537	Oct 9, 2020	438.00 USD	221 >
1	4	Sep 24, 2020	Ryan (594)	American Airlines Inc. (AA)	322	Oct 11, 2020	438.00 USD	223 >
2	1	Dec 12, 2019	Detemple (99)	United Airlines, Inc. (UA)	1537	Dec 14, 2019	438.00 USD	-79 >
2	2	Dec 12, 2019	Sisko (660)	United Airlines, Inc. (UA)	1537	Dec 14, 2019	438.00 USD	-79 >

Flight Bookings List with Virtual Element to Calculate Days to Flight

Developer Activities

1. Adding the virtual element and corresponding annotations to the relevant CDS projection view.

For a more detailed description, see [Modeling Virtual Elements \[page 442\]](#).

2. Creating and implementing an ABAP class that implements the virtual element calculation.

For more detailed information, see [Implementing the Calculation of Virtual Elements \[page 443\]](#).

7.4.1 Modeling Virtual Elements

Virtual elements are defined in the `select` list of CDS projection views. Their values are calculated in a referenced ABAP class. This topic explains how you can model virtual elements in CDS.

Context

A virtual element is declared in the CDS projection view. It enables you to have additional fields that are not persisted on the database in your application scenario. You can also have different virtual elements in different BO projections. With those, you are flexible in providing an accurately tailored projection layer for one specific service without affecting the basic business object.

You can use virtual element for read-only, as well as for transactional scenarios.

The following steps model a virtual element in a CDS projection view using the CDS view `/DMO/I_BOOKING_U` as an underlying CDS view. The projection CDS view defines the subset of the CDS view that is relevant to the respective service and is extended with a virtual field for this service.

Syntax for Defining a Virtual Element in CDS Projection View

```
define view entity CDSProjView
  as projection on CDSEntity
{
  key      elem_1           as Element1,
          [@EndUserText.label: 'Element Label']
          [@EndUserText.quickInfo: 'Quick Information']
          @ObjectModel.virtualElementCalculatedBy: 'ABAP:ABAPClass'
  virtual  ELEMNAME : {DataElement | ABAPType} ,
  ...
}
```

Explanation

To define a virtual element, use the keyword `virtual` and specify a name for the virtual element. Aliases for virtual elements are not allowed, as you can choose the name of the element freely. Since a virtual element does not have database persistence and therefore does not have a defined data type, you have to specify the type of the element. Predefined data elements can be used as well as ABAP types.

For more information about the syntax in CDS projection views, see [CDS Projection View \[page 103\]](#).

Annotate the virtual element with the annotation `@ObjectModel.virtualElementCalculatedBy`, and reference the ABAP class that calculates the values for the virtual element.

i Note

As the `@ObjectModel` annotation references an ABAP resource from CDS, you must use the exact syntax '`'ABAP:<class_name>`' as the value for the annotation. No space between colon and `<class_name>` is allowed.

If you use a basic ABAP type for your virtual element, define end-user information for the element with the `@EndUser.Text` annotation.

The label you choose for the end-user text is used in the metadata of an OData service that uses the CDS views. It is also used as a fallback if no label is maintained in `@UI` annotations to display the element on a UI.

If the virtual element uses a predefined data element, the OData metadata deduces the information from the data element.

• Example

```
define view entity /DMO/C_Booking_VE
    as projection on /DMO/I_Booking_U
{
    ...
    @ObjectModel.virtualElementCalculatedBy: 'ABAP:/DMO/
CL_DAYS_TO_FLIGHT'
        @EndUserText.label: 'Days to Flight'
        @EndUserText.quickInfo: 'Calculates the Relative Flight Date'
    virtual DaysToFlight : abap.int2,
    ...
}
```

Next Steps

1. Creating and implementing an ABAP class that implements the virtual element contract.
More on this: [Implementing the Calculation of Virtual Elements \[page 443\]](#).

7.4.1.1 Implementing the Calculation of Virtual Elements

The values of virtual elements are calculated in a referenced ABAP class. This topic explains how you calculate their values by means of an example implementation.

Context

By using a virtual element in a CDS projection view, you define an additional field for your OData service. As this additional field is not persisted in the database layer, its value must be calculated during runtime. The calculation is implemented in an ABAP class that is referenced in an annotation on the virtual element in the CDS projection view.

The calculation class must implement the interface `IF_SADL_EXIT_CALC_ELEMENT_READ`. This interface provides two methods for the calculation implementation:

- `get_calculation_info`: This method is called before the actual data retrieval. It ensures that all the relevant elements that are needed for the calculation of the virtual element are selected.
- `calculate`: This method is called after the data retrieval. It uses the values of the relevant elements to calculate the value for the virtual element.

Prerequisites

- The virtual element is modeled in the CDS projection view as explained in [Modeling Virtual Elements \[page 442\]](#).
- The virtual element uses the annotation `@ObjectModel.virtualElementCalculatedBy` which references an existing ABAP class.

Procedure

1. Add the interface `IF_SADL_EXIT_CALC_ELEMENT_READ` to the public section of your calculation class and add the two method implementations.

Example

```
CLASS /dmo/cl_days_to_flight DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_sadl_exit_calc_element_read.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_days_to_flight IMPLEMENTATION.
  METHOD if_sadl_exit_calc_element_read~get_calculation_info.
  ENDMETHOD.
  METHOD if_sadl_exit_calc_element_read~calculate.
  ENDMETHOD.
ENDCLASS.
```

2. Implement the method `get_calculation_info`.

With this method, you can check that the calculation for the virtual element is only executed if the relevant element is requested. In addition, you must provide a list of elements that are required for the calculation. You can only add elements of the requested CDS entity.

For more information, see the method description of [get_calculation_info \[page 756\]](#).

If an entity or a virtual element other than the intended is requested, raise adequate exceptions. For more information on application-specific message handling, see `, ,` and `.`

Example

At first, we check if the requested entity is the entity that contains the virtual element for which the calculation is intended. To calculate the remaining time or the passed time compared to the flight date, we need the element `FlightDate`. This element is added to the requested elements list to ensure that the value is available to calculate the virtual element.

It is possible to calculate the values for more than one virtual element in the same ABAP class. Depending on the virtual element, you might have to append different elements to the list of requested elements.

```
CLASS /dmo/cl_days_to_flight IMPLEMENTATION.
  METHOD if_sadl_exit_calc_element_read~get_calculation_info.
    IF iv_entity <> '/DMO/Č_BOOKING_VE'.
      RAISE EXCEPTION TYPE /dmo/cx_virtual_elements
        EXPORTING
          textid = /dmo/cx_virtual_elements=>entity_not_known
          entity = iv_entity.
    ENDIF.
    LOOP AT it_requested_calc_elements ASSIGNING FIELD-
      SYMBOL(<fs_calc_element>).
      CASE <fs_calc_element>.
        WHEN 'DAYSTOFLIGHT'.
          APPEND 'FLIGHTDATE' TO et_requested_orig_elements.
        * WHEN 'ANOTHERELEMENT'.
        * APPEND '' ...
        WHEN OTHERS.
          RAISE EXCEPTION TYPE /dmo/cx_virtual_elements
            EXPORTING
              textid = /dmo/cx_virtual_elements=>virtual_element_not_known
              element = <fs_calc_element>
              entity = iv_entity.
      ENDCASE.
    ENDLOOP.
  ENDMETHOD.
ENDCLASS.
```

i Note

The methods of the interface `IF_SADL_CAL_ELEMENT_READ` import and export their parameters as strings in upper case. The CDS entity and element names must therefore be in upper case as well, so they can be mapped correctly.

3. Implement the method `calculate`.

With this method, you must pass as they are previously retrieved. The parameter `it_requested_calc_elements` contains one or more virtual elements. In the changing parameter `ct_calculated_data`, you must provide the calculated value for the virtual element.

For more information, see the method description of [calculate \[page 757\]](#).

• Example

In our example, we want to calculate how many days are left before the flight or how many days have passed since the flight, so we compare the value of `FLIGHTDATE` with today's date to calculate the value for the virtual element.

The system date can be retrieved via `cl_abap_context_info=>get_system_date()`.

ABAP is able to calculate with date types as long as the date context is clear. The actual calculation is therefore quite easy: We just subtract the today's date from the flight date. The returning parameter can then be filled with the calculated days to flight.

```
METHOD if_sadl_exit_calc_element_read~calculate.
  DATA(lv_today) = cl_abap_context_info=>get_system_date( ).
  DATA lt_original_data TYPE STANDARD TABLE OF /dmo/Č_booking_proc_ve
  WITH DEFAULT KEY.
  lt_original_data = CORRESPONDING #( it_original_data ).
  LOOP AT lt_original_data ASSIGNING FIELD-SYMBOL(<fs_original_data>).
```

```
<fs_original_data>-DaysToFlight = <fs_original_data>-FlightDate -  
lv_today.  
ENDLOOP.  
ct_calculated_data = CORRESPONDING #( lt_original_data ).  
ENDMETHOD.
```

Related Information

[Interface IF_SADL_EXIT_CALC_ELEMENT_READ \[page 756\]](#)

7.5 Using Aggregate Data in SAP Fiori Apps

This topic explains how you can provide aggregate data for your SAP Fiori application. The available aggregate functions operations are sum, minimum, maximum, and average. Alongside this, the framework also provides options for counting.

What is Aggregate Data?

We speak of aggregate data when numerical values are combined to form a single value that signifies meaning. General assumptions can be drawn from this value that is representative for all values that were included in the calculation of the value.

The classic aggregate functions are:

- sum
- minimum
- maximum
- average

These functions determine a value from which you can assume information relating to all the values that were included in the calculation.

Apart from these functions, there is also a counting function available to count distinct values:

- distinct count.

This counting option determines a natural number based on the number of entries in the calculation.

All of these functions are supported and evaluated by the SADL framework.

Aggregate Data in your SAP Fiori App

Aggregate data calculated by the SSDL framework provides additional and enhanced information for your list reporting app.

To display aggregate data in your application, annotate the respective elements in CDS with the annotations described in [Annotating Aggregate Functions in CDS \[page 447\]](#). These annotations cause the CDS entity to be respected as an aggregated entity by OData. A thorough description of how OData interprets the annotations is provided in [OData Interpretation of Aggregation Annotations \[page 451\]](#).

i Note

Aggregate functions are not supported in scenarios with projection CDS views. Aggregation annotations in projection view cause an error on service compilation. Aggregation annotations in projected entities are ignored.

Based on the entity that calculates aggregate data, the SAP Fiori user interface displays aggregate data depending on the settings you choose. The aggregated values are displayed in your list reporting app as an additional field in the relevant column.

Sales Order	Item	Company	Product	Category	Gross Amount	Gross Amount in EUR	Net Amount	Tax Amount	Number of Prod...	Distinct Products
500007257	50	Panorama Studios	HT-1036	Flat Screen Monitors	1,023.40 CAD	725.82 EUR	860.00 CAD	163.40 CAD	1	1
500007257	60	Panorama Studios	HT-1037	Flat Screen Monitors	1,463.70 MXN	159.44 EUR	1,230.00 MXN	233.70 MXN	1	1
500007257	70	Panorama Studios	HT-1000	Notebooks	3,412.92 EUR	3,412.92 EUR	2,868.00 EUR	544.92 EUR	1	1
500007257	80	Panorama Studios	HT-1001	Notebooks	1,486.42 EUR	1,486.42 EUR	1,249.09 EUR	237.33 EUR	1	1
Aggregate Data					Show Details	9,367.78 EUR	Show Details	Show Details	8	7

List Reporting App of Sales Order Items with Aggregate Data

7.5.1 Annotating Aggregate Functions in CDS

The elements for which you want to display aggregate data in your SAP Fiori App must be annotated in the CDS entity with the relevant annotation for the aggregate function.

Metadata for Aggregations

The annotation `@Aggregation.Default: #<AGGR_FUNCTION>` enables the aggregation of the values of the annotated element.

Only measures can be annotated with an aggregation annotation. Measures are elements that either represent numerical values, which means they can be summed, averaged, or otherwise mathematically manipulated. Typically, measures are units that express the size, amount, or degree of something, for example prices. In addition, elements with date values, can also be compared with each other to determine the maximum or minimum. That means date values can also be measures for calculating the minimum or the maximum.

The other elements in a CDS entity are called dimensions. Dimensions provide structured labeling information about otherwise unordered numeric measures. They are relevant for the grouping and the order of the elements in the Fiori App.

The SADL framework supports the following aggregating functions for measures. As soon as one element is annotated with one of these annotations, the CDS entity is considered an analytical entity.

i Note

Only elements with numerical data types can be annotated with @Aggregation annotations.

Annotation and Value	Effect
@Aggregation.Default: #SUM	Calculates the sum of the values of the annotated element.
@Aggregation.Default: #MAX	Calculates the maximum of the values of the annotated element. You can also annotate elements with date types with this annotation.
@Aggregation.Default: #MIN	Calculates the minimum of the values of the annotated element. You can also annotate elements with date types with this annotation.
@Aggregation.Default: #AVG	Calculates the average of the values of the annotated element.
@Aggregation.Default: #COUNT_DISTINCT	Counts the number of distinct values of the annotated element. In combination with the annotation @Aggregation.ReferenceElement: ['elementRef'], you can count the number of distinct values of another element that is referenced.

i Note

Since the distinct count of the referenced element is naturally 1 for single data records, the value of the annotated element is 1 regardless of the actual value of the element. In grouped records the annotated element counts the distinct values of the referenced element. So the actual value of the annotated element is ignored.

For this reason, it is recommended that a new element for the counting of elements be created. Make sure that you use an adequate numerical type for this element.

❖ Example

```
@Aggregation.referenceElement:  
['CustomerID']  
@Aggregation.default: #COUNT_DISTINCT  
cast( 1 as abap.int4 ) as  
DistinctCustomers,
```

This example counts the number of distinct customers into the element DistinctCustomers.

Annotation and Value	Effect
@Aggregation.ReferenceElement: ['elementRef']	References the element that is used for distinct counts.

i Note

Can only be used in combination with
 @Aggregation.Default: #COUNT_DISTINCT.

Only one aggregate function can be used on one element. You cannot display different aggregated values of the same element.

Aggregations in Fiori Elements UIs

In a Fiori Elements UI, records are always grouped by the dimensions that are selected by \$select in an OData request. The elements for \$select in OData requests are the column elements that are selected in the Fiori UI.

If the key elements are selected in the Fiori UI, the records are grouped by these key elements, which means there is no grouping, as there are no records that can be grouped by the same key element. If not all key elements are requested, the records are grouped by the dimensions that are requested and the selected measures are aggregated according to the groups.

If you want to suppress the behavior, that the records are grouped automatically if not all key elements are selected, you can use a @UI annotation that triggers the key elements to be always selected implicitly.

Annotation and Value	Effect
@UI.presentationVariant: [{requestAtLeast: ['<TECHNICAL_KEY>']]}	Defines the properties that must always be included in the result of the queried collection if no grouping is desired.

The UI offers the option to group explicitly. On the UI, go to settings and define the dimensions to group by or click on a dimension column and choose *Group*.

For more information on grouping, see [OData Interpretation of Aggregation Annotations \[page 451\]](#).

Example

The following listing displays a CDS entity in which all the necessary annotations for analytical operations are used. This view describes sales order combined with associated product and customer information.

```
define view ZDEMO_C_SOI_ANLY
  as select from <data source> as Item
    association [0..1] to SEPM_I_Product_E as _Product      on $projection.Product =
      -Product.Product
    - association [0..1] to ZDEMO_C_SO_ANLY as _SalesOrder on
      $projection.SalesOrderID = _SalesOrder.SalesOrder
```

```

{
    key SalesOrder                               as
    SalesOrderID,
    key SalesOrderItem                          as
    ItemPosition,
    Item._SalesOrder.Customer                 as
    CustomerID,
    Item._SalesOrder._Customer.CompanyName   as
    CompanyName,
    @ObjectModel.foreignKey.association: '_Product'
    Product                                     as Product,
    Product.ProductCategory                   as
    ProductCategory,
    @Semantics.currencyCode: true
    TransactionCurrency                      as
    CurrencyCode,
    @Semantics.currencyCode: true
    cast( 'EUR' as abap.cuky )                as
    TargetCurrency,
    @Aggregation.Default: #SUM
    @Semantics.amount.currencyCode: 'TargetCurrency'
    CURRENCY_CONVERSION(
        amount           => Item.GrossAmountInTransacCurrency,
        source_currency  => Item.TransactionCurrency,
        target_currency   => cast( 'EUR' as abap.cuky ),
        exchange_rate_date => cast( '20180315' as abap.dats ),
        error_handling     => 'SET_TO_NULL' )            as
    ConvertedGrossAmount,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    @Aggregation.Default: #AVG
    GrossAmountInTransacCurrency             as
    GrossAmount,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    @Aggregation.Default: #MIN
    NetAmountInTransactionCurrency          as NetAmount,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    @Aggregation.Default: #MAX
    TaxAmountInTransactionCurrency          as TaxAmount,
    @Aggregation.referenceElement: ['Product']
    @Aggregation.Default: #COUNT_DISTINCT
    cast( 15 as abap.int4 )                  as
    DistinctProducts,
    _SalesOrder,
    _Product
}

```

i Note

Aggregate functions only respect values with the same semantics. This means that, if you have prices in different currencies that are annotated with an aggregation annotation, you receive aggregated data for each currency.

The following figure displays the maximum tax amount with regard to the respective currency.

Sales Order	Item	Company	Product	Category	Gross Amount	Gross Amount in EUR	Net Amount
500007270	40	SAP	HT-1003	Notebooks	3,927.00 EUR	3,927.00 EUR	3,300.00 EUR
500007270	50	SAP	HT-1007	PDAs & Organizers	1,067.43 USD	1,135.56 EUR	897.00 USD
500007270	60	SAP	HT-1010	Notebooks	4,757.62 EUR	4,757.62 EUR	3,998.00 EUR
500007270	70	SAP	HT-1011	Notebooks	547.162 JPY	5,040.18 EUR	459,800 JPY
					Show Details	26,339.99 EUR	Show Details

Aggregate Data for Distinct Currencies

Related Information

[Using Aggregate Data in SAP Fiori Apps \[page 446\]](#)

[OData Interpretation of Aggregation Annotations \[page 451\]](#)

7.5.2 OData Interpretation of Aggregation Annotations

The following sections provide an overview of the most prominent features of aggregated entities in OData.

Data models with aggregation annotations are considered as aggregated entities by OData. The behavior of these aggregated OData entities differs from non-aggregated entities.

OData Annotations

The OData entity is given multiple annotations based on the aggregate annotation you use in CDS for your data model. The following figure displays the metadata of an aggregated entity that processes sales order items.

The annotations specific to aggregations are highlighted and labeled. Further descriptions of the annotations in OData are given below.

```

    Aggregated OData Entity
    +-- <EntityType sap:content-version="1" sap:label="Sales Order Items" sap:semantics="aggregate" Name="ZDEMO_C_SOI_ANLYType">
        -<Key>
            <PropertyRef Name="ID"/>
        Generated<Key>
            ID <Property Name="ID" sap:filterable="false" sap:sortable="false" Nullable="false" Type="Edm.String"/>
            <Property sap:label="Sales Order ID" Name="SalesOrderID" Type="Edm.String" sap:quickinfo="EPM: Sales Order Number" sap:display-format="UpperCase" sap:aggregation-role="dimension" MaxLength="10" sap:updatable="false" sap:createable="false" sap:value-list="standard"/>
            <Property sap:label="Item Position" Name="ItemPosition" Type="Edm.String" sap:quickinfo="EPM: Sales Order Item Position" sap:display-format="UpperCase" sap:aggregation-role="dimension" MaxLength="10"/>
            Dimension <Property sap:label="Customer" Name="CustomerID" Type="Edm.String" sap:quickinfo="EPM: Customer ID" sap:display-format="UpperCase" sap:aggregation-role="dimension" MaxLength="10"/>
            <Property sap:label="Company Name" Name="CompanyName" Type="Edm.String" sap:quickinfo="EPM: Company Name" MaxLength="80" sap:attribute-for="CustomerID"/>
            <Property sap:label="Product ID" Name="Product" Type="Edm.String" sap:quickinfo="EPM: Product ID" sap:text="to_Product/Product_Text" sap:display-format="UpperCase" sap:aggregation-role="dimension" MaxLength="10" sap:value-list="standard"/>
            Attribute <Property sap:label="Product Category" Name="ProductCategory" Type="Edm.String" sap:quickinfo="EPM: Product Category" sap:attribute-for="Product" MaxLength="10"/>
            <Property sap:label="ISO Currency Code" sap:semantics="currency-code" Name="CurrencyCode" Type="Edm.String" sap:quickinfo="EPM: Currency Code" sap:aggregation-role="dimension" MaxLength="5"/>
            <Property sap:label="ConvertedGrossAmount" sap:filterable="false" Type="Edm.Decimal" sap:aggregation-role="measure" Precision="16" sap:unit="TargetCurrency" Scale="3"/>
            <Property sap:label="Total Gross Amount" Name="GrossAmount" sap:filterable="false" Type="Edm.Decimal" sap:quickinfo="EPM: Total Gross Amount" sap:aggregation-role="measure" Precision="16" sap:unit="CurrencyCode" Scale="3"/>
            Measure <Property sap:label="Total Net Amount" Name="NetAmount" sap:filterable="false" Type="Edm.Decimal" sap:quickinfo="EPM: Total Net Amount" sap:aggregation-role="measure" Precision="16" sap:unit="CurrencyCode" Scale="3"/>
            <Property sap:label="Total Tax Amount" Name="TaxAmount" sap:filterable="false" Type="Edm.Decimal" sap:quickinfo="EPM: Total Tax Amount" sap:aggregation-role="measure" Precision="16" sap:unit="CurrencyCode" Scale="3"/>
            <Property sap:label="Count" Name="AllItems" sap:filterable="false" Type="Edm.Int32" sap:aggregation-role="measure"/>
            <Property Name="DistinctProducts" sap:filterable="false" Type="Edm.Int32" sap:aggregation-role="measure"/>
            <NavigationProperty Name="to_Product" ToRole="ToRole_assoc_AAF8BC2D7DDDE36AF83ED42DD9AA9D33">
                FromRole="FromRole_assoc_AAF8BC2D7DDDE36AF83ED42DD9AA9D33"
                Relationship="SERVICE_NAME.assoc_AAF8BC2D7DDDE36AF83ED42DD9AA9D33"
            </NavigationProperty>
            <NavigationProperty Name="to_SalesOrder" ToRole="ToRole_assoc_FCD485C2E8FB98D874E8452CC7AB576A">
                FromRole="FromRole_assoc_FCD485C2E8FB98D874E8452CC7AB576A"
                Relationship="SERVICE_NAME.assoc_FCD485C2E8FB98D874E8452CC7AB576A"
            </NavigationProperty>
        </EntityType>

```

Metadata of an Aggregated Entity

Aggregated OData Entities

As soon as one element in the CDS view is annotated with the aggregation annotation

`@Aggregation.Default: #<AGGR_FUNCTION>`, the OData entity is annotated with `sap:semantics="aggregate"`. Hence, the OData entity is identified as an aggregated entity.

In the example of the screenshot above, this OData annotation can be found in the first line of the extract of the metadata.

Measures

The aggregated entity is characterized by measures and dimensions. Measures are those properties that are annotated with the annotation relevant for aggregating data in CDS. Measures are given the OData annotation `sap:aggregation-role="measure"`.

In the example of the screenshot above, there are six properties which are marked as measures: ConvertedGrossAmount, GrossAmount, NetAmount, TaxAmount, AllItems, and DistinctProducts.

Dimensions

Dimensions are all properties that are neither marked as measures nor as attributes. Dimensions are given the OData annotation `sap:aggregation-role="dimension"`.

In the example of the screenshot above, there are six OData properties which are marked as dimensions: SalesOrderID, ItemPosition, Customer ID, Product, CurrencyCode, and TargetCurrency.

Each dimension can have a maximum of one text property. A text property is an element that is defined as a text element in CDS, as described in [Defining Text Elements \[page 422\]](#). Dimensions with a text property are annotated by OData with `sap:text="<TEXT_PROPERTY>"`.

In the example of the screenshot above, the dimension Product has a text property.

Generated ID for Aggregated OData Entities

An aggregated OData entity is given an additional property for a generated ID (<Property Name="ID"/>). The generated ID for the aggregate entity uniquely identifies each record so that every entity request for a given ID always returns the same values.

In the example of the screenshot above, the property for the generated ID can be found as the first property on the list.

The ID is also generated for every group record when it is requested for the first time. Following from this, you can use this ID in a further request.

Example

This behavior is exemplified by a request on the entity that supplies the metadata above. It retrieves sales order items. The following request selects the generated ID (`ID`), the dimension `Product`, and some aggregated measures related to the selected dimension.

```
....sap/opu/odata/SAP/<service_name>/ZDEMO_C_SOI_ANLY?  
$select=ID,Product,GrossAmount,NetAmount,TaxAmount,AllProducts
```

This request retrieves the following result:

```
<?xml version="1.0"?>  
<feed xml:base="https://HOST/sap/opu/odata/SAP/SERVICE_NAME/" 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>https://HOST/sap/opu/odata/SAP/SERVICE_NAME/ZDEMO_C_SOI_ANLY</id>  
  <title type="text">ZDEMO_C_SOI_ANLY</title>  
  <updated>2018-04-03T13:27:26Z</updated>  
  <author>  
    <name/>  
  </author>  
  <link title="ZDEMO_C_SOI_ANLY" rel="self" href="ZDEMO_C_SOI_ANLY'<4~HT%26c1002.6~USD'">  
  <entry>  
    <id>https://HOST/sap/opu/odata/SAP/SERVICE_NAME/ZDEMO_C_SOI_ANLY('4~HT%26c1002.6~USD')</id>  
    <title type="text">ZDEMO_C_SOI_ANLY('4~HT%26c1002.6~USD')</title>  
    <updated>2018-04-03T13:27:26Z</updated>  
    <category scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" term="SERVICE_NAME.ZDEMO_C_SOI_ANLYType"/>  
    <link title="ZDEMO_C_SOI_ANLY" type="application/xml" rel="self" href="ZDEMO_C_SOI_ANLY('4~HT%26c1002.6~USD')?>  
    <content type="application/xml">  
      <m:properties xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:ns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">  
        <d:ID value="4~HT%26c1002.6~USD" />  
        <d:Product value="HT%1002" />  
        <d:GrossAmount value="3529.01" />  
        <d:NetAmount value="1570.00" />  
        <d:TaxAmount value="596.60" />  
        <d:AllProducts value="9" />  
      </m:properties>  
    </content>  
  </entry>  
  <entry>  
    <id>https://HOST/sap/opu/odata/SAP/SERVICE_NAME/ZDEMO_C_SOI_ANLY('4~HT%26c1007.6~USD')</id>  
    <title type="text">ZDEMO_C_SOI_ANLY('4~HT%26c1007.6~USD')</title>  
    <updated>2018-04-03T13:27:26Z</updated>  
    <category scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" term="SERVICE_NAME.ZDEMO_C_SOI_ANLYType"/>  
    <link title="ZDEMO_C_SOI_ANLY" type="application/xml" rel="self" href="ZDEMO_C_SOI_ANLY('4~HT%26c1007.6~USD')?>  
    <content type="application/xml">  
      <m:properties xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:ns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">  
        <d:ID value="4~HT%26c1007.6~USD" />  
        <d:Product value="HT%1007" />  
        <d:GrossAmount value="934.00" />  
        <d:NetAmount value="598.00" />  
        <d:TaxAmount value="170.42" />  
        <d:AllProducts value="8" />  
      </m:properties>  
    </content>  
  </entry>  
</feed>
```

Aggregate Data of Group Record with Generated ID for Group Record

Each group record is given its own generated ID which retrieves the same results when requested again. Based on this group ID, you can also execute other requests, as in `/sap/opu/odata/SAP/<service_name>/ZDEMO_C_SOI_ANLY ('.4~HT%26c1002.6~USD')? $select=AllProducts`,

which will only retrieve the count of this group, which is 9.

Requesting Data from an Aggregated Entity

Results of requesting data from aggregated entities depend on the elements you select in your OData request. Grouping and aggregation are both driven by the elements you request with the parameter `$select` in entity

set queries. The result of a query consists of aggregated entities with distinct values for the requested dimension properties and requested measures are aggregated using the aggregate function with which the measure elements are annotated in CDS.

If an attribute is requested, the result is grouped by its corresponding dimension and within that group it is grouped by the attribute itself.

i Note

If you use a SAP Fiori app, the `$select` statement of the OData request directly depends on the columns you select in the list reporting app. The following descriptions of requesting data with OData can also be managed by selecting the respective columns in your Fiori App.

Requests that Contain All Primary Key Elements

If all primary key elements of the requested CDS entity are included in the query option `$select`, no grouping is possible since every record is unique. Hence, an OData request with all key elements behaves just like a non-aggregated entity. Consequently, no aggregate data is calculated.

Requests that Do not Contain All Primary Key Elements

If not all key elements are included in the OData request, the requested dimensions are grouped and the requested measures are aggregated according to their annotated aggregate function. For every distinct combination of dimension values (after evaluating `$filter`), the result includes an aggregated entity with aggregated values for the requested measures.

i Note

Each group record is given its own generated ID, which can be reused in requests to retrieve the same results.

Only if no dimension and no attribute are requested does the result show the aggregate data of measures for the whole entity set.

You can still execute other query options, such as `$filter` or `$orderby` on grouped requests. The filtering is executed before the grouping, so that the groups are created from the available records after the filtering. The ordering is executed after the grouping, which means the records within a group are ordered according to the query option.

i Note

You can filter for properties that you do not select, but you cannot order by properties that are not included in the `$select` due to the order of executing query options mentioned above.

Navigating and Expanding from a Group Record

You can navigate directly from a group record to one of the properties that are included in the group, for example properties that you have selected previously.

• Example

Assume you have an aggregated entity of sales order items and you group them by product. From this group record, you can navigate directly to the associated entity of products if the association exists in CDS.

The same holds true for the query option `$expand`. Only properties that are selected can be expanded.

i Note

The target entity of the navigation or the expand is also given an aggregated ID if the underlying data model is also aggregated.

Related Information

[Using Aggregate Data in SAP Fiori Apps \[page 446\]](#)

[Annotating Aggregate Functions in CDS \[page 447\]](#)

7.6 Automatically Drawing Primary Key Values in Managed BOs

In **managed** scenarios with UUID keys, the values for primary key fields can be automatically generated by the managed runtime framework.

Context

Values for primary key fields can be generated automatically for managed business objects. This so-called [Managed Early Numbering \[page 63\]](#) is defined in the behavior definition of the business object. During the CREATE operation, the managed runtime framework draws the UUID automatically for the defined key fields. The newly created BO instance is immediately uniquely identifiable after the CREATE operation and saved with the given key during the save sequence.

For more information, see [Numbering \[page 61\]](#).

Prerequisites

- The primary key field has the ABAP type `raw(16)` (UUID).
- The implementation type of the RAP BO is `managed`.

Syntax

You define managed early numbering for primary key fields in the behavior definition by using the following syntax:

```
[implementation] managed [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
  lock (master|dependent() )
  ...
{
  ...
  field ( [read only,] numbering:managed ) KeyField1, [KeyField2];
  ...
}
```

The key field must be read-only if you intend the key fields to be filled internally only. If you do not set the key field to read-only, the key value can also be given by the consumer. However, use cases for this [optional external numbering \[page 63\]](#) are rare, as it is untypical for the consumer to fill in a UUID value.

Example

The following simplified BO exemplifies the use case to have the framework generate a UUID key for a managed BO.

1. Database table /dmo/travel_uuid

The basis for the data model is a database table similar to /DMO/travel (that is included in the [ABAP Flight Reference Scenario \[page 774\]](#)). To demonstrate managed numbering, we need a key field with UUID type (raw(16)).

```
@EndUserText.label : 'Flight Reference Scenario: Managing Travels'
@AbapCatalog.enhancementCategory : #NOT_EXTENSIBLE
@AbapCatalog.tableCategory : #TRANSPARENT
@AbapCatalog.deliveryClass : #A
@AbapCatalog.dataMaintenance : #LIMITED
define table /dmo/travel_uuid {
  key client      : abap.clnt not null;
  key travel_uuid : sysuuid_x16 not null;
  travel_id       : /dmo/travel_id;
  agency_id       : /dmo/agency_id;
  customer_id    : /dmo/customer_id;
  begin_date     : /dmo/begin_date;
  end_date       : /dmo/end_date;
  @Semantics.amount.currencyCode : '/dmo/travel_uuid.currency_code'
  booking_fee    : /dmo/booking_fee;
  @Semantics.amount.currencyCode : '/dmo/travel_uuid.currency_code'
  total_price    : /dmo/total_price;
  currency_code  : /dmo/currency_code;
  description    : /dmo/description;
  overall_status : /dmo/overall_status;
  created_by     : syuname;
  created_at     : timestampl;
  last_changed_by : syuname;
  last_changed_at : timestampl;
}
```

2. CDS root view /DMO/I_TRAVEL_UUID

The data model for the simplified travel scenario is defined in a CDS view that is similar to /DMO/I_TRAVEL_M. The difference in the UUID scenario is that the UUID key field is obligatory. In addition to the

UUID key field, the travel CDS view also defines the non-key field `travel_id`, which assigns an additional semantic identifier. This semantic is not a key element in the scenario. It simply serves as an additional, more simple, identifier that is easier to read by consumers. The value for this field is defined via a determination in the behavior pool.

```

@AbapCatalog.sqlViewName: '/dmo/itraveluuid'
@AbapCatalog.compiler.compareFilter: true
@AbapCatalog.preserveKey: true
@AccessControl.authorizationCheck: #NOT_REQUIRED
@EndUserText.label: 'Root View with Travel UUID'
define view /dmo/i_travel_uuid as select from /dmo/travel_uuid
{ ///dmo/travel_uuid
  key travel_uuid,
  travel_id,
  agency_id,
  customer_id,
  begin_date,
  end_date,
  @Semantics.amount.currencyCode: 'currency_code'
  booking_fee,
  @Semantics.amount.currencyCode: 'currency_code'
  total_price,
  @semantics.currencyCode: true
  currency_code,
  description,
  overall_status,

  @Semantics.user.createdBy: true
  created_by,
  @Semantics.systemDateTime.createdAt: true
  created_at,
  @Semantics.user.lastChangedBy: true
  last_changed_by,
  @Semantics.systemDateTime.lastChangedAt: true
  last_changed_at
}

```

3. Behavior Definition /DMO/I_TRAVEL_UUID

For the root CDS view `/DMO/I_TRAVEL_UUID`, the following behavior definition defines managed numbering for the UUID key field `travel_uuid`.

The key UUID field is defined as `read only`. Not setting the key field to `read only` causes [optional external numbering \[page 63\]](#), in which it is also possible for the consumer to set the value manually. In this scenario, we want to set the value for the additional identifier `travel_id` by a determination. Because of that, this field is also set to `read-only`.

```

managed;
define behavior for /dmo/i_travel_uuid alias Travel
implementation in class /dmo/bp_i_travel_uuid unique
persistent table /DMO/TRAVEL_UUID
lock master
etag master last_changed_at
{
  field ( read only, numbering : managed ) travel_uuid;
  field ( read only ) travel_id;
  create;
  update;
  delete;
  /** optional determination to determine the travel ID
  determination CreateKeys on modify { create; }
}

```

4. Optional: Determination to determine the value for semantic travel identifier

The value for the `travel_id` field is assigned via a determination. In this way, the consumer does not have to enter the semantic identifier manually. It is assigned on `CREATE`.

The determination is defined in the behavior definition and implemented in the behavior pool. For more information, see [Developing Determinations \[page 206\]](#).

For the implementation, get a new travel id and execute an EML `modify` on the newly created instance to assign the new travel ID to the instance. To make sure that the determination result does not change in case it is executed twice under the same circumstances, we have to make sure that a new travel id is only assigned if the `travel_id` field is initial.

i Note

To ensure that one travel ID is only assigned once, you can use a number range object.

The following code example does not show how to get this new travel ID. It is just suggested by the method call of `get_number`.

```
CLASS lhc_Travel_UUID DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
    METHODS CreateKeys FOR DETERMINATION Travel~CreateKeys  
        IMPORTING keys FOR Travel.  
ENDCLASS.  
CLASS lhc_Travel IMPLEMENTATION.  
    METHOD CreateKeys.  
        " Read entity to check if travel_id is initial  
        READ ENTITIES OF zmv_i_travel_uuid IN LOCAL MODE  
        ENTITY travel  
            FIELDS ( travel_id )  
            WITH CORRESPONDING #( keys )  
        RESULT DATA(lt_travel_result).  
  
        " only modify entities where no travel_id is given  
        DELETE lt_travel_result WHERE travel_id IS NOT INITIAL.  
        LOOP AT lt_travel_result ASSIGNING FIELD-SYMBOL(<fs_travel>).  
            " get a new travel ID  
            DATA(lv_travel_id_new) = get_number( ).  
            <fs_travel>-travel_id = lv_travel_id_new.  
        ENDLOOP.  
        MODIFY ENTITIES OF zmv_i_travel_uuid IN LOCAL MODE  
        ENTITY travel  
            UPDATE FROM VALUE #( FOR travel IN lt_travel_result  
                ( travel_uuid = travel->key-travel_uuid  
                  travel_id = travel-travel_id  
                  %control-travel_id = if_abap_behv=>mk-on ) ).  
    ENDMETHOD.  
ENDCLASS.
```

5. UI Preview

The following figure shows the creation of a new travel instance when using managed numbering for the UUID key field and a determination to assign the semantic identifier on `CREATE`.

For this example to run like it is shown below, you must add UI annotations to the elements in the CDS view.

i Note

This example scenario to automatically draw UUID values for primary key fields is highly simplified. Basic important elements of a travel scenario, such as eTag or BO projection, are left aside to focus on the principle of managed early numbering.

Managed Early Numbering in UUID Scenario

Managed Numbering in UUID Scenario

Related Information

[Numbering \[page 61\]](#)

7.7 Adding Feature Control

This topic treats the concept of feature control in the context of ABAP RESTful application development for transactional use cases.

Feature Control in a Nutshell

Feature control is a means that you can provide as information to the service on how data has to be displayed for consumption in a SAP Fiori UI.

In general, the following characteristics are relevant when providing feature control:

- **Read-only** - Will the user be allowed to change the value of a field?
- **Mandatory** - Must the user provide a value?
- **Disabled/Enabled** - Will the user be able to access an operation?
- **Hidden** - Should a field be available for consumption in the UI?

→ Remember

We do not cover this characteristics in this topic since the relevant information is provided by the UI annotation `@UI.hidden: true`. You can use this annotation to prevent fields from being displayed on a UI and in the personalization dialog but leaving the annotated fields available for the consumer. **More on this:** [Field Hiding \[page 554\]](#)

The feature control information is either static or dynamic:

- **Static** - The field (and action) control information is valid for all instances of a business object node, regardless of their state. For example, consider a field `PAID` in an invoice scenario that is always read-only since it can only be changed by an action, not directly by the consumer after an update procedure.
More on this: [Static Feature Control \[page 460\]](#)
- **Dynamic** - The field (and action) control information depends on the state of the node instances. For example, the field `COMMENTS` should be read-only if the invoice's `PAID` property is set to `true`; otherwise the field can be edited by the user.
More on this: [Dynamic Feature Control \[page 463\]](#)

7.7.1 Static Feature Control

Here we explain how you can implement static feature control in the behavior model of a business object.

Overview

In a typical transactional scenario, you have to specify which operations should be provided by the whole entity (entity scope) or you must specify which fields of an entity have specific access restrictions (field scope).

The feature control can relate to a whole entity (business object node instance) or to individual fields (attributes of a business object's node):

- The transactional character of a business object is defined in the behavior definition where all supported transactional operations are specified for each node of the business object's composition tree. Whenever the corresponding root or child entity is going to be created, updated, or deleted, these operations must be declared in the behavior definition. In this way, you specify at the business object node level whether each node instance is enabled for creation, update, or deletion.
- By using field properties in the behavior definition, you can also specify which fields of an entity have specific access restrictions. In this way, you specify the field control at the attribute level of the corresponding business object node.

Defining Static Feature Control at Entity Level

Within the behavior definition, you can specify the following operations for static entity control:

Operation	Effect
create	<p>Specifies that new instances of a business object node that correspond to the underlying (root or child) entity can be created.</p> <p>If this operation is not declared for an entity in the behavior definition, creation of new instances of the corresponding business object nodes is not allowed.</p>
update	<p>Specifies that data of existing instances of a business object node that corresponds to the underlying (root or child) entity can be updated.</p> <p>If this operation is not declared for the entity in the behavior definition, updating existing instances is not allowed.</p>
delete	<p>Specifies that existing instances of a business object node that corresponds to the underlying (root or child) entity can be deleted.</p> <p>If this operation is not declared for an entity in the behavior definition, deletion of existing instances is allowed.</p>

Example

The following behavior definition specifies that `root` instances of a business object node can be updated, but not created or deleted whereas the `subnode` instances can be created, updated and deleted.

Use cases for the only need to update BO instance data could occur, for example, if the instances only need to be archived or if they are to be created and deleted only by means of an action.

```
implementation ...;
define behavior for BO_Root alias root
{
  update;
}
define behavior for BO_Subnode alias subnode
{
  create;
  update;
  delete;
}
```

Defining Static Feature Control at Field Level

Within the behavior definition, you can specify individual fields of an entity that have certain access restrictions.

Syntax

```
implementation {unmanaged | managed | abstract};
```

```

define behavior for Entity [alias AliasName]
...
{
  /* Static field control */
  field (read only | mandatory) field1[, field2, ..., fieldn];
...
}

```

You can use the following field properties to define static field control:

Field Properties

Property	Effect
field (read only)	<p>Restricts the specified fields to non-editable fields</p> <p>This property defines that values of the specified fields must not be created or updated by the consumer. The BO runtime ignores the values for the specified fields in modifying requests when creating or updating the specified fields.</p>
field (mandatory)	<p>Defines that the specified fields are mandatory</p> <p>The specified fields must be filled by the consumer when executing modifying requests. For the relevant fields, the value must be provided in create operations. In update operations, it must not be given the null value.</p>

⚠ Caution

Note that there is no implicit validation by the business object framework. As an application developer, you must ensure that the [validation \[page 822\]](#) is implemented by your application.

Example

For the business object from our flight demo application, the static field control is used to restrict properties of particular fields. Here, the key field `TravelID` cannot be created or updated by the consumer. The fields `AgencyID`, `CustomerID`, `BeginDate`, and `EndDate` are mandatory, that is, the fields must contain a value in modifying requests.

```

implementation ...;
define behavior for /DMO/I_Travel_U alias travel
...
{
  field (read only) TravelID;
  field (mandatory) AgencyID, CustomerID, BeginDate, EndDate;
  create;
  update;
...
}

```

The screenshot shows a form interface for a travel booking. It includes the following fields:

- Travel ID:** 11 — **read only field**
- *Agency ID:** 70005 — **mandatory field**
- *Customer ID:** 582
- *Starting Date:** Aug 18, 2018, 2:00:00 AM
- *End Date:** Aug 18, 2018, 2:00:00 AM
- Booking Fee:** 20.00
- Total Price:** 13,575.26
- Comment:** (empty)

Read-only and mandatory fields on SAP Fiori UI

Related Information

[Defining Static and Dynamic Feature Control \[page 466\]](#)

7.7.2 Dynamic Feature Control

Dynamic Field and Entity Control

Dynamic feature control allows you to handle the changeability of fields or the entire business object entities depending on a feature condition (for example: a value of a field). In the context of the ABAP RESTful programming model, the field and entity control serve as a way to provide information to the (OData) service in a transactional scenario about how data has to be displayed for consumption, for example, on the SAP Fiori UI. The field and entity control can in this context relate to the individual fields of an entity or an entire entity of the business object.

Example

The following app manages invoices. Each invoice has three properties which are the invoice ID, the payment status, and the comments provided. In our example, the fields *Invoice ID* and *Comments* should be read-only if the invoice's *Paid* attribute is set to `true`.

General Information

*Invoice ID:
1,000,066

Paid:

Comments:
SOME COMMENTS

Save **Cancel**

The comments field is editable for invoices that are not paid

General Information

Invoice ID:
1,000,077

Paid:

Comments:
EDIT SOME COMMENTS HERE! ← read-only

Save **Cancel**

The ID and the comments fields are read-only for a paid invoice

Activities Relevant to Developers

The following steps are then required to provide dynamic field (or entity) control:

1. Modeling Dynamic Field and Entity Control in the Behavior Definition
2. Implementing Dynamic Feature Control in a Handler Class

Dynamic Action Control

Specific operations of an entity of a business object can be defined using actions. Similar to standard operations (create, update, delete), you can define actions in the behavior definition. In many cases actions are to be enabled or disabled depending on the state of an entity. Example: If an invoice has already been paid then action *Set to paid* has to be disabled for that invoice.

For dynamic control of actions acting on individual entity instances, the option (features: instance) must be added to the relevant action in the behavior definition. The required implementation must be provided in the referenced behavior pool. For each relevant action, you can specify in the implementing handler of the class pool the following values:

- ENABLED - if the action is enabled
- DISABLED - if the action is disabled.

Example

The following example app is used to manage invoices. Users can change the payment status for individual invoices without having to switch to edit mode. The action *Set to PAID* is disabled for all paid invoices, since the action control information is related to the property *Paid*.

Invoice ID	Paid	Comments	
1,000,066	No	SOME COMMENTS	>
1,000,077	Yes	EDIT SOME COMMENTS HERE!	>
1,000,088	Yes	COMMENT TEXT AND MORE TEXT	>

Action enabled

Invoice ID	Paid	Comments	
1,000,066	No	SOME COMMENTS	>
1,000,077	Yes	EDIT SOME COMMENTS HERE!	>
1,000,088	Yes	COMMENT TEXT AND MORE TEXT	>

Action disabled

Activities Relevant to Developers

The following steps are required to provide dynamic action control:

1. Modeling Dynamic Action Control in the Behavior Definition
2. Implementing Dynamic Action Control in a Handler Class

Related Information

[Defining Static and Dynamic Feature Control \[page 466\]](#)

7.7.3 Defining Static and Dynamic Feature Control

Both, static and dynamic feature control is defined for different levels (entity, field, or action level) in the behavior definition by using the following syntax:

Syntax: Feature Control in the Behavior Definition

```
[implementation] (managed | unmanaged);
define behavior for CDSEntity [alias AliasName]
implementation in class ABAP_CLASS [unique]
...
{
    /* (1) Feature control at entity level */
    /* Static operation control */
    internal create
    internal update
    internal delete
    /* or (instance-based) dynamic operation control for update, delete and create
       by association: implementation required! */
    update (features: instance);
    delete (features: instance);
     _association { create (features: instance); }
    /* (2) Feature control at field level */
    /* Static field control */
    field (read only | mandatory) f1[, f2, ..., fn];
    /* or dynamic field control: implementation required! */
    field (features: instance) f1[, f2, ..., fn];
    /* (3) Feature control for actions */
    /* Static action control */
    internal action ActionName [...]
    /* or dynamic action control: implementation required! */
    action ( features: instance ) ActionName [... ]
}
```

(1) Feature Control at Entity Level

To manage a transactional behavior, an entity of a business object offers standard operations create, update, and delete for external consumption using EML or OData services. To only provide these operations without exposing them to consumers, the option `internal` can be set before the operation name. An **internal operation** can only be accessed from the business logic inside the business object implementation such as from an action, a validation or a determination.

For dynamic control of operations acting on individual entity instances, the option `(features: instance)` must be added to the update or delete operation in question. This is also possible for the create by association operation. However, an implementation in the referenced class pool `ABAP_CLASS` is necessary for this. For each relevant operation, you can specify in the implementing handler of the class pool the following values:

- **ENABLED** - if the operation is enabled
- **DISABLED** - if the operation is disabled.

(2) Feature Control at Element Level

Within the bracket of `define behavior for CDSEntity { ... }`, you can specify for fields of an entity if they should have certain access restrictions.

For static feature control, it is sufficient to define these restrictions in the behavior definition alone:

- The option `field (read only) f1` does not allow the `f1` field to be changed in create and update operations.
- The option `field (mandatory) f2` requires the corresponding field `f2` to be supplied with a value in create operations. For update operations, the field must not have a non-initial value.

i Note

To classify multiple fields in the same way, the comma notation can be used:

```
field(read only) f1, f2, f3;
```

For defining instance-based field control, the option `(features: instance)` must be added to the field in question. In this case however, the implementation of dynamic feature control in the referenced class pool `ABAP_CLASS` is required. When implementing dynamic field control you have the option of specifying the following values for each field that is notated with `(features: instance)`:

- UNRESTRICTED – field has no restrictions
- MANDATORY – field is mandatory
- READ_ONLY – field is read-only
- ALL – All restrictions are requested.

(3) Feature Control for Actions

Specific operations of an entity of a business object can be defined using actions. Similar to standard operations, you can define internal actions in the behavior definition by adding the option `internal` to the operation name. Internal actions can only be accessed from the business logic inside the business object implementation such as from validations, determinations, or from other non-internal actions.

For dynamic control of actions acting on individual entity instances, the option `(features: instance)` must be added to the relevant action in the behavior definition. The required implementation must be provided in the referenced class pool `ABAP_CLASS`. For each relevant action, you can specify in the implementing handler of the class pool the following values:

- ENABLED - if the action is enabled
- DISABLED - if the action is disabled.

Related Information

[Implementing Dynamic Feature Control \[page 468\]](#)

7.7.4 Implementing Dynamic Feature Control

The implementation of dynamic feature control is based on the ABAP language in a local handler class (`lhc_handler`) as part of the behavior pool. As depicted in the listing below, each such local class inherits from the base handler class `CL_ABAP_BEHAVIOR_HANDLER`.

The dynamic feature control for an entity is implemented in this handler class using the method `FOR FEATURES`. For more information, see [<method> FOR FEATURES \[page 729\]](#).

The output parameter `result` is used to return the feature control values.

These include

- field control information: `%field-fieldx`
- action control information: `%features-%action-action_name`
- standard operation control information for update and delete: `%features-(%update|%delete)`
- operation control information for create by association: `%assoc-assoc_name`.

Listing 1: Implementing Feature Control in a Local Handler Class

```
CLASS lhc_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS feature_ctrl_method FOR FEATURES
    IMPORTING keys REQUEST requested_features FOR aliasedEntity RESULT result.
ENDCLASS.

CLASS lhc_handler IMPLEMENTATION.
METHOD feature_ctrl_method.
    " Read data required
    READ ENTITY CDSEntityName
        FIELDS ( field1 field2 )
        WITH VALUE #( FOR keyval IN keys ( %key = keyval-%key ) )
        RESULT DATA(lt_result_variable).

    " Return feature control information
    result = VALUE #( FOR ls_variable IN lt_result_variable
        ( %key
            = ls_variable-%key
        )
    )
    " Field control information
        %field-field1
        %field-field2
    " Action control information
        %features-%action-action_name = COND #( WHEN condition
            THEN if_abap_behv=>fc-o-
        disabled
            ELSE if_abap_behv=>fc-o-
        enabled
            )
        " Operation (example: update) control information
        %features-%update
            = COND #( WHEN condition
                THEN if_abap_behv=>fc-o-
            disabled
                ELSE if_abap_behv=>fc-o-
            enabled
                )
        " Operation control information for create by association
        %assoc-_Assoc
            = COND #( WHEN condition
                THEN if_abap_behv=>fc-o-
            disabled
                ELSE if_abap_behv=>fc-o-
            enabled
                )
        .
ENDMETHOD.

ENDCLASS.
```

For a complete example of dynamic action control, see [Implementing Dynamic Action Control \[page 335\]](#).

For a complete example of dynamic operation control, see [Implementing Dynamic Operation Control for Create By Association \[page 326\]](#).

7.8 Consuming Business Objects with EML

Business objects can be consumed not only by means of the OData protocol (for example, in Fiori UIs) but also directly in ABAP by using Entity Manipulation Language (EML).

This topic offers some code samples to demonstrate how you can access our Travel business object with EML syntax in a simple consumer class. You will get to know the core structure of EML at this point.

Contents:

[EXAMPLE 1: Implementing UPDATE for Travel Data \[page 469\]](#)

[EXAMPLE 2: Executing an Action \[page 471\]](#)

[EXAMPLE 3: Implementing DELETE for Travel Instances and Their Child Instances \[page 472\]](#)

[EXAMPLE 4: Creating Instances Along the Composition Hierarchy \("deep create"\) \[page 473\]](#)

EXAMPLE 1: Implementing UPDATE for Travel Data

In this example, two fields `agencyid` and the `memo` text should be changed to a given travel instance.

Prerequisites

The entity instances can only be updated in a `MODIFY` call if the `update` operation is specified for each relevant entity in the behavior definition and is implemented in the behavior pool accordingly.

Because the change will only affect one entity, we use the short form of the `MODIFY` syntax:

Syntax for UPDATE

```
MODIFY ENTITY EntityName
  UPDATE FIELDS ( field1 field2 ... ) WITH it_instance_u
  [FAILED ls_failed | DATA(ls_failed)]
  [REPORTED ls_reported | DATA(ls_reported)].
```

The `UPDATE` call allows to trigger delta updates on consumer side where only the key field and the fields with new values need to be supplied. From provider side, it allows to identify which fields are overwritten and which need to be kept according to the DB data. The fields to be updated are specified in the field list through addition `FIELDS (field1 field2 ...)`.

The following listing provides you with the source code of an executable consumer class. To enable the `classrun` mode, the consumer class implements the `if_oo_adt_classrun` interface.

Since the result data is not exported as part of the UPDATE statement, a subsequent READ ENTITY call is required to read the changed data from transactional buffer. The result data of the read operation is specified in the target variable lt_received_travel_data.

Listing 1: Implementing UPDATE for Travel Data

```
CLASS /dmo/cl_eml_travel_update DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_oo_adt_classrun.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_eml_travel_update IMPLEMENTATION.
  METHOD if_oo_adt_classrun~main.
    DATA(lv_travel_id)      = '00000011'. " Valid travel ID
    DATA(lv_description)    = 'Changed Travel Agency'.
    DATA(lv_new_agency_id)  = '070017'.   " Valid agency ID
    " UPDATE travel data
    MODIFY ENTITY /DMO/I_Travel_U
      UPDATE FIELDS ( agencyid memo ) WITH VALUE #((
        travelid      = lv_travel_id
        agencyid      = lv_new_agency_id
        memo          = lv_description ) )
    FAILED  DATA(ls_failed)
    REPORTED DATA(ls_reported).
    " Read travel data from transactional buffer
    CLEAR: ls_reported, ls_failed.
    READ ENTITY /DMO/I_Travel_U
      FIELDS ( agencyid memo ) WITH VALUE #( ( travelid = lv_travel_id ) )
      RESULT  DATA(lt_received_travel_data)
    FAILED  ls_failed.
    " Output result data on the console
    OUT->WRITE( lt_received_travel_data ).
    " Persist changed travel data in the database
    COMMIT ENTITIES.
    " Check criteria of success
    IF SY-subrc = 0.
      out->write( 'Successful COMMIT!' ).
    ELSE.
      out->write( 'COMMIT failed!' ).
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

Checking Results

To check the results of the MODIFY call, run the main method of the consumer class /dmo/cl_eml_travel_update by pressing **F9** key and view the received RESULT data (lt_received_travel_data) on the console output. This data contains all fields of the target travel instance (in this example, with the travel ID = 11).

The screenshot shows the ABAP Console interface with a table titled 'Table'. The table has columns: TRAVELID, AGENCYID, CUSTOMERID, BEGINDATE, and ENDDATE. A single row is displayed: TRAVELID 00000011, AGENCYID 070017, CUSTOMERID 000582, BEGINDATE 2018-12-15, and ENDDATE 2018-12-15. Below the table, a message says 'Received RESULT Data from Transactional Buffer'.

TRAVELID	AGENCYID	CUSTOMERID	BEGINDATE	ENDDATE
00000011	070017	000582	2018-12-15	2018-12-15

Received RESULT Data from Transactional Buffer

EXAMPLE 2: Executing an Action

All modify operations in EML that cannot be implemented by standard operations (create, update, delete) are handled by actions.

This example demonstrates the implementation of an action related to a given travel instance. The consumer class (see Listing 2 below) is implemented to change the status of the travel processing to booked.

Prerequisites

The `SET_STATUS_BOOKED` action is specified in the behavior definition at the root entity level and is implemented in the behavior pool accordingly.

The `MODIFY` statement uses the following general syntax for executing an action:

Syntax for UPDATE (short form)

```
MODIFY ENTITY EntityName
  EXECUTE action_name FROM it_instance_a
    [RESULT result_action | DATA(result_action) ]
    [FAILED ls_failed | DATA(ls_failed) ]
    [REPORTED ls_reported | DATA(ls_reported) ].
```

The `action_name` refers to the name of an action as it is defined in the behavior definition for the corresponding entity. The input parameter `it_instance_a` contains the keys of the entities on which the action has to be executed.

The syntax for executing an action allows exporting of result data. The result data of an action execution is specified in the target variable `result_action`.

The following listing provides you with the source code of an executable consumer class implementing the execution of `set_status_booked` action for a selected travel instance. Again, to enable the class-run mode, the consumer class `/dmo/cl_eml_travel_action` implements the `if_oo_adt_classrun` interface.

Listing 2: Executing the SET_STATUS_BOOKED Action

```
CLASS /dmo/cl_eml_travel_action DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_oo_adt_classrun.
  PROTECTED SECTION.
```

```

PRIVATE SECTION.
ENDCLASS.
CLASS /dmo/cl_eml_travel_action IMPLEMENTATION.
METHOD if_oo_adt_classrun~main.
  DATA(lv_travel_id)      = '00000026'. " Valid travel ID
  " EXECUTE action for travel data
  MODIFY ENTITY /DMO/I_Travel_U
    EXECUTE set_status_booked
      FROM VALUE #( ( travelid = lv_travel_id ) )
      RESULT DATA(lt_set_status_booked)
  FAILED  DATA(ls_failed)
  REPORTED DATA(ls_reported).

  " Output result data on the console
  OUT->WRITE( lt_set_status_booked[ 1 ]-%param ).
  " Persist changed travel data in the database
  COMMIT ENTITIES.
  " Check criteria of success
  IF SY-subrc = 0.
    out->write( 'Successful COMMIT!' ).
  ELSE.
    out->write( 'COMMIT failed!' ).
  ENDIF.
ENDMETHOD.
ENDCLASS.

```

Checking Results

To check the results of the `MODIFY` call, run the `main` method of the class from listing above by pressing **F9** key and then view the received `RESULT` data (`lt_set_status_booked`) on the console output.

EXAMPLE 3: Implementing DELETE for Travel Instances and Their Child Instances

This example demonstrates how you can implement multiple `DELETE` operations for different entities in one `MODIFY` call. In this case, we use the long form of the `MODIFY` statement that allows you to collect multiple operations on multiple entities of one business object that is identified by `RootEntityName`.

Prerequisites

The entity instances can only be deleted in a `MODIFY` call if the `delete` operation is specified for each relevant entity in the behavior definition and is implemented in the behavior pool(s) accordingly.

Syntax for DELETE (long form)

```

MODIFY ENTITIES OF RootEntityName
  ENTITY entity_1_name
    DELETE FROM it_instance1_d
  ENTITY entity_2_name
    DELETE FROM it_instance2_d

  ...
  [FAILED  DATA(it_failed)]
  [REPORTED DATA(it_reported)].

```

To delete individual instances of entities, the keys of the entity must be specified in the `FROM` clause of the `MODIFY` statement.

Each `DELETE` operation has a table of instances as input parameters: `it_instance1_d` and `it_instance1_d`, which provide the `MODIFY` call with key information.

The following listing provides you with the source code of an executable consumer class that implements the deletion of a given travel instance `lv_travel_to_delete` and a booking supplement `lv_booksuppl_to_delete`.

Listing 3: Implementing DELETE for Travel and Booking Supplement Instances

```
CLASS /dmo/cl_eml_travel_delete DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_oo_adt_classrun.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_eml_travel_delete IMPLEMENTATION.
  METHOD if_oo_adt_classrun~main.

    DATA(lv_travel_to_delete)      = '00000077'. " Valid IDs
    DATA(lv_travel_id)           = '00000101'.
    DATA(lv_booking_id)          = '0002'.
    DATA(lv_booksuppl_to_delete) = '04'.

    MODIFY ENTITIES OF /dmo/i_travel_U
      " Delete travel and all child instances (booking, booking supplements)
      ENTITY travel
        DELETE FROM VALUE #( ( travelid           = lv_travel_to_delete ) )
      " Delete booking supplement with the ID =04
      ENTITY bookingsupplement
        DELETE FROM VALUE #( ( travelid           = lv_travel_id
                               bookingid          = lv_booking_id
                               bookingsupplementid =
                               lv_booksuppl_to_delete ) )
        REPORTED DATA(ls_reported)
        FAILED   DATA(ls_failed).
      " Persist changed travel data in the database
      COMMIT ENTITIES.
      " Check criteria of success
      IF SY-subrc = 0.
        out->write( 'Successful COMMIT!' ).
      ELSE.
        out->write( 'COMMIT failed!' ).
      ENDIF.
    ENDMETHOD.
ENDCLASS.
```

Checking Results

To check the results of the `MODIFY` call, run the `main` method of the class from listing above by pressing **[F9]** key in the class editor and then search for data of selected instances in the data preview tool (**[F8]** on the CDS root view `DMO/I_TRAVEL_U`.)

EXAMPLE 4: Creating Instances Along the Composition Hierarchy ("deep create")

This example demonstrates how you can implement a direct `CREATE` and multiple `CREATE BY` association operations for different entities in one `MODIFY` call. In this case, the long form of the `MODIFY` statement is used to collect multiple operations on multiple entities of one business object that is identified by the `RootEntityName`.

Prerequisites

The instances of entities (root or child) can only be directly created in a `MODIFY` call if the create operation is specified for the relevant entities in the behavior definition (and is implemented in the behavior pool accordingly - in case of unmanaged implementation type).

The same applies to instances of child entities that are created by association. In this case however, an association to the child entity must be specified in behavior definition (and implemented in a handler method of the behavior pool - in case of unmanaged implementation type).

Syntax for `CREATE (BY association)`

```
MODIFY ENTITIES OF RootEntityName  " name of root CDS view
  ENTITY entity_1_name
    CREATE FIELDS( field1 field2 ... ) WITH it_instance1_c
    CREATE BY \association1_name ( field1 field2 ... ) WITH it_instance1_cba
    ...
  ENTITY entity_2_name
    CREATE BY \association2_name ( field1 field2 ... ) WITH it_instance2_cba
    ...
  [FAILED  DATA(it_failed)]
  [MAPPED  DATA(it_mapped)]
  [REPORTED DATA(it_reported)].
```

→ Remember

When multiple entity instances are created by one `MODIFY` statement, then it is required to provide the content ID `%CID` [\[page 740\]](#) information for all instances to be created. .

In addition to the content ID, the required field values for the instance to be created must be populated. This is done by the input parameter `it_instance_c` in the `CREATE FIELDS () WITH` statement.

If the instances of child entities should to be created by an association, besides the parent key, also the new values for the child entity to be created, must be populated in the create operation. The input parameter `it_instance_cba` in the `CREATE BY \association1_name...` statement is therefore a table type containing the parent key (reference to content ID in case the parent instance is created in the same `MODIFY` call) and the `%target` sub-table that refers to the child instance to be created.

The following listing provides you with the source code of an executable consumer class that implements the creation of a new travel instance including a new booking and the associated booking supplement.

Listing 4: Creating Instances with `%CIDs`

```
CLASS /dmo/cl_eml_travel_subnodes DEFINITION
  PUBLIC
  FINAL
```

```

CREATE PUBLIC .
PUBLIC SECTION.
  INTERFACES if_oo_adt_classrun.
PROTECTED SECTION.
PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_eml_travel_subnodes IMPLEMENTATION.
METHOD if_oo_adt_classrun~main.
  TYPES: BEGIN OF ts_flight,
    carrier_id      TYPE /dmo/carrier_id,
    connection_id   TYPE /dmo/connection_id,
    flight_date     TYPE /dmo/flight_date,
    price           TYPE /dmo/flight_price,
    currency_code   TYPE /dmo/currency_code,
  END OF ts_flight.

  DATA gv_booking_date TYPE /dmo/booking_date.
  DATA gv_customer_id  TYPE /dmo/customer_id.
  DATA gs_flight       TYPE ts_flight.
  DATA(lv_description) = 'Intro to EML'.
  DATA(lv_agency_id)   = '070048'.
  DATA(lv_my_agency_id) = '070017'.
  " Get current date
  gv_booking_date = cl_abap_context_info->get_system_date( ).
  " Get valid customer ID
  SELECT SINGLE customer_id FROM /dmo/customer INTO @gv_customer_id..
  " Get valid flight data
  SELECT SINGLE FROM /dmo/flight FIELDS * INTO @DATA(ls_flight).
  " Get valid supplement data
  SELECT SINGLE FROM /dmo/supplement FIELDS * INTO @DATA(ls_supplement).
  " Create a new travel > booking > booking supplement
  MODIFY ENTITIES OF /dmo/i_travel_u
    ENTITY travel
      CREATE FIELDS ( agencyid customerid begindate enddate memo status )
  WITH
    VALUE #( ( %cid          = 'CID_100'      " Preliminary ID
  for new travel instance
    agencyid      = lv_agency_id
    customerid   = gv_customer_id
    begindate    = '20191212'
    enddate      = '20191227'
    memo         = lv_description
    status        = CONV #( /dmo/
  if_flight_legacy=>travel_status-new ) )
    " Update data of travel instance
    UPDATE FIELDS ( agencyid memo status ) WITH
      VALUE #( ( %cid_ref    = 'CID_100'      " Refers to travel
  instance
    agencyid      = lv_my_agency_id
    memo         = 'Changed Agency and Status!'
    status        = CONV #( /dmo/
  if_flight_legacy=>travel_status-planned ) )
    " Create a new booking by association
    CREATE BY \booking FIELDS ( bookingdate customerid airlineid
connectionid flightdate flightprice currencycode ) WITH
      VALUE #( ( %cid_ref    = 'CID_100'      "refers to the
  root (travel instance)
      %target      = VALUE #( (
        %cid          = 'CID_200' "
  Preliminary ID for new booking instance
      bookingdate  = gv_booking_date
      customerid   = gv_customer_id
      airlineid    = ls_flight-
carrier_id
      connectionid = ls_flight-
connection_id
      flight_date   = ls_flight-
flight_date
      flightprice   = ls_flight-price

```

```

        currencycode = ls_flight-
currency_code ) ) ) )
    " Create a booking supplement by association
ENTITY booking
    CREATE BY \_booksupplement FIELDS ( supplementid price
currencycode ) WITH
        VALUE #( ( %cid_ref = 'CID_200'
            %target = VALUE #( (
                %cid             = 'CID_300'
                supplementid = ls_supplement-
supplement_id
                price          = ls_supplement-
price
                currencycode = ls_supplement-
currency_code ) ) )
        MAPPED    DATA(ls_mapped)
        FAILED   DATA(ls_failed)
        REPORTED DATA(ls_reported).
COMMIT ENTITIES.
" Check criteria of success
IF SY-subrc = 0.
    out->write( 'Successful COMMIT!' ).
ELSE.
    out->write( 'COMMIT failed!' ).
ENDIF.
ENDMETHOD.
ENDCLASS.
```

Checking Results

To check the results, run the main method of the consumer class /dmo/cl_eml_travel_subnodes from listing above by pressing **[F9]** key in the class editor and then search for the newly created travel, booking and booking supplement instances in the data preview tool (**[F8]** on the travel root CDS view /DMO/I_TRAVEL_U.)

The screenshot shows the SAP Data Preview tool interface. The path in the top bar is **/DMO/I_TRAVEL_U**. The current view is **Raw Data**, indicated by the tab. Below it, a filter bar shows **3 rows retrieved - 6 ms**. The data table has columns: TravellID, Age..., Custo..., BeginDate, EndDate. There are three rows of data:

TravellID	Age...	Custo...	BeginDate	EndDate
00004139	070017	000001	2019-03-08	2019-03-27
00004140	070017	000001	2019-03-08	2019-03-27
00004142	070017	000001	2019-03-08	2019-03-27

A context menu is open over the last row (TravellID 00004142). The menu items are: Quick filter on [00004142], Distinct values for [TravellID], Follow Association (highlighted in blue), and Copy row. A tooltip 'new travel instance' points to the highlighted row, and another tooltip 'association to new booking instance' points to the 'Follow Association' menu item.

Checking the Newly Created Instances in the Data Preview

Related Information

[Entity Manipulation Language \(EML\) \[page 116\]](#)

7.9 Using Type and Control Mapping

Whenever existing legacy code and data types are to be reused in behavior pools of business objects, then you need to perform a mapping between CDS field names and types and corresponding legacy field names and types.

Use Case

You can use this kind of mapping in applications that include unmanaged or managed business objects based on CDS entities on the one hand, and legacy data types (ABAP Dictionary types) (and functionality) that are generally older.

This is particularly significant for the unmanaged implementation type, which essentially represents a kind of wrapper for existing legacy functionality.

Also with the managed implementation type, it can happen that, for example, the code for a determination or validation already exists, but is based on "old" (legacy) data types.

When accessing the legacy code, the developer would normally have to use "corresponding with mapping" in many places to map input [derived types \[page 811\]](#) (type table for create, type table for action import) to legacy types and vice versa to map legacy results to output [derived types \[page 811\]](#) (type table for action result, type table for read result, failed).

In addition, the types used to represent the control information also can differ significantly in the legacy code and the current ABAP programming model:

In some legacy scenarios (especially the ones using BAPIs), in addition to the dictionary type directly corresponding to the entity, there is another one that contains (ideally) the same fields as that type, but these all have the type C (1) (like the control structures in BAPIs). This type has the same function as the [%CONTROL \[page 741\]](#) structure in [derived types \[page 811\]](#), which indicates that fields in the main structure that are accessed by an operation (update, read, and so on) Such type pairs are often used in BAPIs, for example BAPIAD2VD/BAPIAD2V рDX. The control data element is BAPIUPDATE with the type C (1).

The solution is now to introduce a central, declarative mapping within the behavior definition instead of many individual corresponding statements. This improves maintainability of the application's source code.

Defining Type and Control Mapping in the Behavior Definition

A mapping sets the entity in relation to any other structured ABAP Dictionary type (legacy type). It is introduced in the behavior definition for an [unmanaged \[page 821\]](#) or [managed \[page 815\]](#) implementation type with the keyword `mapping for`.

Syntax: Mapping statement in the behavior definition

```
[implementation] unmanaged|managed [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
...
```

```
{  
    mapping for ...  
  
    ...  
}
```

Variant 1: Field Mapping

The simplest form for mapping definition is the following:

Syntax: Field mapping with corresponding

```
mapping for LegacyType corresponding  
{  
    EntityField1 = LegacyField1;  
    EntityField2 = LegacyField2;  
    ...  
}
```

The addition `corresponding` automatically maps fields of the same name to each another. The corresponding fields of different names can be specified through explicitly listed field pairs.

The `corresponding` addition can also be omitted, which is not recommended in general, because then the automatic mapping of fields with the same name is lost.

If no renaming of the fields is required, the short form can be used:

Syntax (short form):

```
mapping for LegacyType corresponding;
```

i Note

Mapping can be partial (legacy type contains fields that do not match any fields in CDS).

Variant 2: Field and Control Type Mapping

Using the following syntax, a mapping definition can be made simultaneously for a main field type `LegacyType` and a control type `ControlType`:

Syntax: Field type and control mapping with corresponding

```
mapping for LegacyType control ControlType corresponding  
{  
    EntityField1 = LegacyField1;  
    EntityField2 = LegacyField2;  
    ...  
}
```

If no renaming of the fields is required, the short form can be used:

Syntax (short form):

```
mapping for LegacyType control ControlType corresponding;
```

Usually, it is assumed that the fields in the main and control type have the same name.

A different field mapping in { ... } is specified as:

```
EntityField = LegacyField control ControlField;
```

i Note

For all control type fields, the type C(1) or X(1) is expected.

Using Type Mapping in ABAP

The reference to mappings that are defined in a behavior definition is done by some special variants of the ABAP corresponding operator.

Syntax: Input mapping (entity type to legacy type)

```
data:  
  legacy_var type LegacyType,  
  entity_var type structure|table for create|update|delete entity.  
...  
legacy_var = corresponding #( entity_var mapping from entity ).
```

The addition from entity assigns entity_var field-wise to data object legacy_var according to the mapping definition for LegacyType in the behavior definition.

This mapping works fine if the type of entity_var is an input derived type, or the entity type itself, or the table type of the entity.

Syntax: Output mapping (from legacy type to entity type)

```
data:  
  legacy_var type LegacyType,  
  entity_var type table for read result entity.  
...  
entity_var = corresponding #( legacy_var mapping to entity ).
```

The addition to entity assigns legacy_var field-wise to data object entity_var according to the mapping definition for LegacyType in the behavior definition.

This mapping works fine if the type of entity_var is an output derived type, or the entity type itself, or the table type of the entity.

Using Control Type Mapping in ABAP

Control mapping is about supporting the actual semantics of the %control fields: They indicate which of the fields should be, for example, changed by an update or read by a read operation.

For the move between an input derived type (type table for create, type table for action import) and a target structure, this means that only the fields for which the corresponding control field is set should be moved.

Syntax: Using control

```
target = corresponding #( source using control ).
```

The structure or table `source` must be a derived type that also includes the `%control` structure.

For example, if the `source` has a field `amount`, then `%control-amount` decides whether to move the value of `amount` to the corresponding field of `target` – only if the value of the corresponding `control` field is non-initial.

By combining with the additional mapping `from entity` described above, also field mapping specified in the behavior definition is effective:

Syntax: Using control with field mapping

```
target = corresponding #( source mapping from entity using
control ).
```

Conversely, there is an addition `changing control` that allows the `%control` fields of a derived type to be set from a `%control-less` source structure - based on the criterion of whether the corresponding field is non-initial.

Syntax: Changing control

```
target = corresponding #( source changing control ).
```

Related Information

[Declaration of Derived Data Types \[page 734\]](#)

[Implicit Returning Parameters \[page 738\]](#)

7.10 Using Groups in Large Development Projects

This section introduces the concept of groups that can be used to divide operations, actions and other implementation-relevant parts of the business logic into several groups for behavior implementation.

Use Case

Generally, the implementation of business object entity's operations and actions is done in the *Local Types* include of the behavior pool (ABAP class that implements business object's behavior) associated with that entity – unless the control of the implementation classes using the `IMPLEMENTATION IN CLASS` syntax (at entity level) has been completely dispensed.

Since the *Local Types* include can only be changed by one developer at a time, the efficiency of development would be significantly reduced in case of larger implementations. Let us think about large business objects with extensive business logic implementations, with many entities, each of which may contain a variety of elements.

As a solution, the operations and actions of an entity can be divided into several groups, whereby each of these groups can then be assigned a different behavior pool as implementation class.

Especially in large development projects the responsibilities for the implementation of application logic, are assigned to different members of a development team. To support this approach technically, we introduce the concept of groups. This approach enables that a team of developers can implement parts of business logic independent from each other. In addition, the group concept allows to tailor the functionality of business objects according to semantic considerations.



Relationship Between Entities, Groups and Implementing Classes

Syntax: Defining Groups for Unmanaged Business Objects

Groups can be defined within a behavior definition for a business object of type unmanaged by using the following syntax:

```
[implementation] unmanaged [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
  lock (master|dependent() )
  ...
{
  group groupName_1 implementation in class ABAP_CLASS_1 unique
  {
    // Implementation-relevant content of the entity
  }
  group groupName_2 implementation in class ABAP_CLASS_2 unique
  {
    // Implementation-relevant content of the entity
  }
  // It is possible to assign the same behavior pool as the implementation class
  in different groups
```

```

group groupName_3 implementation in class ABAP_CLASS_1 unique
{
    // Implementation-relevant content of the entity
}
group ...
}

```

Syntax: Defining Groups for Managed Business Objects

Groups can be defined within a behavior definition for a business object of type managed by using the following syntax:

```

[implementation] managed [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
    persistent table DB_TABLE
    ...
{
    [create;]
    [update;]
    [delete;]
    [read;]
    [association AssociationName [abbreviation AbbreviationName] {[create;] } ]]

    group groupName_1 implementation in class ABAP_CLASS_1 unique
    {
        // Implementation-relevant content of the entity
    }
    group groupName_2 implementation in class ABAP_CLASS_2 unique
    {
        // Implementation-relevant content of the entity
    }
    // It is possible to assign the same behavior pool as the implementation class
    // in different groups
    group groupName_3 implementation in class ABAP_CLASS_1 unique
    {
        // Implementation-relevant content of the entity
    }
    group ...
}

```

Explanatory Notes

(1) The group name is defined locally within the entity's behavior definition and must not conflict with actions, determinations, or validations of the same name.

(2) The `implementation in class` syntax can only be used on groups, but no longer on individual entity's behavior definition itself. A group requires a behavior pool `ABAP_CLASS_*` and the addition of `unique`.

i Note

With the addition `implementation in class ABAP_CLASS` in the header of the behavior definition, you have the option to implement the remaining functionality for all entities in a common behavior pool. For example, the save sequence for all entities of a business object could be implemented in a single behavior pool `ABAP_CLASS`.

It is possible to specify the same behavior pool as the implementation class in different groups. In the syntax above, the implementation-relevant content of `groupName_1` and of `groupName_3` must be implemented in `ABAP_CLASS_1`.

(3) The implementation-relevant content of an entity can be:

- Actions
- Instance-based feature control
- Determinations - for managed implementation type only
- Validations - for managed implementation type only.

i Note

In the case of **unmanaged** implementation type, the standard operations (`CREATE`, `UPDATE`, `DELETE`) as well as `READ` and `CREATE` by association must be assigned to a group. In the **managed** case however, the standard operations (that are implemented by the framework) and `READ` and `CREATE` for associations can be specified either inside or outside groups.

(4) Information on mapping (which is never relevant to implementation) must always be specified outside of groups.

(5) Implicit standard operations (defined automatically and did not have to be explicitly specified) must be explicitly specified within one and the same group in the unmanaged case (where they are implementation-relevant):

- `read;` - for `READ` operations of an entity
- `lock;` - for `LOCK` operations of an entity that is defined as lock master.

Examples

Listing: Groups in the behavior definition of an unmanaged business object

```
implementation unmanaged implementation in class /DMO/BP_TRAVEL_U;
// behavior definition for the TRAVEL root entity
define behavior for /DMO/I_Travel_U alias travel
    etag LastChangedAt
    lock master
{
    group travel_cud implementation in class /dmo/bp_travel_cud unique
    {
        field ( read only ) TravelID;
        field ( mandatory ) AgencyID, CustomerID, BeginDate, EndDate;
        field(features:instance) overall_status;
        create;
        update(features:instance);
        delete;
        read; // read and lock must be assigned explicitly to a group
        lock;
    }
    group travel_cba implementation in class /dmo/bp_travel_cba unique
    {
        association _Booking { create; }
    }
    group travel_main_actions implementation in class /dmo/bp_travel_main_a unique
    {
        action(features : instance) set_status_booked result [1] $self;
        action                                getTravel result [1] $self;
        action                                copyTravel result [1] $self;
    }
}
```

```

group travel_aux_actions implementation in class /dmo/bp_travel_aux_a unique
{
    action(features:instance) getMaxDate result [1] $self;
    action(features:instance) getminDate result [1] $self;
}
mapping for /dmo/travel
{
    ...
}

}

// behavior defintion for the BOOKING child entity
define behavior for /DMO/I_Booking_U alias booking
    lock dependent ( travel_id = travel_id )
{

group booking_rud implementation in class /dmo/bp_booking_rud unique
{
    read;
    update;
    delete;
}
group booking_fc implementation in class /dmo/bp_booking_fc unique
{
    field ( read only ) TravelID, BookingID;
    field ( mandatory ) CustomerID, AirlineID, ConnectionID, FlightDate;
    action(features:instance) confirmBooking result [1] $self;
}
mapping for /dmo/booking
{
    ...
}
}

```

Listing: Groups in the behavior definition of a managed business object

```

implementation unmanaged implementation in class /DMO/BP_TRAVEL_M;
// behavior definition for the TRAVEL root entity
define behavior for /DMO/I_Travel_M alias travel
    persistent table /dmo/travel_m
    with additional save
    lock master
    authorization master ( instance )
    etag LastChangedAt
{
    create;
    delete;
    association _Booking { create; }
    group travel_fc implementation in class /dmo/bp_travel_fc unique
    {
        field ( read only )      TravelID;
        field ( mandatory )     AgencyID, CustomerID, BeginDate, EndDate;
        field(features:instance) overall_status;
    }
    group travel_cba implementation in class /dmo/bp_travel_cba unique
    {
        association _Booking { create; }
    }
    group travel_actions implementation in class /dmo/bp_travel_a unique
    {
        action(features : instance)      set_status_booked result [1] $self;
        action                           getTravel result [1] $self;
        action ( authorization : none ) copyTravel result [1] $self;
        action(features:instance)       getMaxDate result [1] $self;
        action(features:instance)       getminDate result [1] $self;
    }
    group booking_det_val implementation in class /dmo/bp_booking_det_val unique

```

```

{
  determination determineDiscount on modify { create; }
  validation validateAgency      on save   { field Agency_ID; }
  validation validateCustomer    on save   { field Customer_ID; }
  validation validateDates       on save   { field Begin_Date, End_Date; }
  validation validateStatus      on save   { field overall_Status; }
}

mapping for /dmo/travel
{
  ...
}

// behavior definition for the BOOKING child entity
define behavior for /DMO/I_Booking_M alias booking
{
  read;
  update;
  delete;
  mapping for /dmo/booking
  {
    ...
  }
  group booking_fc implementation in class /dmo/bp_booking_fc unique
  {
    field ( read only ) TravelID, BookingID;
    field ( mandatory ) CustomerID, AirlineID, ConnectionID, FlightDate;
    action(features:instance) confirmBooking result [1] $self;
  }
  group booking_det_val implementation in class /dmo/bp_booking_det_val unique
  {
    determination totalBookingPrice on modify { field Flight_Price; }
    determination determineCustomerStatus on modify { create; }
    // internal action: triggered by determination
    internal action SetCustomerStatus;
  }
}

```

Implementation-Related Aspects

(1) The name of the group and which operations are associated with this group do not matter to external users (and can therefore be changed retrospectively by the developer). This means that external operations and actions are still accessed by the usual syntax, in which only the name of the entity, the operation, and, if applicable, the action/determination/validation or association plays a role, but not the name of the group.

However, there are exceptions: the name of the group is relevant for the implementation of instance-based feature control - the corresponding implementations then control only those features that are associated with their respective group. (The framework merges the information for the external consumers.) The corresponding syntax `entity-group` can only be used within the implementation class associated with that group. Specifically, the following declarations are concerned:

Syntax for methods ... for features

```

methods method_name for features key_param
  request request_param for entity~group_name
  result result_parameter.

```

Syntax for types/data ... for features

```
type|data ... type table|structure for features key|request|result  
entity~group_name.
```

i Note

This declaration can also be done in the `public` section of the implementation class to make the group-dependent type public outside. Because it makes the changes to group assignment incompatible with external users, such publishing is not recommended.

Example: Declaration of local handler for feature control implementation

Within the implementation class, the syntax `methods ... for features` for instance-based feature control can only be defined by specifying the group name:

```
class lhc_travel_fc definition inheriting from cl_abap_behavior_handler.  
  private section.  
    methods get_features for features  
      importing keys request requested_features for travel~group_name result  
      result.  
  endclass.
```

(2) Because associations with the usual association syntax can only be assigned as a whole to a group, it is not possible to implement the association's `CREATE` operation in an implementation class other than the `READ` operation.

Example: Local behavior implementation of create by association

```
class lhc_travel_cba definition inheriting from cl_abap_behavior_handler.  
  private section.  
    methods create_bookings for modify  
      importing entities for create travel\_Booking.  
    methods read_bookings for read  
      importing keys for read travel\_Booking full result_requested  
      result result link association_links.  
  endclass.  
  class lhc_travel_cba implementation.  
    method create_bookings.  
      ...  
    endmethod.  
    method read_bookings.  
      ...  
    endmethod.  
  endclass.
```

7.11 Adding Authorization Control to Managed Business Objects

Transactional business applications based on business objects require an authorization concept for their data and for the operations on their data. Display and create, update and delete (CUD) operations, as well as specific business-related activities, must be protected from unauthorized access and are therefore allowed for authorized users only.

In a transactional development scenario, you can add authorization checks to various components of an application. In this case, different mechanisms are used to implement the authorization concept.

The following topics deal with authorization control for read and modify operations in the context of managed business objects.

Authorizations Checks for Read Operations

To protect data from unauthorized read access, the ABAP CDS already provides its own authorization concept based on a data control language ([DCL \[page 810\]](#)). The authorization and role concept of ABAP CDS uses conditions defined in CDS access control objects to check the authorizations of users for read access to the data in question. In other words, access control allows you to limit the results returned by a CDS entity to those results you authorize a user to see.

In addition, DCL is also automatically evaluated in case of transactional read access, that is when using EML-based read and read by association operations, as well as when processing feature control.

More on this:

Authorizations Checks for Modify Operations

For managed business objects, also modifying operations such as standard operations create, update, delete, create by associations, and actions must be checked against unauthorized access.

The availability of authorization control is modeled in the behavior definition. The authorization control requires not only a definition but also an implementation in a handler class of the corresponding behavior pool.

The authorization control is checked by the business object runtime as soon as the relevant operation is executed.

Restriction: With the current release, only instance-based authorization control (`authorization master(instance)`) is supported. That means, static authorization is not available. Therefore, you cannot apply authorization checks to `create` operations (static operations).

Activities Relevant to Developers

1. [Modeling Authorization Control \[page 488\]](#)
2. [Implementing Authorization Control \[page 490\]](#)

7.11.1 Modeling Authorization Control

!Restriction

With the current release, only instance-based authorization control is supported: (authorization master(instance)). This means, static authorization (that does not depend on an instance) is not yet available. Therefore, you cannot apply authorization checks to the create operation (that is a static operation).

To define the instance-based authorization control on a CDS entity for managed implementation type, the following syntax is used in the behavior definition:

Syntax for Defining Instance-Based Authorization Control

```
[implementation] managed;
define behavior for RootEntity [alias RootAliasedName]
implementation in class ABAP_CLASS_FOR_ROOT [unique]
authorization master(instance)
...
{
/* (1) Authorization checks cannot be applied on create operation (static
operation) */
    create;
/* (1') Authorization check: is always enabled for update */
    update;
/* (1") Authorization check can be disabled for delete */
    delete (authorization : none);
/* (2) Authorization check: enabled for Action1 */
    action Action1 [...]
/* (2') Authorization check: disabled for Action2 */
    action (authorization : none) Action2 [...]
/* (3) Authorization check: enabled for _Assoc1 */
    association _Assoc1 { create; }
/* (3') Authorization check: disabled for create by _Assoc2 */
    association _Assoc2 { create ( authorization : none); }
...
}
define behavior for ChildEntity [alias ChildAliasedName]
implementation in class ABAP_CLASS_FOR_CHILD [unique]
authorization dependent( key_field_of_child_entity = key_field_of_root_entity )
...
{
/* (4) Operations that are treated as an update operation on the authorization
master */
    update;
    delete;
/* (5) Authorization check: enabled for Action11 */
    action Action11 [...]
/* (5') Authorization check: disabled for Action12 */
    action (authorization : none) Action12 [...]
/* (6) Treated as an update operation on the authorization master */
    association _AssocNamell { create; }
...
}
```

The root entity is always defined as `authorization master`. To enable instance-based authorization control, the parameter `(instance)` is added to the definition of master. The authorization check can then be implemented in the local handler class of the corresponding behavior pool `ABAP_CLASS_FOR_ROOT`. More on this: [Implementing Authorization Control \[page 490\]](#)

For standard operations such as `update`, `delete`, as well as for `create by association` and actions, the authorization control is then checked by the business object runtime as soon as the relevant operation is executed (default behavior). For each relevant operation, you can specify the following values in the implementing handler of the class pool:

- [auth-allowed \[page 493\]](#) - if the consumer is allowed to execute the operation on the current instance
- [auth-unauthorized \[page 493\]](#) - if the consumer is not allowed to execute the operation on the current instance.

However, for selected operations you have the option of suppressing the authorization check execution so that the consumer can access them. To disable the authorization check, the parameter `(authorization : none)` must be added to the operation in question in the behavior definition.

With the current release, the root entity is always defined as `authorization master`, whereas all child entities are defined as `authorization dependent`. If a child entity is modified (`update`, `delete`, `create by association`) on that entity, the authorization check (that is implemented in the corresponding behavior class) of the master is triggered to check if the operation is allowed for being accessed.

The operations `update`, `delete`, `create by association` on child entities are treated as an update on the corresponding root entity (`authorization master`). Thus, the authorization check implementation is triggered to check the authorization for update at master level - despite of the fact that it was a `update`, `delete`, `create by association` request at dependent child entity level. In other words: `create by association`, `update` and `delete` operations on the authorization dependent child entity are checked on their `authorization master` entity as `update`. For example, the `delete` operation of `ChildEntity` instance invokes the `authorization` method of the `RootEntity` and checks the corresponding root instance with `update` operation.

However, actions on the `authorization dependent` child entity are checked by the authorization handler of the `authorization dependent` entity instead.

Example

In the following behavior definition, the `travel` (root) entity acts as `authorization master`, whereas the child entities `booking` and `booksuppl` are defined as `authorization dependent`.

To disable the authorization check for the `createBooking` action of the root entity, the parameter `(authorization : none)` is added to the action definition. The same applies to the `createBookingSupplement` action in the behavior definition of the `booking` entity.

```
managed;
define behavior for /DMO/I_Travel_M alias travel
...
authorization master(instance)
{
  create;
  update;
  delete;
  association Booking { create; }
  action ( authorization : none ) createBooking result [1] $self;
```

```

    ...
}

define behavior for /DMO/I_Booking_M alias booking
...
authorization dependent( travel_id = travel_id )
{
    update;
    delete;
    association _BookSupplement { create; }
    action ( authorization : none ) createBookingSupplement result [1] $self;
    ...
}
define behavior for /DMO/I_BookSuppl_M alias booksuppl
...
authorization dependent( travel_id = travel_id )

{
    update;
    ...
}

```

Related Information

[Implementing Authorization Control \[page 490\]](#)

7.11.2 Implementing Authorization Control

Implementing Instance-Based Authorization Control in the Handler Class of the Behavior Pool

Signature of Authorization Handler Method

```

CLASS lhc_authorization_handler DEFINITION INHERITING FROM
cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS check_authority_for_entity FOR AUTHORIZATION
    IMPORTING it_entity_key REQUEST is_request FOR entity
    RESULT rt_result.
ENDCLASS

```

The instance-based authorization control of a business object's entity is implemented in the behavior pool that is specified in the behavior definition by the keyword `implementation` in class `ABAP_CLASS [unique]`.

The implementation of authorization control is based on the ABAP language and is done in a local handler class (`lhc_authorization_handler`) as part of the behavior pool. As depicted in the listing below, each such local class inherits from the base handler class `cl_abap_behavior_handler`.

The signature of the handler method `check_authority_for_entity` (method name is freely selectable) is introduced by the keyword `FOR AUTHORIZATION`, followed by the input parameter `it_entity_key`, which specifies the set of keys of entity instances that are included in the consumer's authorization request is request.

i Note

Example for authorization request parameter is request:

```
REQUEST is_request FOR travel RESULT result.
```

```
▫ is_request type structure for authorization request /dmo/i_travel_m\travel  
%update type abp_behv_flag  
%action  
    accepttravel type abp_behv_flag  
    rejecttravel type abp_behv_flag  
    createtravelbytemplate type abp_behv_flag  
%assoc
```

F2 information on authorization request

The output parameter `rt_result` (parameter name is freely selectable) is used to return a result containing information whether the consumer is allowed using the relevant operation for the current instance or not.

The output parameters failed and reported for errors or messages are added implicitly (automatically). They can, however, be declared explicitly as CHANGING parameters in the method signature using the generic type DATA:

CHANGING failed TYPE DATA
 reported TYPE DATA.

Template for Implementing the Authorization Handler Method

```

CLASS lhc_authorization_handler IMPLEMENTATION.
...
METHOD check_authority_for_entity.
  DATA ls_result LIKE LINE OF rt_result.
  LOOP AT it_entity_key INTO DATA(ls_entity_key).
    ls_result = VALUE #( entity_key           = ls_entity_key-key
                         %update            = if_abap_behv=>auth-allowed
                         %delete            = if_abap_behv=>auth-
unauthorized
                         %action-Action1     = if_abap_behv=>auth-allowed
                         ...
                         %action-ActionN     = if_abap_behv=>auth-unauthorized
                         %assoc-_Association = if_abap_behv=>auth-allowed
                       ).
    APPEND ls_result to rt_result.
  ENDLOOP.
ENDMETHOD.
...
ENDCLASS.

```

The authorization method is implemented in a generic manner. For each instance, the output parameter `rt_result` contains the result with information whether the consumer is allowed using the relevant operation for the current instance or not.

In general, it contains the following components:

- key fields
- %update - (for authorization master only)
- %delete – (for authorization master only)
- %action – action_xyz
- %assoc – association_abc (for authorization master only)

The components in the result depend on the consumer's authorization request. If an operation is disabled for authorization check by adding the parameter (authorization : none) to the operation in question, then it is neither included in the authorization request nor in the output parameter.

i Note

Example of a result table type rt_result for an authorization master.

The screenshot shows the SAP ABAP F2 information for the output parameter of the rt_result type. The title bar says 'RESULT result'. The main area displays the following table:

F2 information for the output parameter	
◀▶ result	type table for authorization result /dmo/i_travel_m\travel optional
travel_id	type /dmo/travel_id
%update	type abp_behv_flag
%action	
accepttravel	type abp_behv_flag
rejecttravel	type abp_behv_flag
createtravelbytemplate	type abp_behv_flag
%assoc	
_booking	type abp_behv_flag

Example

UI Preview

The following figure shows the effect of the authorization control for an action that is triggered by the consumer using the *Reject Travel* button (the consumer is not authorized to trigger the action).

The screenshot shows a travel list with the following details:

Travels (9) Standard ▾				
Travel ID	Overall Status	Agency ID	Customer ID	Booking Fee
2	O	Sunshine Travel (70001)	Lautenbach (591)	232.00 EUR >

Total Price: 16,784.46 EUR
Description: Business Trip 1234

The 'Reject Travel' button in the header is highlighted with a yellow box.

Accessing an action which the consumer is not allowed to run

Reject Travel

2

ⓘ Action execution not possible; no authority

The BO runtime returns a generic message

Definition

In the behavior definition, the authorization control for the root entity may be defined as follows:

```
managed;
define behavior for /DMO/I_Travel_M alias travel
implementation in class /DMO/BP_TRAVEL_M unique
...
authorization master(instance)
{
    ...
    create;
    update;
    delete ( authorization : none );
    action acceptTravel result [1] $self;
    action rejectTravel result [1] $self;
    action createTravelByTemplate result [1] $self;
    action (authorization : none) createBooking result [1] $self;
    association _Booking { create; }

}
```

Implementation

The following example shows the implementation of authorization control for various operations to be executed on `travel` instances:

The authorization control is checked by the business object runtime as soon as the relevant operation is executed. For each relevant operation, the authorization check must be enabled so that one of the following values can be assigned in the implementing handler method:

- `if_abap_behv=>auth-allowed` - the consumer is allowed to execute the operation on the current instance (for example: `update` operation)
- `if_abap_behv=>auth-unauthorized` - the consumer is not allowed to execute the operation on the current instance (for example: `rejectTravel` action).

Corresponding to the behavior definition in our example, the authorization check is disabled for `delete` operation and `createBooking` action (`authorization : none`). The consumer can access these operations without any authorization check execution.

Listing: Implementing Authorization Control in a Local Handler Class

```
CLASS lhc_auth_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.
PRIVATE SECTION.
METHODS check_authority_for_travel FOR AUTHORIZATION
    IMPORTING it_travel_key REQUEST is_request FOR travel
```

```

        RESULT rt_result.
ENDCLASS.
CLASS lhc_auth_handler IMPLEMENTATION.
...
*****  

*  

* Implements what operations and actions are (not) allowed for travel instances  

*  

*****  

METHOD check_authority_for_travel.
    DATA ls_result LIKE LINE OF rt_result.
    LOOP AT it_travel_key INTO DATA(ls_travel_key).
        ls_result = VALUE #( travel_id
            travel_id
            allowed      "Default setting
            unauthorized
            allowed
            unauthorized
            allowed
        ) .
        APPEND ls_result to rt_result.

    ENDLOOP.
ENDMETHOD.
...
ENDCLASS.

```

7.12 Implementing an Unmanaged Query

An unmanaged query uses an ABAP interface to implement read-only access to persistent or non-persistent data. It enables a more flexible data retrieval than using the SQL push down by the query framework to retrieve data from a database table.

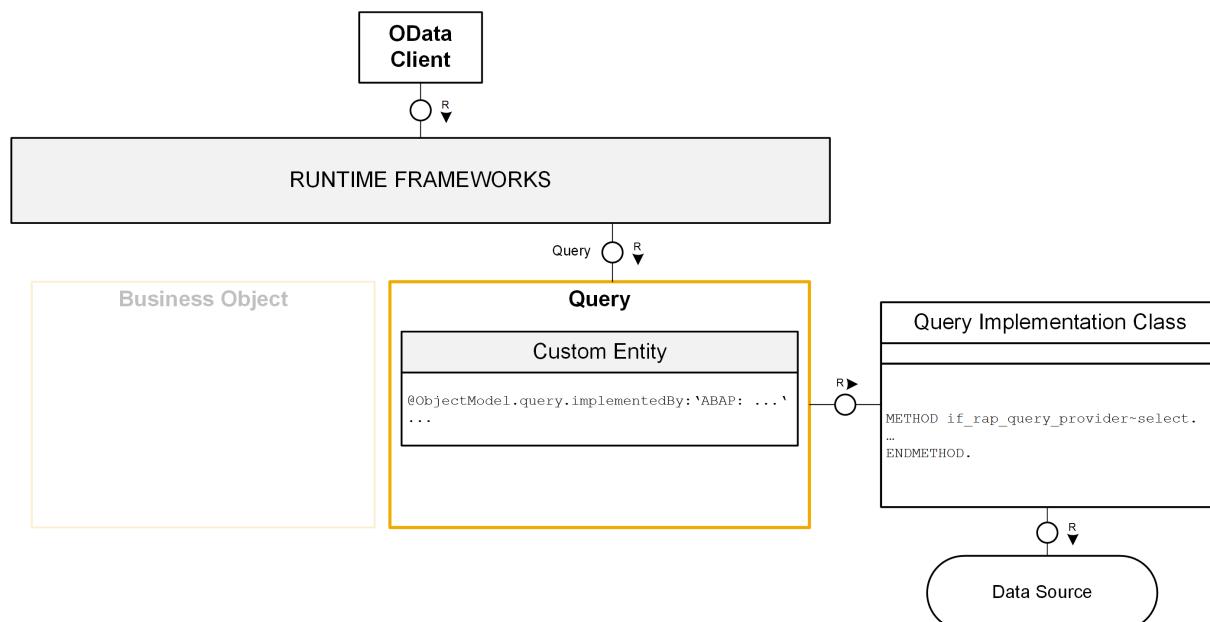
Context

An unmanaged query is implemented for read-only access to a data source whenever the standard SQL push-down by the query framework is not sufficient or not usable at all; or if there is no persistent data source at all.

Use cases for unmanaged queries are

- the data source for an OData request is not a database table, but, for example another OData service, which is reached by an OData client proxy,
- performance optimization with application specific handling,
- using AMDPs with some query push-down parameters in the SQL script implementation,
- forwarding the call to the analytical engines, or
- enrichment of query result data on property or row level, for example when splitting rows for intermediate sums or condensing the filter result.

In these cases, the data model is defined in a custom entity, which references a query implementation class, where the query is implemented using a query provider interface. The runtime of an unmanaged query is illustrated in the following diagram.



For more background information about the unmanaged query and the custom entity, see [Unmanaged Query \[page 51\]](#).

Example Scenario

The following sections offer an example for the implementation of the query provider interface. It is aimed to give an understanding on how to work with the interface `IF_RAP_QUERY_PROVIDER`. The example query implementation retrieves data from a database table, which is not a typical use case.

i Note

Do not use the custom query in the straight-forward case of retrieving data from a database table. The example is only used for demonstration purposes, as no background information about another technology (for example in AMDP implementations) is necessary to understand the example. The recommended implementation for such a scenario is to use a CDS view and the underlying query implementation of the SQL select by the orchestration framework.

The data model is defined in a custom entity. Expand the following codeblock to view the data model of the example scenario. For simplification reasons, the same element names as in the data source are used. (If you use differing names, you must map the elements of the custom entity to the corresponding table fields in the query implementation. Make sure that the types are compatible.)

i Expand the following listing to view the source code of the travel CDS view that is used for the demo example.

Custom Entity /DMO/I_TRAVEL_U

```
@EndUserText.label: 'Custom entity for unmanaged travel query'
```

```

@ObjectModel.query.implementedBy:'ABAP:/dmo/cl_travel_uq'
define custom entity /DMO/I_TRAVEL_UQ
{
  key Travel_ID      : abap.numc( 8 );
  Agency_ID         : abap.numc( 6 );
  Customer_ID       : abap.numc( 6 );
  Begin_Date        : abap.dats;
  End_Date          : abap.dats;
  Booking_Fee       : abap.dec( 17, 3 );
  Total_Price       : abap.dec( 17, 3 );
  Currency_Code    : abap.cuky;
  Status            : abap.char( 1 );
  LastChangedAt     : timestamppl;
}

```

The annotation `@ObjectModel.queryImplementedBy` references the query implementation class. Learn how you implement the unmanaged query for this example scenario in the following section.

The data source in the example scenario is the database table `/dmo/travel`, see [ABAP Flight Reference Scenario \[page 774\]](#).

Implementation

Every method provided by `IF_RAP_QUERY_PROVIDER` is used and explained in this implementation. The complete source code of the query implementation with every method is displayed after the implementation steps.

Prerequisites

- The custom entity references the query implementation class in the annotation `@ObjectModel.queryImplementedBy`.
- The query implementation class implements the interface `IF_RAP_QUERY_PROVIDER` with its `select` method.

i Note

To avoid high resource consumption when calling an OData service without using `$top` each OData service, based on the ABAP RESTful Programming Model uses a default paging. This includes

- Setting a default paging if the client does not provide `$top`
- Reducing the response to the default limit if `$top` exceeds this limit
- Adding a `__next` link with a `skiptoken` to the response if the response represents a partial listing, which is a subset of all available data records. See <https://www.odata.org/documentation/odata-version-2-0/json-format/> → 6. Representing Collections of Entries.

i Note

In scenarios, in which you expose your custom entity for a Fiori UI, you have to include at least the implementation for counting and paging as the UI always requests the count and sets the query options

`$top` and `$skip` for paging. If the corresponding methods are not implemented and the unmanaged query does not return the respective information, there will be an error during runtime.

Implementation Steps

- Check that the query is only executed when the requested entity set matches the custom entity.
For implementation details, see [Returning Requested Entity in an Unmanaged Query \[page 499\]](#).
- Separate your implementation for data retrieval and count.
For implementation details, see [Requesting and Setting Data or Count in an Unmanaged Query \[page 500\]](#).
- Implement filter conditions according to
 - a requested filter,
For implementation details, see [Implementing Filtering in an Unmanaged Query \[page 501\]](#)
 - a requested search term,
For implementation details, see [Implementing Search in an Unmanaged Query \[page 505\]](#)
 - requested parameters.
For implementation details, see [Using Parameters in an Unmanaged Query \[page 503\]](#).
- Get the paging information and retrieve data according to the requested page.
For implementation details, see [Implementing Paging in an Unmanaged Query \[page 507\]](#).
- Get the sorting information and order the retrieved data according to the sort elements and direction.
For implementation details, see [Implementing Sorting in an Unmanaged Query \[page 509\]](#).
- Get the requested elements and select only the relevant elements from the data source.
For implementation details, see [Considering Requested Elements in an Unmanaged Query \[page 510\]](#).
- Get the aggregated and the grouped elements and aggregated and group the records accordingly.
For implementation details, see [Implementing Aggregations in an Unmanaged Query \[page 511\]](#).

i Expand the following listing to view the source code of query implementation class for the demo example.

Query Implementation Class /dmo/cl_travel_uq c

```
CLASS /dmo/cl_travel_uq DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .
  PUBLIC SECTION.
    INTERFACES if_rap_query_provider.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS /dmo/cl_travel_uq IMPLEMENTATION.
  METHOD if_rap_query_provider~select.
    TRY.
      CASE io_request->get_entity_id( ).
        WHEN 7DMO/I_TRAVEL_UQ` .
      **query implementation for travel entity
      **filter
        DATA(lv_sql_filter) = io_request->get_filter( )-
>get_as_sql_string(`).
        TRY.
          DATA(lt_filter) = io_request->get_filter( )->get_as_ranges( ).
          CATCH cx_rap_query_filter_no_range.
            "handle exception
          ENDTRY.
      **parameters
```

```

        DATA(lt_parameters) = io_request->get_parameters( ).
        DATA(lv_next_year) = CONV
syst_datum( cl_abap_context_info->get_system_date( ) + 365 ) .
        DATA(lv_par_filter) = | BEGIN_DATE >=
'{ cl_abap_dyn_prg=>escape_quotes( VALUE #( lt_parameters[ parameter_name =
'P_START_DATE' ]-value

        DEFAULT cl_abap_context_info->get_system_date( ) ) )' | &&
                           | AND | &&
                           | END_DATE <=
'{ cl_abap_dyn_prg=>escape_quotes( VALUE #( lt_parameters[ parameter_name =
'P_END_DATE' ]-value

        DEFAULT lv_next_year ) )' | .
        IF lv_sql_filter IS INITIAL.
            lv_sql_filter = lv_par_filter.
        ELSE.
            lv_sql_filter = |({ lv_sql_filter } AND { lv_par_filter } )| .
        ENDIF.

**search
        DATA(lv_search_string) = io_request->get_search_expression( ).
        DATA(lv_search_sql) = |DESCRIPTION LIKE '%
{ cl_abap_dyn_prg=>escape_quotes( lv_search_string ) }%'|.
        IF lv_sql_filter IS INITIAL.
            lv_sql_filter = lv_search_sql.
        ELSE.
            lv_sql_filter = |( { lv_sql_filter } AND { lv_search_sql } )| .
        ENDIF.

**request data
        IF io_request->is_data_requested( ).

**paging
        DATA(lv_offset) = io_request->get.paging()->get_offset( ).
        DATA(lv_page_size) = io_request->get.paging()->get_page_size( ).
        DATA(lv_max_rows) = COND #( WHEN lv_page_size =
if_rap_query_paging=>page_size_unlimited
                                         THEN 0 ELSE lv_page_size ).

**sorting
        DATA(sort_elements) = io_request->get_sort_elements( ).
        DATA(lt_sort_criteria) = VALUE string_table( FOR sort_element IN
sort_elements
                                         ( sort_element-
element_name && COND #( WHEN sort_element-descending = abap_true THEN `

descending` ELSE ` ascending` ) ) .
        DATA(lv_sort_string) = COND #( WHEN lt_sort_criteria IS INITIAL
THEN `primary key` .

ELSE concat_lines_of( table = lt_sort_criteria sep = `, ` ) .

**requested_elements
        DATA(lt_req_elements) = io_request->get_requested_elements( ).

**aggregate
        DATA(lt_aggr_element) = io_request->get_aggregation( )-
>get_aggregated_elements( ).
        IF lt_aggr_element IS NOT INITIAL.
            LOOP AT lt_aggr_element ASSIGNING FIELD-
SYMBOL(<fs_aggr_element>).
                DELETE lt_req_elements WHERE table_line = <fs_aggr_element>-
result_element.
                DATA(lv_aggregation) = |{ <fs_aggr_element>-
aggregation_method }| ( { <fs_aggr_element>-input_element } ) as
{ <fs_aggr_element>-result_element }|.
                APPEND lv_aggregation TO lt_req_elements.
            ENDLOOP.
        ENDIF.
        DATA(lv_req_elements) = concat_lines_of( table = lt_req_elements
sep = `, ` ) .

****grouping

```

```

        DATA(lt_grouped_element) = io_request->get_aggregation( )-
>get_grouped_elements( ).
        DATA(lv_grouping) = concat_lines_of(  table = lt_grouped_element
sep = ` , ` ).
**select data
        DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.
        SELECT (lv_req_elements) FROM /dmo/travel
                WHERE (lv_sql_filter)
                GROUP BY (lv_grouping)
                ORDER BY (lv_sort_string)
                INTO CORRESPONDING FIELDS OF TABLE
@lt_travel_response
                OFFSET @lv_offset UP TO @lv_max_rows ROWS.

**fill response
        io_response->set_data( lt_travel_response ).
ENDIF.
**request count
        IF io_request->is_total_numb_of_rec_requested( ). .
**select count
        SELECT COUNT( * ) FROM /dmo/travel
                WHERE (lv_sql_filter)
                INTO @DATA(lv_travel_count).

**fill response
        io_response->set_total_number_of_records( lv_travel_count ). .
ENDIF.
WHEN `/DMO/I_BOOKING_UQ`.
**query implementation for booking entity
ENDCASE.
CATCH cx_rap_query_provider.
ENDTRY.
ENDMETHOD.
ENDCLASS.
```

7.12.1 Returning Requested Entity in an Unmanaged Query

Getting the requested entity ID into your query implementation class can be helpful to ensure that the query is only executed if a specific entity is queried. You can also differentiate between query implementations of different custom entities in one query implementation class. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the entity ID, which is requested.

Prerequisites

For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).

Implementation Steps

The following steps provide an example on how to use the method `get_entity_id` in your query implementation class.

1. Call the method `get_entity_id` of the interface `IF_RAP_QUERY_REQUEST`, which returns the requested CDS entity name.
2. Use the returned value to compare to the custom entity the query implementation is aimed at, or to define query implementation for different custom entities in one query implementation class.

The following codeblock illustrates the implementation within the `SELECT` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```
CASE io_request->get_entity_id( ).  
  WHEN `/DMO/I_TRAVEL_UQ` .  
  ****query implementation for travel entity  
    WHEN `/DMO/I_BOOKING_UQ` .  
  ****query implementation for booking entity  
ENDCASE.
```

For API information, see [Method `get_entity_id` \[page 744\]](#).

7.12.2 Requesting and Setting Data or Count in an Unmanaged Query

The interface `IF_RAP_QUERY_REQUEST` provides methods to indicate whether data or the count is requested. These methods can be used to separate the implementations for data retrieval and count or to ensure that the query implementation is only executed if the respective request is made.

The interface `IF_RAP_QUERY_RESPONSE` provides methods to return the requested data or the count as a response for the request.

i Note

If data is requested, the method `set_data` must be called. If the total number of records is requested, the method `set_total_number_of_records` must be called. Otherwise there will be an error during runtime.

In UI scenarios with Fiori Elements the total number of records is always requested by the UI.

Prerequisites

For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).

Implementation Steps

Requesting and Setting Data

The following steps provide an example on how to use the method `is_data_requested` in your query implementation class.

1. Call the method `is_data_requested` of the interface `IF_RAP_QUERY_REQUEST`, which returns a boolean value.
2. Create the SQL `SELECT` to retrieve the requested data into a local variable.
3. Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` and use the retrieved data as importing parameter. The result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the implementation for data requests within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class..

```
IF io_request->is_data_requested( ).  
  DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.  
    SELECT * FROM /dmo/travel  
      INTO CORRESPONDING FIELDS OF TABLE @lt_travel_response.  
    io_response->set_data( lt_travel_response ).  
ENDIF.
```

For API information, see [Method `is_data_requested` \[page 744\]](#) and [Method `set_data` \[page 755\]](#).

Requesting and Setting Count

The following steps provide an example on how to use the method `is_total numb_of_rec_requested` in your query implementation class.

1. Call the method `is_total numb_of_rec_requested` of the interface `IF_RAP_QUERY_REQUEST`, which returns a boolean value.
2. Create the SQL `SELECT` to retrieve the requested count into a local variable.
3. Call the method `set_total_number_of_records` of the interface `IF_RAP_QUERY_RESPONSE` and use the retrieved data as importing parameter. The count is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the implementation for data requests within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class. .

```
IF io_request->is_total_numb_of_rec_requested( ).  
  SELECT COUNT( * ) FROM /dmo/travel  
    INTO @DATA(lv_travel_count).  
  io_response->set_total_number_of_records( lv_travel_count ).  
ENDIF.
```

For API information, see [Method `is_total_numb_of_rec_requested` \[page 745\]](#) and [Method `set_total_number_of_records` \[page 755\]](#).

7.12.3 Implementing Filtering in an Unmanaged Query

To retrieve filtered data or a filtered number of records in an unmanaged query, you need to implement a filter in the query implementation class. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the filter for the request with two options to retrieve the filter conditions:

- Get the filter condition as an SQL string.
- Get the filter condition in a ranges table.

Depending on your use case, the one or the other option is more useful. If you retrieve your data from the data source with an `SQL SELECT`, you can include the SQL filter string directly in the `WHERE` clause of the `SELECT` statement. If you want to manipulate the filter conditions before executing the filter, a ranges table can be the better choice.

i Note

If the filter is not feasible as a ranges table, an exception is raised. You then have to handle the error appropriately

Prerequisites

For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).

Implementation Steps

Using the Filter as an SQL String

The following steps provide an example on how to use the method `get_as_sql_string` in your query implementation class.

1. Call the method `get_filter` of the interface `IF_RAP_QUERY_REQUEST`, which returns an interface instance of `IF_RAP_QUERY_FILTER`. Use the method `get_as_sql_string` to use the filter directly in an SQL string.
2. Check if data is requested.
3. Use the filter condition in the `WHERE` clause of the SQL statement to retrieve the filtered data from the data source.
4. Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the filtered data. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.
5. Check if count is requested.
6. Use the filter condition in the `WHERE` clause of the SQL statement to retrieve the count for the data records that match the request.
7. Call the method `set_total_number_of_records` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the filtered count. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the filter implementation within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```
DATA(lv_sql_filter) = io_request->get_filter( )->get_as_sql_string( ).  
IF io_request->is_data_requested( ).  
  DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.  
  SELECT * FROM /dmo/travel  
    WHERE (lv_sql_filter)  
    ORDER BY ('primary key')  
    INTO CORRESPONDING FIELDS OF TABLE @lt_travel_response.
```

```

    io_response->set_data( lt_travel_response ) .
ENDIF.
IF io_request->is_total numb_of_rec_requested( ) .
  SELECT COUNT( * ) FROM /dmo/travel
    WHERE (lv_sql_filter)
    INTO @DATA(lv_travel_count).
  io_response->set_total_number_of_records( lv_travel_count ) .
ENDIF.

```

i Note

It is recommended to implement a default sort order to return consistent results from the data source.

For API information, see [Method get_filter \[page 745\]](#).

Getting the Filter as Ranges Table

The following steps provide an example on how to use the method `get_as_ranges` in your query implementation class.

1. In a TRY – CATCH block, call the method `get_filter` of the interface `IF_RAP_QUERY_REQUEST`, which returns an interface instance of `IF_RAP_QUERY_FILTER`. Use the method `get_as_ranges` to get the filter as a ranges table. The format of the returning ranges table is described in [Method get_as_ranges \[page 749\]](#).
2. Use the filter condition of the ranges table in your implementation.
3. Catch the exception `CX_RAP_QUERY_FILTER_NO_RANGE`, which is raised if the filter cannot be expressed as a ranges table.
4. Handle the exception appropriately. For example:
 1. Throw an error.
 2. Use the filter SQL string as a fall back, see [Method get_as_sql_string \[page 751\]](#)

```

TRY.
  DATA(lt_ranges) = io_request->get_filter( )->get_as_ranges( ) .
*****filter manipulation
  CATCH cx_rap_query_filter_no_range.
    ""error handling
ENDTRY.

```

For API information, see [Method get_filter \[page 745\]](#).

7.12.4 Using Parameters in an Unmanaged Query

To retrieve data dependent on an entity parameter that is set in the custom entity, you need to implement a handling for the parameters in the query implementation class. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the parameters. It returns a string table with the parameter names and values.

It is up to the application developer how to implement the parameter logic. One option is to implement the parameters as filter criteria. To use the parameter values as a filter in the `WHERE` clause of an SQL `SELECT` statement, you need to create a filter string from the parameter values.

The interface `IF_RAP_QUERY_RESPONSE` provides a method to set the filtered data or the filtered total number of records for the query response after retrieving data from the data source.

Prerequisites

- The custom entity has one or more entity parameters.

Example

```
define custom entity /DMO/I_Travel_UQ
with parameters p_start_date : /dmo/begin_date,
      p_end_date   : /dmo/end_date
```

- For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).

Implementation Steps

The following steps describe the procedure of using parameters as additional filter criteria. The parameters `p_start_date` and `p_end_date` are used as a filter on the elements `Begin_Date` and `End_Date`.

- Call the method `get_parameters` of the interface `IF_RAP_QUERY_REQUEST`, which returns a string table of the parameters and their values.
- Define default parameter values in case the parameters are not given in the request. The implementation example uses the system date for `Begin_Date` and the system date the following year for `End_Date`.
- Define a variable for the filter string and fill it with the filter condition for the SQL `WHERE` clause on the element `Begin_Date` and `End_Date` and integrate the default values.

i Note

To avoid security risks via SQL string injections, use the method `escape_quotes` of the public class `CL_ABAP_DYN_PRG`.

- If there are other filter conditions (from filter requests or parameters), concatenate the filter strings with `AND`.
- Check if data is requested.
- Use the filter condition in the `WHERE` clause of the SQL statement to retrieve the filtered data from the data source.
- Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the filtered data. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.
- Check if count is requested.
- Use the filter condition in the `WHERE` clause of the SQL statement to retrieve the count for the data records that match the request.
- Call the method `set_total_number_of_records` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the filtered count. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates an implementation for parameters within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```
DATA(lt_parameters) = io_request->get_parameters( ).
```

```

DATA(lv_next_year) =  CONV
syst_datum( cl_abap_context_info->get_system_date( ) + 365 ) .
  DATA(lv_par_filter) = |( BEGIN_DATE >=
'{"cl_abap_dyn_prg=>escape_quotes("VALUE #( lt_parameters[ parameter_name =
'P_START_DATE' ]-value

      DEFAULT cl_abap_context_info->get_system_date( ) ) }'| &&
      | AND | &&
      | END_DATE <= '{ cl_abap_dyn_prg=>escape_quotes( VALUE
#( lt_parameters[ parameter_name = 'P_END_DATE' ]-value

DEFAULT lv_next_year ) ) }'| .
IF lv_sql_filter IS INITIAL.
  lv_sql_filter = lv_par_filter.
ELSE.
  lv_sql_filter = |( { lv_sql_filter } AND { lv_par_filter } )| .
ENDIF.
IF io_request->is_data_requested( ).
  DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.
  SELECT * FROM /dmo/travel
    WHERE (lv_sql_filter)
    ORDER BY ('primary key')
    INTO CORRESPONDING FIELDS OF TABLE @lt_travel_response
  io_response->set_data( lt_travel_response ).
ENDIF.
IF io_request->is_total numb_of_rec_requested( ).
  SELECT COUNT( * ) FROM /dmo/travel
    WHERE (lv_sql_filter)
    INTO @DATA(lv_travel_count).
  io_response->set_total_number_of_records( lv_travel_count ).
ENDIF.

```

i Note

It is recommended to implement a default sort order to return consistent results from the data source.

For API information, see [Method get_parameters \[page 747\]](#).

7.12.5 Implementing Search in an Unmanaged Query

To retrieve data according to the search term in the OData request, you need to implement a search logic in your query implementation class. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the search expression from the request.

It is up to the application developer how the search logic is implemented. One option is to use the search expression as (additional) filter criteria for one or more elements. To use the search expression as a filter in the `WHERE` clause of an SQL `SELECT` statement, you need to create a filter string from the search expression and combine it with other possible filter strings.

The interface `IF_RAP_QUERY_RESPONSE` provides a method to set the filtered data for the query response after retrieving data from the data source.

Prerequisites

- For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).
- To send query requests with search conditions from a Fiori Elements UI, you need to annotate the custom entity with `@Search.searchable:true`.

i Note

The annotation `@Search.searchable: true` requires using the annotation `@Search.defaultSearchElement: true` on at least one element in the custom entity. This element annotation does not have any influence on the search, as it is up to the application developer on which element(s) the search logic is implemented.

Implementation Steps

The following steps provide an example on how to implement the search as a filter for the element `Description`. If a filter is also requested, you need to ensure that the SQL filter string has the right syntax for the SQL `SELECT` statement.

1. Check if data is requested.
2. Call the method `get_search_expression` of the interface `IF_RAP_QUERY_REQUEST`, which returns a string of the requested search expression.
3. Create the filter string for the `WHERE` clause of SQL `SELECT` statement with the filter for the element `Description`.

i Note

To avoid security risks via SQL string injections use the method `escape_quotes` of the public class `CL_ABAP_DYN_PRG`.

4. If there are other filter conditions (from filter requests or parameters), concatenate the filter strings with `AND`.
5. Check if data is requested.
6. Use the filter condition in the `WHERE` clause of the SQL statement to retrieve the filtered data from the data source.
7. Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the filtered data. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.
8. Check if count is requested.
9. Use the filter condition in the `WHERE` clause of the SQL statement to retrieve the count for the data records that match the request.
10. Call the method `set_total_number_of_records` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the filtered count. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the search implementation within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```
DATA(lv_search_string) = io_request->get_search_expression( ).  
DATA(lv_search_sql) = '|DESCRIPTION LIKE '%  
{ cl_abap_dyn_prg=>escape_quotes( lv_search_string ) }%'|.  
IF lv_sql_filter IS INITIAL.  
    lv_sql_filter = lv_search_sql.  
ELSE.  
    lv_sql_filter = |( { lv_sql_filter } AND { lv_search_sql } )|.br/>ENDIF.  
IF io_request->is_data_requested( ).  
    DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.  
    SELECT * FROM /dmo/travel  
        WHERE (lv_sql_filter)  
        ORDER BY ('primary key')  
        INTO CORRESPONDING FIELDS OF TABLE @lt_travel_response  
    io_response->set_data( lt_travel_response ).  
ENDIF.  
IF io_request->is_total_numb_of_rec_requested( ).  
    SELECT COUNT( * ) FROM /dmo/travel  
        WHERE (lv_sql_filter)  
        INTO @DATA(lv_travel_count).  
    io_response->set_total_number_of_records( lv_travel_count ).  
ENDIF.
```

i Note

It is recommended to implement a default sort order to return consistent results from the data source.

For API information, see [Method `get_search_expression` \[page 748\]](#).

7.12.6 Implementing Paging in an Unmanaged Query

To retrieve data in packages, you need to implement paging in the query implementation class. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the paging information. It returns an interface instance of `IF_RAP_QUERY_PAGING` with two methods for the beginning and the number of records to be retrieved.

The method `get_offset` defines the number of records that are dropped.

i Note

In accordance to the OData query option `$skip`, the method `get_offset` does not return the position of the first data record to retrieve, but the number of records that are not taken into account before the retrieval starts. That means, the first line of the data source that is retrieved is the returning value of `get_offset` plus 1.

i Note

For retrieving all available data records, the method `get_page_size` returns the constant `page_size_unlimited` of the interface `IF_RAP_QUERY_PRAGING`. This has to be converted when using the paging information in an SQL string.

The method `get_page_size` defines the number of records that are retrieved.

The paging information can then be used in the `OFFSET` and the `UP TO n ROWS` clause of the SQL `SELECT` statement.

The interface `IF_RAP_QUERY_RESPONSE` provides a method to set the reduced data records for the query response after retrieving data from the data source.

Prerequisites

For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#)

Implementation Steps

The following steps provide an example on how to implement paging.

1. Call the method `get_paging` of the interface `IF_RAP_QUERY_REQUEST`, which returns an interface instance of `IF_RAP_QUERY_PAGING`. Use the method `get_offset` to get the number of records to drop into a local variable.
2. Call the method `get_paging` of the interface `IF_RAP_QUERY_REQUEST`, which returns an interface instance of `IF_RAP_QUERY_PAGING`. Use the method `get_page_size` to get the number of records to retrieve.
3. Convert the number for infinite numbers of records to be compatible with the definition of the SQL `SELECT` statement to retrieve an infinite number.

i Note

The SQL `SELECT` addition `UP TO 0 ROWS` retrieves all available data sets.

4. Use the additions `OFFSET` and `UP TO n ROWS` in the SQL `SELECT` to retrieve the data records in packages.
5. Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the reduced data records. The filtered result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the paging implementation within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```
IF io_request->is_data_requested( ).  
  DATA(lv_offset) = io_request->get_paging()->get_offset( ).  
  DATA(lv_page_size) = io_request->get_paging()->get_page_size( ).  
  DATA(lv_max_rows) = COND #( WHEN lv_page_size =  
    if_rap_query_paging->page_size_unlimited THEN 0  
      ELSE lv_page_size .. ).  
  DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.  
  SELECT * FROM /dmo/travel  
    ORDER BY ('primary key')  
    INTO CORRESPONDING FIELDS OF TABLE @lt_travel_response  
    OFFSET @lv_offset UP TO @lv_max_rows ROWS.  
  io_response->set_data( lt_travel_response ).  
ENDIF.
```

i Note

It is required to implement a default sort order to return consistent results from the data source. Sorted results are essential in combination with paging. If you do not provide a default order, the data records for a certain page might not be consistent.

For API information, see [Method `get_paging` \[page 746\]](#).

7.12.7 Implementing Sorting in an Unmanaged Query

To retrieve sorted data in an unmanaged query, you need to add sorting criteria to the SQL `SELECT` statement. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the sort element and the sort order for the request.

The method returns an ordered list of elements with their sort order. If there is more than one sort element, the sorting priority is in order of appearance. To use the sorting criteria in an SQL `SELECT` clause, the sorting criteria has to be transformed into a string that has comma-separated pairs of sort element and sort order.

Whereas `abap.bool` indicates the sort order for the element `descending` in the sorted table that you get from the query request, the sort order must be indicated with the string `ascending` or `descending` in the `ORDER BY` clause of the SQL statement.

• Example

The string for the SQL select statement must look like `element1 ascending, element2 descending, ...`.

The interface `IF_RAP_QUERY_RESPONSE` provides a method to set the sorted data for the query response after retrieving the data records from the data source.

Prerequisites

For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).

Implementation Steps

The following steps provide an example on how to implement sorting criteria.

1. Call the method `get_sort_elements` of the interface `IF_RAP_QUERY_REQUEST`, which returns an ordered list of sort elements with the respective sort direction.
2. Write the elements of the returning value into a string table and map `abap.bool` to '`ascending`' or '`descending`' respectively.

3. Concatenate the lines of the string table with comma separation into a string variable.
4. To achieve a consistent result if there is a query request with initial sort order, provide a default sort order, for example 'primary key'.
5. Use the sort condition in the ORDER BY clause of the SQL select.
6. Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the sorted data. The sorted result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the sort implementation within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```

    IF io_request->is_data_requested( ).
      DATA(sort_elements) = io_request->get_sort_elements( ).
      DATA(lt_sort_criteria) = VALUE string_table( FOR sort_element IN
sort_elements
                                         ( sort_element-element_name &&
COND #( WHEN sort_element-descending = abap_true
      THEN ` descending`
      ELSE ` ascending` ) ) .
      DATA(lv_sort_string) = COND #( WHEN lt_sort_criteria IS INITIAL THEN
`primary key`
                                         ELSE
concat_lines_of( table = lt_sort_criteria sep = `, ` ) ) .
      DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.
      SELECT * FROM /dmo/travel
        ORDER BY (lv_sort_string)
        INTO CORRESPONDING FIELDS OF TABLE @lt_travel_response.
      io_response->set_data( lt_travel_response ).
ENDIF.

```

For API information, see [Method `get_sort_elements` \[page 746\]](#).

7.12.8 Considering Requested Elements in an Unmanaged Query

You can optimize the performance for your unmanaged query you can add an implementation to retrieve only the elements that are requested in the OData request. If you do not specify requested element, the query retrieves the value for every element in the custom entity.

To retrieve only the elements that are requested in the OData request, you need to implement an element restriction in the SQL `SELECT` statement. The interface `IF_RAP_QUERY_PROVIDER` provides a method to get the requested elements for the request.

To select only the requested element in the SQL `SELECT` clause, you must transform the requested elements into a string with comma separations.

The interface `IF_RAP_QUERY_RESPONSE` provides a method to set the data for the query response after retrieving the relevant data from the data source.

Prerequisites

For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).

Implementation Steps

The following steps provide an example on how to select only the requested elements from a data source.

1. Check if data is requested.
2. Call the method `get_requested_elements` of the interface `IF_RAP_QUERY_REQUEST`, which returns a table with the requested elements.
3. Concatenate the lines of the table with comma separation into a string variable.
4. Include the string variable in the SQL `SELECT` clause to retrieve only the requested elements from the data source.
5. Call the method `set_data` of the interface `IF_RAP_QUERY_RESPONSE` to respond to the OData request with the requested elements. The result is then returned to the OData client, for example the SAP Fiori Elements UI.

The following codeblock illustrates the implementation to retrieve only the requested elements within the `select` method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```
IF io_request->is_data_requested( ).  
  DATA(lt_req_elements) = io_request->get_requested_elements( ).  
  DATA(lv_req_elements)  = concat_lines_of( table = lt_req_elements sep = `,  
  ).  
  DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.  
  SELECT (lv_req_elements) FROM /dmo/travel  
    ORDER BY ('primary key')  
    INTO CORRESPONDING FIELDS OF TABLE  
@lt_travel_response.  
  io_response->set_data( lt_travel_response ).  
ENDIF.
```

i Note

It is recommended to implement a default sort order to return consistent results from the data source.

For API information, see [Method `get_requested_elements` \[page 749\]](#).

7.12.9 Implementing Aggregations in an Unmanaged Query

To retrieve aggregate data, you need to implement a logic to retrieve and group data according to the requested aggregations in the OData request. The interface `IF_RAP_QUERY_REQUEST` provides a method to get the aggregation information. It returns an interface instance of `IF_RAP_QUERY_AGGREGATION` with two methods to get the elements to be aggregated or grouped.

To select the requested elements according to the requested aggregation, you need to transform the requested aggregation elements into an aggregation string for the SQL SELECT. The aggregation string must look as follows:

```
<AGGR_METHOD> ( <aggr_element> ) as <result_element>
```

For more information about aggregate functions in SQL expressions, see [ABAP SQL -Aggregate Expressions agg_exp \(ABAP Keyword Documentation\)](#).

To avoid double selecting the result element, you need to delete it from the list of elements that are requested as the aggregation string is used in the SQL SELECT together with the other requested elements.

To group the response data by the requested grouped elements, you need to create an SQL string for the SQL GROUP BY clause.

The interface `IF_RAP_QUERY_RESPONSE` provides a method to set the aggregated and grouped data for the query response after retrieving data from the data source.

Prerequisites

- For general prerequisites of an unmanaged query implementation, see [Prerequisites Unmanaged Query \[page 496\]](#).
- To use aggregation in a Fiori Elements UI and send requests for aggregate values, you need to annotate the related elements with `@Aggregation.default: <aggr_method>`.

Implementation Steps

This procedure combines the implementation of requested elements and aggregation, as both are used in the select list of the SQL SELECT. The steps also provide an example on how to use the grouped elements in the SQL SELECT.

1. Check if data is requested.
2. Call the method `get_requested_elements` of the interface `IF_RAP_QUERY_REQUEST`, which returns a table with the requested elements.
3. Call the method `get_aggregation` of the interface `IF_RAP_QUERY_REQUEST`, which returns an interface instance of `IF_RAP_QUERY_AGGREGATION`. Use the method `get_aggregated_elements`, which returns a list of the elements to be aggregated, their respective aggregation method and the result element for the aggregated value.
4. Delete the result element from the requested elements.
5. Create the aggregation string for the SQL SELECT with the aggregation method, the element to be aggregated and the result element.
6. Append the aggregation string to the string of requested elements.
7. Call the method `get_aggregation` of the interface `IF_RAP_QUERY_REQUEST`, which returns an interface instance of `IF_RAP_QUERY_AGGREGATION`. Use the method `get_grouped_elements`, which returns a list of the elements that are requested as group reference.
8. Concatenate the lines of the string table with comma separation into a string variable.

9. Use the requested elements in the select list of the SQL SELECT.
10. Use the string with the grouped elements in the GROUP BY clause of the SELECT statement.

The following codeblock illustrates the implementation for aggregation and grouping select method of `IF_RAP_QUERY_PROVIDER` in the query implementation class.

```

IF io_request->is_data_requested( ).
  DATA(lt_req_elements) = io_request->get_requested_elements( ).
  DATA(lt_aggr_element) = io_request->get_aggregation( )-
>get_aggregated_elements( ).
  IF lt_aggr_element IS NOT INITIAL.
    LOOP AT lt_aggr_element ASSIGNING FIELD-SYMBOL(<fs_aggr_element>).
      DELETE lt_req_elements WHERE table_line = <fs_aggr_element>-
result_element.
      DATA(lv_aggregation) = |{ <fs_aggr_element>-aggregation_method }|
( { <fs_aggr_element>-input_element } ) as { <fs_aggr_element>-result_element }|.
      APPEND lv_aggregation TO lt_req_elements.
    ENDLOOP.
  ENDIF.
  DATA(lv_req_elements) = concat_lines_of( table = lt_req_elements sep =
` ` )..
  DATA(lt_grouped_element) = io_request->get_aggregation( )-
>get_grouped_elements( ).
  DATA(lv_grouping) = concat_lines_of( table = lt_grouped_element sep =
` ` )..
  DATA lt_travel_response TYPE STANDARD TABLE OF /dmo/i_travel_uq.
  SELECT (lv_req_elements) FROM /dmo/travel
    GROUP BY (lv_grouping)
    ORDER BY ('primary key')
    INTO CORRESPONDING FIELDS OF TABLE
@lt_travel_response.
  io_response->set_data( lt_travel_response ).
ENDIF.

```

i Note

It is recommended to implement a default sort order to return consistent results from the data source.

i Note

For API information, see [Method `get_aggregation` \[page 748\]](#).

7.13 Adding Field Labels and Descriptions

End-user texts, such as field labels or field descriptions, are taken from ABAP Dictionary data elements to which the corresponding element is bound - unless you redefine the texts using CDS annotations. Unlike technical element names, the header texts, field labels and descriptions are **language-dependent**. For example, the field 'Airline' would have a language-dependent label 'Airline Code'.

Such texts must be translated. Therefore, the CDS development infrastructure is able to extract them from the source code and transfer the extracted texts to the actual translation infrastructure of the corresponding delivery package.

Relevant Annotations

Annotation	Effect
@EndUserText.label: '<text>'	This annotation is used to define translatable semantic short texts (with maximum 60 characters) for an element of the CDS view (label text).
@EndUserText.quickInfo: '<text>'	The annotation defines a human-readable <text> that provides additional information about an element. The quick info is used for accessibility hints or the mouse over function.

→ Remember

The <text> specified in the source code should consist of text in the original language of the CDS source code and will be translated into the required languages.

Example

The listing below demonstrates the usage of @EndUserText annotations at the view and element (field) level:

```
...
@EndUserText.label: 'Overview of available flights' -- Annotation at the view
level

DEFINE VIEW <CDS_view> as Flights {
  ...
  -- Annotation at the field level
  @EndUserText: { label: 'Airline Code',
    quickinfo: 'Code to identify which airline operates the flight' }
  carrier_id as CarrierID;
  ...
}
```

→ Tip

Press **F1** in the CDS source code editor for extended help content on @EndUserText annotation!

i Note

If @UI labeling annotations are used, they will be evaluated primarily. That means, they will overwrite the text given with the @EndUserText annotations.

OData Metadata

To verify that the additional information of labels and descriptions is pushed correctly to the OData service, you can check the OData metadata document. This can also be helpful to find out which label information is used if you maintain @UI and @EndUserText in your CDS view.

If no UI annotations are used, the OData metadata document of the example above should contain the annotations that are marked in the following image:

```
<EntityType sap:content-version="1" sap:label="Overview of Available Flights" Name="FlightsType">
  <Key/>
  <Property sap:label="Airline Code" Name="CarrierID" sap:quickinfo="Code to identify which airline operates the flight" sap:display-format="UpperCase"
```

7.14 Defining CDS Annotations for Metadata-Driven UIs

Metadata-driven UIs are dynamic UIs because metadata, namely CDS annotations in this context, are stored in a repository and can be retrieved from the client as needed. CDS annotations depend on the UI in which they are supposed to be used.

UIs might differ from user to user. Even though if, for example, three different users use the same application, each of them might have different permissions or different preferences, which results in different UI perspectives. Users might want to personalize their UIs and see different columns in tables, for example. CDS annotations offer default views for modelling UIs, however, CDS annotations can be overruled by personalization preferences.

The following chapters inform you about CDS annotations that you can use to define metadata-driven UIs, and answer the following questions:

- [How can I define the title of a field or a table? \[page 516\]](#)
- [How can I define the columns of a field or a table? \[page 518\]](#)
- [How can I define fields for filtering? \[page 519\]](#)
- [How can I define the header of an object-page floorplan? \[page 521\]](#)
- [How can I define the body of an object-page floorplan? \[page 523\]](#)
- [How can I group fields? \[page 525\]](#)
- [How can I expose elements to UIs? \[page 526\]](#)
- [How can I overwrite default labels? \[page 527\]](#)
- [How can I position fields? \[page 527\]](#)
- [How can I prioritize fields? \[page 528\]](#)
- [How can I define charts? \[page 530\]](#)
- [How can I visualize criticality? \[page 534\]](#)
- [How can I visualize trends? \[page 535\]](#)
- [How can I visualize criticality based on trend calculation? \[page 537\]](#)
- [How can I visualize a person responsible and a reference period? \[page 540\]](#)
- [How can I use the dataField type #AS_DATAPOINT? \[page 542\]](#)
- [How can I define a contact? \[page 543\]](#)
- [How can I define navigation between screens? \[page 544\]](#)
- [How can I define navigation to external web sites? \[page 546\]](#)
- [How can I define navigation based on actions executed on semantic objects? \[page 548\]](#)
- [How can I define actions? \[page 550\]](#)
- [How can I display fields in a text area? \[page 552\]](#)
- [How can I mask fields? \[page 553\]](#)
- [How can I hide fields? \[page 554\]](#)

- How can I define interaction between annotations? [page 555]

7.14.1 Tables and Lists

Get an overview of how to use UI annotations for lists and tables for SAP Fiori UIs.

In Fiori we distinguish tables and list. Both mostly hold homogeneous data, but lists hold in general rather simple data whereas tables hold usually more complex ones.

Lists are mostly used in the master list section of the master-detail floorplan and in popovers or dialogs. Sure there is also the possibility to have them in full-screen floorplans for certain use cases.

A table contains a set of line items and usually comprises rows (one row showing one line item) and columns. Line items can contain data of any kind, but also interactive controls, for example, for editing the data, navigating, or triggering actions relating to the line item.

To display large amounts of data in tabular form, several table controls are provided. These are divided into two groups, each of which is defined by a consistent feature set:

- Fully responsive tables
- Desktop-centric tables

In order to expose a CDS view in a table-like or list-like format, you can use the annotations explained in the following sections::

- Title [page 516]
- Columns [page 518]
- Selection Fields [page 519]

7.14.1.1 Title

Get information about what UI annotations to use to work with titles of lists or tables for SAP Fiori UIs.

If necessary, you can provide a header for your list or table.

The screenshot shows a SAP Fiori application interface. At the top, there is a filter bar with a dropdown set to "Standard * ⓘ", a search input field labeled "Company:" containing "SAP", and buttons for "Hide Filter Bar", "Filters", and "Go". Below the filter bar is a table header row with columns: "Sales Order ID", "Company", "Currency Code", and "Gross Amount". The table body contains 16 rows of sales order data. Each row includes a small arrow icon at the end of the "Gross Amount" column. The table has a yellow header bar with the text "Sales Orders (1,190) Standard * ⓘ". On the right side of the table, there are icons for settings and export. A dark footer bar at the bottom right contains a small arrow icon.

Sales Order ID	Company	Currency Code	Gross Amount
500000000	SAP	EUR	25,867.03 EUR >
500000001	DelBont Industries	EUR	14,602.49 EUR >
500000002	TECUM	EUR	5,631.08 EUR >
500000003	Asia High tech	EUR	1,704.04 EUR >
500000004	Asia High tech	EUR	761.24 EUR >
500000005	AVANTEL	EUR	101,299.22 EUR >
500000006	Talpa	EUR	250.73 EUR >
500000007	Panorama Studios	EUR	10,311.35 EUR >
500000008	Telecomunicaciones Star	EUR	195.16 EUR >
500000009	SAP	EUR	3,972.22 EUR >
500000010	DelBont Industries	EUR	827.95 EUR >
500000011	Panorama Studios	EUR	325.94 EUR >
500000012	TECUM	EUR	24,704.40 EUR >
500000013	Asia High tech	EUR	8,256.22 EUR >
500000014	Asia High tech	EUR	3,459.33 EUR >
500000015	AVANTEL	EUR	862.73 EUR >
500000016	Panorama Studios	EUR	70.18 EUR >

Example of title of table

You can use the following UI annotation to define what can be displayed in the title of a table or a list:

- [@UI.headerInfo.typeNamePlural \[page 623\]](#)

↳ Sample Code

```
...
@UI.headerInfo: { typeNamePlural: 'Sales Orders' }
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    ...
}
```

Related Information

[Tables and Lists \[page 516\]](#)

[Columns \[page 518\]](#)

Selection Fields [page 519]

7.14.1.2 Columns

Get information about what UI annotations to use to work with columns of lists or tables for SAP Fiori UIs.

What columns are needed for a table or list depends on the use case, for example, an overview table may require more fields than a value help list. For this reason, you can define several list layouts and table layouts that are distinguished by a qualifier, for example 'ValueList'.

If a CDS view contains analytical annotations, for example the `@AggregationDefault` annotation, the UI automatically takes this into consideration and no additional UI annotations are required.

<input type="checkbox"/>	Sales Order ID	Company	Currency Code	Gross Amount
>	Company: African Gold ...	African Gold And Diamond Corporation	EUR	84,676.32 EUR
>	Company: Alpine Systems	Alpine Systems	EUR	84,676.32 EUR
▼	Company: Anav Ideon			
<input type="checkbox"/>	500000105	Anav Ideon	EUR	101,299.22 EUR
<input type="checkbox"/>	500000115	Anav Ideon	EUR	862.73 EUR
<input type="checkbox"/>	500000123	Anav Ideon	EUR	411.50 EUR
<input type="checkbox"/>	500000128	Anav Ideon	EUR	1,704.04 EUR
<input type="checkbox"/>	500000140	Anav Ideon	EUR	541.31 EUR
<input type="checkbox"/>	500000147	Anav Ideon	EUR	521.22 EUR
<input type="checkbox"/>	500000355	Anav Ideon	EUR	101,299.22 EUR
<input type="checkbox"/>	500000365	Anav Ideon	EUR	862.73 EUR
<input type="checkbox"/>	500000373	Anav Ideon	EUR	411.50 EUR
<input type="checkbox"/>	500000378	Anav Ideon	EUR	1,704.04 EUR
<input type="checkbox"/>	500000390	Anav Ideon	EUR	541.31 EUR
<input type="checkbox"/>	500000397	Anav Ideon	EUR	521.22 EUR
<input type="checkbox"/>	500000605	Anav Ideon	EUR	101,299.22 EUR
<input type="checkbox"/>	500000615	Anav Ideon	EUR	862.73 EUR
<input type="checkbox"/>	500000623	Anav Ideon	EUR	411.50 EUR
<input type="checkbox"/>	500000628	Anav Ideon	EUR	1,704.04 EUR
<input type="checkbox"/>	500000640	Anav Ideon	EUR	541.31 EUR
<input type="checkbox"/>	500000647	Anav Ideon	EUR	521.22 EUR
<input type="checkbox"/>	500000855	Anav Ideon	EUR	101,299.22 EUR

Example of `@DefaultAggregation` annotation

For more information about the `@AggregationDefault` annotation, see section *AggregationDefault Annotations* linked below.

You can use the following UI annotation to define what can be displayed in the title of a table or a list:

- [@UI.lineItem \[page 683\]](#)

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
  @UI.lineItem: [ { position: 10 }, { qualifier: 'ValueList', position:
10 } ]
  key so.sales_order_id as SalesOrder,
  @UI.lineItem: [ { position: 20 }, { qualifier: 'ValueList', position:
20 } ]
  so.customer.company_name as CompanyName,
  @UI.lineItem: [ { position: 30 } ]
  so.currency_code as CurrencyCode,
  @AggregationDefault: #SUM
  @UI.lineItem: [ { position: 40 } ]
  so.gross_amount as GrossAmount
}
```

For more information about positioning, see section *Positioning Fields* linked below.

Related Information

[Aggregation Annotations \[page 559\]](#)

[Positioning Fields \[page 527\]](#)

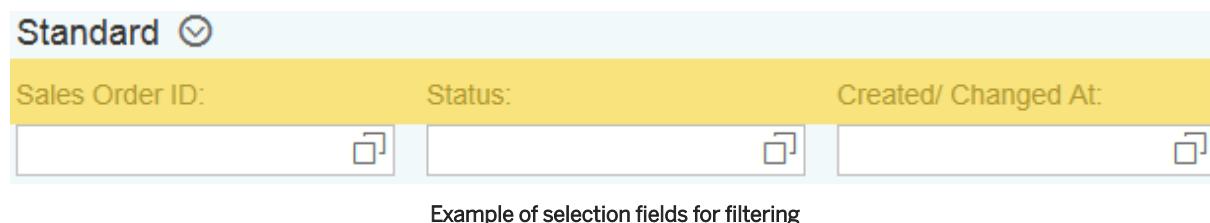
[Tables and Lists \[page 516\]](#)

[Title \[page 516\]](#)

[Selection Fields \[page 519\]](#)

7.14.1.3 Selection Fields

Get information about what UI annotations to use to work with selection fields for SAP Fiori UIs.



Example of selection fields for filtering

If a CDS is annotated as [@Search.searchable \[page 603\]](#), the UI automatically takes this into consideration and no additional UI annotations are required to expose a search field or a value help. If your table or list contains many data and therefore many rows, it gets hard to find the information you need. To facilitate finding the desired information, you can use selection fields to specify the range of information that you are looking for.

The screenshot shows a SAP Fiori application interface. At the top, there is a search bar with placeholder text "Search" and a "Go" button. To the right of the search bar are links for "Hide Filter Bar", "Filters", and another "Go" button. Below the search bar is a "Company:" field with a dropdown arrow. The main area displays a table titled "Sales Orders (1,190) | Standard *". The table has columns: "Sales Order ID", "Company", "Currency Code", and "Gross Amount". The data is grouped by company, with African Gold, Alpine Systems, and Anav Ideon expanded. The table contains 1,190 rows of sales order data.

	Sales Order ID	Company	Currency Code	Gross Amount
>	Company: African Gold ...	African Gold And Diamond Corporation	EUR	84,676.32 EUR
>	Company: Alpine Systems	Alpine Systems	EUR	84,676.32 EUR
▼	Company: Anav Ideon			
<input type="checkbox"/>	500000105	Anav Ideon	EUR	101,299.22 EUR
<input type="checkbox"/>	500000115	Anav Ideon	EUR	862.73 EUR
<input type="checkbox"/>	500000123	Anav Ideon	EUR	411.50 EUR
<input type="checkbox"/>	500000128	Anav Ideon	EUR	1,704.04 EUR
<input type="checkbox"/>	500000140	Anav Ideon	EUR	541.31 EUR
<input type="checkbox"/>	500000147	Anav Ideon	EUR	521.22 EUR
<input type="checkbox"/>	500000355	Anav Ideon	EUR	101,299.22 EUR
<input type="checkbox"/>	500000365	Anav Ideon	EUR	862.73 EUR
<input type="checkbox"/>	500000373	Anav Ideon	EUR	411.50 EUR
<input type="checkbox"/>	500000378	Anav Ideon	EUR	1,704.04 EUR
<input type="checkbox"/>	500000390	Anav Ideon	EUR	541.31 EUR
<input type="checkbox"/>	500000397	Anav Ideon	EUR	521.22 EUR
<input type="checkbox"/>	500000605	Anav Ideon	EUR	101,299.22 EUR
<input type="checkbox"/>	500000615	Anav Ideon	EUR	862.73 EUR
<input type="checkbox"/>	500000623	Anav Ideon	EUR	411.50 EUR
<input type="checkbox"/>	500000628	Anav Ideon	EUR	1,704.04 EUR
<input type="checkbox"/>	500000640	Anav Ideon	EUR	541.31 EUR
<input type="checkbox"/>	500000647	Anav Ideon	EUR	521.22 EUR
<input type="checkbox"/>	500000855	Anav Ideon	EUR	101,299.22 EUR

Example for @Search.searchable annotation

If your table or list contains many data and therefore many rows, it gets hard to find the You can use the following UI annotation to enable specific elements for selection, for example using a filter bar:

- [@UI.selectionField \[page 659\]](#)

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    key so.sales_order_id as SalesOrder,
    @UI.selectionField: [ { position: 10 } ]
    so.customer.company_name as CompanyName,
    ...
}
```

Related Information

[Tables and Lists \[page 516\]](#)

[Title \[page 516\]](#)

[Columns \[page 518\]](#)

7.14.2 Detail Pages

Get an overview of how to use UI annotations for object-page floorplans for SAP Fiori UIs.

You can use the object-page floorplan if you need to display, create, or edit any object regardless of its complexity level. You can use the object-page floorplan with either a facet (tabs) or flat (anchors) approach.

To expose a CDS view in an object-page floorplan, you can use the annotations explained in the following sections:

- [Page Header \[page 521\]](#)
- [Page Body \[page 523\]](#)

7.14.2.1 Page Header

Get information about what UI annotations to use to work with page headers of object-page floorplans for SAP Fiori UIs.

The page header contains information on the object you are editing in the object-page floorplan, for example.

The screenshot shows a SAP Fiori detail page for a Sales Order. At the top, there is a header bar with a back arrow on the left and a 'Sales Order' tab on the right. Below the header, the page title is 'Sales Order: 5000000000' and the customer is listed as 'Customer: SAP'. The main content area is titled 'GENERAL INFORMATION' and contains a section for 'General Information'. Under this section, several fields are listed with their values: Sales Order ID: 5000000000, Company: SAP, Currency Code: EUR, and Gross Amount: 25,867.03 EUR. The entire header and part of the main content are highlighted with a light blue background, illustrating the scope of the @UI.headerInfo annotation.

Example of `@UI.headerInfo` for page header of object-page floorplan

You can use the following UI annotations to use several properties to influence the header section of the object-page floorplan:

- [@UI.headerInfo \[page 622\]](#)

« Sample Code

```
@UI.headerInfo: {
    typeName: 'Sales Order',
    title: {
        label: 'Sales Order',
        value: 'SalesOrder'      -- Reference to element in element list
    },
    description: {
        label: 'Customer',
        value: 'CompanyName'     -- Reference to element in element list
    }
}
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    key so.sales_order_id as SalesOrder,
    so.customer.company_name as CompanyName,
    ...
}
```

- [@UI.badge \[page 632\]](#)

This UI annotation can be considered as the combination of the UI annotations [@UI.headerInfo \[page 622\]](#) and [@UI.identification \[page 690\]](#). The properties imageUrl, typeImageURL and title should usually correspond to the properties of the UI annotation [@UI.headerInfo \[page 622\]](#). In addition to the title, a headLine, mainInfo and secondaryInfo of the same format can be specified.

« Sample Code

```
@UI.badge: {
    title: {
        label: 'Sales Order',
        value: 'SalesOrderID'      -- Reference to element in element list
    },
    headLine: {
        label: 'Customer',
        value: 'CompanyName'     -- Reference to element in element list
    },
    mainInfo: {
        label: 'Gross Amount',
        value: 'GrossAmount'      -- Reference to element in element list
    },
    secondaryInfo: {
        label: 'Billing Status',
        value: 'BillingStatus'    -- Reference to element in element list
    }
}
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    key so.sales_order_id as SalesOrder,
    so.customer.company_name as CompanyName,
    so.gross_amount as GrossAmount,
    so.billing_status as BillingStatus,
    ...
}
```

- [@UI.statusInfo \[page 697\]](#)

This UI annotation can be used to display status information of an entity on the UI, for example the delivery status or payment status of an entity. This annotation is similar to the [@UI.lineItem \[page 683\]](#)

annotation. However, the [@UI.statusInfo \[page 697\]](#) annotation is usually used together with the `criticality` property instead of the `qualifier` property.

« Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    key so.sales_order_id as SalesOrder,
    @UI.statusInfo: [ { position: 10 } ]
    so.delivery_status as DeliveryStatus,
    @UI.statusInfo: [ { position: 20 } ]
    so.billing_status as BillingStatus,
    @UI.statusInfo: [ { position: 30 } ]
    so.lifecycle_status as LifecycleStatus,
    ...
}
```

Related Information

[Detail Pages \[page 521\]](#)

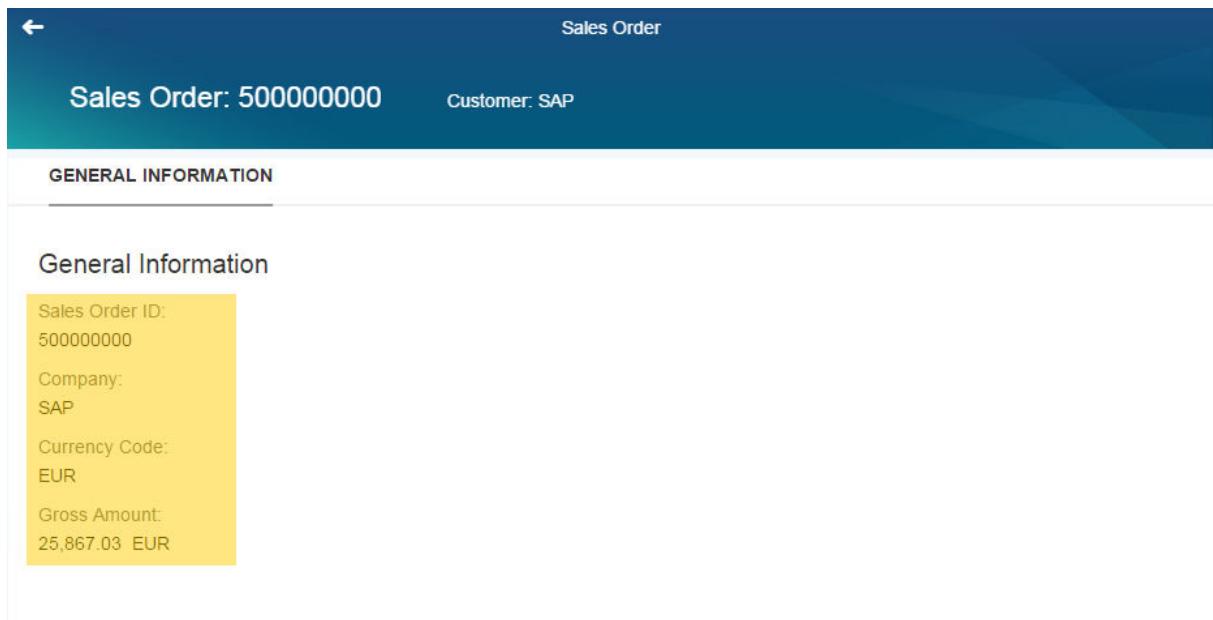
[Page Body \[page 523\]](#)

[Defining Criticality of Field Values \[page 529\]](#)

7.14.2.2 Page Body

Get information about what UI annotations to use to work with page bodies of object-page floorplans for SAP Fiori UIs.

The page body can consist of a list or a table, for example, in which you can see and edit details of an object from the master-detail floorplan.



Example of columns of table on object-page floorplan

You can use the following UI annotation to define what elements are displayed in the page body of the object-page floorplan:

- [@UI.identification \[page 690\]](#)

This annotation is similar to the UI annotation [@UI.lineItem \[page 683\]](#), but [@UI.identification \[page 690\]](#) has no qualifier.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    @UI.identification: [ { position: 10 } ]
    key so.sales_order_id as SalesOrder,
    @UI.identification: [ { position: 20 } ]
    so.customer.company_name as CompanyName,
    @UI.identification: [ { position: 30 } ]
    so.currency_code as CurrencyCode,
    @UI.identification: [ { position: 40 } ]
    so.gross_amount as GrossAmount
}
```

Related Information

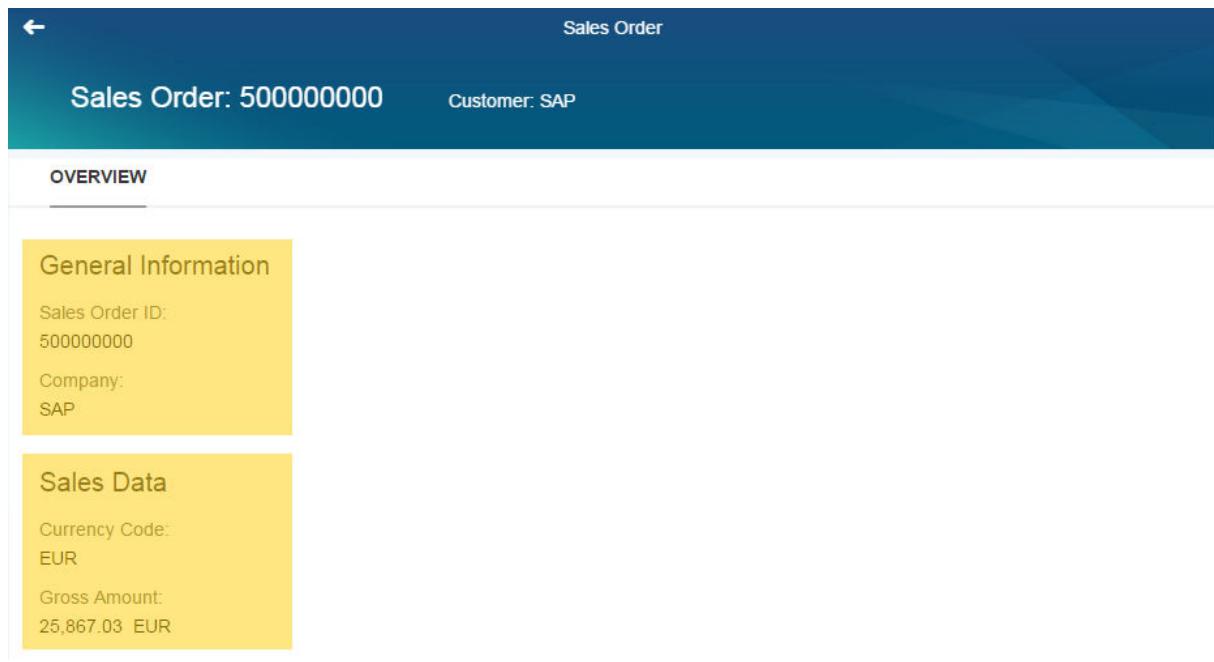
[Detail Pages \[page 521\]](#)

[Page Header \[page 521\]](#)

7.14.3 Field Groups

Get information about what UI annotations to use to work with field groups for SAP Fiori UIs.

If you want to group fields under one heading to consolidate semantically connected information, you can use field groups. With field groups, you can build sections for forms, for example.



Example of field group

You can use the following UI annotation to group several fields:

- [@UI.fieldGroup \[page 704\]](#)

This annotation is similar to the UI annotation [@UI.lineItem \[page 683\]](#) because the different field groups have unique qualifiers.

↳ Sample Code

```
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {  
    @UI.fieldGroup: [ { qualifier: 'GeneralInformation', position: 10 } ]  
    key so.sales_order_id as SalesOrder,  
  
    @UI.fieldGroup: [ { qualifier: 'GeneralInformation', position: 20 } ]  
    so.customer.company_name as CompanyName,  
  
    @UI.fieldGroup: [ { qualifier: 'SalesData', position: 10 } ]  
    so.currency_code as CurrencyCode,  
    @UI.fieldGroup: [ { qualifier: 'SalesData', position: 20 } ]  
    so.gross_amount as GrossAmount,  
}
```

7.14.4 Annotations Similar to `dataField`

Get an overview of how to use UI annotations that are similar to the OData annotation `dataField`.

The OData annotation `dataField` refers to a property of the OData service that is used.

Some annotations are syntactically similar or even identical. These annotations are the following:

- [@UI.lineItem \[page 683\]](#)
- [@UI.selectionField \[page 659\]](#)
- [@UI.statusInfo \[page 697\]](#)
- [@UI.identification \[page 690\]](#)
- [@UI.fieldGroup \[page 704\]](#)

These annotations are called **dataField-like annotations** in the following sections:

- [Exposing Elements \[page 526\]](#)
- [Overwriting Default Labels \[page 527\]](#)
- [Positioning Fields \[page 527\]](#)
- [Prioritizing UI Elements \[page 528\]](#)

7.14.4.1 Exposing Elements

Get information about how to expose elements to SAP Fiori UIs.

You can use dataField-like annotations to reference elements from a different CDS view using to-one-associations. You therefore need to explicitly define the elements with a value property. These elements are then exposed to the UI.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so
    association [0..1] to sepm_cds_business_partner as _BusinessPartner
        on $projection.buyer_guid = _BusinessPartner.business_partner_key
{
    key so.sales_order_id as SalesOrder,
    so.buyer_guid,
    ...
    @UI.Identification: [
        { value: '_BusinessPartner.company_name', position: 110 },
        { value: '_BusinessPartner.bp_role', position: 120 }
    ]
    _BusinessPartner
}
```

Related Information

[Annotations Similar to `dataField` \[page 526\]](#)

[Overwriting Default Labels \[page 527\]](#)

[Positioning Fields \[page 527\]](#)

[Prioritizing UI Elements \[page 528\]](#)

7.14.4.2 Overwriting Default Labels

Get information about how to overwrite default labels for SAP Fiori UIs.

If a CDS element is exposed via a dataField-like annotation, the label is by default derived from the CDS annotation `@EndUserText.label` if available, or from a DDIC element.

If you want a default label to be overwritten by a specific label, for example *Customer* instead of *Business*, you can use the label property.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so
    association [0..1] to sepm_cds_business_partner as _BusinessPartner
        on $projection.buyer_guid = _BusinessPartner.business_partner_key
{
    key so.sales_order_id as SalesOrder,
    so.buyer_guid,
    ...

    @UI.Identification: [
        { value: '_BusinessPartner.company_name', position: 110, label: 'Customer
Name' },
        { value: '_BusinessPartner.bp_role', position: 120, label: 'Customer
Role' }
    ]
    _BusinessPartner
}
```

Related Information

[Annotations Similar to dataField \[page 526\]](#)

[Exposing Elements \[page 526\]](#)

[Positioning Fields \[page 527\]](#)

[Prioritizing UI Elements \[page 528\]](#)

7.14.4.3 Positioning Fields

Get information about how to change the position of fields on SAP Fiori UIs.

To define the order of fields in the UI, you can use the `position` property of dataField-like annotations. Only the positioning order is relevant, so you can use any decimal number as value for the positioning property.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    @UI.identification: [ { position: 1 } ]
    key so.sales_order_id as SalesOrder,
    @UI.identification: [ { position: -5 } ]
    so.customer.company_name as CompanyName,
    @UI.identification: [ { position: 9999 } ]
    so.currency_code as CurrencyCode,
    @UI.identification: [ { position: '1.1' } ]
    so.gross_amount as GrossAmount
}
```

Related Information

[Annotations Similar to dataField \[page 526\]](#)

[Exposing Elements \[page 526\]](#)

[Overwriting Default Labels \[page 527\]](#)

[Prioritizing UI Elements \[page 528\]](#)

7.14.4.4 Prioritizing UI Elements

Get information about how to set the priority of elements displayed on SAP Fiori UIs.

To define the priority of elements, you can use the `importance` property of dataField-like annotations. This information is relevant for adaptive UIs. If a UI is displayed on a small screen, elements with low priority can automatically be hidden. To define importance, you can choose the following values:

- `#HIGH`
- `#MEDIUM`
- `#LOW`
- not set

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    @UI.identification: [ { position: 10, importance: "#HIGH" } ]-
    key so.sales_order_id as SalesOrder,
    @UI.identification: [ { position: 20, importance: #MEDIUM } ]
    so.customer.company_name as CompanyName,
    @UI.identification: [ { position: 30, importance: #LOW } ]
    so.currency_code as CurrencyCode,
    @UI.identification: [ { position: 40 } ]
```

```
    so.gross_amount as GrossAmount  
    ...  
}
```

Related Information

[Annotations Similar to dataField \[page 526\]](#)

[Exposing Elements \[page 526\]](#)

[Overwriting Default Labels \[page 527\]](#)

[Positioning Fields \[page 527\]](#)

7.14.4.5 Defining Criticality of Field Values

Get information about how to define the criticality of field values for SAP Fiori UIs.

To define if a field value is negative, critical, or positive, you can use the `criticality` property of `dataField`-like annotations. This property must refer to a CDS element that has the value 1 (negative), 2 (critical), or 3 (positive).

Sample Code

```
...  
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {  
    @UI.identification: [ { position: 10, importance: #HIGH } ]  
    key so.sales_order_id as SalesOrder,  
    ...  
    @UI.statusInfo: [ { position: 10, criticality: 'GrossAmountCrit' } ]  
    so.billing_status as BillingStatus,  
    so.billing_status_crit as BillingStatusCrit,  
    ...  
}
```

Related Information

[Annotations Similar to dataField \[page 526\]](#)

[Exposing Elements \[page 526\]](#)

[Overwriting Default Labels \[page 527\]](#)

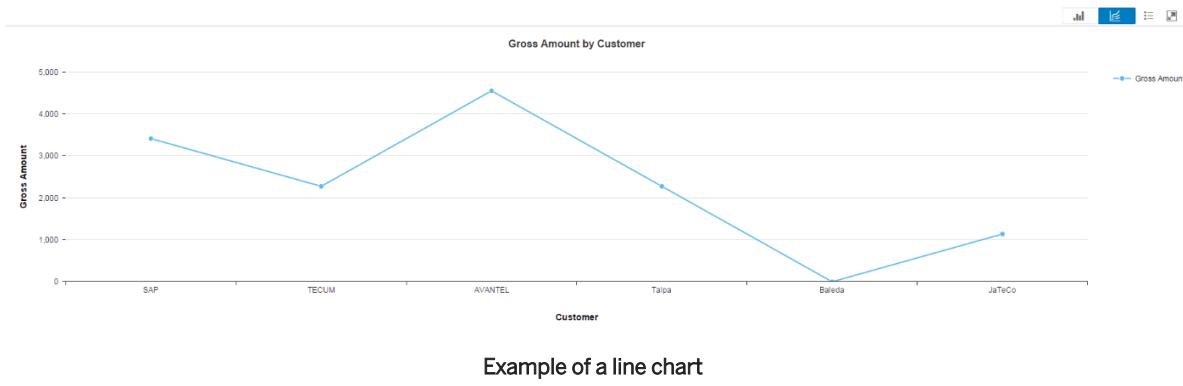
[Positioning Fields \[page 527\]](#)

[Prioritizing UI Elements \[page 528\]](#)

7.14.5 Charts

Get information about what UI annotations to visualize data on SAP Fiori UIs.

If you want to visualize data, you can use a chart.



You can use the following UI annotation to define the properties of a chart:

- [@UI.chart \[page 661\]](#)

You define this UI annotation at view level. It refers to the elements that are to be used in the chart. Additionally, you can provide a title and description.

↳ Sample Code

```
...
@UI.chart: {
    title: 'Gross Amount by Customer',
    description: 'Line-chart displaying the gross amount by customer',
    chartType: #LINE,
    dimensions: [ 'CompanyName' ],      -- Reference to one element
    measures: [ 'GrossAmount' ]         -- Reference to one or more elements
}
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
    key so.sales_order_id as SalesOrder,
    so.customer.company_name as CompanyName,
    so.currency_code as CurrencyCode,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    so.gross_amount as GrossAmount,
    ...
}
```

Related Information

[Charts \[page 531\]](#)

7.14.5.1 Charts

Get an overview of what chart types you can use to visualize data on SAP Fiori UIs.

Each chart type has different restrictions referring to how many dimensions and measures are required or allowed. The following table lists the admissible types and their restrictions.

Chart Types for Data Visualization

Type	Dimensions	Measures
COLUMN	One dimension	One or more measures
COLUMN_STACKED	Displayed on the x-axis	Displayed on the y-axis
COL- UMN_STACKED_100		
AREA		
AREA_STACKED		
AREA_100		
LINE		
BAR	One dimension	One or more measures
BAR_STACKED	Displayed on the y-axis	Displayed on the x-axis
BAR_STACKED_100		
HORIZONTAL_AREA		
HORIZON- TAL_AREA_STACKED		
HORIZON- TAL_AREA_100		
PIE	One dimension	One Measure
DONUT	For segmentation	For size of segment
SCATTER	Two dimensions	Up to two measures (symbol and color)
BUBBLE	One for the x-axis, one for the y-axis	One measure (size of bubble)
RADAR	Three or more dimensions	No measures

Type	Dimensions	Measures
HEAT_MAP	Two dimensions One for the x-axis, one for the y-axis	One measure (color)
TREE_MAP	One or more hierarchical dimensions	One measure (rectangle size)
		One optional measure (color)
WATERFALL	One dimension Displayed on the x-axis	One measure Displayed on the y-axis

Related Information

[Charts \[page 530\]](#)

7.14.6 Data Points

Get an overview of how to use data points to display criticality, trends, and references to people and time periods on SAP Fiori UIs.

In some cases, you want to visualize a single point of data that typically is a number that can be enriched with business-relevant data but may also be textual, for example a status value.

Gross Amount

Gross Amount per Customer

→ **34.5 kEUR**

Reference Period: 2015 Q3

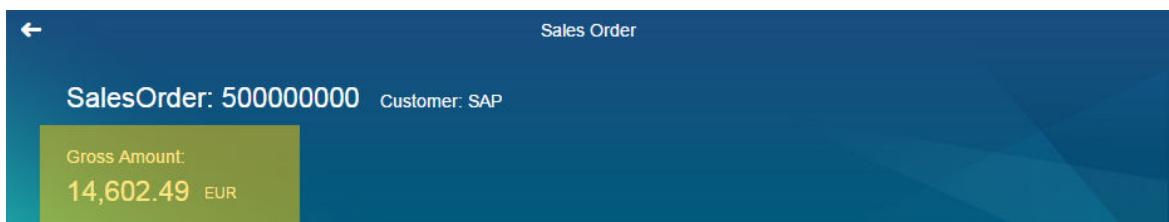
Responsible: John Doe

Example of data points

You can use the following UI annotation to define a single point of data:

- [@UI.dataPoint \[page 649\]](#)

You can, for example, express if a high or a low value is desired, or if a value is increasing or decreasing. The simplest variant of the UI annotation [@UI.dataPoint \[page 649\]](#) consists of the `title` property.



Example of simple variant of data point

In the following example, only the title is exposed to the UI.

↳ Sample Code

```
...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    key so.buyer_guid as BuyerGuid,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,
    @UI.dataPoint: { title: 'Gross Amount' }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    so.actual_amount as ActualAmount
}
```

Related Information

[Criticality \[page 534\]](#)

[Trends \[page 535\]](#)

[Trend-Criticality Calculation \[page 537\]](#)

[Person Responsible and Reference Period \[page 540\]](#)

[DataField Type: #AS_DATAPOINT \[page 542\]](#)

7.14.6.1 Criticality

Get information about how to use data points to display criticality on SAP Fiori UIs.

A more usable variant of the UI annotation [@UI.dataPoint \[page 649\]](#) also contains information about the criticality, the trend, and the name of a person responsible.

You can use the sub-annotation [@dataPoint.criticality \[page 653\]](#) to express if a value is positive or negative, for example.

You can use the sub-annotation [@dataPoint.trend \[page 656\]](#) to express if a value has decreased or increased, for example.

In this case, the properties `targetValue`, `criticality`, and `trend` are already evaluated in the CDS view. In the CDS view, the target value is already calculated, and if the current value thus is negative or positive, and if the current value has improved or declined, for example. These values are only referred to from the [@UI.dataPoint \[page 649\]](#) annotation.

Data can be defined as being either positive, critical, or negative. These data can be statuses, for example.

You can use the following sub-annotation to highlight criticality:

- [UI.dataPoint.criticality \[page 653\]](#)

You define this UI annotation at view level. It refers to the elements that are to be used in the chart.

Additionally, you can provide a title and description.

The table below lists the values that are valid for the UI annotation [@UI.dataPoint.criticality \[page 653\]](#), and shows how these values are visualized on the UI:

Values and Visualization of Criticality

Value	Description	Visualization in Color
1	Negative	Red
2	Critical	Yellow
3	Positive	Green

↳ Sample Code

```
...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    key so.buyer_guid as BuyerGuid,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,
```

```

@UI.dataPoint: {
    title: 'Gross Amount',
    targetValueElement: 'TargetAmount',      -- Reference to element
    criticality: 'AmountCriticality',       -- Reference to element
    trend: 'AmountTrend',                  -- Reference to element
}
@Semantics.amount.currencyCode: 'CurrencyCode'
so.actual_amount as ActualAmount,

@Semantics.amount.currencyCode: 'CurrencyCode'
so.target_amount as TargetAmount,

so.criticality as AmountCriticality,
so.trend as AmountTrend
}

```

Related Information

[Data Points \[page 532\]](#)

[Trends \[page 535\]](#)

[Trend-Criticality Calculation \[page 537\]](#)

[Person Responsible and Reference Period \[page 540\]](#)

[DataField Type: #AS_DATAPOINT \[page 542\]](#)

7.14.6.2 Trends

Get information about how to use data points to display trends on SAP Fiori UIs.

Data can be defined as being either increasing, decreasing, or stable. These data can be measured over a certain period of time and visualized on the UI.

You can use the following sub-annotations to highlight trends:

- [@UI.dataPoint.trend \[page 656\]](#)

❖ Example

For an example, see the example code in section *Criticality* linked below.

- [@UI.dataPoint.trendCalculation \[page 656\]](#)

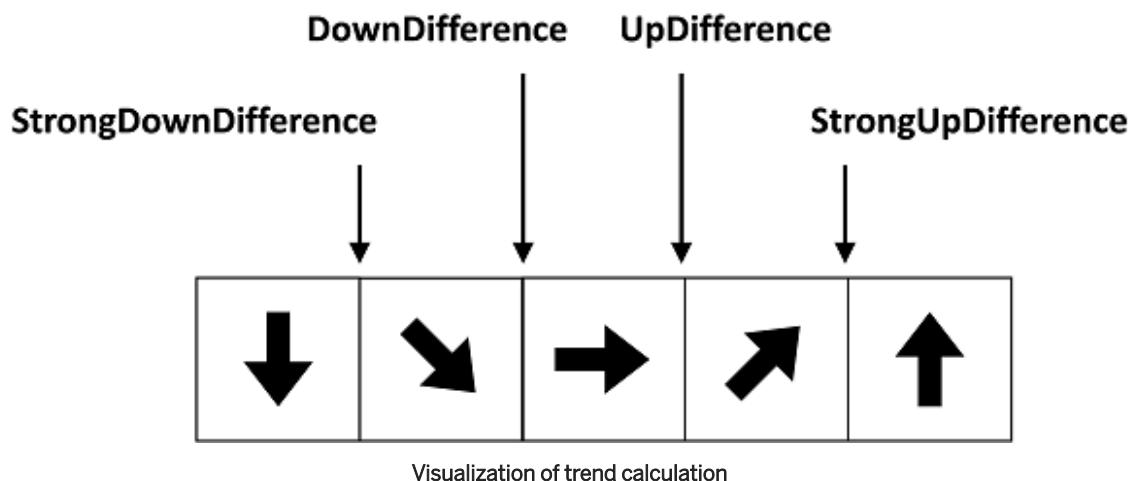
❖ Example

The table below lists the values that are valid for the UI annotation [@UI.dataPoint.trendCalculation \[page 656\]](#), and shows how these values are visualized on the UI:

Values and Visualization of Trend

Value	Description	Visualization
1	Strong up	↑
2	Up	↗
3	Sideways	→
4	Down	↘
5	Strong down	↓

For the trend calculation, the flag `isRelativeDifference` indicates whether the absolute or the relative difference between the actual value and the reference value is used to calculate the trend.



Sample Code

```
...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    key so.buyer_guid as BuyerGuid,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,

    @UI.dataPoint: {
        title: 'Gross Amount',
        //...
        trendCalculation: {
            referenceValue: 'ReferenceAmount', -- Reference to element
            isRelativeDifference: true, -- Comparison of ratio
            strongUpDifference: 1.25,
            upDifference: 1.1,
            downDifference: 0.9,
        }
    }
}
```

```

        strongDownDifference: 0.75
    }
}
@Semantics.amount.currencyCode: 'CurrencyCode'
so.target_amount as TargetAmount,
@Semantics.amount.currencyCode: 'CurrencyCode'
so.reference_amount as ReferenceAmount
}

```

Related Information

[Data Points \[page 532\]](#)

[Criticality \[page 534\]](#)

[Trend-Criticality Calculation \[page 537\]](#)

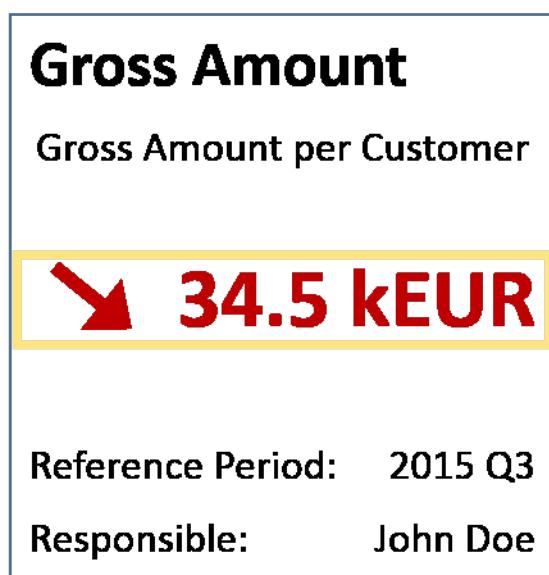
[Person Responsible and Reference Period \[page 540\]](#)

[DataField Type: #AS_DATAPOINT \[page 542\]](#)

7.14.6.3 Trend-Criticality Calculation

Get information about how to use data points to calculate and display trend-criticality relations.

Another way to specify properties of criticality and trend is to define rules for criticality and trend within the UI annotation [@UI.dataPoint \[page 649\]](#).



Example of visualization of trend-criticality calculation

You can use the following sub-annotations to calculate trends and derive from these calculation the criticality of data:

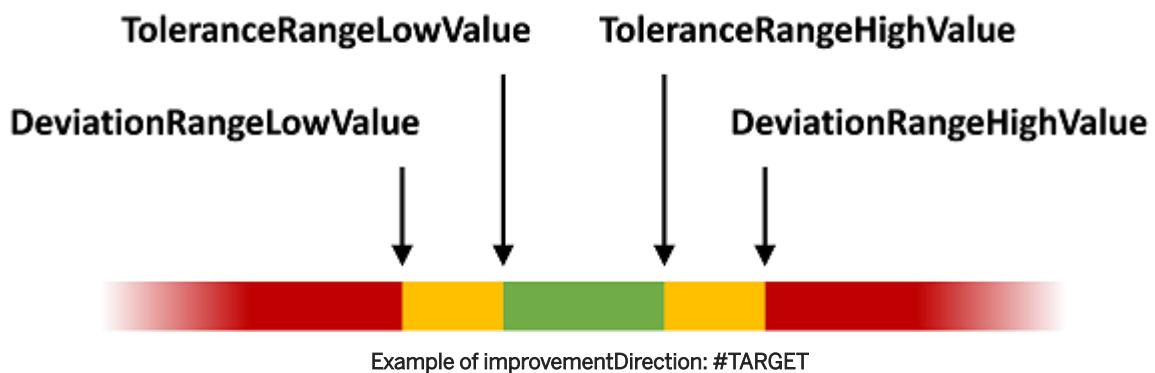
- [@UI.dataPoint.trendCalculation \[page 656\]](#)
- [@UI.dataPoint.criticalityCalculation \[page 656\]](#)

↳ Sample Code

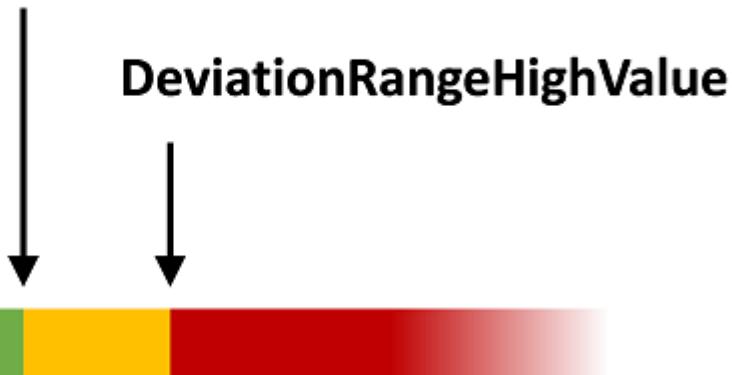
```
...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    key so.buyer_guid as BuyerGuid,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,

    @UI.dataPoint: {
        title: 'Gross Amount',
        targetValue: 9216,
        criticalityCalculation: {
            improvementDirection: #TARGET,
            toleranceRangeLowValue: 9200,
            toleranceRangeHighValue: 9300,
            deviationRangeLowValue: 8800,
            deviationRangeHighValue: 9700
        },
        trendCalculation: {
            referenceValue: 'ReferenceAmount',      -- Reference to element
            isRelativeDifference: false,           -- Comparison of difference
            strongUpDifference: 100,
            upDifference: 10,
            downDifference: -10,
            strongDownDifference: -100
        }
    }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    so.target_amount as TargetAmount,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    so.reference_amount as ReferenceAmount
}
```

For the criticality calculation, the value of the property `improvementDirection` is crucial because this value determines what further properties are needed. If, for example, the value is `#MINIMIZE`, the properties `ToleranceRangeHighValue` and `DeviationRangeHighValue` are relevant.



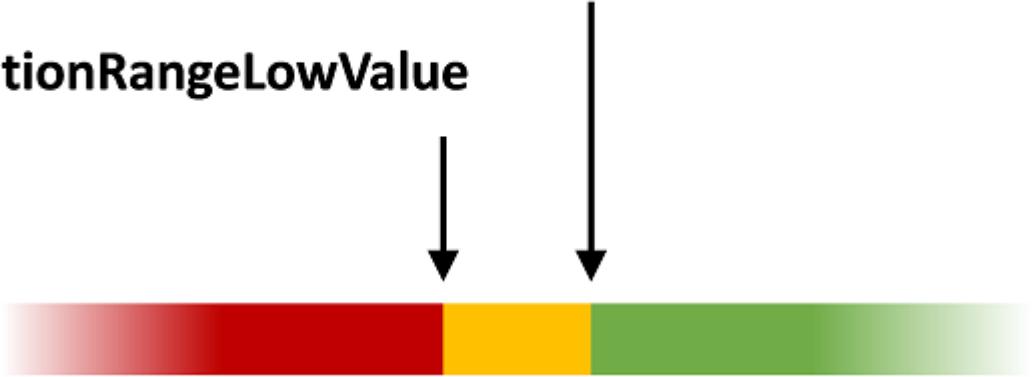
ToleranceRangeHighValue



Example of improvementDirection: # MINIMIZE

ToleranceRangeLowValue

DeviationRangeLowValue



Example of improvementDirection: # MAXIMIZE

The properties of the sub-annotation [@UI.dataPoint.criticalityCalculation \[page 656\]](#) can have either constant values or derive values from referencing to other elements. If a property references to another element, the suffix Element must be added to the name of the property.

i Note

This also applies to the properties of the sub-annotation [@UI.dataPoint.trendCalculation \[page 656\]](#), except for the property `referenceValue`. This property always references to another element.

• Example

`toleranceRangeLowValue` becomes `toleranceRangeLowValueElement`.

Related Information

[Data Points \[page 532\]](#)

[Criticality \[page 534\]](#)

[Trends \[page 535\]](#)

[Person Responsible and Reference Period \[page 540\]](#)

[DataField Type: #AS_DATAPOINT \[page 542\]](#)

7.14.6.4 Person Responsible and Reference Period

Get information about how to use data points to display references to persons responsible and to reference periods on SAP Fiori UIs.

You can add the following properties to the UI annotation `@UI.dataPoint [page 649]`:

- `referencePeriod`
- `responsibleName`

You can define both properties either in the UI annotation directly, or in another element and reference from the UI annotation to this element.

Example

In the following example, the data point has a static reference period and a static person responsible. The value of the gross amount is formatted with the `valueFormat` property. The value is thus scaled with factor 1000 and is displayed with one decimal place, this is the value 34500 EUR would be displayed as `34.5 kEUR`.



Example of visualization of person responsible, reference period, and value format

↳ Sample Code

```
...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    key so.buyer_guid as BuyerGuid,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,

    @UI.dataPoint: {
        title: 'Gross Amount',
        description: 'Gross Amount per Customer',
        longDescription: 'The gross amount per customer ...',
        valueFormat: {
            scaleFactor: 1000,
            numberOfFractionalDigits: 1
        },
        referencePeriod: { description: '2015 Q3' },
        responsibleName: 'John Doe'
    }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    so.actual_amount as ActualAmount
}
```

❖ Example

In the following example, a dynamic reference period is used that is supplied by the following parameters:

- start
- end

These parameters have to be aliased in the element list before they can be used in the [@UI.dataPoint \[page 649\]](#) annotation. The responsible property must refer to a to-one-association. The target entity of this association should contain the contact data of the person responsible.

↳ Sample Code

```
...
define view ZExample_SalesOrdersByCust
with parameters p_StartDate : abap.dats,
                  p_EndDate   : abap.dats
as select from ... as so
association [0..1] to Employees as _PersonResponsible
    on _PersonResponsible.EmployeeId = $projection.PersonResponsible
{
    ...
    $parameters.p_StartDate as StartDate,      -- Alias is required for
annotation
    $parameters.p_EndDate as EndDate,          -- Alias is required for
annotation
    so.person_responsible as PersonResponsible,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,

    @UI.dataPoint: {
        title: 'Gross Amount',
        referencePeriod: {
            start: 'StartDate',           -- Reference to element
            end: 'EndDate'              -- Reference to element
        },
        responsible: '_PersonResponsible'       -- Reference to association
    }
    @Semantics.amount.currencyCode: 'CurrencyCode'
    @DefaultAggregation: #SUM
}
```

```

        so.actual_amount as ActualAmount,
        _PersonResponsible
    }
where so.validity_date >= $parameters.p_StartDate
and   so.validity_date <= $parameters.p_EndDate

```

For a definition of element list, see section *Glossary* linked below.

Related Information

[Data Points \[page 532\]](#)

[Criticality \[page 534\]](#)

[Trends \[page 535\]](#)

[Trend-Criticality Calculation \[page 537\]](#)

[DataField Type: #AS_DATAPOINT \[page 542\]](#)

7.14.6.5 DataField Type: #AS_DATAPOINT

Get information about how to use the type #AS_DATAPOINT to refer to other annotations.

The type #AS_DATAPOINT maps to *DataFieldForAnnotation*. *DataFieldForAnnotation* is used to refer to other annotations using the *Edm.AnnotationPath* abstract type. The annotation path must end in *vCard.Address* or *UI.dataPoint*.

You can use the following type to reference an exposed data point from dataField-like annotations:

- #AS_DATAPOINT

You use this type to include a microchart in the UI annotation [@UI.lineItem \[page 683\]](#), for example.

❖ Example

In this example, the UI annotation [@UI.lineItem \[page 683\]](#) has to be defined at the same CDS element as the UI annotation [@UI.dataPoint \[page 649\]](#) itself.

↳ Sample Code

```

...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    key so.buyer_guid as BuyerGuid,
    ...
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,
    ...
    @UI.dataPoint: { title: 'Gross Amount' }
    @UI.lineItem: [ { type: #AS_DATAPOINT } ]
    @Semantics.amount.currencyCode: 'CurrencyCode'
    so.actual_amount as ActualAmount
}

```

Related Information

[Data Points \[page 532\]](#)

[Criticality \[page 534\]](#)

[Trends \[page 535\]](#)

[Trend-Criticality Calculation \[page 537\]](#)

[Person Responsible and Reference Period \[page 540\]](#)

7.14.7 Contact Data

Get information about what UI annotations to use to display contact data on SAP Fiori UIs.

In some cases users of an application need to see contact data, for example, of business partners, customers, or employees.

You can use the following annotation set to inform a client that an entity contains contact information and map the CDS elements to the corresponding address field:

- [@Semantics](#)

This annotation set contains annotations to inform about telephone numbers, email addresses, names, addresses, and contacts.

❖ Example

The following example contains sub-annotations belonging to the annotation set [@Semantics](#). For a complete list, see section [Semantics Annotations](#) linked below.

↳ Sample Code

```
...
define view Employees as select from ...
{
    key EmployeeId,
    @Semantics.name.givenName
    FirstName,
    @Semantics.name.additionalName
    MiddleName,
    @Semantics.name.familyName
    LastName,
    GenderCode,
    @Semantics.telephone.type: [#WORK, #PREF]
    PhoneNumber,
    @Semantics.telephone.type: [#FAX]
    FaxNumber,
    @Semantics.telephone.type: [#CELL]
    MobilePhoneNumber,
    @Semantics.eMail.address
    EmailAddress,
    PreferredLanguage,
    @Semantics.contact.birthDate
    BirthDate
}
```

Related Information

[Semantics Annotations \[page 606\]](#)

7.14.8 Navigation

Get an overview of how to use *dataField* types to provide means of navigation on SAP Fiori UIs.

It often is not sufficient to stay on one screen. Users might need to navigate between screens or even to web sites outside an application. You can use the following *dataField* types to include navigation concepts:

- [#WITH_NAVIGATION_PATH \[page 544\]](#)
Used for navigation within an application.
- [#WITH_URL \[page 546\]](#)
Used for navigation from an application to an external web site.
- [#FOR_INTENT_BASED_NAVIGATION \[page 548\]](#)
Used for navigation based on an action that is related to a semantic object.

7.14.8.1 With Navigation Path

Get information about how to provide navigation between UI screens and pages on SAP Fiori UIs.

This navigation type contains either a navigation property or a term cast. The term either is of type Edm.EntityType, a concrete entity type, or a collection of these types.

The screenshot shows a SAP Fiori application interface. At the top, there is a search bar with placeholder text 'Search' and a dropdown menu labeled 'Standard *'. To the right of the search bar are buttons for 'Hide Filter Bar', 'Filters', and 'Go'. Below the search bar is a field labeled 'Company:' with a dropdown arrow icon.

The main area displays a table titled 'Sales Orders (1,216) | Demo'. The table has four columns: 'Sales Order ID', 'Company', 'Currency Code', and 'Gross Amount'. The 'Business Partner' column header is highlighted with a yellow box. The table lists 15 rows of sales order data, each with a radio button next to the ID, the company name, the currency code (all listed as EUR), and the gross amount.

Sales Order ID	Company	Currency Code	Gross Amount
500000000	SAP	EUR	25,867.03 EUR >
500000001	DelBont Industries	EUR	14,602.49 EUR >
500000002	TECUM	EUR	5,631.08 EUR >
500000003	Asia High tech	EUR	1,704.04 EUR >
500000004	Asia High tech	EUR	761.24 EUR >
500000005	AVANTEL	EUR	101,299.22 EUR >
500000006	Talpa	EUR	250.73 EUR >
500000007	Panorama Studios	EUR	10,311.35 EUR >
500000008	Telecomunicaciones Star	EUR	195.16 EUR >
500000009	SAP	EUR	3,972.22 EUR >
500000010	DelBont Industries	EUR	827.95 EUR >
500000011	Panorama Studios	EUR	325.94 EUR >
500000012	TECUM	EUR	24,704.40 EUR >
500000013	Asia High tech	EUR	8,256.22 EUR >
500000014	Asia High tech	EUR	3,459.33 EUR >
500000015	AVANTEL	EUR	862.73 EUR >

Example of dataField of type #WITH_NAVIGATION_PATH

You can use the following dataField type to expose a link to other pages of a UI:

- #WITH_NAVIGATION_PATH

❖ Example

In the following example, `CompanyName` is displayed as link referring to the association `_BusinessPartner`.

≡ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as
so
    association [0..1] to sepm_cds_business_partner as _BusinessPartner
    on $projection.buyer_guid = _BusinessPartner.business_partner_key
{
    key so.sales_order_id as SalesOrder,
    so.buyer_guid,
    ...
    @UI.lineItem: [ {
        position: 20,
```

```
    type: #WITH_NAVIGATION_PATH,
    targetElement: '_BusinessPartner'      -- Reference to association
  }
  so.customer.company_name as CompanyName,
  ...
  _BusinessPartner
}
```

Related Information

[Navigation \[page 544\]](#)

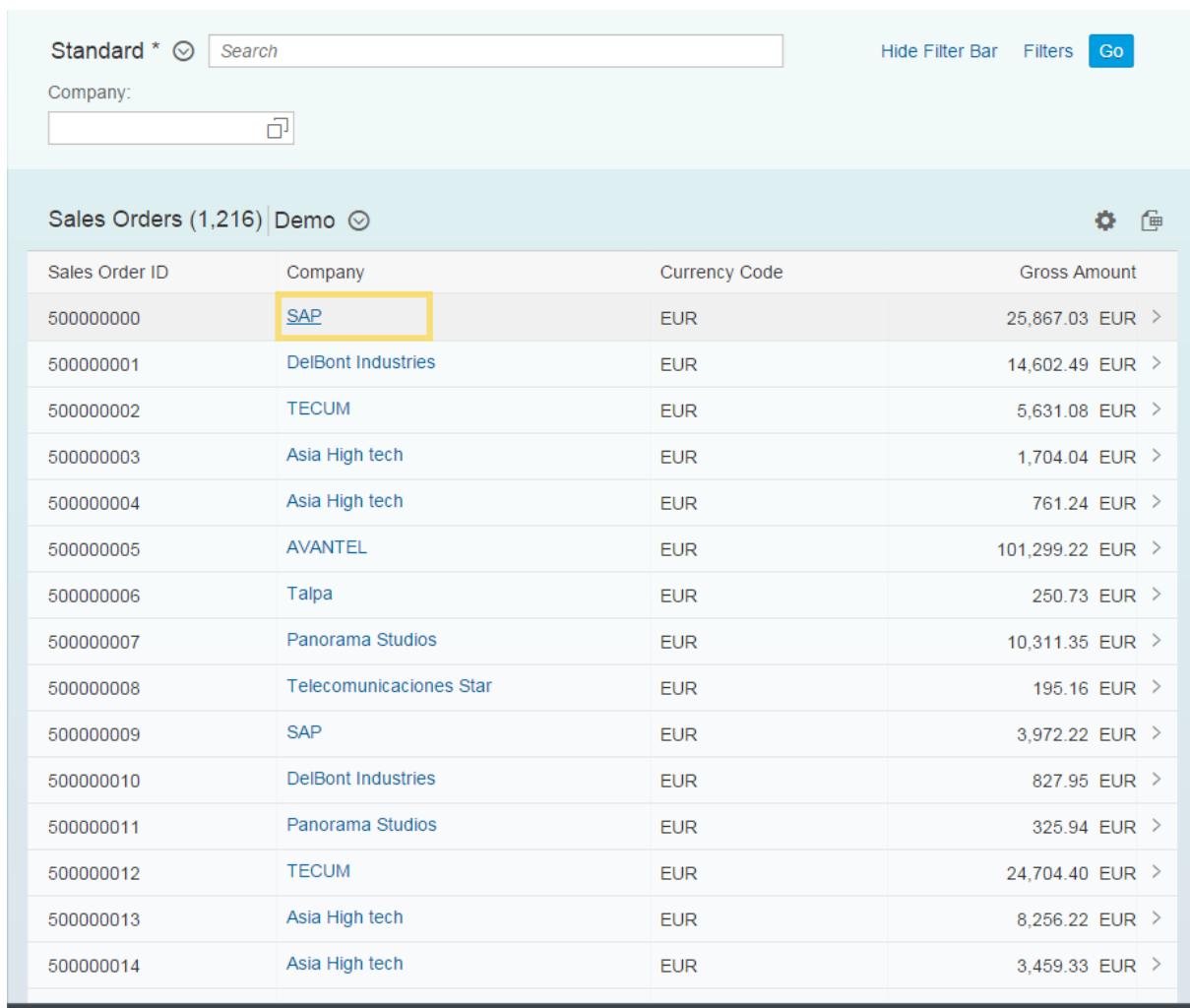
[With URL \[page 546\]](#)

[Based on Intent \[page 548\]](#)

7.14.8.2 With URL

Get information about how to provide navigation from SAP Fiori UIs to external web sites, for example.

This type navigation contains a reference to a URL to navigate to specific web sites, for example.



Sales Order ID	Company	Currency Code	Gross Amount
500000000	SAP	EUR	25,867.03 EUR >
500000001	DelBont Industries	EUR	14,602.49 EUR >
500000002	TECUM	EUR	5,631.08 EUR >
500000003	Asia High tech	EUR	1,704.04 EUR >
500000004	Asia High tech	EUR	761.24 EUR >
500000005	AVANTEL	EUR	101,299.22 EUR >
500000006	Talpa	EUR	250.73 EUR >
500000007	Panorama Studios	EUR	10,311.35 EUR >
500000008	Telecomunicaciones Star	EUR	195.16 EUR >
500000009	SAP	EUR	3,972.22 EUR >
500000010	DelBont Industries	EUR	827.95 EUR >
500000011	Panorama Studios	EUR	325.94 EUR >
500000012	TECUM	EUR	24,704.40 EUR >
500000013	Asia High tech	EUR	8,256.22 EUR >
500000014	Asia High tech	EUR	3,459.33 EUR >

Example of dataField of type #WITH_URL

You can use the following `dataField` type to display links to external websites:

- `#WITH_URL`

Example

In the following example, `CompanyName` is displayed as link referring to the CDS element `WebsiteUrl`.

Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as
so
{
  key so.sales_order_id as SalesOrder,
  ...

  @UI.lineItem: [ {
    position: 20,
    type: #WITH_URL,
    url: 'WebsiteUrl'      -- Reference to element
  } ]
  so.customer.company_name as CompanyName,
```

```
    so.customer.web_address as WebsiteUrl,  
    ...  
}
```

Related Information

[Navigation \[page 544\]](#)

[With Navigation Path \[page 544\]](#)

[Based on Intent \[page 548\]](#)

7.14.8.3 Based on Intent

Get information about how to provide navigation related on actions that are executed on SAP Fiori UIs.

This navigation type contains an action that is related to a semantic object. This combination of action and semantic object is an **intent**. The annotation [@Consumption.semanticObject \[page 563\]](#) is required for navigation based on intent. The client decides how to react when this navigation is triggered.

The screenshot shows a SAP Fiori application interface. At the top, there is a header with the SAP logo and a user dropdown set to 'Default User'. Below the header, the page title is 'AnnoDokuExmaple'. A search bar with placeholder 'Search' and a 'Go' button are on the right. Underneath, a filter bar includes 'Standard *' with a dropdown, 'Company:' with a dropdown menu, and buttons for 'Hide Filter Bar', 'Filters', and 'Go'. The main content area displays a table titled 'Sales Orders (1,216) | Demo'. The table has four columns: 'Sales Order ID', 'Company', 'Currency Code', and 'Gross Amount'. Each row contains a radio button, the sales order ID, the company name, the currency code (EUR), and the gross amount. The first row, which has a selected radio button, corresponds to the data shown in the detailed view below. The detailed view shows the same four columns for the selected row: Sales Order ID 500000000, Company SAP, Currency Code EUR, and Gross Amount 25,867.03 EUR. There is also a 'Show customer-details' button with a yellow border, a gear icon, and a copy icon. At the bottom right of the main content area is a small blue arrow icon.

Example of dataField of type #FOR_INTENT_BASED_NAVIGATION

You can use the following ***dataField*** type to expose the intent to navigate without specifying how this navigation is to be resolved:

- #FOR_INTENT_BASED_NAVIGATION

❖ Example

In the following example, the intent 'Show' (action) 'BusinessPartner' (semantic object) is expressed. The client can, for example, open a separate application to display the details of the corresponding business partner.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as
so
{
```

```
key so.sales_order_id as SalesOrder,  
...  
  
@UI.lineItem: [ {  
    position: 20,  
    label: 'Show customer-details',  
    type: '#FOR_INTENT_BASED_NAVIGATION',  
    semanticObjectAction: 'Show'           -- Action  
} ]  
@Consumption.semanticObject: 'BusinessPartner'   -- Semantic Object  
so.customer.company_name as CompanyName,  
...  
}
```

Related Information

[Navigation \[page 544\]](#)

[With Navigation Path \[page 544\]](#)

[With URL \[page 546\]](#)

7.14.9 Actions

Get information about how to use *dataField* types to provide means of executing actions on SAP Fiori UIs.

Actions are directly related to items that you can see in a table on a master-detail floorplan, for example. Users can select items and execute certain actions on the selected items.

The screenshot shows a SAP Fiori application interface. At the top, there's a header with a house icon, the SAP logo, and a user dropdown labeled "Default User". Below the header, the page title is "AnnoDokuExmaple". A navigation bar includes a back arrow, a search input field, and buttons for "Hide Filter Bar", "Filters", and "Go". A "Company:" filter is present. The main content area displays a table titled "Sales Orders (1,216)" with a "Demo" link. The table has columns: Sales Order ID, Company, Currency Code, and Gross Amount. A "Copy" button is highlighted with a yellow border. The table data is as follows:

Sales Order ID	Company	Currency Code	Gross Amount
5000000000	SAP	EUR	25,867.03 EUR >
5000000001	DelBont Industries	EUR	14,602.49 EUR >
5000000002	TECUM	EUR	5,631.08 EUR >
5000000003	Asia High tech	EUR	1,704.04 EUR >
5000000004	Asia High tech	EUR	761.24 EUR >
5000000005	AVANTEL	EUR	101,299.22 EUR >
5000000006	Talpa	EUR	250.73 EUR >
5000000007	Panorama Studios	EUR	10,311.35 EUR >
5000000008	Telecomunicaciones Star	EUR	195.16 EUR >
5000000009	SAP	EUR	3,972.22 EUR >
5000000010	DelBont Industries	EUR	827.95 EUR >
5000000011	Panorama Studios	EUR	325.94 EUR >
5000000012	TECUM	EUR	24,704.40 EUR >
5000000013	Asia High tech	EUR	8,256.22 EUR >
5000000014	Asia High tech	EUR	3,459.33 EUR >
5000000015	AVANTEL	EUR	862.73 EUR >

Example of action 'Copy' on master-detail floorplan

You can use the following `dataField` type to expose actions to the client:

- `#FOR_ACTION`

This property has to be assigned to some arbitrary element. It is thereby irrelevant if the property refers to the element to which the property is assigned.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so {
  @UI.lineItem: [
    -- Standard Lineitem
    { position: 10 },
    -- Action Lineitem
    { type: #FOR_ACTION, dataAction: 'Copy', label: 'Copy' }
  ]
  key so.sales_order_id as SalesOrder,
  ...
}
```

```
}
```

7.14.10 Field Manipulation

Get information about what UI annotations to use to manipulate fields for SAP Fiori UIs.

This chapter describes annotations that influence the appearance exposed fields. When a field is marked with these annotations, it is manipulated no matter in what other annotations the field is used. The reason for this is that annotations for manipulation are self-contained annotations and not properties of other annotations.

For example, when a field is marked with the [@UI.masked \[page 676\]](#) annotation, the field is masked regardless if it is used in a [@UI.lineItem \[page 683\]](#) annotation or a [@UI.identification \[page 690\]](#) annotation.

To manipulate the appearance of fields on SAP Fiori UIs, you can use the annotations explained in the following sections:

- [Multi-Line Text \[page 552\]](#)
- [Field Masking \[page 553\]](#)
- [Field Hiding \[page 554\]](#)
- [Interaction with Other Annotations \[page 555\]](#)

7.14.10.1 Multi-Line Text

Get information about what UI annotations to use to display fields as multi-line text on SAP Fiori UIs.

You can use the following annotation to mark a field to be displayed by a control that supports multi-line input, for example a text area:

- [@UI.multiLineText \[page 677\]](#)

↳ Sample Code

```
...
define view Product as select from ... {
    @UI.identification: [ { position: 10 } ]
    key ProductID,
    @UI.identification: [ { position: 20 } ]
    ProductName,
    @UI.identification: [ { position: 30 } ]
    @UI.multiLineText: true
    Description,
}
...
```

Related Information

[Field Manipulation \[page 552\]](#)

[Field Masking \[page 553\]](#)

[Field Hiding \[page 554\]](#)

[Interaction with Other Annotations \[page 555\]](#)

7.14.10.2 Field Masking

Get information about what UI annotations to use to mask fields, for example for password input, on SAP Fiori UIs.

In some cases, data of fields need to be consumed by the client, but must not be visible on the UI. This field behavior is required when users need to enter passwords, for example.

You can use the following annotation to mark a field to not to be displayed in clear text by the client because, for example, it contains sensitive data:

- [@UI.Masked \[page 676\]](#)

This annotation does not influence how data is transferred. If a field is marked with the [@UI.masked \[page 676\]](#) annotation, the data belonging to this field is still transferred to the client like any other property in clear text.

↳ Sample Code

```
...
define view Destination as select from ... {
    @UI.identification: [ { position: 10 } ]
    key DestinationID,
    ...
    @UI.identification: [ { position: 20 } ]
    AuthType,          -- None, Basic, SSO, ...
    @UI.identification: [ { position: 30 } ]
    BasicAuthUserName,
    @UI.identification: [ { position: 40 } ]
    @UI.masked
    BasicAuthPassword,
    ...
}
```

Related Information

[Field Manipulation \[page 552\]](#)

[Multi-Line Text \[page 552\]](#)

[Field Hiding \[page 554\]](#)

[Interaction with Other Annotations \[page 555\]](#)

7.14.10.3 Field Hiding

Get information about what UI annotations to use to hide fields from SAP Fiori UIs.

Generally, all fields that are exposed by the OData service are available to the client, regardless if the fields are exposed explicitly using UI annotations. To enable end-user personalization, the client may offer the possibility to add fields that are hidden by default, for example to a list report.

You can use the following annotation to prevent fields from being displayed on a UI and in the personalization dialog, but leaving the field available for client:

- [@UI.hidden \[page 676\]](#)

You can use this annotation if, for example, a CDS view contains technical keys, for example GUIDs, that have to be exposed to the OData service to work. These keys are usually not supposed to be displayed on the UI. You can also use this annotation if fields are required in calculations, but are not supposed to be displayed on a UI.

❖ Example

In the following example, the annotation [@UI.dataPoint \[page 649\]](#) with pre-calculated criticality and trend is exposed. The hidden fields `AmountCriticality` and `AmountTrend` are required by the client to calculate the corresponding values, but are not supposed to be displayed on the UI.

≡ Sample Code

```
...
define view ZExample_SalesOrdersByCustomer as select from ... as so {
    @UI.hidden
    key so.buyer_guid as BuyerGuid,
    ...

    @UI.dataPoint: {
        criticality: 'AmountCriticality',          -- Reference to element
        trend: 'AmountTrend',                      -- Reference to element
    }
    so.actual_amount as ActualAmount,
    @UI.hidden
    so.criticality as AmountCriticality,
    @UI.hidden
    so.trend as AmountTrend
}
```

Related Information

[Consumption Annotations \[page 562\]](#)

[Field Manipulation \[page 552\]](#)

[Multi-Line Text \[page 552\]](#)

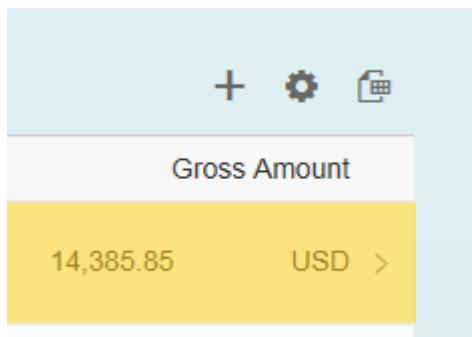
[Field Masking \[page 553\]](#)

[Interaction with Other Annotations \[page 555\]](#)

7.14.10.4 Interaction with Other Annotations

Get information about how to provide interaction between annotations.

In addition to UI annotation, you can use model-specific annotations that affect the desired client behavior. You can implement, for example, unit-currency-mappings, ID-text-mappings, or properties for field control that are represented by model-specific annotations. These model-specific annotations can be evaluated by the client and **no** additional UI annotations are required.



Example of interaction between CDS annotations

• Example

In the following example, the field `CurrencyCode` is marked with a [@Semantics.currencyCode \[page 607\]](#) and is referenced by field `GrossAmount`. This means that the field `GrossAmount` is always displayed with the corresponding currency. The field `CurrencyCode` does not need to be exposed explicitly.

Furthermore the field `GrossAmount` is marked as being mandatory. This means that the field is treated accordingly by the client: The field is marked with an asterisk, and if users do not fill in a value for this property, an error is raised. The administrative fields such as `CreatedAt` and `CreatedBy`, are set in a back-end validation and must **not** be changed by the client. For this reason, these fields are marked as being read-only.

↳ Sample Code

```
...
define view ZExample_SalesOrder as select from sepm_cds_sales_order as so
{
    @UI.identification: [ { position: 10 } ]
    key so.sales_order_id as SalesOrder,
    @Semantics.currencyCode: true
    so.currency_code as CurrencyCode,

    @Semantics.amount.currencyCode: 'CurrencyCode'
    @ObjectModel.mandatory: true
    @UI.identification: [ { position: '20' } ]
    so.gross_amount as GrossAmount,
    ...
    @ObjectModel.readOnly: true
    @UI.identification: [ { position: '110' } ]
    so.created_at as CreatedAt,
    ...
    @ObjectModel.readOnly: true
    @UI.identification: [ { position: '120' } ]
    so.created_by as CreatedBy,
    ...
    @ObjectModel.readOnly: true
}
```

```

@UI.identification: [ { position: '130' } ]
so.changed_at as ChangedAt,
@ObjectModel.readOnly: true
@UI.identification: [ { position: '140' } ]
so.changed_by as ChangedBy,
}

```

Related Information

[Field Manipulation \[page 552\]](#)

[Multi-Line Text \[page 552\]](#)

[Field Masking \[page 553\]](#)

[Field Hiding \[page 554\]](#)

7.14.10.5 Inheritance of Annotations

Get information about what property to use to prevent elements from being inherited from an underlying CDS view.

By default, all UI annotation elements are inherited from the underlying CDS view. You can explicitly disable this behavior. You can use the following property to prevent a UI annotation element from being inherited:

- *exclude*

The following sample code depicts the CDS view `ZP_SalesOrder` that inherits elements from the underlying CDS view `SEPM_CDS_SALES_ORDER`, and uses the UI annotation [@UI.identification \[page 690\]](#):

↳ Sample Code

```

...
define view ZP_SalesOrder as select from sepm_cds_sales_order as so {
    @UI.identification: [ { position: 10 } ]
    key so.sales_order_id as SalesOrder,
    @UI.identification: [ { position: 20 } ]
    so.customer.company_name as CompanyName,
}
...
```

The following sample code depicts the CDS view `ZI_SalesOrder` that inherits elements from the underlying CDS view `ZP_SalesOrder`. In this view, the element `key SalesOrder` is inherited from the underlying CDS view as UI annotation [@UI.identification \[page 690\]](#) by default. The element `so.customer.company_name as CompanyName`, however, is not inherited as UI annotation [@UI.identification \[page 690\]](#) because of the property `exclude`:

↳ Sample Code

```

...
define view ZI_SalesOrder as select from ZP_SalesOrder as so {
    key SalesOrder,
```

```
@UI.identification: [ { exclude } ]
so.customer.company_name as CompanyName,
...
}
```

8 Reference

8.1 CDS Annotations

The following list summarizes SAP annotations of the Data Definition Language (DDL) of ABAP CDS that are relevant in the context of ABAP RESTful programming model and released for ABAP Cloud Platform.

SAP CDS annotations are evaluated by SAP frameworks and can be either **ABAP annotations** or **framework-specific annotations**.

ABAP CDS - ABAP Annotations

CDS annotations that are evaluated by **ABAP runtime**:

- AbapCatalog Annotations
- AccessControl Annotations
- ClientHandling Annotations
- EndUserText Annotations
- Environment Annotations
- MappingRole Annotations
- Metadata Annotations
- Semantics Annotations

See also: [ABAP CDS - View Annotations \(ABAP Keyword Documentation\)](#)

→ Tip

To access help for an ABAP annotation, position the cursor on the relevant annotation in the DDL editor and choose **F1**.

Framework-Specific Annotations

Framework-specific CDS annotations (as a rule) are exposed for OData and evaluated during runtime.

- Aggregation Annotations [\[page 559\]](#)
- AccessControl Annotations [\[page 561\]](#)
- Consumption Annotations [\[page 562\]](#)
- ObjectModel Annotations [\[page 565\]](#)
- OData Annotations [\[page 568\]](#)
- Search Annotations [\[page 602\]](#)

- Semantics Annotations [page 606]
- UI Annotations [page 610]

Related Information

[CDS Annotation Syntax](#)

8.1.1 Aggregation Annotations

Specifies aggregation behavior on element level.

Scope and Definition

```
@Scope:[#ELEMENT]
annotation Aggregation
{
    default: String(30) enum
    {
        NONE;
        SUM;
        MIN;
        MAX;
        AVG;
        COUNT_DISTINCT;
        NOP;
        FORMULA;
    };
    referenceElement : array of ElementRef;
};
```

Usage

Annotation	Meaning
Aggregation.Default	When the <i>Aggregation</i> annotation has been specified for an element, the corresponding elements are used as so called <i>measures</i> (elements that can be aggregated) in analytical scenarios. These measures are aggregated automatically with the Aggregation. In SQL SELECT statements you have to specify the aggregation behavior explicitly. Scope: [ELEMENT] Evaluation Runtime (Engine): SADL
Value	Description

Annotation	Meaning
SUM, MAX, MIN, AVG	All these values determine the aggregation of the measure.
COUNT_DISTINCT	Counts the number of distinct values of the annotated element. In combination with the annotation {@Aggregation.ReferenceElement: ['elementRef']}, you can count the number of distinct values of another element that is referenced.
Aggregation.ReferenceElementRef ceElement	References the element that is used for distinct counts.

i Note
Can only be used in combination with
{@Aggregation.Default: #COUNT_DISTINCT}.

• Example

The example demonstrates how you can use the aggregate functions on measure elements.

```
define view /DMO/I_TRAVEL_Aggr as select from /dmo/travel
{
    key travel_id           as TravelID,
    agency_id                as AgencyID,
    customer_id              as CustomerID,
    @Aggregation.default: #MIN
    begin_date                as BeginDate,
    @Aggregation.default: #MAX
    end_date                  as EndDate,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    @Aggregation.default: #SUM
    booking_fee                as BookingFee,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    @Aggregation.default: #AVG
    total_price                 as TotalPrice,
    @Semantics.currencyCode: true
    currency_code               as CurrencyCode,
    @Aggregation.referenceElement: ['CustomerID']
    @Aggregation.default: #COUNT_DISTINCT
    cast(1 as abap.int4) as DistintCustomers
}
```

8.1.2 AccessControl Annotations

Enable application developers to define how the authorization check for a CDS entity is executed

Scope and Definition

```
@Scope:[#VIEW, #TABLE_FUNCTION]
AccessControl.authorizationCheck : String(20) enum { NOT_REQUIRED; NOT_ALLOWED;
CHECK; PRIVILEGED_ONLY; } default #CHECK;
```

Usage

Annotation	Meaning
AccessControl.authorizationCheck	This element defines the behavior of the authorization check. Scope: [#VIEW] Engine Behavior: The runtime and design-time engines handle the authorization check based on the value of the element. Values:
<hr/>	
#NOT_REQUIRED	During the authorization runtime, an authorization check is executed if a DCL role exists for the entity. If no role exists there is no check and no protection. This behavior is the same behavior at runtime as for value #CHECK. However in this case it is intended by the developer that no role exists. During development, no warning occurs when activating the entity.
#NOT_ALLOWED	During the authorization runtime, no authorization check is executed. During development, a warning occurs if a developer activates a role for an entity, which has this annotation value.
#CHECK	During the authorization runtime, an authorization check is executed if a DCL role exists for the entity. If no role exists there is no check and no protection. During development, a warning occurs if a developer activates the entity and no DCL role exists for the entity. This value is the default value.

NOTE

The value `#NOT_REQUIRED` is recommended for entities for which no authorization checks are planned yet, but might be needed by the developer or customer later. To prohibit roles for the entity, use the value `#NOT_ALLOWED`.

Example

When the developer activates the following DDL document, since an authorization check is not required, ABAP development tools do not produce a warning. It does not matter whether a role exists for the entity or not.

At runtime, if there is a role for the entity, then ABAP performs an authorization check with the role. If there is no role, there is no check and no protection for the entity.

↳ Sample Code

```
@AbapCatalog.sqlViewName: 'DEMO_CDS_PRJCTN'  
@AccessControl.authorizationCheck: #NOT_REQUIRED  
define view demo cds spfli  
as select from spfli  
{ key spfli.carrid,  
  key spfli.connid,  
  spfli.cityfrom,  
  spfli.cityto }
```

8.1.3 Consumption Annotations

Define a specific behavior that relates to the consumption of CDS content through domain-specific frameworks.

Usage

Via these annotations, the specific behavior is defined which is related to the consumption of CDS content. This metadata makes no assumptions about the concrete consumption technology/infrastructure, but it is applicable across multiple consumption technologies (e.g. Analytics or OData).

Annotation	Definition
Consumption.semanticObject:	<p>This annotation leverages enhanced interoperability across applications. The semantic semantic object that is defined in the Fiori Launchpad must be specified as the value.</p>
<p>Example</p> <p>SAP Fiori has introduced the concept of intent-based navigation, whereby an intent is a combination of <semanticObject> <action>. A semanticObject annotation is used in SAP Fiori UIs to dynamically derive navigation targets for the annotated view as a source.</p>	
<p>Scope: #ELEMENT, #PARAMETER, #ENTITY</p> <p>For more information, see Based on Intent [page 548].</p>	
<p>Annotations belonging to <i>Consumption.valueHelpDefinition</i> directly establish a relationship to an entity that acts as a value help provider.</p>	
<p>The value help can be consumed without an association to the target value help provider.</p>	
Scope: [ELEMENT, PARAMETER]	
Evaluation Runtime (Engine): Interpreted by ABAP Runtime Environment	
Consumption.valueHelpDefinition.addition alBinding[]	Defines an additional binding condition for the value help on the same target value help provider entity for filtering the value help result list and/or returning values from the selected value help record.
element	Specifies the element in the target value help provider entity that is linked to the local element or parameter for the additional binding.
localElement	Specifies the local element that is linked to the element or parameter in the target value help provider entity for the additional binding.
localParameter	Specifies the local parameter that is linked to the element or parameter in the target value help provider entity for the additional binding.
parameter	Specifies the parameter in the target value help provider entity that is linked to the local element or parameter for the additional binding.
usage	The binding may either specify an additional filter-criterion on the value help list (#FILTER), or an additional result mapping for the selected value help record (#RESULT) or a combination thereof (#FILTER_AND_RESULT). If not specified explicitly the usage is #FILTER_AND_RESULT. If <i>distinctValues</i> is set to true, additional bindings must specify the usage as #FILTER.

Annotation	Definition	
Consumption.valueHelpDefinition.distinctValues	Specifies whether the value help result list shall only contain distinct values for the annotated field or parameter. If set to true all mappings will be used for filtering, but only the value for the field/parameter which the value help was requested for will be returned by the value help.	
Consumption.valueHelpDefinition.entity[]	Defines the binding for the value help to the value help providing entity. It requires specification of the entity and the element providing the value help for the annotated element.	
element	Value: elementRef	Description: Specifies the element in the entity referenced in name that provides the value help for the annotated element.
name	Value: entityRef	Description: Specifies the entity which contains the element that provides the value help.
Consumption.valueHelpDefinition.label	<p>This annotation contains a language-dependent text that is used to label the value list.</p> <p>If not specified the label of the value help defining entity is used.</p>	
Consumption.valueHelpDefinition.presentationVariantQualifier	The Presentation Variant indicates how the value help result should be displayed.	
Consumption.valueHelpDefinition.qualifier	<p>Uniquely identifies alternative values for an annotation.</p> <p>Omission means the OData term is applied without explicit qualifier.</p> <p>If more than one value help is defined for one element, a qualifier must be used.</p>	

Example 1

The annotation `Consumption.valueHelpDefinition` is used to define a value help for the annotated element. The value help provider can be a different CDS entity without association. To consume the value help, the value help provider entity must be added to the respective OData service.

You can filter the available value help options by defining an additional binding. In the following example case, only the business partners are displayed that use the same currency code.

↳ Sample Code

```
DEFINE VIEW BuPaView AS SELECT FROM db_bp
{
    key bp_bp_id
        as BusinessPartnerID,
    ...
    bp.currency_code
        as CurrencyCode,
```

```

        bp.company_name           as CompanyName,
    }
DEFINE VIEW SOView AS SELECT FROM db_so as so
{
    so.sales_order_id as SalesOrderID,
    ...
    so.CurrencyCode as CurrCode,
    @Consumption.valueHelpDefinition: [{ entity : { name      :
'I_AIVS_BusinessPartner',
                           element      :
'BusinessPartnerID'},
                                         additionalBinding   :
[{ localElement   : 'CurrCode',
  element       : 'CurrencyCode'
} ]
                           _BusinessPartner.BusinessPartnerID
}

```

Example 2

↳ Sample Code

```

define view sales_order_vh as select from SalesOrder as so
{
    ...
    @Consumption.valueHelpDefinition: [{ qualifier: 'ValueHelp2',
                                         entity : { name      :
'I_BusinessPartner',
                           element      :
'BusinessPartnerID',
                           label     : 'Business Partner Value Help'
},
                                         entity : { name      :
'I_BuPa',
                           element      :
'BusinessPartnerID',
                           label     : 'Business Partner VH'
} ]
                           _BusinessPartner.BusinessPartnerID
}

```

8.1.4 ObjectModel Annotations

Provide definitions of text-related aspects of the CDS-based data model

Scope and Definition

```

@Scope:[#ELEMENT]
@API.state: [ #RELEASED_FOR_SAP_CLOUD_PLATFORM ]
text
{
    element : array of ElementRef;
    association : AssociationRef;
}

```

```

control : String(60) enum { NONE; ASSOCIATED_TEXT_UI_HIDDEN; };
reference
{
    association : AssociationRef;
};
};

@Scope:[#VIEW, #CUSTOM_ENTITY]
query
{
    implementedBy : String(255);
};

@Scope:[#ELEMENT]
virtualElementCalculatedBy : String(255);

```

Usage

Annotation	Meaning
ObjectModel. text.elemen t[]	<p>Establishes the conjunction of a field with its descriptive language-independent texts.</p> <p>Scope: [ELEMENT]</p> <p>Evaluation Runtime (Engine): SADL - First text field listed in the annotation array will be handled as descriptive text of the annotated field in OData exposure scenarios.</p>
ObjectModel. text.associa tion	<p>Defines the associated view, which provides textual descriptions for the annotated field.</p> <p>Scope: [ELEMENT]</p> <p>Evaluation Runtime (Engine):</p> <ul style="list-style-type: none"> • SADL - Enriches the OData entity type of the view with the textual description of the target view applying an automated language filtering. The name of the auto-generated text property will be composed out of the annotated field name and the constant suffix <code>_Text</code>. This OData property is mapped onto the first text field of the associated target CDS view annotated with <code>@Semantics.text:true</code>. • Analytic Manager - Uses the associated view as TEXT view for annotated field.
ObjectModel. text.control	<ul style="list-style-type: none"> • Value: { ASSOCIATED_TEXT_UI_HIDDEN } Scope: [VIEW, ELEMENT] Suppresses the text element of the text association to be visible on the UI. • Value: { NONE } Scope: [ELEMENT] Overrules the annotation <code>ObjectModel.text.control: { ASSOCIATED_TEXT_UI_HIDDEN }</code> that is specified on entity level. For the annotated element the text element of the text association is visible on the UI.
ObjectModel. text.referen ce.associati on	<p>Scope: [VIEW, ELEMENT]</p> <p>Value: 'AssocRef'</p> <p>The annotations defines an associated view which defines textual description for the annotated field. The corresponding code field in the associated view is defined by the on-condition of the association.</p>

Annotation	Meaning
ObjectModel.query.implmentedBy:	References the query implementation class for the unmanaged query. Scope: [VIEW, CUSTOM_ENTITY]
	This annotation is evaluated when the unmanaged query is executed whereby the query implementation class is called to perform the query.
	To reference the query implementation class, ABAP: must be added to the string reference.
Example	
@ObjectModel.query.implementedBy: 'ABAP:<query_impl_class>'.	
Note	
As of SAP Cloud Platform ABAP Environment 1908, this annotation substitutes the deprecated annotation @QueryImplementedBy: ''.	
ObjectModel.virtualElementCalculatedBy:	References the calculation class for the annotated virtual element. Scope: [ELEMENT]
	This annotation defines the code exit for virtual elements. The query framework is left during runtime to retrieve the values for the virtual element in the calculation class.
	To reference the calculation class, ABAP: must be added to the string reference.

Examples

Example 1

This example demonstrates how you can define language-dependent texts with a text association.

Sample Code

```
define view I_Material
  association-[0..*] to I_MaterialText as _Text ... {
    @ObjectModel.text.association: '_Text'
    key Material,
    _Text, ...
}
define view I_MaterialText ... {
  key Material,
  @Semantics.language: true
  key Language,
  @Semantics.text: true
  MaterialName,
  @Semantics.text: true
  MaterialDescription, ...
}
```

Example 2

This example demonstrates how you can define language-independent texts within the same view.

↳ Sample Code

```
define view I_Plant ... {
    @ObjectModel.text.element: ['PlantName']
    key Plant,
    @Semantics.text: true
    PlantName, ...
}
```

8.1.5 OData Annotations

Capture OData-related aspects to expose data gained from a CDS entity in an OData service.

Scope and Definition

```
Annotation OData
{
    @Scope:[#ELEMENT]
    etag : Boolean default true;

    @Scope:[#ENTITY]
    entitySet
    {
        name : String(30);
    };
    entityType
    {
        name : String(128);
    };
    action: array of {
        name : String(128);
        localName : String(30);
    };
    property
    {
        name : String(128);
    };
    @Scope:[#SERVICE]
    schema
    {
        name : String(128);
    };
}
```

Usage

OData annotations define OData specific properties in backend development objects.

Runtime:

- Exposed for OData

D
Ae
ns
nc
or
ti
ap
tā
ili
oo
na
-
as
rc
Dc
a
ap
ty
o
[
a
M
C
t
E
:
M
C
r
N
q
I
y
D
V
e
a
n
ou
te
es
s
t
h• l
e o
e c
x a
t l
e N
r a
n m
a e
l N
n a
a m
m m
e e
o

-

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
—

f
a
n
a
c
t
i
o
n
f
o
r
a
n
a
r
b
i
t
r
a
r
y
O
D
a
t
a
s
e
r
v
i
c
e
.
T
h
—

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
—

e
a
n
n
o
t
a
t
i
o
n
m
a
p
s
t
h
e
l
o
c
a
l
N

a
m
e
t
o
t
h
e
e
x
t
e
r
—

D
Ae
ns
nc
or
ti
ap
ta
il
co
re

-
n
a
l
O
D
a
t
a
n
a
m
e

-

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ra
-
B
de
d
d
d
t
h
e
m
s
s
s
s
n
e
a
c
t
i
o
n
n
a
m
e
f
o
r
m
a
p
p
i
n
g
t
o
t
h
e
O
D
a
t
a
a
c
t
i
-

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
-
o
n
n
a
m
e
(
n
a
m
e
)
.
-
B
d
n
d
g
e
s
s
t
o
e
a
c
t
i
o
n
n
a
m
e
f
o
r
O
D
a
t
a
.

D
Ae
ns
nc
or
ti
ap
ta
il
co
re

-
S
D
ar
ti
p
e
g
B
r
E
N
i
T
Y
S
e
t]

.D
n
e
a
n
e
t
e
s
t
h
e
e
x
t
e
r
n
a
l
n
a
m
e
o

-

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
—

f
t
h
e
e
n
t
i
t
y
s
e
t
f
o
r
a
n
a
r
b
i
t
r
a
r
y
O
D
a
t
a
s
e
r
v
i
c
—

D
Ae
ns
nc
or
ti
ap
ta
ili
co
me

-

e

i
N
o
t
e
Y
o
u
c
a
n
a
l
s
o
d
e
fi
n
e
t
h
e
n
a
m
e
o
f
t
h
e

-

D
Ae
ns
nc
or
ti
ap
ta
ili
co
me

- entity sets in the service definition by using

D
Ae
ns
nc
or
ti
ap
ta
ili
co
na

a
n
a
l
i
a
s
:
e
x
p
o
s
e
/
D
M
O
/
C
—
T
r
a
v
e
1
—
U
a
s
T
r
a
v

D
Ae
ns
nc
or
ti
ap
ta
ili
co
me

e
l
.T
h
e
n
a
m
e
o
f
t
h
e
a
l
i
a
s
i
n
t
h
e
s
e
r
v
i
c
e
d
e
fi
n
i

D
Ae
ns
nc
or
ti
ap
ta
ili
co
na

- t
i
o
n
w
i
l
l
b
e
u
s
e
d
e
v
e
n
i
f
y
o
u
u
s
e
t
h
e
a
n
n
o
t
a
t
i

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ma

-
o
n
i
n
t
h
e
C
D
S
v
i
e
w
.

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
-
g
D
ar
ti
p
e
g
1
r
t
j
N
t
y
T
y
p
e
D
.e
n
a
m
e
t
e
s
t
h
e
e
x
t
e
r
n
a
l
n
a
m
e
o
-

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ne
-
f
t
h
e
e
n
t
i
t
y
t
y
p
e
f
o
r
a
n
a
r
b
i
t
r
a
r
y
O
D
a
t
a
s
e
r
v
i
-

D
Ae
ns
nc
or
ti
ap
ta
ili
co
na
—
c
e

.

A vertical bar consisting of a thick blue line at the bottom and a thin grey line extending upwards from it.

E

x
a
m
p
l
e

T
h
i
s
e
x
a
m
p
l
e

d
e
m
o
n
s
t
r
a
t
e

D
Ae
ns
nc
or
ti
ap
ta
li
co
re

s
h
o
w
O
D
a
t
a
a
n
n
o
t
a
t
i
o
n
s
c
a
n
b
e
u
s
e
d
t
o
c
h
a
n
g
e

D
Ae
ns
nc
or
ti
ap
ta
ili
co
na

t
h
e
n
a
m
e
o
f
t
h
e
e
n
t
i
t
y
t
y
p
e
i
n
t
h
e
m
e
t
a
d
a
t
a
o
f

D
Ae
ns
nc
or
ti
ap
ta
ili
co
na

- t
h
e O
D a
t a s
e r v
i c e :
:

@ O
D a t a . e n
t i t y
S e t . n a
m e :
' T r

D
Ae
ns
nc
or
ti
ap
ta
il
co
re

- a
v
e
l
'@O
D
a
t
a
.
e
n
t
i
t
Y
T
Y
p
e
.
n
a
m
e
:
'
T
r
a
v
e
l
T
Y
p
e
'
d
e
f
i
n
e
r
o

D
Ae
ns
nc
or
ti
ap
ta
il
co
ne

-

```
  o
  t
  e
  n
  t
  i
  t
  y
  /
  D
  M
  O
  /
  I
  -
  T
  R
  A
  V
  E
  L
  {
    k
    e
    Y
  T
  r
  a
  v
  e
  l
  I
  D
  :
    a
    b
```

-

D
Ae
ns
nc
or
ti
ap
ta
il
co
re

```
- a  
p  
.  
n  
u  
m  
c (
```

```
8
```

```
)
```

```
;
```

```
...
```

```
}
```

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
-
as
is
a
tP
ae
.a
an
at
af
a
y
i
t
to
rn
u
o
t
u
s
e
t
h
i
s
a
n
n
o
t
a
t
i
o
n
. .
D
e
-

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
—

c
l
a
r
e
E
T
a
g
s
i
n
b
e
h
a
v
i
o
r
d
e
fi
n
i
t
i
o
n
s
.

—

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
-
S
D
ar
ti
p
e
g
p
E
p
e
rM
tE
yN
.T
n]
ai
m
e
b
s
t
r
a
c
t
e
n
t
i
t
i
e
s
D
e
n
o
-

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ne
—
t
e
s
t
h
e
e
x
t
e
r
n
a
l
n
a
m
e
o
f
C
D
S
e
l
e
m
e
n
t
s
,
p
a
r
a
m
e
—

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ne
—
t
e
r
s
o
r
a
s
s
o
c
i
a
t
i
o
n
s
f
o
r
t
h
e
p
r
o
p
e
r
t
i
e
s
o
f
a
n
—

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ne
—

a
b
s
t
r
a
c
t
e
n
t
i
t
y
.
—

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
-
S
D
ar
o
ti
p
n
e
g
(
sl
z
h
F
m
R
a
V
.I
n
C
a
E
m
e
D
e
n
o
t
e
s
t
h
e
e
x
t
e
r
n
a
l
n
a
m
e

D
Ae
ns
nc
or
ti
ap
ta
ili
co
ne
-

o
f
t
h
e
s
e
r
v
i
c
e
i
n
a
n
O
D
a
t
a
r
e
p
r
e
s
e
n
t
a
t
i
o
n

.

D
Ae
ns
nc
or
ti
ap
ta
il
co
re
-

T
h
i
s
a
n
n
o
t
a
t
i
o
n
i
s
u
s
e
d
i
n
s
e
r
v
i
c
e
d
e
fi
n
i
t
i
o
n

D
Ae
ns
nc
or
ti
ap
ta
il
oo
re
—
S
—

8.1.6 Search Annotations

This annotation marks a view as searchable. You define the fuzziness threshold as well as the specifics of term mappings at element level.

Scope and Definition

```
@Scope:[#ENTITY]
Annotation Search
{
    searchable : Boolean default true;
};

@Scope:[#ELEMENT]
Annotation Search
{
    defaultSearchElement : Boolean default true;
    ranking : String(6) enum { HIGH = 'high'; MEDIUM = 'medium'; LOW = 'low'; }
    default #MEDIUM;
    fuzzinessThreshold : Decimal(3,2);
    termMappingDictionary : String(128);
    termMappingListId : array of String(32);
};
```

Usage

Annotation	Meaning								
Search.searchable	<p>Defines if a CDS entity is generally relevant for search scenarios. This annotation must be set in case other search-related annotations are being defined for elements of the respective CDS entity. The annotation offers a general switch and a means to quickly detect whether a view is relevant or not.</p> <p>Scope: #Entity</p> <p>Evaluation Runtime (Engine): Interpreted by Enterprise Search and SADL</p> <p>Values:</p> <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td><i>Boolean (true, false)</i></td><td>Defines whether a view is relevant for search or not. Default: true</td></tr> </tbody> </table>	Value	Description	<i>Boolean (true, false)</i>	Defines whether a view is relevant for search or not. Default: true				
Value	Description								
<i>Boolean (true, false)</i>	Defines whether a view is relevant for search or not. Default: true								
Search.defaultSearchElement	<p>Specifies that the element is to be considered in a freestyle search (for example a SELECT...) where no columns are specified.</p> <p>Usually, such a search must not operate on all elements – for performance reasons, and because not all elements (e.g. internal keys) do qualify for this kind of access.</p> <p>Scope: #Element</p> <p>Evaluation Runtime (Engine): Interpreted by Enterprise Search and SADL</p> <p>Values:</p> <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td><i>Boolean (true, false)</i></td><td>Defines weather the element is to be considered in a free-style search. Default: true</td></tr> </tbody> </table>	Value	Description	<i>Boolean (true, false)</i>	Defines weather the element is to be considered in a free-style search. Default: true				
Value	Description								
<i>Boolean (true, false)</i>	Defines weather the element is to be considered in a free-style search. Default: true								
Search.ranking	<p>Specifies how relevant the values of an element are for ranking, if the freestyle search terms match the element value.</p> <p>Scope: #Element</p> <p>Evaluation Runtime (Engine) : Interpreted by Enterprise Search</p> <p>Values:</p> <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>HIGH</td><td>The element is of high relevancy; this holds usually for ID and their descriptions.</td></tr> <tr> <td>MEDIUM</td><td>The element is of medium relevancy; this holds usually for other, important element. This is the default.</td></tr> <tr> <td>LOW</td><td>Although the element is relevant for freestyle search, a hit in this element has no real significance for a result item's ranking.</td></tr> </tbody> </table>	Value	Description	HIGH	The element is of high relevancy; this holds usually for ID and their descriptions.	MEDIUM	The element is of medium relevancy; this holds usually for other, important element. This is the default.	LOW	Although the element is relevant for freestyle search, a hit in this element has no real significance for a result item's ranking.
Value	Description								
HIGH	The element is of high relevancy; this holds usually for ID and their descriptions.								
MEDIUM	The element is of medium relevancy; this holds usually for other, important element. This is the default.								
LOW	Although the element is relevant for freestyle search, a hit in this element has no real significance for a result item's ranking.								

Annotation	Meaning				
Search.fuzzinessThreshold	<p>Specifies the least level of fuzziness (with regard to some comparison criteria passed at run-time) the element has to have to be considered in a fuzzy search at all.</p> <p>i Note</p> <p>A fuzzy search enables a certain degree of error tolerance and returns records even if the search term contains additional or missing characters or other types of spelling errors.</p> <p>i Note</p> <p>To perform a fuzzy search you have to set the search mode to <code>fuzzy</code> in the customizing settings of your ABAP system. Find the customizing node under  SAP NetWeaver Implementation Guide  Search and Operational Analytics  Enterprise Search  Search Configuration  Set Parameters for Federated Search.</p> <p>If in the customizing a value for <code>Fuzzy Similarity</code> is present, the value of the parameter <code>Search.fuzzinessThreshold</code> will become void.</p>				
	<p>Scope: #Element</p> <p>Evaluation Runtime (Engine): Interpreted by SADL</p> <p>Values:</p> <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td><i>Decimal (3,2)</i></td><td> <p>The least level of fuzziness the element has to have to be considered in a fuzzy search at all, e.g. <code>0 . 7</code>.</p> <p>The value can be between 0 and 1.</p> <p>We recommend using the default value <code>0 . 7</code> 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. A value between <code>0 . 7</code> and <code>0 . 99</code> would be most useful. Use 1 for exact matches.</p> </td></tr> </tbody> </table>	Value	Description	<i>Decimal (3,2)</i>	<p>The least level of fuzziness the element has to have to be considered in a fuzzy search at all, e.g. <code>0 . 7</code>.</p> <p>The value can be between 0 and 1.</p> <p>We recommend using the default value <code>0 . 7</code> 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. A value between <code>0 . 7</code> and <code>0 . 99</code> would be most useful. Use 1 for exact matches.</p>
Value	Description				
<i>Decimal (3,2)</i>	<p>The least level of fuzziness the element has to have to be considered in a fuzzy search at all, e.g. <code>0 . 7</code>.</p> <p>The value can be between 0 and 1.</p> <p>We recommend using the default value <code>0 . 7</code> 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. A value between <code>0 . 7</code> and <code>0 . 99</code> would be most useful. Use 1 for exact matches.</p>				
Search.termMappingDictionary	<p>Specifies the table that holds the term mappings (synonyms) to be considered in the context of a search on this view.</p> <p>Scope: #Element</p> <p>Evaluation Runtime (Engine): No engine usage right now. Reserved for future usage.</p> <p>Values:</p> <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td><i>String(128)</i></td><td>Defines the term mapping dictionary, e.g. a table or entity.</td></tr> </tbody> </table>	Value	Description	<i>String(128)</i>	Defines the term mapping dictionary, e.g. a table or entity.
Value	Description				
<i>String(128)</i>	Defines the term mapping dictionary, e.g. a table or entity.				

Annotation	Meaning
Search.termMappingListID	<p>Specifies one or multiple list IDs within the term mapping dictionary mentioned before.</p> <p>The list is implemented as a column of the term mapping table, with the list ID as content of this column. This concept has the aim to enable overarching term mapping dictionaries while being able to separate domain-specific content at the same time.</p> <p>Scope: #Element</p> <p>Evaluation Runtime (Engine): No engine usage right now. Reserved for future usage.</p> <p>Values:</p>
Value	Description
<i>Array of String(32)</i>	Defines one or more columns of the term mapping dictionary.

Example

The following example demonstrates how the search annotations are used in a CDS view.

Sample Code

```
@Search.searchable: true
define view demo_search
  as select from db_flight
{
  @Search.defaultSearchElement: true
  @Search.fuzzinessThreshold: 0.7
  key carrid,
  key connid,
  @Search.defaultSearchElement: true
  @Search.fuzzinessThreshold: 0.7
  fldate,
  price,
  currencycode
}
```

The search is executed primarily on the elements `carrid` and `fldate` with a fuzziness threshold of 0.7.

8.1.7 Semantics Annotations

Used by the core engines for data processing and data consumption

Scope and Definition

```
@Scope: [#ELEMENT, #PARAMETER]
Annotation Semantics
    text           : Boolean default true;
    language       : Boolean default true;

@Scope: [#ELEMENT]
Annotation Semantics
    amount
    {
        currencyCode   : ElementRef;
    };
    quantity
    {
        unitOfMeasure : ElementRef;
    };
    currencyCode   : Boolean default true;
    language       : Boolean default true;
    unitOfMeasure  : Boolean default true;
    signReversalIndicator : Boolean default true;
    systemDateTime
    {
        createdAt     : Boolean default true;
        lastChangedAt : Boolean default true;
    };
    text           : Boolean default true;
    user
    {
        createdBy     : Boolean default true;
        lastChangedBy : Boolean default true;
    };
}
```

Usage

Semantic annotations complement the concept of semantic data types, while semantic data types always introduce specific behavior in the provider/core infrastructure (through dedicated operations or conversion functions).

Semantic annotations allow the standardizing of semantics that only have an impact on the consumption side (such as currency code representation together with the amount).

Annotation	Meaning		
Semantics.amount.currencyCode	<p>This annotation tags an element that contains a currency amount.</p> <p>Scope: [ELEMENT]</p> <p>Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations.</p> <p>Values:</p> <table> <tr> <td>elementRef</td><td>The annotated field contains a monetary amount, and the corresponding currency code is contained in the referenced field.</td></tr> </table>	elementRef	The annotated field contains a monetary amount, and the corresponding currency code is contained in the referenced field.
elementRef	The annotated field contains a monetary amount, and the corresponding currency code is contained in the referenced field.		
Semantics.currencyCode	<p>This annotation tags a field containing a currency code</p> <p>This can be either an ISO code or an SAP currency code (data type CUKY).</p> <p>Scope: [ELEMENT]</p> <p>Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations.</p> <p>Values: Boolean default true</p>		
Semantics.language	<p>This annotation identifies a language.</p> <p>Scope: [ELEMENT, PARAMETER]</p> <p>Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations.</p> <p>Values: Boolean default true</p>		
Semantics.quantity.unitOfMeasure	<p>This annotation tags an element that contains a measured quantity.</p> <p>Scope: [ELEMENT, PARAMETER]</p> <p>Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations.</p> <p>Values:</p> <table> <tr> <td>elementRef</td><td>The value of the annotated field references a unit of measure related to a measured quantity.</td></tr> </table>	elementRef	The value of the annotated field references a unit of measure related to a measured quantity.
elementRef	The value of the annotated field references a unit of measure related to a measured quantity.		
Semantics.signReversalIndicator	<p>This annotation reverses the sign of the annotated view element</p> <p>Scope: [ELEMENT, PARAMETER]</p> <p>This annotation is used in analytical queries of CDS view with @ObjectModel.dataCategory: #HIERARCHY annotations.</p> <p>Values: Boolean (default true)</p>		

Annotation	Meaning
Annotations belonging to Semantics.systemDate tag elements that specify the date/time that is recorded by the technical infrastructure/database.	
Scope: [ELEMENT, PARAMETER]	
Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations.	
Semantics.systemDateTime e.createdAt	The annotated element contains a timestamp indicating when the database record was created. Values: Boolean default true
Semantics.systemDateTime e.lastChangedAt	The annotated element contains a timestamp indicating when the database record was last changed. Values: Boolean default true
Semantics.text	This annotation identifies a human-readable text that is not necessarily language-dependent. Scope: [ELEMENT, PARAMETER] Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations. Values: Boolean default true
Semantics.unitOfMeasure	This annotation tags a field as containing a unit of measure. Scope: [ELEMENT, PARAMETER] Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations. Values: Boolean default true
Annotations belonging to Semantics.user tag elements that define the ID of the user related to the data record.	
Scope: [ELEMENT, PARAMETER]	
Evaluation Runtime (Engine): Interpreted by the orchestration framework (SADL). Translates CDS annotations into the corresponding OData annotations.	
Semantics.user.createdBy Y	The value of the annotated field specifies the user who created a data record. Values: Boolean default true
Semantics.user.lastChangedBy	The value of the annotated field specifies the user who changed a data record at last. Values: Boolean default true

Examples

Example 1

The following CDS view fetches sales order items. Here, the annotations assign the units and currencies to the corresponding fields.

↳ Sample Code

```
DEFINE VIEW SalesOrderItem as select from ...
{
    ...
    @Semantics.currencyCode
    currency_code as CurrencyCode,
    @Semantics.amount.currencyCode: 'CurrencyCode'
    gross_amount as GrossAmount,
    @Semantics.unitOfMeasure
    unit_of_measure as UnitOfMeasure,
    @Semantics.quantity.unitOfMeasure: 'UnitOfMeasure'
    quantity as Quantity,
    ...
}
```

Example 2

The following CDS view fetches language-dependant data annotating the corresponding language fields and text fields:

↳ Sample Code

```
DEFINE VIEW chartOfAccountsTexts AS SELECT FROM ...
{
    key ktopl AS chartOfAccounts,
    @Semantics.language: true
    key spras AS language,
    @Semantics.text: true
    ktplt AS chartOfAccountsName
}
```

8.1.8 UI Annotations

Represent semantic views on business data through the use of specific patterns that are completely independent of UI technologies.

Scope and Definition

```
@MetadataExtension.usageAllowed : true
{
    @Scope:[#ENTITY]
    headerInfo
    {
        @LanguageDependency.maxLength : 40
        typeName : String(60);
        @LanguageDependency.maxLength : 40
        typeNamePlural : String(60);
        typeImageUrl : String(1024);
        imageUrl : ElementRef;
        title
        {
            type : String(40) enum
            {
                STANDARD;
                WITH_INTENT_BASED_NAVIGATION;
                WITH_NAVIGATION_PATH;
                WITH_URL;
            } default #STANDARD;
            @LanguageDependency.maxLength : 40
            label : String(60);
            iconUrl : String(1024);
            criticality : ElementRef;
            criticalityRepresentation : String(12) enum
            {
                WITHOUT_ICON;
                WITH_ICON;
            } default #WITHOUT_ICON;
            value : ElementRef;
            targetElement : ElementRef;
            url : ElementRef;
        };
        description
        {
            type : String(40) enum
            {
                STANDARD;
                WITH_INTENT_BASED_NAVIGATION;
                WITH_NAVIGATION_PATH;
                WITH_URL;
            } default #STANDARD;
            @LanguageDependency.maxLength : 40
            label : String(60);
            iconUrl : String(1024);
            criticality : ElementRef;
            criticalityRepresentation : String(12) enum
            {
                WITHOUT_ICON;
                WITH_ICON;
            } default #WITHOUT_ICON;
            value : ElementRef;
            targetElement : ElementRef;
            url : ElementRef;
        };
    };
}
```

```

    };
};

@Scope:[#ENTITY]
badge
{
    headLine
    {
        type : String(40) enum
        {
            STANDARD;
            WITH_INTENT_BASED_NAVIGATION;
            WITH_NAVIGATION_PATH;
            WITH_URL;
        } default #STANDARD;
        @LanguageDependency.maxLength : 40
        label : String(60);
        iconUrl : String(1024);
        criticality : ElementRef;
        criticalityRepresentation : String(12) enum
        {
            WITHOUT_ICON;
            WITH_ICON;
        } default #WITHOUT_ICON;
        value : ElementRef;
        targetElement : ElementRef;
        url : ElementRef;
    };
    title
    {
        type : String(40) enum
        {
            STANDARD;
            WITH_INTENT_BASED_NAVIGATION;
            WITH_NAVIGATION_PATH;
            WITH_URL;
        } default #STANDARD;
        @LanguageDependency.maxLength : 40
        label : String(60);
        iconUrl : String(1024);
        criticality : ElementRef;
        criticalityRepresentation : String(12) enum
        {
            WITHOUT_ICON;
            WITH_ICON;
        } default #WITHOUT_ICON;
        value : ElementRef;
        targetElement : ElementRef;
        url : ElementRef;
    };
    typeImageUrl : String(1024);
    imageUrl : ElementRef;
    mainInfo
    {
        type : String(40) enum
        {
            STANDARD;
            WITH_INTENT_BASED_NAVIGATION;
            WITH_NAVIGATION_PATH;
            WITH_URL;
        } default #STANDARD;
        @LanguageDependency.maxLength : 40
        label : String(60);
        iconUrl : String(1024);
        criticality : ElementRef;
        criticalityRepresentation : String(12) enum
        {
            WITHOUT_ICON;
            WITH_ICON;
        }
    }
}

```

```

        } default #WITHOUT_ICON;
        value : ElementRef;
        targetElement : ElementRef;
        url : ElementRef;
    };
    secondaryInfo
    {
        type : String(40) enum
        {
            STANDARD;
            WITH_INTENT_BASED_NAVIGATION;
            WITH_NAVIGATION_PATH;
            WITH_URL;
        } default #STANDARD;
        @LanguageDependency.maxLength : 40
        label : String(60);
        iconUrl : String(1024);
        criticality : ElementRef;
        criticalityRepresentation : String(12) enum
        {
            WITHOUT_ICON;
            WITH_ICON;
        } default #WITHOUT_ICON;
        value : ElementRef;
        targetElement : ElementRef;
        url : ElementRef;
    };
};
@Scope:[#ENTITY]
chart : array of
{
    qualifier : String(120);
    @LanguageDependency.maxLength : 40
    title : String(60);
    @LanguageDependency.maxLength : 80
    description : String(120);
    chartType : String(40) enum
    {
        COLUMN;
        COLUMN_STACKED;
        COLUMN_STACKED_100;
        COLUMN_DUAL;
        COLUMN_STACKED_DUAL;
        COLUMN_STACKED_DUAL_100;
        BAR;
        BAR_STACKED;
        BAR_STACKED_100;
        BAR_DUAL;
        BAR_STACKED_DUAL;
        BAR_STACKED_DUAL_100;
        AREA;
        AREA_STACKED;
        AREA_STACKED_100;
        HORIZONTAL_AREA;
        HORIZONTAL_AREA_STACKED;
        HORIZONTAL_AREA_STACKED_100;
        LINE;
        LINE_DUAL;
        COMBINATION;
        COMBINATION_STACKED;
        COMBINATION_STACKED_DUAL;
        HORIZONTAL_COMBINATION_STACKED;
        HORIZONTAL_COMBINATION_STACKED_DUAL;
        PIE;
        DONUT;
        SCATTER;
        BUBBLE;
        RADAR;
    };
};

```

```

        HEAT_MAP;
        TREE_MAP;
        WATERFALL;
        BULLET;
        VERTICAL_BULLET;
        HORIZONTAL_WATERFALL;
        HORIZONTAL_COMBINATION_DUAL;
        DONUT_100;
    };
    dimensions : array of ElementRef;
    measures : array of ElementRef;
    dimensionAttributes : array of
    {
        dimension : ElementRef;
        role : String(10) enum
        {
            CATEGORY;
            SERIES;
            CATEGORY2;
        };
        valuesForSequentialColorLevels: array of String(1024);
        emphasizedValues: array of String(1024);
    };
    measureAttributes : array of
    {
        measure : ElementRef;
        role : String(10) enum
        {
            AXIS_1;
            AXIS_2;
            AXIS_3;
        };
        asDataPoint : Boolean default true;
        useSequentialColorLevels: Boolean default true;
    };
    actions : array of
    {
        type : String(40) enum
        {
            FOR_ACTION;
            FOR_INTENT_BASED_NAVIGATION;
        };
        @LanguageDependency.maxLength : 40
        label : String(60);
        dataAction : String(120);
        requiresContext : Boolean default true;
        invocationGrouping : String(12) enum
        {
            ISOLATED;
            CHANGE_SET;
        } default #ISOLATED;
        semanticObjectAction : String(120);
    };
};
@Scope:[#ENTITY]
selectionPresentationVariant : array of
{
    qualifier : String(120);
    id : String(120);
    @LanguageDependency.maxLength : 40
    text : String(60);
    selectionVariantQualifier : String(120);
    presentationVariantQualifier : String(120);
};
@Scope:[#ENTITY]
selectionVariant : array of
{
    qualifier : String(120);
}

```

```

id : String(120);
@LanguageDependency.maxLength : 40
text : String(60);
parameters : array of
{
    name : ParameterRef;
    value : String(1024);
};
filter : String(1024);
};

@Scope:[#ENTITY]
presentationVariant : array of
{
    qualifier : String(120);
    id : String(120);
    @LanguageDependency.maxLength : 40
    text : String(60);
    maxItems : Integer;
    sortOrder : array of
    {
        by : ElementRef;
        direction : String(4) enum
        {
            ASC;
            DESC;
        };
    };
    groupBy : array of ElementRef;
    totalBy : array of ElementRef;
    total : array of ElementRef;
    includeGrandTotal : Boolean default true;
    initialExpansionLevel : Integer;
    requestAtLeast : array of ElementRef;
    visualizations : array of
    {
        type : String(40) enum
        {
            AS_LINEITEM;
            AS_CHART;
            AS_DATAPOINT;
        };
        qualifier : String(120);
        element : ElementRef;
    };
    selectionFieldsQualifier : String(120);
};
@Scope:[#ELEMENT, #PARAMETER]
hidden : Boolean default true;
@Scope:[#ELEMENT]
masked : Boolean default true;
@Scope:[#ELEMENT]
multiLineText : Boolean default true;
@Scope:[#ELEMENT]
lineItem : array of
{
    @Scope: [#ELEMENT, #ENTITY]
    qualifier : String(120);
    position : DecimalFloat;
    exclude : Boolean default true;
    hidden : Boolean default true;
    importance : String(6) enum { HIGH; MEDIUM; LOW; };
    type : String(40) enum
    {
        AS_ADDRESS;
        AS_CHART;
        AS_CONNECTED_FIELDS;
        AS_CONTACT;
        AS_DATAPOINT;
    };
}

```

```

        AS_FIELDGROUP;
        FOR_ACTION;
        FOR_INTENT_BASED_NAVIGATION;
        STANDARD;
        WITH_INTENT_BASED_NAVIGATION;
        WITH_NAVIGATION_PATH;
        WITH_URL;
    } default #STANDARD;
@LanguageDependency.maxLength : 40
label : String(60);
iconUrl : String(1024);
@Scope: [#ELEMENT, #ENTITY]
criticality : ElementRef;
criticalityRepresentation : String(12) enum
{
    WITHOUT_ICON;
    WITH_ICON;
} default #WITHOUT_ICON;
dataAction : String(120);
requiresContext : Boolean default true;
invocationGrouping : String(12) enum { ISOLATED; CHANGE_SET; } default
#ISOLATED;
semanticObjectAction : String(120);
value : ElementRef;
valueQualifier : String(120);
targetElement : ElementRef;
url : ElementRef;
};

@Scope:[#ELEMENT]
identification : array of
{
    position : DecimalFloat;
    exclude : Boolean default true;
    importance : String(6) enum { HIGH; MEDIUM; LOW; };
    type : String(40) enum
    {
        AS_ADDRESS;
        AS_CHART;
        AS_CONNECTED_FIELDS;
        AS_CONTACT;
        AS_DATAPPOINT;
        AS_FIELDGROUP;
        FOR_ACTION;
        FOR_INTENT_BASED_NAVIGATION;
        STANDARD;
        WITH_INTENT_BASED_NAVIGATION;
        WITH_NAVIGATION_PATH;
        WITH_URL;
    } default #STANDARD;
@LanguageDependency.maxLength : 40
label : String(60);
iconUrl : String(1024);
criticality : ElementRef;
criticalityRepresentation : String(12) enum
{
    WITHOUT_ICON;
    WITH_ICON;
} default #WITHOUT_ICON;
dataAction : String(120);
requiresContext : Boolean default true;
invocationGrouping : String(12) enum { ISOLATED; CHANGE_SET; } default
#ISOLATED;
semanticObjectAction : String(120);
value : ElementRef;
valueQualifier : String(120);
targetElement : ElementRef;
url : ElementRef;
};

```

```

@Scope:[#ELEMENT]
statusInfo : array of
{
    position : DecimalFloat;
    exclude : Boolean default true;
    importance : String(6) enum { HIGH; MEDIUM; LOW; };
    type : String(40) enum
    {
        AS_ADDRESS;
        AS_CHART;
        AS_CONNECTED_FIELDS;
        AS_CONTACT;
        AS_DATAPOINT;
        AS_FIELDGROUP;
        FOR_ACTION;
        FOR_INTENT_BASED_NAVIGATION;
        STANDARD;
        WITH_INTENT_BASED_NAVIGATION;
        WITH_NAVIGATION_PATH;
        WITH_URL;
    } default #STANDARD;
    @LanguageDependency.maxLength : 40
    label : String(60);
    iconUrl : String(1024);
    criticality : ElementRef;
    criticalityRepresentation : String(12) enum
    {
        WITHOUT_ICON;
        WITH_ICON;
    } default #WITHOUT_ICON;
    dataAction : String(120);
    requiresContext : Boolean default true;
    invocationGrouping : String(12) enum { ISOLATED; CHANGE_SET; } default
#ISOLATED;
    semanticObjectAction : String(120);
    value : ElementRef;
    valueQualifier : String(120);
    targetElement : ElementRef;
    url : ElementRef;
};

@Scope:[#ELEMENT]
fieldGroup : array of
{
    qualifier : String(120);
    @LanguageDependency.maxLength : 40
    groupLabel : String(60);
    position : DecimalFloat;
    exclude : Boolean default true;
    importance : String(6) enum { HIGH; MEDIUM; LOW; };
    type : String(40) enum
    {
        AS_ADDRESS;
        AS_CHART;
        AS_CONNECTED_FIELDS;
        AS_CONTACT;
        AS_DATAPOINT;
        AS_FIELDGROUP;
        FOR_ACTION;
        FOR_INTENT_BASED_NAVIGATION;
        STANDARD;
        WITH_INTENT_BASED_NAVIGATION;
        WITH_NAVIGATION_PATH;
        WITH_URL;
    } default #STANDARD;
    @LanguageDependency.maxLength : 40
    label : String(60);
    iconUrl : String(1024);
    criticality : ElementRef;
}

```

```

criticalityRepresentation : String(12) enum
{
    WITHOUT_ICON;
    WITH_ICON;
} default #WITHOUT_ICON;
dataAction : String(120);
requiresContext : Boolean default true;
invocationGrouping : String(12) enum { ISOLATED; CHANGE_SET; } default
#ISOLATED;
semanticObjectAction : String(120);
value : ElementRef;
valueQualifier : String(120);
targetElement : ElementRef;
url : ElementRef;
};
@Scope:[#ELEMENT]
dataPoint
{
    qualifier : String(120);
    @LanguageDependency.maxLength : 40
    title : String(60);
    @LanguageDependency.maxLength : 80
    description : String(120);
    @LanguageDependency.maxLength : 193
    longDescription : String(250);
    targetValue : DecimalFloat;
    targetValueElement : ElementRef;
    forecastValue : ElementRef;
    minimumValue : DecimalFloat;
    maximumValue : DecimalFloat;
    visualization : String(12) enum
    {
        NUMBER;
        BULLET_CHART;
        DONUT;
        PROGRESS;
        RATING;
    };
    valueFormat
    {
        scaleFactor : DecimalFloat;
        numberOfFractionalDigits : Integer;
    };
    referencePeriod
    {
        @LanguageDependency.maxLength : 80
        description : String(120);
        start : ElementRef;
        end : ElementRef;
    };
    criticality : ElementRef;
    criticalityValue : Integer enum
    {
        NEGATIVE;
        CRITICAL;
        POSITIVE;
    };
    criticalityRepresentation : String(12) enum
    {
        WITHOUT_ICON;
        WITH_ICON;
    } default #WITHOUT_ICON;
    criticalityCalculation
    {
        improvementDirection : String(8) enum
        {
            MINIMIZE;
            TARGET;
        };
    };
}

```

```

        MAXIMIZE;
    };
    acceptanceRangeLowValue : DecimalFloat;
    acceptanceRangeHighValue : DecimalFloat;
    toleranceRangeLowValue : DecimalFloat;
    toleranceRangeLowValueElement : ElementRef;
    toleranceRangeHighValue : DecimalFloat;
    toleranceRangeHighValueElement : ElementRef;
    deviationRangeLowValue : DecimalFloat;
    deviationRangeLowValueElement : ElementRef;
    deviationRangeHighValue : DecimalFloat;
    deviationRangeHighValueElement : ElementRef;
    constantThresholds: array of
    {
        aggregationLevel: array of ElementRef;
        acceptanceRangeLowValue: DecimalFloat;
        acceptanceRangeHighValue: DecimalFloat;
        toleranceRangeLowValue: DecimalFloat;
        toleranceRangeHighValue: DecimalFloat;
        deviationRangeLowValue: DecimalFloat;
        deviationRangeHighValue: DecimalFloat;
    };
    trend : ElementRef;
    trendCalculation
    {
        referenceValue : ElementRef;
        isRelativeDifference : Boolean default true;
        upDifference : DecimalFloat;
        upDifferenceElement : ElementRef;
        strongUpDifference : DecimalFloat;
        strongUpDifferenceElement : ElementRef;
        downDifference : DecimalFloat;
        downDifferenceElement : ElementRef;
        strongDownDifference : DecimalFloat;
        strongDownDifferenceElement : ElementRef;
    };
    responsible : ElementRef;
    responsibleName : String(120);
};
@Scope:[#ELEMENT]
selectionField : array of
{
    qualifier : String(120);
    position : DecimalFloat;
    exclude : Boolean default true;
    element : ElementRef;
};
@Scope:[#ELEMENT]
facet : array of
{
    qualifier : String(120);
    @CompatibilityContract: {
        c1: { usageAllowed: false },
        c2: { usageAllowed: true,
            allowedChanges.annotation: [ #REMOVE ],
            allowedChanges.value: [ #NONE ] } }
    feature : String(40);
    id : String(120);
    purpose : String(40) enum
    {
        STANDARD;
        HEADER;
        QUICK_VIEW;
        QUICK_CREATE;
        FILTER;
    } default #STANDARD;
}

```

```

parentId : String(120);
position : DecimalFloat;
exclude : Boolean default true;
hidden : Boolean default true;
isPartOfPreview : Boolean default true;
isSummary : Boolean default true;
isMap : Boolean default true;
importance : String(6) enum
{
    HIGH;
    MEDIUM;
    LOW;
};
@LanguageDependency.maxLength : 40
label : String(60);
type : String(40) enum
{
    COLLECTION;
    ADDRESS_REFERENCE;
    BADGE_REFERENCE;
    CHART_REFERENCE;
    CONTACT_REFERENCE;
    DATAPOINT_REFERENCE;
    FIELDGROUP_REFERENCE;
    HEADERINFO_REFERENCE;
    IDENTIFICATION_REFERENCE;
    LINEITEM_REFERENCE;
    STATUSINFO_REFERENCE;
    URL_REFERENCE;
};
targetElement : ElementRef;
targetQualifier : String(120);
url : ElementRef;
};
@Scope:[#ENTITY, #ELEMENT]
textArrangement : String(13) enum
{
    TEXT_FIRST;
    TEXT_LAST;
    TEXT_ONLY;
    TEXT_SEPARATE;
};
//=====
// Version 7.69
//=====
@Scope: [#ELEMENT]
kpi : array of
{
    qualifier : String(120);
    id : String(120);
    @LanguageDependency.maxLength: 10
    shortDescription : String(20);
    selectionVariantQualifier : String(120);
    detail
    {
        defaultPresentationVariantQualifier : String(120);
        alternativePresentationVariantQualifiers : array of String(120);
        semanticObject : String(120);
        semanticObjectAction : String(120);
    };
    dataPoint
    {
        @LanguageDependency.maxLength : 40
        title : String(60);
        @LanguageDependency.maxLength : 80
        description : String(120);
        @LanguageDependency.maxLength : 193
        longDescription : String(250);
    };
}

```

```

targetValue      : DecimalFloat;
forecastValue    : DecimalFloat;
minimumValue     : DecimalFloat;
maximumValue     : DecimalFloat;
valueFormat
{
    scaleFactor           : DecimalFloat;
    number_of_fractional_digits : Integer;
};
visualization : String(12) enum
{
    NUMBER;
    BULLET_CHART;
    DONUT;
    PROGRESS;
    RATING;
};
referencePeriod {
    @LanguageDependency.maxLength: 80
    description : String(120);
    start       : ElementRef;
    end         : ElementRef;
};
criticality          : ElementRef;
criticalityValue     : Integer enum
{
    NEGATIVE;
    CRITICAL;
    POSITIVE;
};
criticalityRepresentation : String(12) enum
{
    WITHOUT_ICON;
    WITH_ICON;
} default #WITHOUT_ICON;
criticalityCalculation
{
    improvementDirection : String(8) enum
    {
        MINIMIZE;
        TARGET;
        MAXIMIZE;
    };
    acceptanceRangeLowValue   : DecimalFloat;
    acceptanceRangeHighValue : DecimalFloat;
    toleranceRangeLowValue   : DecimalFloat;
    toleranceRangeHighValue  : DecimalFloat;
    deviationRangeLowValue   : DecimalFloat;
    deviationRangeHighValue  : DecimalFloat;
    constantThresholds       : array of
    {
        aggregationLevel      : array of ElementRef;
        acceptanceRangeLowValue : DecimalFloat;
        acceptanceRangeHighValue : DecimalFloat;
        toleranceRangeLowValue  : DecimalFloat;
        toleranceRangeHighValue : DecimalFloat;
        deviationRangeLowValue  : DecimalFloat;
        deviationRangeHighValue : DecimalFloat;
    };
};
trend : ElementRef;
trendCalculation
{
    referenceValue      : ElementRef;
    isRelativeDifference : Boolean ;
    upDifference        : DecimalFloat;
    strongUpDifference  : DecimalFloat;
    downDifference      : DecimalFloat;
}

```

```

        strongDownDifference : DecimalFloat;
    };
    responsible      : ElementRef;
    responsibleName: String(120);
};

};

@Scope: [#ELEMENT]
valueCriticality: array of
{
    qualifier      : String(120);
    value          : String(120);
    criticality   : Integer enum
    {
        NEGATIVE;
        CRITICAL;
        POSITIVE;
    };
};

@Scope: [#ELEMENT]
criticalityLabels : array of {
    qualifier: String(120);
    criticality: Integer enum
    {
        NEGATIVE;
        CRITICAL;
        POSITIVE;
    };
}
@LanguageDependency.maxLength: 40
label: String(60);
};

@Scope: [#ELEMENT]
connectedFields : array of
{
    qualifier : String(120);
    @LanguageDependency.maxLength : 40
    groupLabel : String(60);
    @LanguageDependency.maxLength : 197
    template   : String(255);
    name       : String(120);
    exclude    : Boolean default true;
    hidden     : Boolean default true;
    importance : String(6) enum { HIGH; MEDIUM; LOW; };
    type       : String(40) enum
    {
        AS_ADDRESS;
        AS_CHART;
        AS_CONNECTED_FIELDS;
        AS_CONTACT;
        AS_DATAPOINT;
        AS_FIELDGROUP;
        FOR_ACTION;
        FOR_INTENT_BASED_NAVIGATION;
        STANDARD;
        WITH_INTENT_BASED_NAVIGATION;
        WITH_NAVIGATION_PATH;
        WITH_URL;
    } default #STANDARD;
    @LanguageDependency.maxLength : 40
    label : String(60);
    iconUrl : String(1024);
    criticality : ElementRef;
    criticalityRepresentation : String(12) enum
    {
        WITHOUT_ICON;
    };
};

```

```

        WITH_ICON;
    } default #WITHOUT_ICON;
dataAction           : String(120);
requiresContext      : Boolean default true;
invocationGrouping  : String(12) enum { ISOLATED; CHANGE_SET; } default
#ISOLATED;
semanticObjectAction : String(120);
value               : ElementRef;
valueQualifier      : String(120);
targetElement       : ElementRef;
url                 : ElementRef;
};

}

```

Usage

The focus of OData UI vocabulary developed by SAP is on usage patterns of data in UIs, not on UI patterns. The vocabulary is completely independent of the UI technologies or devices that consume the data. The usage patterns of data used by the OData UI vocabulary represent certain semantic views on business data. Some of them are very generic, others are based on the concept of an entity, something tangible to end-users. Examples for entities are semantic object instances or business object instances. Looking at different UI patterns, these data usage patterns reoccur again and again. To generate OData annotations from CDS views, CDS annotations are reused from different domains, for example Consumption, Communication, Semantics, EndUserText. The CDS annotations that are additionally required in a UI domain are listed in the following table.

UI Annotations

Parent Annotation	Annotation	Type	Value	Description
<i>UI.headerInfo</i>				Annotations belonging to <i>UI.headerInfo</i> describe an entity, its title, and an optional short description, the name of its entity in singular and plural form, and optional image URLs for the individual entity.
				Scope: ENTITY
				Evaluation Runtime (Engine): SADL: Translates CDS annotations into the corresponding OData annotations
				Values:

Parent Annotation	Annotation	Type	Value	Description
UI.headerInfo	.typeName	String (60)	#mandatory	<p>i Note</p> <p>This annotation can be omitted only when the @End-UserText.label is specified on view level.</p>
UI.headerInfo	typeNamePlural	String (60)	#mandatory	<p>This annotation represents a list title, for example.</p>
UI.headerInfo	typeImageUrl	String (1024)	#optional	<p>This annotation contains the URL of an image representing an entity.</p> <p>i Note</p> <p>When users open a SAP Fiori application, they can see an image related to the entity type to which all items displayed on that page belong to.</p>
UI.headerInfo	imageUrl	elementRef	#optional	<p>This annotation represents a path to an element containing the URL of an image representing the entity instance.</p>
UI.headerInfo	imageData	elementRef		<p>This annotation allows referencing an element that contains the image BLOB and is itself annotated with <code>Semantics.large Object</code>.</p>
UI.headerInfo	title.			

Parent Annotation	Annotation	Type	Value	Description
UI.headerInfo	title.type	String (40) enum	<ul style="list-style-type: none"> • STANDARD: Maps to standard <i>DataField</i>. You use this type if you want a field to be displayed without any additional functionality. A standard <i>DataField</i> refers to a property of the OData service used. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> ○ <i>criticality</i> • WITH_URL: Maps to <i>DataFieldWithURL</i>. <i>DataFieldWithURL</i> is based on <i>DataField</i>, and defines a label–value pair that refers a property of the OData service used. The definition consists a URL to navigate to a new target, that is a URL. For more information, see With URL [page 546]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> When you use this type, you must use the following elements: <ul style="list-style-type: none"> ○ <i>url</i> 	Annotations belonging to UI.headerInfo.title represent a property of type UI.DataFieldAbstract restricted to the types STANDARD, WITH_NAVIGATION_PATH, WITH_URL, and WITH_INVENTORY_BASED_NAVIGATION. @UI.headerInfo.title annotations are mandatory and are usually used to represent the title of an item on the header of an item's object page.. The OData annotations DataFieldAbstract are the basis for all DataField types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> • WITH_NAVIGATION_PATH: Maps to <i>DataFieldWithNavigationPath</i>. <i>DataFieldWithNavigationPath</i> is based on <i>DataField</i>, and defines a label-value pair that refers to a property of the OData service used. The definition consists of a link to navigate to a new target, based on a navigation property provided by the OData service, or defined in the annotation file. For more information, see With Navigation Path [page 544]. <p>When you use this type, you can use the following elements:</p> <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> <p>When you use this type, you must use the following elements:</p> <ul style="list-style-type: none"> ○ <i>targetElement</i> <ul style="list-style-type: none"> • WITH_INVENTENT_BASED_NAVIGATION; 	<p>Default: STANDARD;</p>

Parent Annotation	Annotation	Type	Value	Description
UI.headerInfo	title.label	String (60)	#optional	<p>This annotation contains a language-dependent text that can be used for titles in page headers of object-page floorplans. Object-page floorplans are SAP Fiori floorplan to view, edit and create objects.</p> <p>If omitted, the label of the annotated element, or the label of the element are referenced via the value.</p>
UI.headerInfo	title.iconUrl	String (1024)	#optional	This annotation contains the URL to an icon image.
UI.headerInfo	title.criticality	elementRef	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • Neutral: 0 • Negative: 1 • Critical: 2 • Positive: 3 	<p>i Note</p> <p>This annotation can be specified if the badge headline type is STANDARD.</p>
UI.headerInfo	title.criticalityRepresentation	String (12) enum	#mandatory	<ul style="list-style-type: none"> • WITHOUT_ICON • WITH_ICON <p>Default:</p> <p>WITHOUT_ICON</p>
UI.headerInfo	title.value	ElementRef		This annotation refers to a value. If you refer to a value that is in the same view, specify the element name. If you use an association to refer to a value, specify the path to the element.

Parent Annotation	Annotation	Type	Value	Description
UI.headerInfo	title.targetElement	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using This annotation, you can link from the header part of an object view floorplan to a target element. You need to specify UI.badge.headLine.targetElement when you use the annotation UI.badge.headLine.type of type WITH_NAVIGATION_PATH. You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
UI.headerInfo	title.url	ElementRef		<p>This annotation represents the path to a structural element that contains a navigation URL. You need to specify UI.badge.headLine.url when you use the annotation UI.badge.headLine.type of type WITH_URL.</p>
UI.headerInfo	description			

Parent Annotation	Annotation	Type	Value	Description
UI.headerInfo	description.type	String (40) enum	<ul style="list-style-type: none"> • STANDARD: Maps to standard <i>DataField</i>. You use this type if you want a field to be displayed without any additional functionality. A standard <i>DataField</i> refers to a property of the OData service used. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> ○ <i>criticality</i> • WITH_URL: Maps to <i>DataFieldWithURL</i>. <i>DataFieldWithURL</i> is based on <i>DataField</i>, and defines a label–value pair that refers a property of the OData service used. The definition consists a URL to navigate to a new target, that is a URL. For more information, see With URL [page 546]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> When you use this type, you must use the following elements: <ul style="list-style-type: none"> ○ <i>url</i> 	Annotations belonging to <code>UI.headerInfo.title</code> represent a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code> , <code>WITH_NAVIGATION_PATH</code> , <code>WITH_URL</code> , and <code>WITH_INVENTORY_BASED_NAVIGATION</code> . <code>@UI.headerInfo.title</code> annotations are mandatory and are usually used to represent the title of an item on the header of an item's object page.. The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> • WITH_NAVIGATION_PATH: Maps to <i>DataFieldWithNavigationPath</i>. <i>DataFieldWithNavigationPath</i> is based on <i>DataField</i>, and defines a label-value pair that refers to a property of the OData service used. The definition consists of a link to navigate to a new target, based on a navigation property provided by the OData service, or defined in the annotation file. For more information, see With Navigation Path [page 544]. <p>When you use this type, you can use the following elements:</p> <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> <p>When you use this type, you must use the following elements:</p> <ul style="list-style-type: none"> ○ <i>targetElement</i> <ul style="list-style-type: none"> • WITH_INVENTENT_BASED_NAVIGATION; 	<p>Default: STANDARD;</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.headerInfo</code>	<code>description.label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for titles in page headers of object-page floorplans. Object-page floorplans are SAP Fiori floorplan to view, edit and create objects.</p> <p>If omitted, the label of the annotated element, or the label of the element are referenced via the value.</p>
<code>UI.headerInfo</code>	<code>description.iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.
<code>UI.headerInfo</code>	<code>description.criticality</code>	<code>elementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	<p>i Note</p> <p>This annotation can be specified if the badge headline type is <code>STANDARD</code>.</p>
<code>UI.headerInfo</code>	<code>description.criticalityR</code>	<code>String(12) enum epresentation</code>	<p><code>#mandatory</code></p> <ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> <p>Default:</p> <p><code>WITHOUT_ICON</code></p>	
<code>UI.headerInfo</code>	<code>description.value</code>	<code>ElementRef</code>		This annotation refers to a value. If you refer to a value that is in the same view, specify the element name. If you use an association to refer to a value, specify the path to the element.

Parent Annotation	Annotation	Type	Value	Description
UI.headerInfo	description.targetElement	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using This annotation, you can link from the header part of an object view floorplan to a target element. You need to specify UI.badge.headLine.targetElement when you use the annotation UI.badge.headLine.type of type WITH_NAVIGATION_PATH. You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
UI.headerInfo	description.url	ElementRef		<p>This annotation represents the path to a structural element that contains a navigation URL. You need to specify UI.badge.headLine.url when you use the annotation UI.badge.headLine.type of type WITH_URL.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>.headLine.type</code>	String (40) enum	<ul style="list-style-type: none"> • STANDARD: Maps to standard <code>DataField</code>. You use this type if you want a field to be displayed without any additional functionality. A standard <code>DataField</code> refers to a property of the OData service used. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> ◦ <code>criticality</code> • WITH_URL: Maps to <code>DataFieldWithURL</code>. <code>DataFieldWithURL</code> is based on <code>DataField</code>, and defines a label–value pair that refers a property of the OData service used. The definition consists a URL to navigate to a new target, that is a URL. For more information, see With URL [page 546]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> When you use this type, you must use the following elements: <ul style="list-style-type: none"> ◦ <code>url</code> 	<p>This annotation is used to define, what can be shown in the title of a table or list. <code>UI.badge.headLine</code> represent a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code>, <code>WITH_NAVIGATION_PATH</code>, and <code>WITH_URL</code>.</p> <p>The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.</p>

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> • WITH_NAVIGATION_PATH: Maps to <i>DataFieldWithNavigationPath</i>. <i>DataFieldWithNavigationPath</i> is based on <i>DataField</i>, and defines a label-value pair that refers to a property of the OData service used. The definition consists of a link to navigate to a new target, based on a navigation property provided by the OData service, or defined in the annotation file. For more information, see With Navigation Path [page 544]. <p>When you use this type, you can use the following elements:</p> <ul style="list-style-type: none"> ○ <i>label</i> ○ <i>value</i> <p>When you use this type, you must use the following elements:</p> <ul style="list-style-type: none"> ○ <i>targetElement</i> <ul style="list-style-type: none"> • WITH_INVENTENT_BASED_NAVIGATION; 	<p>Default: STANDARD;</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>.headLine.label</code>	String (60)	#Optional	This annotation contains a language-dependent text. If omitted, the label of the annotated element, or the label of the element referenced via the value is used. The element is optional.
<code>UI.badge</code>	<code>.headLine.iconURL</code>	String (1024)	#Optional	This annotation contains the URL to an icon image. This annotation is optional .
<code>UI.badge</code>	<code>.headLine.criticality</code>	ElementRef	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	<p>i Note</p> <p>This annotation can be specified if the badge headline type is STANDARD.</p>
<code>UI.badge</code>	<code>.headLine.criticalityRe</code>	String(12) enum presentation	<ul style="list-style-type: none"> • WITHOUT_ICON • WITH_ICON 	<p>Default:</p> <p>WITHOUT_ICON</p>
<code>UI.badge</code>	<code>.headLine.value</code>	ElementRef		This annotation refers to a value. If you refer to a value that is in the same view, specify the element name. If you use an association to refer to a value, specify the path to the element.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>.headLine.targetElement</code>	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData <code>NavigationPropertyPath</code>. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.badge.headLine.targetElement</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_NAVIGATION_PATH</code>. You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.badge</code>	<code>.headLine.url</code>	ElementRef		<p>This annotation represents the path to a structural element that contains a navigation URL. You need to specify <code>UI.badge.headLine.url</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_URL</code>.</p>
<code>UI.badge</code>	<code>title.</code>			

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>title.type</code>	String (40) enum	<ul style="list-style-type: none"> • STANDARD: Maps to standard <code>DataField</code>. You use this type if you want a field to be displayed without any additional functionality. A standard <code>DataField</code> refers to a property of the OData service used. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> ◦ <code>criticality</code> • WITH_URL: Maps to <code>DataFieldWithURL</code>. <code>DataFieldWithURL</code> is based on <code>DataField</code>, and defines a label–value pair that refers a property of the OData service used. The definition consists a URL to navigate to a new target, that is a URL. For more information, see With URL [page 546]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> When you use this type, you must use the following elements: <ul style="list-style-type: none"> ◦ <code>url</code> 	Annotations belonging to <code>UI.headerInfo.title</code> represent a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code> , <code>WITH_NAVIGATION_PATH</code> , <code>WITH_URL</code> , and <code>WITH_INVENTORY_BASED_NAVIGATION</code> . <code>@UI.headerInfo.title</code> annotations are mandatory and are usually used to represent the title of an item on the header of an item's object page.. The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> • WITH_NAVIGATION_PATH: Maps to <i>DataFieldWithNavigationPath</i>. <i>DataFieldWithNavigationPath</i> is based on <i>DataField</i>, and defines a label-value pair that refers to a property of the OData service used. The definition consists of a link to navigate to a new target, based on a navigation property provided by the OData service, or defined in the annotation file. For more information, see With Navigation Path [page 544]. <p>When you use this type, you can use the following elements:</p> <ul style="list-style-type: none"> ◦ <i>label</i> ◦ <i>value</i> <p>When you use this type, you must use the following elements:</p> <ul style="list-style-type: none"> ◦ <i>targetElement</i> <ul style="list-style-type: none"> • WITH_INVENTENT_BASED_NAVIGATION; 	<p>Default: STANDARD;</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>title.label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for titles in page headers of object-page floorplans. Object-page floorplans are SAP Fiori floorplan to view, edit and create objects.</p> <p>If omitted, the label of the annotated element, or the label of the element are referenced via the value.</p>
<code>UI.badge</code>	<code>title.iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.
<code>UI.badge</code>	<code>title.criticality</code>	<code>elementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	<p>i Note</p> <p>This annotation can be specified if the badge headline type is <code>STANDARD</code>.</p>
<code>UI.badge</code>	<code>title.criticalityRepresentation</code>	<code>String(12) enum</code>	<code>#mandatory</code>	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> <p>Default: <code>WITHOUT_ICON</code></p>
<code>UI.badge</code>	<code>title.value</code>	<code>ElementRef</code>		This annotation refers to a value. If you refer to a value that is in the same view, specify the element name. If you use an association to refer to a value, specify the path to the element.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>title.targetElement</code>	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData <code>NavigationPropertyPath</code>. Using This annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.badge.headLine.targetElement</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_NAVIGATION_PATH</code>. You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.badge</code>	<code>title.url</code>	ElementRef		<p>This annotation represents the path to a structural element that contains a navigation URL. You need to specify <code>UI.badge.headLine.url</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_URL</code>.</p>
<code>UI.badge</code>	<code>typeImageUrl</code>	String(1024) ; #optional		<p>This annotation contains the URL of an image representing an entity.</p>
<code>UI.badge</code>	<code>imageUrl</code>	ElementRef	#optional	<p>This annotation represents a path to an element containing the URL of an image representing the entity instance.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>mainInfo</code>			<p>The content of <code>UI.badge.mainInfo</code> annotations is highlighted on the badge. These annotations represent a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code>, <code>WITH_NAVIGATION_PATH</code>, and <code>WITH_URL</code>.</p> <p>The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>mainInfo.type</code>	String (40) enum	<ul style="list-style-type: none"> • STANDARD: Maps to standard <code>DataField</code>. You use this type if you want a field to be displayed without any additional functionality. A standard <code>DataField</code> refers to a property of the OData service used. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> ◦ <code>criticality</code> • WITH_URL: Maps to <code>DataFieldWithURL</code>. <code>DataFieldWithURL</code> is based on <code>DataField</code>, and defines a label–value pair that refers a property of the OData service used. The definition consists a URL to navigate to a new target, that is a URL. For more information, see With URL [page 546]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> When you use this type, you must use the following elements: <ul style="list-style-type: none"> ◦ <code>url</code> 	Annotations belonging to <code>UI.headerInfo.title</code> represent a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code> , <code>WITH_NAVIGATION_PATH</code> , <code>WITH_URL</code> , and <code>WITH_INVENTORY_BASED_NAVIGATION</code> . <code>@UI.headerInfo.title</code> annotations are mandatory and are usually used to represent the title of an item on the header of an item's object page.. The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> • WITH_NAVIGATION_PATH: Maps to <i>DataFieldWithNavigationPath</i>. <i>DataFieldWithNavigationPath</i> is based on <i>DataField</i>, and defines a label-value pair that refers to a property of the OData service used. The definition consists of a link to navigate to a new target, based on a navigation property provided by the OData service, or defined in the annotation file. For more information, see With Navigation Path [page 544]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <i>label</i> ◦ <i>value</i> • WITH_INVENTENT_BASED_NAVIGATION; 	<p>Default: STANDARD;</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>mainInfo.label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for titles in page headers of object-page floorplans. Object-page floorplans are SAP Fiori floorplan to view, edit and create objects.</p> <p>If omitted, the label of the annotated element, or the label of the element are referenced via the value.</p>
<code>UI.badge</code>	<code>mainInfo.iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.
<code>UI.badge</code>	<code>mainInfo.criticality</code>	<code>elementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	<p>i Note</p> <p>This annotation can be specified if the badge headline type is <code>STANDARD</code>.</p>
<code>UI.badge</code>	<code>mainInfo.criticalityRep</code>	<code>String(12) enum representation</code>	<p><code>#mandatory</code></p> <ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> <p>Default:</p> <p><code>WITHOUT_ICON</code></p>	
<code>UI.badge</code>	<code>mainInfo.value</code>	<code>ElementRef</code>		This annotation refers to a value. If you refer to a value that is in the same view, specify the element name. If you use an association to refer to a value, specify the path to the element.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>mainInfo.targetElement</code>	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData <code>NavigationPropertyPath</code>. Using This annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.badge.headLine.targetElement</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_NAVIGATION_PATH</code>. You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.badge</code>	<code>mainInfo.url</code>	ElementRef		<p>This annotation represents the path to a structural element that contains a navigation URL. You need to specify <code>UI.badge.headLine.url</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_URL</code>.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>secondaryInfo</code>			<p>The content of <code>UI.badge.secondaryInfo</code> annotations is subordinate to the content of the <code>UI.badge.mainInfo</code> annotations. This annotation represents a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code>, <code>WITH_NAVIGATION_PATH</code>, and <code>WITH_URL</code>.</p> <p>The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>secondaryInfo.type</code>	<code>String(40)</code> enum	<ul style="list-style-type: none"> • STANDARD: Maps to standard <code>DataField</code>. You use this type if you want a field to be displayed without any additional functionality. A standard <code>DataField</code> refers to a property of the OData service used. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> ◦ <code>criticality</code> • WITH_URL: Maps to <code>DataFieldWithURL</code>. <code>DataFieldWithURL</code> is based on <code>DataField</code>, and defines a label–value pair that refers a property of the OData service used. The definition consists a URL to navigate to a new target, that is a URL. For more information, see With URL [page 546]. When you use this type, you can use the following elements: <ul style="list-style-type: none"> ◦ <code>label</code> ◦ <code>value</code> When you use this type, you must use the following elements: <ul style="list-style-type: none"> ◦ <code>url</code> 	Annotations belonging to <code>UI.headerInfo.title</code> represent a property of type <code>UI.DataFieldAbstract</code> restricted to the types <code>STANDARD</code> , <code>WITH_NAVIGATION_PATH</code> , <code>WITH_URL</code> , and <code>WITH_INVENTORY_BASED_NAVIGATION</code> . <code>@UI.headerInfo.title</code> annotations are mandatory and are usually used to represent the title of an item on the header of an item's object page.. The OData annotations <code>DataFieldAbstract</code> are the basis for all <code>DataField</code> types and represent values with optional labels that can trigger navigation to related data, or execute actions on data.

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> • WITH_NAVIGATION_PATH: Maps to <i>DataFieldWithNavigationPath</i>. <i>DataFieldWithNavigationPath</i> is based on <i>DataField</i>, and defines a label-value pair that refers to a property of the OData service used. The definition consists of a link to navigate to a new target, based on a navigation property provided by the OData service, or defined in the annotation file. For more information, see With Navigation Path [page 544]. <p>When you use this type, you can use the following elements:</p> <ul style="list-style-type: none"> ◦ <i>label</i> ◦ <i>value</i> <p>When you use this type, you must use the following elements:</p> <ul style="list-style-type: none"> ◦ <i>targetElement</i> <ul style="list-style-type: none"> • WITH_INVENTENT_BASED_NAVIGATION; 	<p>Default: STANDARD;</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>secondaryInfo.label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for titles in page headers of object-page floorplans. Object-page floorplans are SAP Fiori floorplan to view, edit and create objects.</p> <p>If omitted, the label of the annotated element, or the label of the element are referenced via the value.</p>
<code>UI.badge</code>	<code>secondaryInfo.iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.
<code>UI.badge</code>	<code>secondaryInfo.criticality</code>	<code>elementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	<p>i Note</p> <p>This annotation can be specified if the badge headline type is <code>STANDARD</code>.</p>
<code>UI.badge</code>	<code>secondaryInfo.criticalityRepresentation</code>	<code>String(12)</code> enum	<code>#mandatory</code>	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> <p>Default: <code>WITHOUT_ICON</code></p>
<code>UI.badge</code>	<code>secondaryInfo.value</code>	<code>ElementRef</code>		This annotation refers to a value. If you refer to a value that is in the same view, specify the element name. If you use an association to refer to a value, specify the path to the element.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.badge</code>	<code>secondaryInfo.targetElement</code>	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData <code>NavigationPropertyPath</code>. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.badge.headLine.targetElement</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_NAVIGATION_PATH</code>. You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.badge</code>	<code>secondaryInfo.url</code>	ElementRef		<p>This annotation represents the path to a structural element that contains a navigation URL. You need to specify <code>UI.badge.headLine.url</code> when you use the annotation <code>UI.badge.headLine.type</code> of type <code>WITH_URL</code>.</p>
<code>UI.datapoint</code>	<code>qualifier</code>	String(120)		
<code>UI.datapoint</code>	<code>title</code>	String(60)	#mandatory	<p>i Note The element can be omitted only if the <code>@EndUserText.label</code> is specified.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.datapoint</code>	<code>description</code>	<code>String(120)</code>	<code>#optional</code>	This annotation contains a description of the data point. If omitted, the <code>@EndUserText.quickinfo</code> is used, if specified.
<code>UI.datapoint</code>	<code>longDescription</code>	<code>String(250)</code>	<code>#optional</code>	This annotation contains a detailed description of the data point.
<code>UI.datapoint</code>	<code>targetValue</code>	<code>DecimalFloat</code>	<code>#not compatible with UI.dataPoint.targetValue</code>	<p>⚠ Caution If you use this annotation, do not use the element <code>UI.dataPoint.targetValueElement</code>.</p> <p>→ Tip You create a KPI in which you specify a certain revenue that needs to be reached at the end of a specific year. This is the <code>UI.dataPoint.targetValue</code> that is a static value</p>
<code>UI.datapoint</code>	<code>forecastValue</code>	<code>ElementRef</code>		This annotation references a value such as predicted or intended quarterly results, for example.
<code>UI.datapoint</code>	<code>minimumValue</code>	<code>DecimalFloat</code>		This annotation specifies the minimum value of a threshold.
<code>UI.datapoint</code>	<code>maximumValue</code>	<code>DecimalFloat</code>		This annotation specifies the maximum value of a threshold.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.datapoint</code>	<code>valueFormat</code>	DecimalFloat	#optional	All <code>UI.dataPoint.valueFormat</code> annotations are optional. For more information about value formats, see Person Responsible and Reference Period [page 540] .
<code>UI.datapoint</code>	<code>valueFormat.scaleFact</code> or	DecimalFloat	#optional	This annotation contains the scale factor for the value, e.g. A value 1000 displayed with scaleFactor = 1000 is displayed as 1k.
<code>UI.datapoint</code>	<code>valueFormat.numberOfFractionalDigits</code>	Integer	#optional	This annotation contains the number of fractional digits to be displayed. If the element value is 1, one decimal place is rendered, for example, 34.5.

Parent Annotation	Annotation	Type	Value	Description
UI.datapoint	visualization	String(12) enum	<ul style="list-style-type: none"> • NUMBER: A data point is visualized as a number. <p>⚠ Caution The following visualizations require the annotation UI.dataPoint.targetValue.</p> <ul style="list-style-type: none"> • BULLET_CHART: A data point is visualized as a bullet chart. • DONUT: A data point is visualized as a donut chart. • PROGRESS: A data point is visualized as a progress indicator. • RATING: A data point is visualized as partly or completely filled symbols such as stars or hearts. 	This annotation defines the preferred visualization of a data point.
UI.datapoint	referencePeriod	#optional		All UI.dataPoint.referencePeriod annotations are optional. You either use UI.dataPoint.referencePeriod.description, or UI.dataPoint.referencePeriod.start and UI.dataPoint.referencePeriod.end.

Parent Annotation	Annotation	Type	Value	Description
<i>UI.datapoint</i>	<i>referencePeriod.description</i>	String (120)	#optional	This annotation describes the business period of evaluation, for example "Oct 2012". Typical patterns are calendar dates or fiscal dates.
<i>UI.datapoint</i>	<i>referencePeriod.start</i>	elementRef	#optional	This annotation contains a reference to the end date of the reference period.
<i>UI.datapoint</i>	<i>referencePeriod.end</i>	elementRef	#optional	This annotation contains a reference to the start date of the reference period.
<i>UI.datapoint</i>	<i>criticality</i>	elementRef		<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • Neutral: 0 • Negative: 1 • Critical: 2 • Positive: 3
<i>UI.datapoint</i>	<i>criticalityValue</i>	Integer enum		<ul style="list-style-type: none"> • Negative • Critical • Positive
<i>UI.datapoint</i>	<i>criticalityRepresentation</i>	String enum	#mandatory	<ul style="list-style-type: none"> • WITHOUT_ICON • WITH_ICON <p>Default: WITHOUT_ICON</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.datapoint</code>	<code>criticalityCalculation</code>			Annotations belonging to <code>UI.dataPoint.criticalityCalculation</code> can be used as an alternative to specifying the criticality in the criticality element. The criticality can be calculated based on the values of the criticalityCalculation annotations.
<code>UI.datapoint</code>	<code>criticalityCalculation.improvementDirection</code>	<code>String(8) enum</code>	<ul style="list-style-type: none"> • <code>MINIMIZE</code> • <code>TARGET</code> • <code>MAXIMIZE</code> 	<p>Description: This annotation calculates the criticality based on a specified improvement direction. For more information, see Trend-Criticality Calculation [page 537].</p>
<code>UI.datapoint</code>	<code>criticalityCalculation.acceptanceRangeLowValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.acceptanceRangeHighValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.toleranceRangeLowValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.toleranceRangeHighValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.deviationRangeLowValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.deviationRangeHighValue</code>	<code>DecimalFloat</code>		

Parent Annotation	Annotation	Type	Value	Description
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds</code>	(array of)		Thresholds depending on the aggregation level as a set of constant values. (Constant thresholds should only be used in order to refine constant values given for the datapoint overall (aggregationLevel []), but not if the thresholds are based on other measure elements.)
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds.agg</code>	<code>elementRef</code>		An unordered tuple of dimensions, i.e. elements which are intended to be used for group-by in aggregating requests. (In analytical UIs, e.g. an analytical chart, the aggregation level typically corresponds to the visible dimensions.)
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds.acceptanceRangeLowValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds.acceptanceRangeHighValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds.toleranceRangeLowValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds.toleranceRangeHighValue</code>	<code>DecimalFloat</code>		
<code>UI.datapoint</code>	<code>criticalityCalculation.constantThresholds.deviationRangeLowValue</code>	<code>DecimalFloat</code>		

Parent Annotation	Annotation	Type	Value	Description
<i>UI.datapoint</i>	<i>criticalityCalculation.constantThresholds.deviationRangeHighValue</i>	DecimalFloat		
<i>UI.datapoint</i>	<i>trend</i>	elementRef	<ul style="list-style-type: none"> • 1: StrongUp • 2: Up • 3: Sideways • 4: Down • 5: StrongDown 	<p>Reference to an element to visualize a trend in form of an arrow.</p> <p>For more information, see Trends [page 535].</p>
<i>UI.datapoint</i>	<i>trendCalculation</i>			<p>As an alternative to specifying the trend in the <i>trend</i> element, the trend could be calculated based on the <i>trendCalculation</i> annotations.</p>
<i>UI.datapoint</i>	<i>trendCalculation.referenceValue</i>	elementRef		This annotation specifies the reference value for the trend calculation as a reference to an element.
<i>UI.datapoint</i>	<i>trendCalculation.isRelativeDifference</i>	boolean	Default: False	This boolean constant expresses whether the difference between the value and <i>referenceValue</i> is absolute or relative.
<i>UI.datapoint</i>	<i>trendCalculation.upDifference</i>	DecimalFloat		This annotation specifies a threshold as a constant.
<i>UI.datapoint</i>	<i>trendCalculation.strongUpDifference</i>	DecimalFloat		This annotation specifies a threshold as a constant.
<i>UI.datapoint</i>	<i>trendCalculation.downDifference</i>	DecimalFloat		This annotation specifies a threshold as a constant.
<i>UI.datapoint</i>	<i>trendCalculation.strongDownDifference</i>	DecimalFloat		This annotation specifies a threshold as a constant.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.datapoint</code>	<code>responsible</code>	ElementRef	# not compatible with <code>UI.dataPoint.responsibleName</code>	<p>This annotation contains an association to an entity that is annotated with <code>@Semantics.name</code>, <code>@Semantics.eMail</code>, <code>@Semantics.telephone</code>, <code>@Semantics.address</code>, or <code>@Semantics.organization</code>.</p> <p>⚠ Caution</p> <p>If you use this annotation, you can't use the annotation <code>UI.dataPoint.responsibleName</code>.</p> <p>For more information, see Person Responsible and Reference Period [page 540].</p> <p>For an overview of <code>@Semantics</code> annotations, see Semantics Annotations [page 606].</p>
<code>UI.datapoint</code>	<code>responsibleName</code>	String (120)	# not compatible with <code>UI.dataPoint.responsible</code>	<p>This annotation can be used as an alternative to the responsible element. Only the name of the responsible person can be specified here.</p> <p>⚠ Caution</p> <p>If you use this annotation, you can't use the annotation <code>UI.dataPoint.responsible</code>.</p>

Parent Annotation	Annotation	Type	Value	Description
<i>UI.kpi</i>				Annotations belonging to UI.KPI represent a single point of data, specialized for a specific data selection and extended with information about KPI details, especially the first level of drilldown, for a progressive disclosure. Scope: [ELEMENT]
<i>UI.kpi</i>	<i>qualifier</i>	String(120)		
<i>UI.kpi</i>	<i>id</i>	String(120)		
<i>UI.kpi</i>	<i>shortDescription</i>	String(120)		
<i>UI.kpi</i>	<i>selectionVariantQualifier</i>	String(120)		This element refers to a UI.selectionVariant annotation at the same view via its qualifier.
<i>UI.kpi</i>	<i>detail</i>	String(120)		This annotation bundles additional settings for a KPI which are relevant for a separate KPI detail display or for progressive disclosure of the KPI.
<i>UI.kpi</i>	<i>detail.defaultPresentationVariantQualifier</i>	String(120)		This element refers to a UI.presentationVariant annotation at the same view via its qualifier. This presentation variant should be used to initially represent the KPI detail.

Parent Annotation	Annotation	Type	Value	Description
<i>UI.kpi</i>	<i>detail.alternativePresentationVariantQualifier</i>	(array of) String (120)s		This element refers to a UI.presentationVariant annotations at the same view via their qualifiers. These presentation variants should be used/offered as alternative representations of the KPI detail
<i>UI.kpi</i>	<i>detail.semanticObject</i>	String (120)		
<i>UI.kpi</i>	<i>detail.semanticObject</i>	String (120) Action		
<i>UI.kpi</i>	<i>dataPoint</i>			For the <i>UI.kpi.dataPoint</i> annotations you can refer to <i>UI.dataPoint</i> .
<i>UI.selectionField</i>		(array of)		Annotations belonging to <i>UI.selectionField</i> allow filtering a list of data. <i>UI.selectionField</i> annotations are usually used in an initial page floorplan as filter bar. Scope: [ELEMENT] Evaluation Runtime (Engine): SADL: Translates CDS annotations into the corresponding OData annotations
<i>UI.selectionField</i>	<i>qualifier</i>	String (120)		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a selection field to ensure that the correct selection field can be referenced by the UI.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.selectionField</code>	<code>position</code>	DecimalFloat	#mandatory	With this annotation you specify the order of selection fields that are used for filtering.
<code>UI.selectionField</code>	<code>exclude</code>	Boolean Default: true	#optional	This annotation allows excluding the element from the OData annotation on the derived view by setting it to true. For more information, see Inheritance of Annotations [page 556]
<code>UI.selectionField</code>	<code>element</code>	elementRef	#mandatory: Must be used when an association is annotated, the value is a path to an element of the associated view. You use this option if you want to filter a table for a column that is not defined in your CDS view but in another CDS view.	
				<p>⚠ Caution</p> <p>Must not be used when a structured element is annotated, in this case the annotated element is the value.</p>
<code>UI.valueCriticality</code>		(array of)		Scope: [ELEMENT]
<code>UI.valueCriticality</code>	<code>qualifier</code>	String(120);		
<code>UI.valueCriticality</code>	<code>value</code>	Expression		
<code>UI.valueCriticality</code>	<code>criticality</code>	Integer enum	<ul style="list-style-type: none"> • 1: <code>Negative</code> • 2: <code>Critical</code> • 3: <code>Positive</code> 	
<code>UI.criticalityLabels</code>		(array of)		Scope: [ELEMENT]
<code>UI.criticalityLabels</code>	<code>qualifier</code>	String(120);		

Parent Annotation	Annotation	Type	Value	Description
UI.criticalityLabels	<i>criticality</i>	Integer enum	<ul style="list-style-type: none"> • 1: Negative • 2: Critical • 3: Positive 	
UI.criticalityLabels	<i>label</i>	String(60);		
UI.chart				<p>Annotations belonging to UI.chart are used to show a visual representation of aggregated data.</p> <p>Scope: [ENTITY]</p> <p>Evaluation Runtime (Engine): SADL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: array of</p>
UI.chart	<i>qualifier</i>	String(120)		<p>This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a chart to ensure that the correct chart can be referenced by the UI.</p>
UI.chart	<i>title</i>	String(60)	#optional	<p>This annotation contains a language-dependent text. If omitted, the @EndUserText.label of the annotated entity or view is used.</p>
UI.chart	<i>description</i>	String(120)	#optional	<p>This annotation contains a language-dependent text. If omitted, the @EndUserText.quickInfo of the annotated entity or view is used.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.chart</code>	<code>chartType</code>	String enum	COLUMN; COLUMN_STACKED; COL- UMN_STACKED_100; COLUMN_DUAL; COL- UMN_STACKED_DUAL; COL- UMN_STACKED_DUAL_ 100; BAR; BAR_STACKED; BAR_STACKED_100; BAR_DUAL; BAR_STACKED_DUAL; BAR_STACKED_DUAL_1 00; AREA; AREA_STACKED; AREA_STACKED_100; HORIZONTAL_AREA; HORIZON- TAL_AREA_STACKED; HORIZON- TAL_AREA_STACKED_1 00; LINE; LINE_DUAL; COMBINATION; COMBINA- TION_STACKED; COMBINA- TION_STACKED_DUAL; HORIZONTAL_COMBI- NATION_STACKED;	

Parent Annotation	Annotation	Type	Value	Description
			HORIZONTAL_COMBI- NA- TION_STACKED_DUAL; PIE; DONUT; SCATTER; BUBBLE; RADAR; HEAT_MAP; TREE_MAP; WATERFALL; BULLET; VERTICAL_BULLET; HORIZONTAL_WATER- FALL; HORIZONTAL_COMBI- NATION_DUAL; DONUT_100;	
<i>UI.chart</i>	<i>dimensions</i>	elementRef		This annotation is an array of one or more element references for the discrete axes of a chart. The exact semantics depend on the chart type.
<i>UI.chart</i>	<i>measures</i>	elementRef		This annotation is an array of zero or more element references for the numeric axes of a chart. The exact semantics depend on the chart type.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.chart</code>	<code>dimensionAttributes.dimension</code>	<code>elementRef</code>		This annotation defines the dimensions used in a chart. This annotation must reference an element that is contained in <code>UI.chart.dimensions</code> .
<code>UI.chart</code>	<code>dimensionAttributes.role</code>	<code>String(10) enum</code>	<p>These annotations determine the visualization of a chart.</p> <ul style="list-style-type: none"> • CATEGORY: For example: Line chart: Dimensions for which the role is set to CATEGORY, make up the X-axis (category axis). If no dimension is specified with this role, the first dimension is used as the X-axis. • SERIES: For example: Line chart: Dimensions for which the role is set to SERIES make up the line segments of the chart, with different colors assigned to each dimension value. If multiple dimensions are assigned to this role, the values of all such dimensions together are considered as one dimension and a color is assigned. • CATEGORY2 	This annotation defines the manner in which a dimension is used within a chart. This is configured differently for each chart type.
<code>UI.chart</code>	<code>dimensionAttributes.vacValuesForSequentialColorLevels</code>	<code>array of String(1024)</code>		

Parent Annotation	Annotation	Type	Value	Description
<code>UI.chart</code>	<code>dimensionAttributes.e mphasizedValues</code>	array of String (1024)		
<code>UI.chart</code>	<code>measureAttributes</code>	(array of)		Annotations belonging to <code>UI.chart.measureAttributes</code> are used to specify the measure attributes of a chart.
<code>UI.chart</code>	<code>measureAttributes.measure</code>	ElementRef		This annotation defines the measures used in a chart. This annotation must reference an element that is contained in <code>UI.chart.measures</code> and has a <code>UI.dataPoint</code> annotation.
<code>UI.chart</code>	<code>measureAttributes.role</code>	String (10) enum	This annotation determines the visualization of a chart. <ul style="list-style-type: none">• AXIS_1: Example Bubble chart: The first measure for which the role is set to AXIS_1, or if none exists, the first measure for which the role is set to AXIS_2, or if none exists, the first measure for which the role is set to AXIS_3, is assigned to the feed UID valueAxis. This makes up the X-axis.• AXIS_2: For an example, see the description of AXIS_1.• AXIS_3: For an example, see the description of AXIS_1.	This annotation defines the manner in which a measure is used within a chart. This is configured differently for each chart type.

Parent Annotation	Annotation	Type	Value	Description
<i>UI.chart</i>	<i>measureAttributes.asDataPoint</i>	Boolean	Default : true	This annotation defines whether or not measures are displayed as data points in addition to a chart. The element annotated with this UI annotation needs to have an annotation to a data point.
<i>UI.chart</i>	<i>measureAttributes.useSequentialColorLevels</i>	Boolean	Default : false	
<i>UI.chart</i>	<i>actions</i>			
<i>UI.chart</i>	<i>actions.type</i>	String(40) enum	<ul style="list-style-type: none"> • FOR_ACTION • FOR_INVENTENT_BASED_NAVIGATION 	
<i>UI.chart</i>	<i>actions.label</i>	String(60)		
<i>UI.chart</i>	<i>actions.dataAction</i>	String(120)		
<i>UI.chart</i>	<i>actions.requiresContext</i>	Boolean	Default: true	

Parent Annotation	Annotation	Type	Value	Description
<code>UI.chart</code>	<code>actions. invocationGrouping</code>	String(12) enum	<ul style="list-style-type: none"> • ISOLATED: Describes the error handling when an action cannot be executed on all selected instances: The action is executed on all instances except for instance on which the action cannot be executed. • CHANGE_SET: Describes the error handling when an action cannot be executed on all selected instances: If an action cannot be executed on one of the selected instances, the action is executed on none of the selected instances. 	<p>This annotation expresses how multiple invocations of the same action on multiple instances are grouped.</p> <p>ISOLATED Example: A user selects five items in a list and wants to copy them. One item cannot be copied. This item will not be copied, the other four items are copied.</p> <p>CHANGE_SET Example: A user selects five items in a list and wants to copy them. One item cannot be copied. None of the selected items are copied.</p> <p>Default: ISOLATED</p>
<code>UI.chart</code>	<code>actions.semanticObjec tAction</code>	String(120)		<p>Annotations belonging to <code>UI.selectionPresentationVariant</code> are used to bundle annotations of <code>UI.presentationVariant</code> and <code>UI.selectionVariant</code>.</p> <p>Scope: [ENTITY]</p> <p>Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: array of</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI. selectionPresentationVariant</code>	<code>qualifier</code>	<code>String(120)</code>		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a selection presentation variant to ensure that the correct selection presentation variant can be referenced by the UI.
<code>UI. selectionPresentationVariant</code>	<code>id</code>	<code>String(120)</code>		This annotation contains an identifier to reference this instance from an external context.
<code>UI. selectionPresentationVariant</code>	<code>text</code>	<code>String(60)</code>		This annotation contains the language-dependent name of the selection presentation variant.
<code>UI. selectionPresentationVariant</code>	<code>selectionVariantQualifier</code>	<code>String(120)</code>		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a selection variant to ensure that the correct selection variant can be referenced by the selection presentation variant

Parent Annotation	Annotation	Type	Value	Description
<i>UI.selectionPresentationVariant</i>	<i>presentationVariantQualifier</i>	String (120)		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a presentation variant to ensure that the correct presentation variant can be referenced by the selection presentation variant.
UI selectionVariant				<p>Annotations belonging to <i>UI.selectionVariant</i> are used to denote a combination of parameters and filters used to query the annotated entity set.</p> <p>Scope: [ENTITY]</p> <p>Evaluation Runtime (Engine): SADL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: array of</p>
<i>UI.selectionVariant</i>	<i>qualifier</i>	String (120)		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a selection variant to ensure that the correct selection variant can be referenced by the UI.
<i>UI.selectionVariant</i>	<i>id:</i>	String (120)		This annotation can contain an identifier to reference this instance from an external context.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.selectionVariant</code>	<code>text</code>	String (60)		This annotation contains the language-dependent name of the selection variant.
<code>UI.selectionVariant</code>	<code>parameters</code>	(array of)		Annotations belonging to <code>UI.selectionVariant.parameters</code> represent a collection of parameters used to query the annotated entity set.
<code>UI.selectionVariant</code>	<code>parameters.name</code>	ParameterRef		This annotation references to a parameter name.
<code>UI.selectionVariant</code>	<code>parameters.value</code>	String (1024)		This annotation contains a parameter value.
<code>UI.selectionVariant</code>	<code>filter</code>	String (1024)		This annotation contains a filter used to query the annotated entity set.
<code>UI.presentationVariant</code>	<code>qualifier</code>	String (120)		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a presentation variant to ensure that the correct presentation variant can be referenced by the UI.
<code>UI.presentationVariant</code>	<code>id:</code>	String (120)		This annotation contains an identifier to reference this instance from an external context.
<code>UI.presentationVariant</code>	<code>text</code>	String (60)		This annotation contains the language-dependent name of the presentation variant.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.presentationVariant</code>	<code>maxItems</code>	Integer		This annotation defines the maximum number of items that should be included in the result.
<code>UI.presentationVariant</code>	<code>sortOrder</code>	(array of)		Annotations belonging to <code>UI.presentationVariant.sortOrder</code> represent a collection of sorting parameters that can be provided inline or by a reference to a <code>Common.SortOrder</code> annotation (syntax is identical to <code>AnnotationPath</code>).
<code>UI.presentationVariant</code>	<code>sortOrder.by</code>	elementRef		This annotation defines by what property queried collections can be sorted.
<code>UI.presentationVariant</code>	<code>sortOrder.direction</code>	String(4) enum	<ul style="list-style-type: none"> • <code>ASC</code>: Sort in ascending order. • <code>DESC</code>: Sort in descending order. 	This annotation defines the sorting direction of queried collections.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.presentationVariant groupBy</code>		array of elementRef		<p>This annotation defines a sequence of groupable properties (p1, p2, pn) that define how the result of a queried collection is composed of instances that represent groups, one group for each combination of value properties in the queried collection.</p> <p>The sequence specifies a certain level of aggregation for the queried collection, and every group instance provides aggregated values for properties that are aggregatable. Moreover, the series of sub-sequences, for example (p1), (p1, p2), ..., forms a leveled hierarchy that can be used in combination with the annotation <code>UI.presentationVariant.initialExpansionLevel</code>.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.presentationVariant</code>	<code>totalBy</code>	array of elementRef		This annotation defines the sub-sequence q1, q2, qn of the properties p1, p2, pn specified in the annotation <code>UI.presentationVariant.groupBy</code> . With this, additional levels of aggregation are requested in addition to the most granular level defined by the annotation <code>UI.presentationVariant.groupBy</code> . Every element in the series of sub-sequences, for example (q1), (q1, q2), ..., introduces an additional aggregation level included in the result of the queried collection.
<code>UI.presentationVariant</code>	<code>total</code>	array of elementRef		This annotation contains aggregatable properties for which aggregated values are to be provided for the additional aggregation levels specified in the annotation <code>UI.presentationVariant.totalBy</code> .
<code>UI.presentationVariant</code>	<code>includeGrandTotal</code>	Boolean Default: true		This annotation specifies that the result of the queried collection includes a grand total for the properties specified in the annotation <code>UI.presentationVariant.total</code> .

Parent Annotation	Annotation	Type	Value	Description
<code>UI.presentationVariant</code>	<code>initialExpansionLevel</code>	Integer		This annotation contains the initial number of expansion levels of a hierarchy defined for the queried collection. The hierarchy can be implicitly imposed by the sequence of the annotation <code>UI.presentationVariant.groupBy</code> , or by an explicit hierarchy annotation.
<code>UI.presentationVariant</code>	<code>requestAtLeast</code>	array of elementRef		This annotation defines the properties that should always be included in the result of the queried collection.
<code>UI.presentationVariant</code>	<code>visualizations</code>	(array of)		<p>Annotations belonging to <code>UI.presentationVariant.visualizations</code> represent a collection of available visualization types. The following types are supported:</p> <ul style="list-style-type: none"> <code>UI.lineItem</code> <code>UI.chart</code> <code>UI.dataPoint</code>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.presentationVariant</code>	<code>visualizations.type</code>	<code>String(40)</code> enum	<ul style="list-style-type: none"> • <code>AS_LINEITEM</code>: The queried collection is visualized as line item. • <code>AS_CHART</code>: The queried collection is visualized as chart. • <code>AS_DATAPOINT</code>: The queried collection is visualized as data point. <p>Default: AS_LINEITEM</p>	This annotation defines the representation type. For each type, only one single annotation is meaningful. Multiple instances of the same visualization type shall be modeled with different presentation variants. A reference to the annotation <code>UI.lineItem</code> should always be part of the collection (least common denominator for renderers).
<code>UI.presentationVariant</code>	<code>visualizations.qualifier</code>	<code>String(120)</code>		This annotation is used to group and uniquely identify annotations. You need to specify a qualifier as name of a visualization to ensure that the correct visualization can be referenced by the UI.
<code>UI.presentationVariant</code>	<code>visualizations.element</code>	<code>ElementRef</code>		This annotation references the annotation <code>UI.lineItem</code> .
<code>UI.presentationVariant</code>	<code>selectionFieldsQualifier</code>	<code>String(120)</code>		

Parent Annotation	Annotation	Type	Value	Description
UI.hidden		Boolean		This annotation allows to show or hide data fields based on the state of the data instance. For more information, see Field Hiding.
		Default true		Scope: [ELEMENT, PARAMETER]
				Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations
UI.masked		Boolean		This annotation refers to, for example, passwords or pass phrases. The user interface may offer to show the value in clear text upon explicit user interaction. For more information, see Field Masking.
		Default true		Scope: [ELEMENT]
				Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations

Parent Annotation	Annotation	Type	Value	Description
<i>UI.multiLineText</i>		Boolean		UI.multiLineText
		Default true		This annotation contains text that is rendered as multiple lines. For more information, see Multi-Line Text [page 552] .
				Scope: [ELEMENT] Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations
<i>UI.textArrangement</i>		String (13) enum	<ul style="list-style-type: none"> • <i>TEXT_FIRST</i>: The text is displayed in front of the code. • <i>TEXT_LAST</i>: The code is displayed in front of the text. • <i>TEXT_ONLY</i>: The text is displayed without the code. • <i>TEXT_SEPARATE</i>: TEXT_SEPARATE The text and the code are displayed separately. 	<p>Description: This annotation specifies the arrangement of code-text pairs.</p> <p>Scope: [ENTITY, ELEMENT]</p> <p>Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: String</p>
<i>UI.facet</i>	<i>qualifier</i>	String (120)		This element uniquely identifies alternative values for an annotation; only allowed if parentId is not specified.
<i>UI.facet</i>	<i>feature</i>	String (40)		This annotation specifies the name of a feature toggle that determines visibility of the facet.
<i>UI.facet</i>	<i>id</i>	String (120)		This element is the identifier of the facet

Parent Annotation	Annotation	Type	Value	Description
<code>UI.facet</code>	<code>purpose</code>	String(40) enum	#not compatible with UI.facet.parentID STANDARD; HEADER; QUICK_VIEW; QUICK_CREATE; FILTER; default: STANDARD;	This annotation specifies the purpose of a facet; only allowed if parentId is not specified
			⚠ Caution You can only use this annotation, if parentId is not specified.	This annotation identifies the parent facet. only allowed if purpose is not specified
<code>UI.facet</code>	<code>parentId</code>	String(120)	#not compatible with UI.facet.purpose STANDARD; HEADER; QUICK_VIEW; QUICK_CREATE; FILTER; default: STANDARD;	
			⚠ Caution You can only use this annotation, if purpose is not specified.	This annotation specifies the relative position of the Facet within its parent or term. Must be specified to allow interspersing extension elements via extend views.
<code>UI.facet</code>	<code>position</code>	DecimalFloat		
<code>UI.facet</code>	<code>exclude</code>	Boolean Default: true		Marker to explicitly exclude a CDS element from the collection of DataField for a derived view.

Parent Annotation	Annotation	Type	Value	Description
<i>UI.facet</i>	<i>hidden</i>	Boolean		The annotation <i>hidden</i> allows providing a static Boolean value (for facets hidden by default and e.g. made visible via code exit in the UI), but usually the value will be provided by referencing a Boolean element of the same view using the #(...) syntax. The latter allows to dynamically switch the facet on or off.
<i>UI.facet</i>	<i>isPartOfPreview</i>	Boolean	Default: true	This annotation determines that this facet and all included facets are part of the thing preview.
<i>UI.facet</i>	<i>isSummary</i>	Boolean	Default: true	This annotation specifies that this facet and all included facets are the summary of the thing. At most one facet of a thing can be tagged with this term
<i>UI.facet</i>	<i>isMap</i>	Boolean	Default: true	This annotation specifies that this facet is best represented as a geographic map.
<i>UI.facet</i>	<i>importance</i>	String (6) enum	<ul style="list-style-type: none"> • HIGH • MEDIUM • LOW 	<p>This expresses the importance of dataFields or other annotations. It can be used e.g. in dynamic rendering approaches with responsive design.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.facet</code>	<code>label</code>	String (60)	#optional	This annotation can contain a language-dependent text; if omitted, the label of the annotated element or the label of the element referenced via value which is used.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.facet</code>	<code>type</code>	String(40) enum	<ul style="list-style-type: none"> • COLLECTION: <ul style="list-style-type: none"> • id element is required • no additional elements are available • ADDRESS_REFERENCE <ul style="list-style-type: none"> - targetElement element is available - targetQualifier element is available - isMap element is available • BADGE_REFERENCE <ul style="list-style-type: none"> - targetElement element is available • CHART_REFERENCE <ul style="list-style-type: none"> - targetElement element is available - targetQualifier element is available • CONTACT_REFERENCE <ul style="list-style-type: none"> - targetElement element is available - targetQualifier element is available • DATAPOINT_REFERENCE <ul style="list-style-type: none"> - targetElement element is available - targetQualifier element is required • HEADERINFO_REFERENCE <ul style="list-style-type: none"> - targetElement element is available - targetQualifier element is required • IDENTIFICATION_REFERENCE <ul style="list-style-type: none"> - targetElement element is available • LINEITEM_REFERENCE 	This enumeration element specifies the concrete type of the facet. The enumeration type value determines which CDS annotation elements are required or available

Parent Annotation	Annotation	Type	Value	Description
			<ul style="list-style-type: none"> - targetElement element is available - targetQualifier element is available • STATUSINFO_REFERENCE - targetElement element is available - targetQualifier element is available • URL_REFERENCE - url element is available 	
<i>UI.facet</i>	<i>targetElement</i>	elementRef		This annotation specifies a path to the CDS view whose annotation is referenced. Not specified means the current view. The path is converted to an OData Annotation-Path.
<i>UI.facet</i>	<i>targetQualifier</i>	String (120)		This annotation is the qualifier of the referenced annotation.
<i>UI.facet</i>	<i>url</i>	ElementRef		This annotation specifies a path to structural element containing the navigation URL.

Parent Annotation	Annotation	Type	Value	Description
UI.lineItem				<p>Annotations belonging to UI.lineItem represent an ordered collection of data fields that is used to represent data from multiple data instances in a table or a list. For more information, see Columns.</p> <p>Scope: [ELEMENT, ENTITY]</p> <p>Evaluation Runtime (Engine): SADL: Translates CDS annotations into the corresponding OData annotations</p>
UI.lineItem	<i>qualifier</i>	String (120)		<p>This annotation is used to group and uniquely identify annotations.</p> <p>If you want to use more than one table, you need a qualifier to distinguish them on the UI.</p>
UI.lineItem	<i>position</i>	DecimalFloat	#mandatory	<p>With this annotation you specify the order of the columns of a list.</p>
UI.lineItem	<i>exclude</i>	Boolean Default: True		<p>This annotation allows excluding the element from the OData annotation on the derived view by setting it to true. The element is optional.</p> <p>For more information, see Inheritance of Annotations [page 556]</p>

Parent Annotation	Annotation	Type	Value	Description
UI.lineItem	<i>hidden</i>	Boolean		The annotation <code>hidden</code> allows providing a static boolean value, but usually the value will be provided by referencing a Boolean element of the same view using the <code>#(...)</code> syntax.
UI.lineItem	<i>importance</i>	String(6) enum		<p>i Note</p> <p>If no importance is defined, the line item is treated like having importance LOW.</p> <ul style="list-style-type: none"> • HIGH • MEDIUM • LOW <p>This annotation expresses the importance of dataFields or other annotations. The element can be used, for example, in dynamic rendering approaches with responsive design patterns.</p> <p>Example:</p> <p>You defined a table with several columns. The columns that need to be displayed always, get importance HIGH. This ensures that these columns are displayed in a table when this table is rendered on a small display.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.lineItem</code>	<code>type</code>	<code>String(40) enum</code>	<code>AS_ADDRESS;</code> <code>AS_CHART;</code> <code>AS_CONNECTED_FIELDS;</code> <code>AS_CONTACT;</code> <code>AS_DATAPOINT;</code> <code>AS_FIELDGROUP;</code> <code>FOR_ACTION;</code> <code>FOR_IN-</code> <code>TENT_BASED_NAVIGA-</code> <code>TION;</code> <code>STANDARD;</code> <code>WITH_IN-</code> <code>TENT_BASED_NAVIGA-</code> <code>TION;</code> <code>WITH_NAVIGA-</code> <code>TION_PATH;</code> <code>WITH_URL;</code> <code>} default #STANDARD;</code> <code>label</code>	
<code>UI.lineItem</code>	<code>label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for column titles in tables headers.</p> <p>If omitted, the label of the annotated element, or the label of the element referenced via the value is used.</p>
<code>UI.lineItem</code>	<code>iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.lineItem</code>	<code>criticality</code>	<code>ElementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	
<code>UI.lineItem</code>	<code>criticalityRepresentation</code>	<code>String(12)</code> enum	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> 	<p>Default:</p> <p><code>WITHOUT_ICON</code></p>
<code>UI.lineItem</code>	<code>dataAction</code>	<code>String(120)</code>		<p>This annotation can be used if the line item type is <code>FOR_ACTION</code>. The element references the technical name of an action of the Business Object Processing Framework (BOPF), for example. In this case, the string pattern is <code>BOPF:<technical name of action in BOPF></code>.</p>
<code>UI.lineItem</code>	<code>requiresContext</code>	<code>Boolean</code>		<p>Default: True</p>

Parent Annotation	Annotation	Type	Value	Description
UI.lineItem	invocationGrouping	String(12) enum	#optional	<p>i Note</p> <p>This annotation needs to be specified if you use UI.lineItem.type of type FOR_ACTION.</p> <ul style="list-style-type: none"> • ISOLATED: Describes the error handling when an action cannot be executed on all selected instances: The action is executed on all instances except for instance on which the action cannot be executed. • CHANGE_SET: Describes the error handling when an action cannot be executed on all selected instances: If an action cannot be executed on one of the selected instances, the action is executed on none of the selected instances. <p>ISOLATED Example: A user selects five items in a list and wants to copy them. One item cannot be copied. This item will not be copied, the other four items are copied.</p> <p>CHANGE_SET Example: A user selects five items in a list and wants to copy them. One item cannot be copied. None of the selected items are copied.</p> <p>Default: ISOLATED</p>
UI.lineItem	semanticObjectAction	String(120)		This annotation refers to the name of an action on the semantic object. The semantic object is taken from @Consumption.semanticObject or derived via an association from the defining view.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.lineItem</code>	<code>value</code>	<code>ElementRef</code>	<p>For type AS_ADDRESS:</p> <ul style="list-style-type: none"> Value element must not be used when a structural element is annotated. Use instead <code>@com.sap.vocabularies.Communication.v1.Address</code> (or a shorter alias-qualified name) as value. Value element must be used when an element of an associated CDS view is annotated. A value of '.' refers to <code>@Semantics.address</code> on the view that is directly associated. If you want to reference <code>@Semantics.address</code> on a view that is indirectly associated, use a path starting with a dot as value. <p>All other types:</p> <ul style="list-style-type: none"> Value element must not be used when an element is annotated, in this case the annotated element is the value. Value element must be used when an association is annotated. The value is a path to an element of the associated view. 	This annotation refers to a value.
<code>UI.lineItem</code>	<code>valueQualifier</code>	<code>String (120)</code>		

Parent Annotation	Annotation	Type	Value	Description
<code>UI.lineItem</code>	<code>targetElement</code>	<code>ElementRef</code>		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.lineItem.targetElement</code> when you use the annotation <code>UI.lineItem.type</code> of type <code>WITH_NAVIGATION_PATH</code>.</p> <p>You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.lineItem</code>	<code>url</code>	<code>ElementRef</code>		<p>⚠ Caution</p> <p>You need to specify <code>UI.lineItem.url</code> when you use the annotation <code>UI.lineItem.type</code> of type <code>WITH_URL</code>.</p> <p>This annotation represents the path to a structural element that contains a navigation URL.</p>

Parent Annotation	Annotation	Type	Value	Description
<i>UI.identification</i>				<p>Annotation belonging to <i>UI.identification</i> represent an ordered collection of specific data fields that together with headerInfo identifies an entity to an end user.</p> <p>Example</p> <p>This annotation is displayed in the General Information section in the body of the object view floorplan of an item, for example.</p> <p>Scope: [ELEMENT]</p> <p>Evaluation Runtime (Engine): SADL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: array of</p>
<i>UI.identification</i>	<i>qualifier</i>	String(120)		<p>This annotation is used to group and uniquely identify annotations.</p> <p>If you want to use more than one table, you need a qualifier to distinguish them on the UI.</p>
<i>UI.identification</i>	<i>position</i>	DecimalFloat	#mandatory	With this annotation you specify the order of the columns of a list.

Parent Annotation	Annotation	Type	Value	Description
<i>UI.identification</i>	<i>exclude</i>	Boolean Default: True		<p>This annotation allows excluding the element from the OData annotation on the derived view by setting it to <code>true</code>. The element is optional.</p> <p>For more information, see Inheritance of Annotations [page 556]</p>
<i>UI.identification</i>	<i>hidden</i>	Boolean Default: True		<p>The annotation <code>hidden</code> allows providing a static boolean value, but usually the value will be provided by referencing a Boolean element of the same view using the <code>#(...)</code> syntax.</p>
<i>UI.identification</i>	<i>importance</i>	String(6) enum	<p>i Note</p> <p>If no importance is defined, the line item is treated like having importance <code>LOW</code>.</p> <ul style="list-style-type: none"> • <code>HIGH</code> • <code>MEDIUM</code> • <code>LOW</code> 	<p>This annotation expresses the importance of dataFields or other annotations. The element can be used, for example, in dynamic rendering approaches with responsive design patterns.</p> <p>Example:</p> <p>You defined a table with several columns. The columns that need to be displayed always, get importance <code>HIGH</code>. This ensures that these columns are displayed in a table when this table is rendered on a small display.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.identification</code>	<code>type</code>	<code>String(40) enum</code>	<code>AS_ADDRESS;</code> <code>AS_CHART;</code> <code>AS_CONNECTED_FIELDS;</code> <code>AS_CONTACT;</code> <code>AS_DATAPOINT;</code> <code>AS_FIELDGROUP;</code> <code>FOR_ACTION;</code> <code>FOR_IN-</code> <code>TENT_BASED_NAVIGA-</code> <code>TION;</code> <code>STANDARD;</code> <code>WITH_IN-</code> <code>TENT_BASED_NAVIGA-</code> <code>TION;</code> <code>WITH_NAVIGA-</code> <code>TION_PATH;</code> <code>WITH_URL;</code> <code>} default #STANDARD;</code> <code>label</code>	
<code>UI.identification</code>	<code>label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for column titles in tables headers.</p> <p>If omitted, the label of the annotated element, or the label of the element referenced via the value is used.</p>
<code>UI.identification</code>	<code>iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.identification</code>	<code>criticality</code>	<code>ElementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	
<code>UI.identification</code>	<code>criticalityRepresentation</code>	<code>String(12)</code> enum	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> 	<p>Default:</p> <p><code>WITHOUT_ICON</code></p>
<code>UI.identification</code>	<code>dataAction</code>	<code>String(120)</code>		<p>This annotation can be used if the line item type is <code>FOR_ACTION</code>. The element references the technical name of an action of the Business Object Processing Framework (BOPF), for example. In this case, the string pattern is <code>BOPF:<technical name of action in BOPF></code>.</p>
<code>UI.identification</code>	<code>requiresContext</code>	<code>Boolean</code>		<p>Default: True</p>

Parent Annotation	Annotation	Type	Value	Description
UI.identification	invocationGrouping	String(12) enum	#optional	<p>i Note</p> <p>This annotation needs to be specified if you use UI.lineItem.type of type FOR_ACTION.</p> <ul style="list-style-type: none"> • ISOLATED: Describes the error handling when an action cannot be executed on all selected instances: The action is executed on all instances except for instance on which the action cannot be executed. • CHANGE_SET: Describes the error handling when an action cannot be executed on all selected instances: If an action cannot be executed on one of the selected instances, the action is executed on none of the selected instances. <p>ISOLATED Example: A user selects five items in a list and wants to copy them. One item cannot be copied. This item will not be copied, the other four items are copied.</p> <p>CHANGE_SET Example: A user selects five items in a list and wants to copy them. One item cannot be copied. None of the selected items are copied.</p> <p>Default: ISOLATED</p>
UI.identification	semanticObjectAction	String(120)		This annotation refers to the name of an action on the semantic object. The semantic object is taken from @Consumption.semanticObject or derived via an association from the defining view.

Parent Annotation	Annotation	Type	Value	Description
<i>UI.identification</i>	<i>value</i>	ElementRef	<p>For type AS_ADDRESS:</p> <ul style="list-style-type: none"> • Value element must not be used when a structural element is annotated. Use instead @com.sap.vocabularies.Communication.v1.Address (or a shorter alias-qualified name) as value. • Value element must be used when an element of an associated CDS view is annotated. A value of '.' refers to @Semantics.address on the view that is directly associated. • If you want to reference @Semantics.address on a view that is indirectly associated, use a path starting with a dot as value. <p>All other types:</p> <ul style="list-style-type: none"> • Value element must not be used when an element is annotated, in this case the annotated element is the value. • Value element must be used when an association is annotated. The value is a path to an element of the associated view. 	This annotation refers to a value.
<i>UI.identification</i>	<i>valueQualifier</i>	String (120)		

Parent Annotation	Annotation	Type	Value	Description
<code>UI.identification</code>	<code>targetElement</code>	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.lineItem.targetElement</code> when you use the annotation <code>UI.lineItem.type</code> of type <code>WITH_NAVIGATION_PATH</code>.</p> <p>You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.identification</code>	<code>url</code>	ElementRef		<p>⚠ Caution</p> <p>You need to specify <code>UI.identification.url</code> when you use the annotation <code>UI.identification.type</code> of type <code>WITH_URL</code>.</p> <p>This annotation represents the path to a structural element that contains a navigation URL.</p>

Parent Annotation	Annotation	Type	Value	Description
UI.statusInfo				<p>Annotations belonging to UI.statusInfo represent a list of abstract data fields that convey the status of an entity. UI.statusInfo annotations are usually used in the header section of an item's object view floorplan.</p> <p>Scope: [ELEMENT]</p> <p>Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: array of</p>
UI.statusInfo	<i>qualifier</i>	String(120)		<p>This annotation is used to group and uniquely identify annotations.</p> <p>If you want to use more than one table, you need a qualifier to distinguish them on the UI.</p>
UI.statusInfo	<i>position</i>	DecimalFloat	#mandatory	<p>With this annotation you specify the order of the columns of a list.</p>
UI.statusInfo	<i>exclude</i>	Boolean Default: True		<p>This annotation allows excluding the element from the OData annotation on the derived view by setting it to true. The element is optional.</p> <p>For more information, see Inheritance of Annotations [page 556]</p>

Parent Annotation	Annotation	Type	Value	Description
UI.statusInfo	<i>hidden</i>	Boolean		The annotation <code>hidden</code> allows providing a static boolean value, but usually the value will be provided by referencing a Boolean element of the same view using the <code>#(...)</code> syntax.
UI.statusInfo	<i>importance</i>	String(6) enum		<p>i Note</p> <p>If no importance is defined, the line item is treated like having importance LOW.</p> <ul style="list-style-type: none"> • HIGH • MEDIUM • LOW <p>This annotation expresses the importance of dataFields or other annotations. The element can be used, for example, in dynamic rendering approaches with responsive design patterns.</p> <p>Example:</p> <p>You defined a table with several columns. The columns that need to be displayed always, get importance HIGH. This ensures that these columns are displayed in a table when this table is rendered on a small display.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.statusInfo</code>	<code>type</code>	<code>String(40) enum</code>	<pre>AS_ADDRESS; AS_CHART; AS_CONNECTED_FIELDS; AS_CONTACT; AS_DATAPOINT; AS_FIELDGROUP; FOR_ACTION; FOR_IN- TENT_BASED_NAVIGA- TION; STANDARD; WITH_IN- TENT_BASED_NAVIGA- TION; WITH_NAVIGA- TION_PATH; WITH_URL; } default #STANDARD; label</pre>	
<code>UI.statusInfo</code>	<code>label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for column titles in tables headers.</p> <p>If omitted, the label of the annotated element, or the label of the element referenced via the value is used.</p>
<code>UI.statusInfo</code>	<code>iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.statusInfo</code>	<code>criticality</code>	<code>ElementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	
<code>UI.statusInfo</code>	<code>criticalityRepresentation</code>	<code>String(12)</code> enum	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> 	<p>Default:</p> <p><code>WITHOUT_ICON</code></p>
<code>UI.statusInfo</code>	<code>dataAction</code>	<code>String(120)</code>		<p>This annotation can be used if the line item type is <code>FOR_ACTION</code>. The element references the technical name of an action of the Business Object Processing Framework (BOPF), for example. In this case, the string pattern is <code>BOPF:<technical name of action in BOPF></code>.</p>
<code>UI.statusInfo</code>	<code>requiresContext</code>	<code>Boolean</code>		<p>Default: True</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.statusInfo</code>	<code>invocationGrouping</code>	String(12) enum	#optional	<p>i Note</p> <p>This annotation needs to be specified if you use <code>UI.lineItem.type</code> of type <code>FOR_ACTION</code>.</p> <ul style="list-style-type: none"> • <code>ISOLATED</code>: Describes the error handling when an action cannot be executed on all selected instances: The action is executed on all instances except for instance on which the action cannot be executed. • <code>CHANGE_SET</code>: Describes the error handling when an action cannot be executed on all selected instances: If an action cannot be executed on one of the selected instances, the action is executed on none of the selected instances. <p>ISOLATED Example: A user selects five items in a list and wants to copy them. One item cannot be copied. This item will not be copied, the other four items are copied.</p> <p>CHANGE_SET Example: A user selects five items in a list and wants to copy them. One item cannot be copied. None of the selected items are copied.</p> <p>Default: ISOLATED</p>
<code>UI.statusInfo</code>	<code>semanticObjectAction</code>	String(120)		This annotation refers to the name of an action on the semantic object. The semantic object is taken from <code>@Consumption.semanticObject</code> or derived via an association from the defining view.

Parent Annotation	Annotation	Type	Value	Description
UI.statusInfo	value	ElementRef	<p>For type AS_ADDRESS:</p> <ul style="list-style-type: none"> Value element must not be used when a structural element is annotated. Use instead <code>@com.sap.vocabularies.Communication.v1.Address</code> (or a shorter alias-qualified name) as value. Value element must be used when an element of an associated CDS view is annotated. A value of '.' refers to <code>@Semantics.address</code> on the view that is directly associated. If you want to reference <code>@Semantics.address</code> on a view that is indirectly associated, use a path starting with a dot as value. <p>All other types:</p> <ul style="list-style-type: none"> Value element must not be used when an element is annotated, in this case the annotated element is the value. Value element must be used when an association is annotated. The value is a path to an element of the associated view. 	This annotation refers to a value.
UI.statusInfo	valueQualifier	String (120)		

Parent Annotation	Annotation	Type	Value	Description
<code>UI.statusInfo</code>	<code>targetElement</code>	<code>ElementRef</code>		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.lineItem.targetElement</code> when you use the annotation <code>UI.lineItem.type</code> of type <code>WITH_NAVIGATION_PATH</code>.</p> <p>You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.statusInfo</code>	<code>url</code>	<code>ElementRef</code>		<p>⚠ Caution</p> <p>You need to specify <code>UI.statusInfo.url</code> when you use the annotation <code>UI.statusInfo.type</code> of type <code>WITH_URL</code>.</p> <p>This annotation represents the path to a structural element that contains a navigation URL.</p>

Parent Annotation	Annotation	Type	Value	Description
UI.fieldGroup				<p>Annotations belonging to UI.fieldGroup is an ordered collection of data fields with a label for the group.</p> <p>UI.fieldGroup annotations are used to represent parts of a single data instance in a form.</p> <p>Scope: [ELEMENT]</p>
				<p>Evaluation Runtime (Engine): SSDL: Translates CDS annotations into the corresponding OData annotations</p> <p>Values: array of</p>
UI.fieldGroup	qualifier	String(120)		<p>This annotation is used to group and uniquely identify annotations.</p> <p>If you want to use more than one table, you need a qualifier to distinguish them on the UI.</p>
UI.fieldGroup	groupLabel		#optional	<p>This annotation contains language-dependent text that is used as label for the field group. The first occurrence for a given qualifier wins. Other occurrences for the same qualifier are redundant.</p>
UI.fieldGroup	position	DecimalFloat	#mandatory	<p>With this annotation you specify the order of the columns of a list.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.fieldGroup</code>	<code>exclude</code>	Boolean		This annotation allows excluding the element from the OData annotation on the derived view by setting it to <code>true</code> . The element is optional.
			Default: True	For more information, see Inheritance of Annotations [page 556]
<code>UI.fieldGroup</code>	<code>hidden</code>	Boolean		The annotation <code>hidden</code> allows providing a static boolean value, but usually the value will be provided by referencing a Boolean element of the same view using the <code>#(...)</code> syntax.
			Default: True	
<code>UI.fieldGroup</code>	<code>importance</code>	String(6) enum	<p>i Note</p> <p>If no importance is defined, the line item is treated like having importance LOW.</p> <ul style="list-style-type: none"> • HIGH • MEDIUM • LOW 	<p>This annotation expresses the importance of dataFields or other annotations. The element can be used, for example, in dynamic rendering approaches with responsive design patterns.</p> <p>Example:</p> <p>You defined a table with several columns. The columns that need to be displayed always, get importance HIGH. This ensures that these columns are displayed in a table when this table is rendered on a small display.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.fieldGroup</code>	<code>type</code>	<code>String(40) enum</code>	<code>AS_ADDRESS;</code> <code>AS_CHART;</code> <code>AS_CONNECTED_FIELDS;</code> <code>AS_CONTACT;</code> <code>AS_DATAPOINT;</code> <code>AS_FIELDGROUP;</code> <code>FOR_ACTION;</code> <code>FOR_IN-</code> <code>TENT_BASED_NAVIGA-</code> <code>TION;</code> <code>STANDARD;</code> <code>WITH_IN-</code> <code>TENT_BASED_NAVIGA-</code> <code>TION;</code> <code>WITH_NAVIGA-</code> <code>TION_PATH;</code> <code>WITH_URL;</code> <code>} default #STANDARD;</code> <code>label</code>	
<code>UI.fieldGroup</code>	<code>label</code>	<code>String(60)</code>	<code>#optional</code>	<p>This annotation contains a language-dependent text that can be used for column titles in tables headers.</p> <p>If omitted, the label of the annotated element, or the label of the element referenced via the value is used.</p>
<code>UI.fieldGroup</code>	<code>iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.fieldGroup</code>	<code>criticality</code>	<code>ElementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	
<code>UI.fieldGroup</code>	<code>criticalityRepresentation</code>	<code>String(12)</code> enum	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> 	<p>Default:</p> <p><code>WITHOUT_ICON</code></p>
<code>UI.fieldGroup</code>	<code>dataAction</code>	<code>String(120)</code>		<p>This annotation can be used if the line item type is <code>FOR_ACTION</code>. The element references the technical name of an action of the Business Object Processing Framework (BOPF), for example. In this case, the string pattern is <code>BOPF:<technical name of action in BOPF></code>.</p>
<code>UI.fieldGroup</code>	<code>requiresContext</code>	<code>Boolean</code>		<p>Default: True</p>

Parent Annotation	Annotation	Type	Value	Description
UI.fieldGroup	invocationGrouping	String(12) enum	#optional	<p>i Note</p> <p>This annotation needs to be specified if you use UI.lineItem.type of type FOR_ACTION.</p> <ul style="list-style-type: none"> • ISOLATED: Describes the error handling when an action cannot be executed on all selected instances: The action is executed on all instances except for instance on which the action cannot be executed. • CHANGE_SET: Describes the error handling when an action cannot be executed on all selected instances: If an action cannot be executed on one of the selected instances, the action is executed on none of the selected instances. <p>ISOLATED Example: A user selects five items in a list and wants to copy them. One item cannot be copied. This item will not be copied, the other four items are copied.</p> <p>CHANGE_SET Example: A user selects five items in a list and wants to copy them. One item cannot be copied. None of the selected items are copied.</p> <p>Default: ISOLATED</p>
UI.fieldGroup	semanticObjectAction	String(120)		This annotation refers to the name of an action on the semantic object. The semantic object is taken from @Consumption.semanticObject or derived via an association from the defining view.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.fieldGroup</code>	<code>value</code>	<code>ElementRef</code>	<p>For type AS_ADDRESS:</p> <ul style="list-style-type: none"> Value element must not be used when a structural element is annotated. Use instead <code>@com.sap.vocabularies.Communication.v1.Address</code> (or a shorter alias-qualified name) as value. Value element must be used when an element of an associated CDS view is annotated. A value of '.' refers to <code>@Semantics.address</code> on the view that is directly associated. If you want to reference <code>@Semantics.address</code> on a view that is indirectly associated, use a path starting with a dot as value. <p>All other types:</p> <ul style="list-style-type: none"> Value element must not be used when an element is annotated, in this case the annotated element is the value. Value element must be used when an association is annotated. The value is a path to an element of the associated view. 	This annotation refers to a value.
<code>UI.fieldGroup</code>	<code>valueQualifier</code>	<code>String (120)</code>		

Parent Annotation	Annotation	Type	Value	Description
UI.fieldGroup	targetElement	ElementRef		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify UI.lineItem.targetElement when you use the annotation UI.lineItem.type of type WITH_NAVIGATION_PATH.</p> <p>You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
UI.fieldGroup	url	ElementRef		<p>⚠ Caution</p> <p>You need to specify UI.fieldGroup.url when you use the annotation UI.fieldGroup.type of type WITH_URL..</p> <p>This annotation represents the path to a structural element that contains a navigation URL.</p>
UI.connectedFields				
UI.connectedFields	qualifier	String(120)		<p>This annotation is used to group and uniquely identify annotations.</p> <p>If you want to use more than one table, you need a qualifier to distinguish them on the UI.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>groupLabel</code>		#optional	This annotation contains language-dependent text that is used as label for the field group. The first occurrence for a given qualifier wins. Other occurrences for the same qualifier are redundant.
<code>UI.connectedFields</code>	<code>template</code>		<code>String(255)</code>	
<code>UI.connectedFields</code>	<code>name</code>		<code>String(120)</code>	
<code>UI.connectedFields</code>	<code>exclude</code>		<code>Boolean</code> Default: True	This annotation allows excluding the element from the OData annotation on the derived view by setting it to <code>true</code> . The element is optional. For more information, see Inheritance of Annotations [page 556]
<code>UI.connectedFields</code>	<code>hidden</code>		<code>Boolean</code> Default: True	The annotation <code>hidden</code> allows providing a static boolean value, but usually the value will be provided by referencing a Boolean element of the same view using the <code>#(...)</code> syntax.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>importance</code>	<code>String(6) enum</code>	<p>i Note</p> <p>If no importance is defined, the line item is treated like having importance LOW.</p> <ul style="list-style-type: none"> • HIGH • MEDIUM • LOW 	<p>This annotation expresses the importance of dataFields or other annotations. The element can be used, for example, in dynamic rendering approaches with responsive design patterns.</p> <p>Example:</p> <p>You defined a table with several columns. The columns that need to be displayed always, get importance HIGH. This ensures that these columns are displayed in a table when this table is rendered on a small display.</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>type</code>	<code>String(40) enum</code>	<pre>AS_ADDRESS; AS_CHART; AS_CONNECTED_FIELDS; AS_CONTACT; AS_DATAPOINT; AS_FIELDGROUP; FOR_ACTION; FOR_IN- TENT_BASED_NAVIGA- TION; STANDARD; WITH_IN- TENT_BASED_NAVIGA- TION; WITH_NAVIGA- TION_PATH; WITH_URL; } default #STANDARD; label</pre>	This annotation contains a language-dependent text that can be used for column titles in tables headers. If omitted, the label of the annotated element, or the label of the element referenced via the value is used.
<code>UI.connectedFields</code>	<code>label</code>	<code>String(60)</code>	<code>#optional</code>	
<code>UI.connectedFields</code>	<code>iconUrl</code>	<code>String(1024)</code>	<code>#optional</code>	This annotation contains the URL to an icon image.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>criticality</code>	<code>ElementRef</code>	<p>This annotation references to another element that has the values 0, 1, 2, or 3.</p> <ul style="list-style-type: none"> • <code>Neutral</code>: 0 • <code>Negative</code>: 1 • <code>Critical</code>: 2 • <code>Positive</code>: 3 	
<code>UI.connectedFields</code>	<code>criticalityRepresentation</code>	<code>String(12)</code> enum	<ul style="list-style-type: none"> • <code>WITHOUT_ICON</code> • <code>WITH_ICON</code> 	<p>Default:</p> <p><code>WITHOUT_ICON</code></p>
<code>UI.connectedFields</code>	<code>dataAction</code>	<code>String(120)</code>		<p>This annotation can be used if the line item type is <code>FOR_ACTION</code>. The element references the technical name of an action of the Business Object Processing Framework (BOPF), for example. In this case, the string pattern is <code>BOPF:<technical name of action in BOPF></code>.</p>
<code>UI.connectedFields</code>	<code>requiresContext</code>	<code>Boolean</code>		<p>Default: True</p>

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>invocationGrouping</code>	String(12) enum	#optional	<p>i Note</p> <p>This annotation needs to be specified if you use <code>UI.lineItem.type</code> of type <code>FOR_ACTION</code>.</p> <ul style="list-style-type: none"> • <code>ISOLATED</code>: Describes the error handling when an action cannot be executed on all selected instances: The action is executed on all instances except for instance on which the action cannot be executed. • <code>CHANGE_SET</code>: Describes the error handling when an action cannot be executed on all selected instances: If an action cannot be executed on one of the selected instances, the action is executed on none of the selected instances. <p>ISOLATED Example: A user selects five items in a list and wants to copy them. One item cannot be copied. This item will not be copied, the other four items are copied.</p> <p>CHANGE_SET Example: A user selects five items in a list and wants to copy them. One item cannot be copied. None of the selected items are copied.</p> <p>Default: ISOLATED</p>
<code>UI.connectedFields</code>	<code>semanticObjectAction</code>	String(120)		This annotation refers to the name of an action on the semantic object. The semantic object is taken from <code>@Consumption.semanticObject</code> or derived via an association from the defining view.

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>value</code>	<code>ElementRef</code>	<p>For type AS_ADDRESS:</p> <ul style="list-style-type: none"> Value element must not be used when a structural element is annotated. Use instead <code>@com.sap.vocabularies.Communication.v1.Address</code> (or a shorter alias-qualified name) as value. Value element must be used when an element of an associated CDS view is annotated. A value of '.' refers to <code>@Semantics.address</code> on the view that is directly associated. If you want to reference <code>@Semantics.address</code> on a view that is indirectly associated, use a path starting with a dot as value. <p>All other types:</p> <ul style="list-style-type: none"> Value element must not be used when an element is annotated, in this case the annotated element is the value. Value element must be used when an association is annotated. The value is a path to an element of the associated view. 	This annotation refers to a value.
<code>UI.connectedFields</code>	<code>valueQualifier</code>	<code>String (120)</code>		

Parent Annotation	Annotation	Type	Value	Description
<code>UI.connectedFields</code>	<code>targetElement</code>	<code>ElementRef</code>		<p>This annotation represents the path to an element of an associated CDS view. The path is converted to an OData NavigationPropertyPath. Using this annotation, you can link from the header part of an object view floorplan to a target element. You need to specify <code>UI.lineItem.targetElement</code> when you use the annotation <code>UI.lineItem.type</code> of type <code>WITH_NAVIGATION_PATH</code>.</p> <p>You might, for example, provide background information to an item that is opened on the object view floorplan.</p>
<code>UI.connectedFields</code>	<code>url</code>	<code>ElementRef</code>		<p>⚠ Caution</p> <p>You need to specify <code>UI.fieldGroup.url</code> when you use the annotation <code>UI.connectedFields..type</code> of type <code>WITH_URL</code>.</p> <p>This annotation represents the path to a structural element that contains a navigation URL.</p>

Example

8.2 API Documentation

The ABAP RESTful Programming Model provides components and interfaces to implement the REST contract. This allows the ABAP developer to implement specific use cases that are not executed automatically.

[Unmanaged BO Contract \[page 718\]](#)

[Unmanaged Query API \[page 741\]](#)

8.2.1 Unmanaged BO Contract

For the implementation type **unmanaged**, application developers must implement essential components of the REST contract themselves. For this, all desired operations (create, update, delete, or any application-specific actions) must be specified in the corresponding behavior definition artifact by using the [Behavior Definition Language \(BDL\) \[page 805\]](#) before they are implemented with ABAP.

This implementation is carried out in a special type of class pool, the behavior pool, which refers to the behavior definition. The global class is defined with the following syntax:

```
CLASS ClassName DEFINITION
PUBLIC ABSTRACT FINAL
FOR BEHAVIOR OF BehaviorDefinition.
ENDCLASS.
CLASS ClassName IMPLEMENTATION.
ENDCLASS.
```

The concrete implementation of the business logic is based on the ABAP language and the Business Object Behavior API.

The implementation tasks are roughly divided into an **interaction phase** and a **save sequence**. The interaction phase is represented by the local handler class and the save sequence by the local saver class.

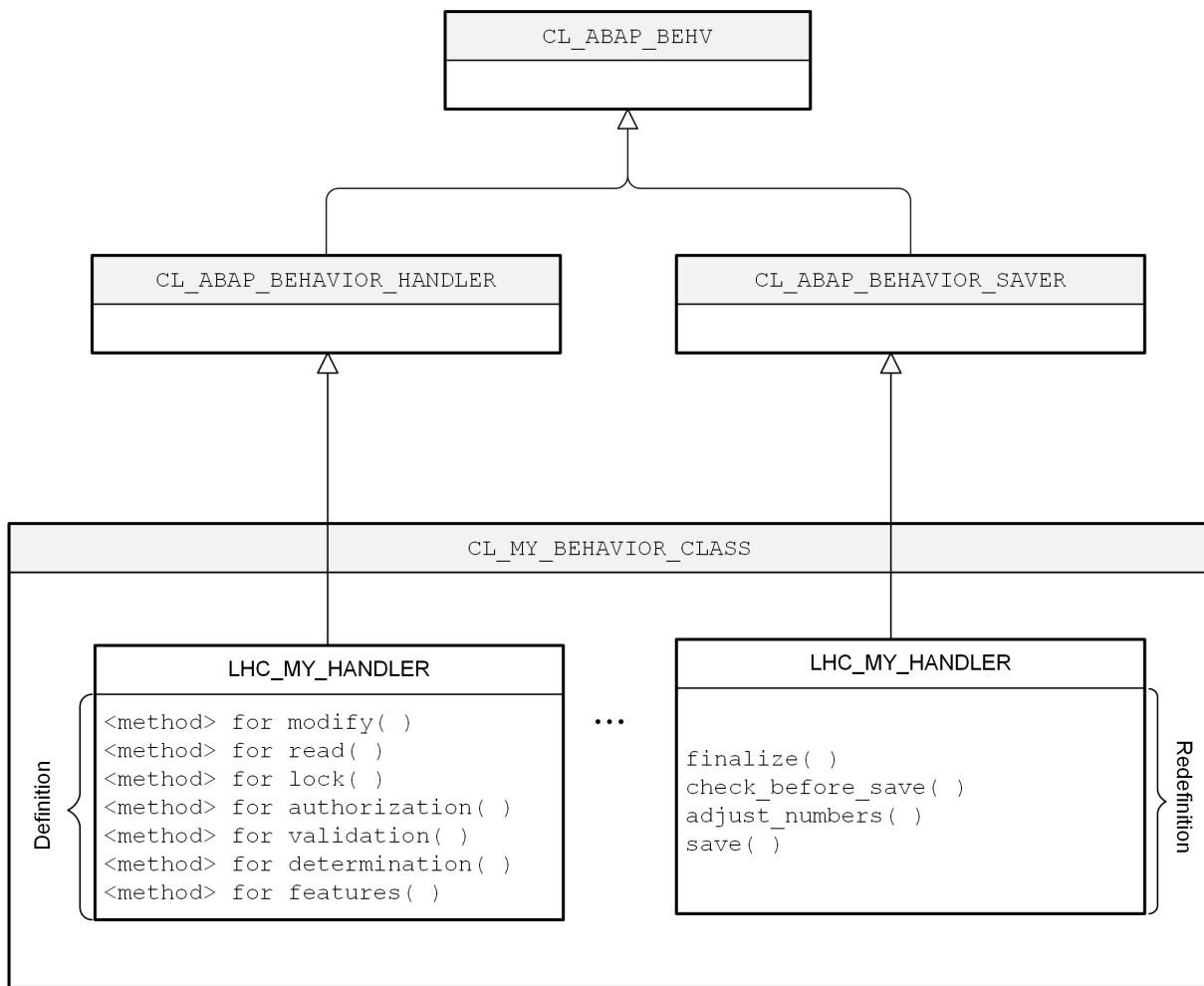
→ Remember

There are some specific rules for assigning names to local classes in a behavior pool. Both handler classes and saver classes are recognized by derivation from the respective system base class. The names LCL_HANDLER and LCL_SAVER are suggested by ADT when you create the class pool, but can be changed. We recommend applying naming conventions for behavior pools and local handler and saver classes corresponding to [Naming Conventions for Development Objects \[page 780\]](#).

At the top of the class hierarchy for the BO API is the class CL_ABAP_BEHV. This class is the foundation class for the handler and the saver class. It defines some fundamental data types to be used in the behavior processing (such as field names in derived type structures) and also provides message creation methods.

The classes that derive from this base class are:

- CL_ABAP_BEHAVIOR_HANDLER – This class is the base class for the handler.
- CL_ABAP_BEHAVIOR_SAVER – This class is the base class for the saver. It specifies the signature of all methods used to implement the save sequence of a business object provider.



Hierarchy Tree of the BO Behavior API

Detailed Information

- Implementing [Handler Classes](#) [page 719]
- Implementing [Saver Classes](#) [page 730]
- Using [Implicit Returning Parameters](#) [page 738]
- Declaration of Derived Data Types [page 734]

8.2.1.1 Handler Classes

To implement the behavior specified in the behavior definition, a special global ABAP class, the behavior pool is used. This global class is implicitly defined as `ABSTRACT` and `FINAL`. So, the behavior implementation cannot be found from outside the BO. A behavior pool can have static methods, `CLASS-DATA`, `CONSTANTS` and `TYPES`. The application may place common or even public aspects of its implementation here.

The real substance of a behavior pool is located in [Local Types](#). Here you can define two types of special local classes: handler classes for the operations within the interaction phase and saver classes for the operations

within the save sequence. These classes can be instantiated or invoked only by the [ABAP VM \[page 804\]](#). (All local class source code within one global class is stored within one include, the CCIMP include.)

Within the global behavior pool one or multiple local handler classes are defined. Each such local class inherits from the base class CL_ABAP_BEHAVIOR_HANDLER. The signature of the handler methods FOR MODIFY, FOR FEATURES, FOR LOCK, and FOR READ are typed based on the entity that is defined by the keyword FOR [OPERATION] entity. If there is an alias defined in the behavior definition, the alias has to be used.

Syntax: Definition of the Local Handler Class

```
CLASS lcl_handler DEFINITION INHERITING FROM cl_abap_behavior_handler.  
PRIVATE SECTION.  
/* FOR MODIFY method */  
METHODS modify_method FOR MODIFY  
[IMPORTING]  
    create_import_parameter      FOR CREATE entity  
    update_import_parameter     FOR UPDATE entity  
    delete_import_parameter    FOR DELETE entity  
    action_import_parameter    FOR ACTION entity~action_name  
    [REQUEST requested-fields]  
    [RESULT action_export_parameter]  
    create_ba_import_parameter FOR CREATE entity\association.  
/* FOR FEATURES method declaration */  
METHODS feature_ctrl_method FOR FEATURES  
    [IMPORTING] keys REQUEST requested_features FOR entity  
    RESULT result_parameter.  
/* FOR LOCK method declaration */  
METHODS lock_method FOR LOCK  
    [IMPORTING] lock_import_parameter FOR LOCK entity.  
/* FOR READ method declaration */  
METHODS read_method FOR READ  
    [IMPORTING] read_import_parameter FOR READ entity  
    RESULT read_export_parameter.  
/* FOR READ by association method */  
METHODS read_by_assoc_method_name FOR READ  
    [IMPORTING] read_ba_import_parameter FOR READ entity\association  
    FULL full_read_import_parameter  
    RESULT read_result_parameter  
    LINK read_link_parameter.  
ENDCLASS.
```

Method Summary

Method	Description
<method> FOR MODIFY	Handles all changing operations (create, update, delete, and specific actions as they are specified in the behavior definition) of an entity
<method> FOR FEATURES	Implements the dynamic feature control of entities

Method	Description
<method> FOR LOCK	Implements the locking of entities corresponding to the lock properties in the behavior definition
<method> FOR READ	Handles the processing of reading requests

Method Details

- <method> FOR MODIFY [page 721]
- <method> FOR FEATURES [page 729]
- <method> FOR LOCK [page 96]
- <method> FOR READ [page 726]

8.2.1.1.1 <method> FOR MODIFY

Handles all changing operations of an entity.

The FOR MODIFY method implements the standard operations create, update, delete, and application-specific actions, as they are specified in the behavior definition.

→ Tip

The FOR MODIFY method can handle multiple entities (root, item, sub item) and multiple operations during one processing step. In some cases, it might be useful to split the handler implementation into separate methods. Then, multiple behavior handlers, that is, multiple local behavior classes within one global behavior pool or even in multiple global behavior pools, can be defined.

Declaration of <method> FOR MODIFY

The declaration of the <method> FOR MODIFY expresses what changing operations this method is responsible for. In extreme cases, this is the total number of all changing operations that are possible according to the behavior definition.

Each individual specification within the declaration of `modify_method FOR MODIFY` consists of a combination of an operation with an entity or an entity part. To refer to the entities, the alias given in behavior definition is used - if there is any.

Each operation type has an import parameter `<operation>_import_parameter` for the incoming instance data and. Its name is freely selectable. The method includes an export parameter `action_export_parameter` if the operation type expects one. Action, for example, can have export parameters for their results.

The import parameters for CREATE, UPDATE and CREATE by association include the control structure %control to identify which fields have been filled by the caller.

You can declare all operations in a single <method> FOR MODIFY:

```
METHODS modify_method FOR MODIFY
  [IMPORTING]
    create_import_parameter      FOR CREATE entity
    update_import_parameter     FOR UPDATE entity
    delete_import_parameter     FOR DELETE entity
    action_import_parameter     FOR ACTION entity~action_name
      [REQUEST requested-fields]
      [RESULT action_export_parameter]
    create_ba_import_parameter   FOR CREATE entity\association.
```

You can also declare a <method> FOR MODIFY for each operation. In many cases, it can be beneficial to implement the individual MODIFY operations in separate methods. This may be particularly the case if the implementations for the respective operations are more extensive.

```
METHODS:
  create_entity_method FOR MODIFY
    [IMPORTING] create_import_parameter      FOR CREATE entity,
    update_entity_method FOR MODIFY
      [IMPORTING] update_import_parameter     FOR UPDATE entity,
    delete_entity_method FOR MODIFY
      [IMPORTING] delete_import_parameter     FOR DELETE entity,
    action_method FOR MODIFY
      [IMPORTING] action_import_parameter     FOR ACTION entity~action_name
      RESULT      action_export_parameter,
    create_by_association FOR MODIFY
      [IMPORTING] create_ba_import_parameter   FOR CREATE entity\association.
```

For the sake of better readability, the keyword IMPORTING can be specified before the first import parameter.

The parameters can also be explicitly declared as REFERENCE (...); However, the declaration as VALUE (...) is not allowed and therefore the importing parameters cannot be changed in the method.

i Note

The data types with which the parameters are implicitly provided by the ABAP compiler are **derived types** resulting from the behavior definition. They usually contain at least the instance key according to the CDS definition, or even the full row type, as well as other components that result from the model (action parameter) or other features of the BO, such as %pid in case of late numbering. For more information, see [Declaration of Derived Data Types \[page 734\]](#).

The method FOR MODIFY has three **implicit changing parameters** failed, mapped, and reported. These parameters can (but do not need to) be explicitly declared (developers may find this explicit declaration helpful), like this:

```
METHODS method_name FOR MODIFY
  [IMPORTING]
    create_import_parameter      FOR CREATE entity
    ...
  CHANGING  failed    TYPE DATA
            mapped    TYPE DATA
            reported  TYPE DATA.
```

Since the derived types also come here into play, you cannot explicitly write them down. The ABAP compiler accepts the generic type DATA and replaces it with the respective derived types resulting from the behavior definition.

Each of FAILED, MAPPED, REPORTED is a structure type with one component per entity from the behavior definition (that is, per entity in a business object). The names of the components are the aliases defined in the behavior definition or else the original entity names.

All parameters and components of these structures are tables to allow mass processing. Together with the bundling of multiple operations in a method, it is possible to implement large modification requests in a single FOR MODIFY method call.

Implementation of <method> FOR MODIFY

When is the FOR MODIFY Method Called?

The FOR MODIFY method is called when the BO framework processes a change request that contains at least one of the operations defined in the method FOR MODIFY.

The FOR MODIFY method can determine which operations are specifically given, for example, in this way:

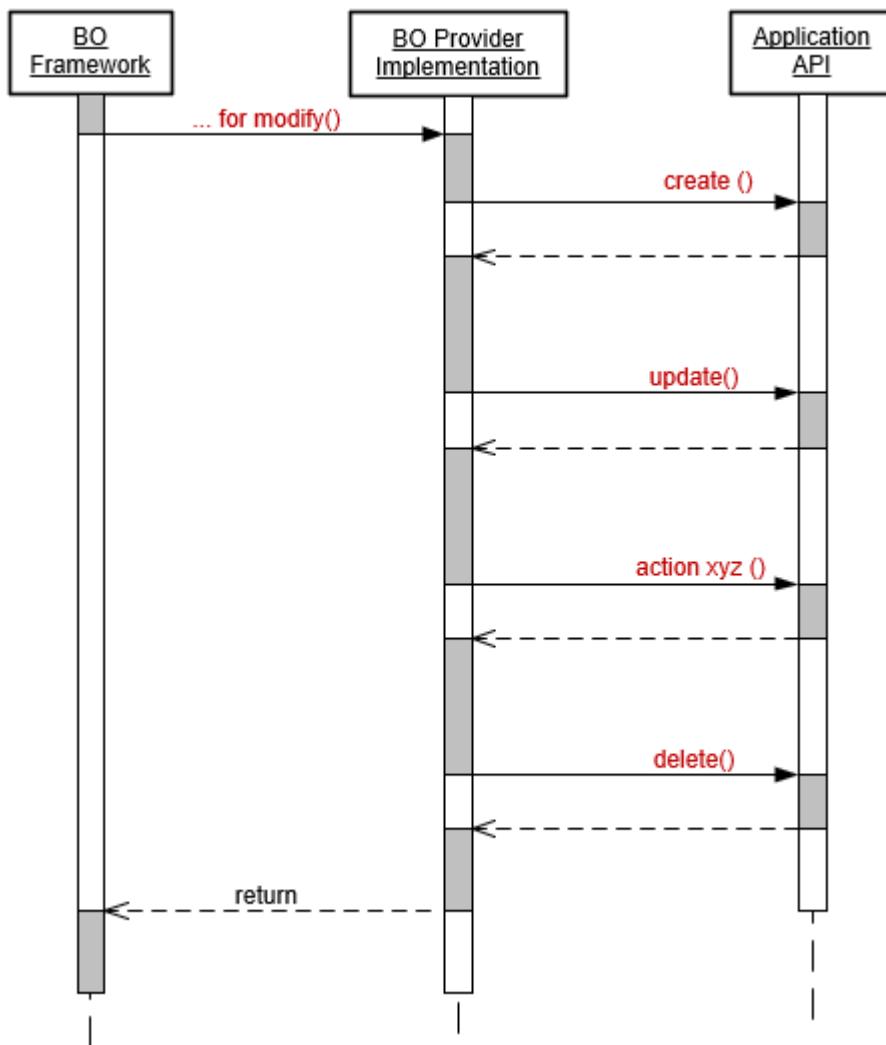
```
... parameter IS [NOT] INITIAL.
```

• Example

```
IF create_import_parameter IS NOT INITIAL.  
  ... " code for creating entities  
ENDIF.
```

Sequence of Processing of Individual Operations

The BO framework does not specify an order for the processing of individual operations within a FOR MODIFY call. It is therefore assumed that the application layer processes all the individual operations that are passed in a meaningful order for them. For example, it is usually useful to process create operations before update operations.



Sequence of Operations Processing within a FOR MODIFY Method call – An Example

Retrieving Results from Operation Processing

To get the output of an action call with a defined RESULT, the named export parameter `action_export_parameter` must be filled.

There are no explicit return parameters to be filled for all other operations. However, the three returning structures `failed`, `reported`, and `mapped` must be filled when the corresponding events happen. Their construction results in a fairly readable pattern, for example, to report failed instances or to store messages for instances:

```

APPEND ... TO failed-Item.
APPEND ... TO reported-Root.

```

All derived types also contain components that do not originate from the line type of the entity and begin with the character % to avoid naming conflicts with original components. For example, the row type of a `failed` table contains a component `%fail` to store the symptom for a failed instance; Also, an include structure `%key` that summarizes the primary key fields of the entity. `%key` is part of almost all derived types, including operation parameters. An overview of all derived types is given in

Thus, the above pattern can be as follows:

```
APPEND VALUE #( %KEY = <item>-%KEY %FAIL = IF_ABAP_BEHV=>CAUSE-... %CID = ... )
TO failed-Item.
```

Related Information

[Implicit Returning Parameters \[page 738\]](#)

8.2.1.1.2

<method> FOR LOCK

Implements the lock for entities in accordance with the lock properties specified in the behavior definition.

The `FOR LOCK` method is automatically called by the [orchestration framework \[page 817\]](#) framework before a changing (`MODIFY`) operation such as `update` is called.

Declaration of

<method> FOR LOCK

In the behavior definition, you can determine which entities support direct locking by defining them as `lock master`.

i Note

The definition of `lock master` is currently only supported for root nodes of business objects.

In addition, you can define entities as `lock dependent`. This status can be assigned to entities that depend on the locking status of a parent or root entity. The specification of `lock dependent` contains the association by which the runtime automatically determines the corresponding `lock master` whose method `FOR LOCK` is executed when change requests for the dependent entities occur.

The declaration of the predefined `LOCK` method in the behavior definition is the following:

```
METHODS lock method FOR LOCK
  [IMPORTING] lock_import_parameter FOR LOCK entity.
```

The keyword `IMPORTING` can be specified before the import parameter. The name of the import parameter `lock_import_parameter` can be freely selected.

The placeholder `entity` refers to the name of the entity (such as a CDS view) or to the alias defined in the behavior definition.

Import Parameters

The row type of the import table provides the following data:

- ID fields

All elements that are specified as a key in the related CDS view.

i Note

The compiler-generated structures %CID, %CID_REF, and %PID are not relevant in the context of locking since locking only affects persisted (non-transient) instances.

Changing Parameters

The `LOCK` method also provides the implicit CHANGING parameters `failed` and `reported`.

- The `failed` parameter is used to log the causes when a lock fails.
- The `reported` parameter is used to store messages about the fail cause.

You have the option of explicitly declaring these parameters in the `LOCK` method as follows:

```
METHODS lock method FOR LOCK
  IMPORTING lock_import_parameter FOR LOCK entity
  CHANGING failed TYPE DATA
    reported TYPE DATA.
```

Implementation of `<method> FOR LOCK`

The RAP lock mechanism requires the instantiation of a lock object. A lock object is an ABAP dictionary object, with which you can enqueue and dequeue locking request. For tooling information about lock objects, see .

The `enqueue` method of the lock object writes an entry in the global lock tables and locks the required entity instances.

An example on how to implement the `<method> FOR LOCK` is given in [Implementing the LOCK Operation \[page 330\]](#).

Related Information

[Implicit Returning Parameters \[page 738\]](#)

8.2.1.1.3 `<method> FOR READ`

Implements a handler for processing reading requests.

The `FOR READ` method is used to return the data from the application buffer. If the buffer is empty, the data is read from the database (which typically populates the application buffer).

There are two options to read data from the application buffer:

- Direct `READ`, see [Declaration of `<method> FOR READ` \[page 727\]](#).

- READ by association, see [Declaration of <method> FOR READ By Association \[page 728\]](#).

Declaration of <method> FOR READ

Similar to <method> FOR MODIFY, the handler <method> FOR READ is also implemented to handle mass requests. It is also designed to bundle multiple operations.

```
METHODS method_name FOR READ
  [IMPORTING] read_import_parameter FOR READ entity
  RESULT read_result_parameter.
```

Again, for the sake of better readability, the keyword IMPORTING can be specified before the import parameter. The name of the import parameter `read_import_parameter` can be freely selected. It imports the key(s) of the instance entity to be read and indicates which elements are requested.

The placeholder `entity` refers to the name of the entity (such as a CDS view) that you want to read from or to the alias defined in the behavior definition.

The parameter `RESULT` is a changing parameter. Its name can be freely selected.

Import Parameters

The row type of the import table `read_import_parameter` provides the following:

- ID fields
All elements that are specified as a key in the related CDS view.
- %CONTROL
The control structure reflects which elements are requested by the consumer.

Exporting Parameters

- `read_result_parameter`
Returns the successfully read data.
The row type of this table provides all elements that are specified in the element list of the entity that is read. Only the requested elements, which are indicated in the `%control` structure, must be filled.

Changing Parameters

In addition to the explicitly declared return parameter, the READ method also provides the implicit CHANGING parameters `failed`, `mapped` and `reported`.

- The `failed` parameter is used to log the entries that could not be read. You can specify the fail cause for the READ, for example `not_found`.
- The `mapped` parameter must not be filled in READ implementations.
- The `reported` parameter must not be filled in READ implementations.

For more information about the implicitly declared parameters, see [Implicit Returning Parameters \[page 738\]](#).

Declaration of <method> FOR READ By Association

The syntax for the READ by association is similar to the one for the direct READ.

```
METHODS
  method_name    FOR READ
    [IMPORTING] read_ba_import_parameter FOR READ entity\_association
    FULL full_read_import_parameter
    RESULT read_result_parameter
    LINK read_link_parameter.
```

As for the other operation implementations, the keyword IMPORTING can optionally be specified explicitly. All parameter names can be freely selected.

The importing parameter `read_ba_import_parameter` imports the key(s) of the entity instance whose associated entity instance shall be read. In addition, it indicates which elements from the associated entity shall be read.

The placeholder `entity` refers to the name of the entity (such as a CDS view) or to the alias defined in the behavior definition, which is the source of the association.

The placeholder `association` refers to the association along which you want to read data, for example `_booking` if you want to read all bookings associated to one travel instance.

The parameter `full_read_import_parameter` indicates whether the `RESULT` parameter must be filled or if only the `LINK` parameter must be filled. It has a boolean value.

The parameter `RESULT` is an exporting parameter. It returns the requested elements that are indicated in the importing parameter if the `FULL` parameter is set.

The parameter `LINK` is also an exporting parameter. It returns the key elements of the source and target entities no matter if the `FULL` parameter is set.

Import Parameters

- The row type of the import table `read_ba_import_parameter` provides the following data:
 - ID fields
All elements that are specified as a key in the related CDS view.
 - %CONTROL
The control structure reflects which elements of the associated entity are requested by the consumer.
- The type of the import parameter `full_read_import_parameter` is a character with boolean value.

Exporting Parameters

- `read_ba_import_parameter`: Returns the successfully read data from the associated entity if the `FULL` parameter is set.
The row type of this table provides all elements that are specified in the element list of the entity that is read. Only the requested elements, which are indicated in the `%control` structure, must be filled.
- `read_link_parameter`: Returns the source and target key of the successfully read entity instances.

For more information about the implicitly declared parameters, see [Implicit Returning Parameters \[page 738\]](#).

Changing Parameters

In addition to the explicitly declared parameters, the method for READ by association also provides the implicit CHANGING parameters `failed`, `mapped`, and `reported`.

- `read_ba_import_parameter`: Returns the successfully read data from the associated entity if the `FULL` parameter is set.
The row type of this table provides all elements that are specified in the element list of the entity that is read. Only the requested elements, which are indicated in the `%control` structure, must be filled.
- `read_link_parameter`: Returns the source and target key of the successfully read entity instances.
- The `failed` parameter is used to log the entries that could not be read. You can specify the fail cause for the READ, for example `not_found`.
- The `mapped` parameter must not be filled in `READ` implementations.
- The `reported` parameter must not be filled in `READ` implementations.

For more information about the implicitly declared parameters, see [Implicit Returning Parameters \[page 738\]](#).

8.2.1.1.4

`<method> FOR FEATURES`

This method implements the dynamic feature control for entities.

Dynamic feature control can be used for the standard operations `update` and `delete`, for actions and on field level. Depending on the feature conditions is the operation executable or not. The `<method> FOR FEATURES` is called by the orchestration framework for every operation or field that is dynamically controlled.

Declaration of `<method> FOR FEATURES`

The operations or fields that are dynamically controlled are defined in the behavior definition with `features`:
instance.

The dynamic feature control for an entity is implemented in a handler class using the method `feature_ctrl_method`. The signature of this handler method is defined by the keyword `FOR FEATURES`, followed by the input parameters `keys` and the `requested_features` of the entity.

```
METHODS feature_ctrl_method FOR FEATURES
  [IMPORTING] keys REQUEST requested_features FOR entity
  RESULT result.
```

Again, for the sake of better readability, the keyword `IMPORTING` can be specified before the import parameter.

i Note

The name of the `<method> FOR FEATURES` can be freely chosen. Often `get_features` is used as method name.

Import Parameters

- `keys`
The table type of `keys` includes all elements that are specified as a key for the related entity.
- `requested_features`

The structure type of `requested_features` reflects which elements (fields, standard operations and actions) of the entity are requested for dynamic feature control by the consumer.

Export Parameters

The export parameter `result` is used to return the feature control values. It includes, besides the key fields, all the fields of the entity, standard operations and actions for which the features control was defined in the behavior definition.

Export Parameters

In addition to the explicitly declared export parameter, the `FOR FEATURE` method also provides the implicit `CHANGING` parameters `failed` and `reported`.

Example

For a fully implemented example of dynamic feature control, see [Feature control for action set_status_booked \[page 469\]](#)

Related Information

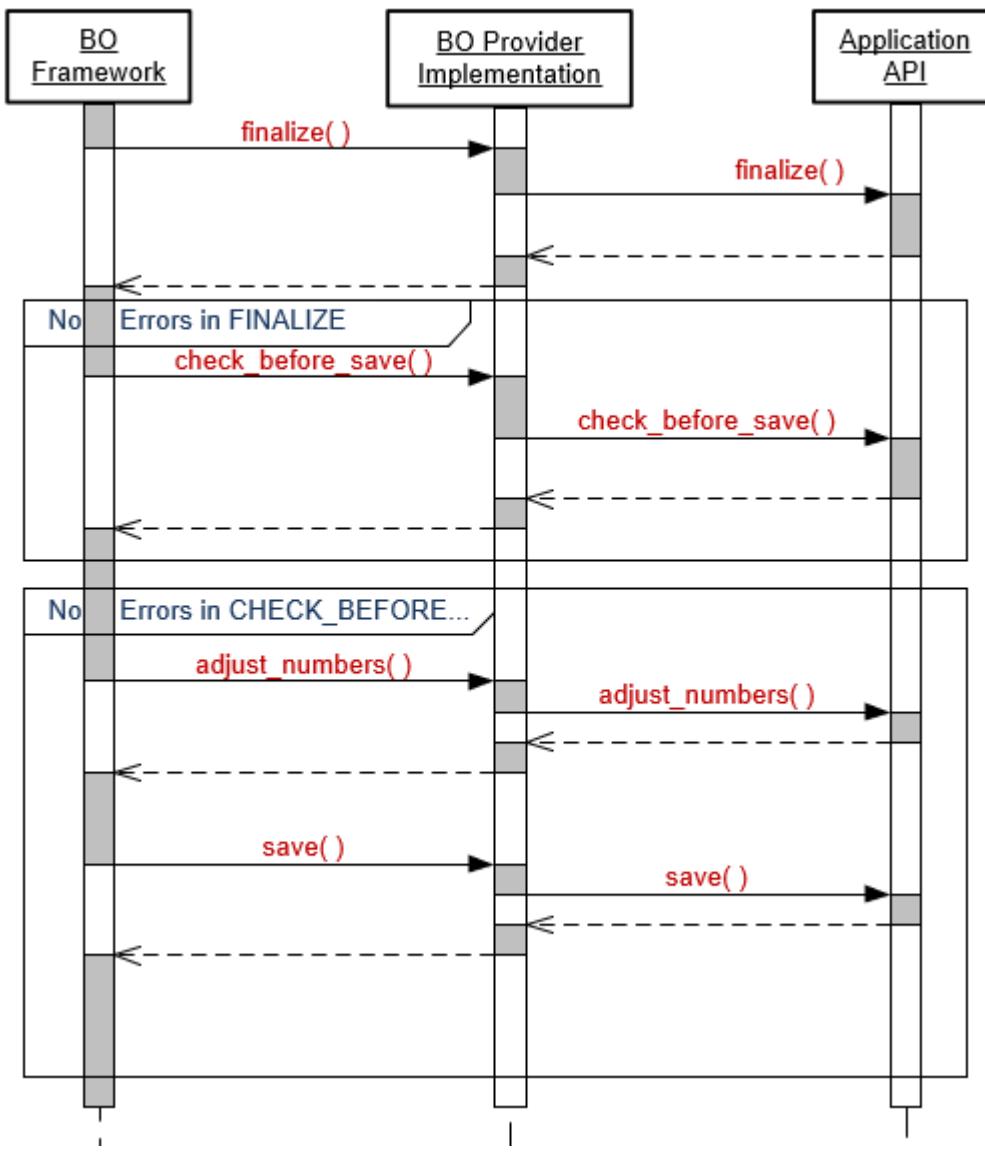
[Implicit Returning Parameters \[page 738\]](#)

8.2.1.2 Saver Classes

Save Sequence

The save sequence is called for each business object after at least one successful modification was performed using the BO behavior APIs in the current LUW.

As depicted in the figure below, the save sequence starts with `finalize()` performing the final calculations before data can be persisted. If the subsequent `check_before_save()` call is positive for all transactional changes, the **point-of-no-return** is reached. From now on, a successful `save()` is guaranteed by all involved BOs. After the point-of-no-return, the `adjust_numbers()` call can occur to take care of [late numbering \[page 814\]](#). The `save()` call persists all BO instance data from the transactional buffer in the database.



Save Sequence and Transactional Method Processing

All transactional methods are implemented in the local saver class that is a part of a global behavior pool. Each local saver class of this type inherits from the base class `CL_ABAP_BEHAVIOR_SAVER`. This superclass provides the transactional methods that need to be redefined in the local saver class.

Syntax: Definition of the Transactional Methods

```

CLASS lcl_saver DEFINITION INHERITING FROM cl_abap_behavior_saver.
  PROTECTED SECTION.
    METHODS finalize          REDEFINITION.
    METHODS check_before_save REDEFINITION.
    METHODS adjust_numbers   REDEFINITION.
    METHODS save             REDEFINITION.
  ENDCLASS.

```

Method Summary

Method	Description
FINALIZE	Finalizes data changes before they can be persisted on the database.
CHECK_BEFORE_SAVE	Checks the application buffer for consistency .
ADJUST_NUMBERS	Implements late numbering.
SAVE	Saves the data from the transactional buffer to the database.

Method Details

- Method FINALIZE [page 732]
- Method CHECK BEFORE SAVE [page 733]
- Method ADJUST NUMBERS [page 733]
- Method SAVE [page 734]

8.2.1.2.1 Method FINALIZE

Finalizes data changes before they can be persisted on the database.

The implementation of `finalize()` is optional.

You can use this method to perform final calculations with determinations before data is persisted on the database with the `save()` call.

Example

Let us assume that a `SalesOrder` triggers the calculation of the `Pricing`, which is quite complex and time consuming to be called for each modification during the consumer-BO interaction (to be precise, pricing is called by default for each `modify()`, but the customer can configure it to be executed only in `finalize()` to optimize the performance).

Changing Parameters

- MAPPED
- FAILED
The parameter `FAILED` is filled to log the entities for which `finalize` went wrong.
- REPORTED

You can fill the parameter `REPORTED` to return messages in case of failure.

More on this: [Implicit Returning Parameters \[page 738\]](#)

8.2.1.2.2 Method `CHECK_BEFORE_SAVE`

Checks the application buffer for consistency.

The implementation of `check_before_save()` is optional.

To enable a successful `save()`, the BO runtime must provide feedback in `check_before_save()` based on all transactional changes. This is done by validations that are called within the `check_before_save` method.

If the `check_before_save()` of all involved BOs returns positive feedback, the **point-of-no-return** is reached. From now on, a successful `save()` is guaranteed for all involved BOs and the data is persisted.

If, on the other hand, errors are reported in the changing parameter `FAILED`, the save chain is canceled.

Changing Parameters

- `MAPPED`
- `FAILED`
The parameter `FAILED` is filled to log the entities for which there is no positive feedback.
- `REPORTED`
You can fill the parameter `REPORTED` to return messages in case of failure.

More on this: [Implicit Returning Parameters \[page 738\]](#)

8.2.1.2.3 Method `ADJUST_NUMBERS`

Implements late numbering.

The implementation of `adjust_numbers()` is only required if late numbering is modeled in the behavior definition.

Late numbering is a common concept for drawing gap-free numbers. In some cases, it can be business critical that identifier numbers are gap-free, for example invoice numbers. The third phase of the save sequence is implemented in `adjust_numbers()`. The output is a link table which maps %PIDs to the related drawn numbers. These final IDs are provided by means of the `MAPPED` exporting parameter so that temporary numbers can be exchanged. The implementation of this method assigns the final keys for the remaining content IDs.

Changing Parameter

- `MAPPED`
- `REPORTED`
Messages can be reported via the implicit returning parameter `REPORTED`. As consumer errors must not appear after `CHECK_BEFORE_SAVE`, `REPORTED` should only contain success or information messages, such as *Material stock is low*.

i Note

- The method must not fail and thus does not return any failed keys since the exchange of temporary IDs takes place after the point-of-no-return. If the application needs to stop the transaction, it can only produce a short dump.

More on this: [Implicit Returning Parameters \[page 738\]](#)

8.2.1.2.4 Method SAVE

Saves the data from the transactional buffer to the database.

The implementation of `save()` is mandatory.

The actual `save()` implementation gets access to the link table of the content IDs (`%CID`) and their numbers. Often these numbers are used as foreign keys, so that they need to be replaced before the data is persisted.

After the data is persisted, it is expected that the transactional buffer is cleared, since the same ABAP session might be used for more than one [LUW \[page 815\]](#) and any remaining changes in the transactional buffer could lead to inconsistencies. Of course, the persisted transactional changes can be transferred to a read cache, once they are successfully saved.

Changing Parameter

-  REPORTED

Messages can be reported via the implicit returning parameter `REPORTED`. As consumer errors must not appear after `CHECK_BEFORE_SAVE`, `REPORTED` should only contain success or information messages, such as *Booking has been saved*.

i Note

- The method must not fail and thus does not return any failed keys since the exchange of temporary IDs takes place after the point-of-no-return. If the application needs to stop the transaction, it can only produce a short dump.

More on this: [Implicit Returning Parameters \[page 738\]](#)

8.2.1.3 Declaration of Derived Data Types

What is a Derived Data Type?

For the type-safe parameterization of the BO provider code, the ABAP compiler derives data types from the involved CDS views and the behavior definition. These are called derived types because they are implicitly derived by the compiler from CDS entity types and their behavior definition. Derived types usually contain at

least the instance key according to the CDS definition, or even the full row type, as well as other components that result from the model (such as action parameters).

When implementing a BO provider, you can use specific derived types in method signatures in the context of the behavior implementation. This means you have the option of creating both local and global derived data types by using a new syntax for declaring import or export parameters.

Type Declaration for Import Parameters

Each individual type declaration consists of a combination of an operation with an entity or an entity part, such as an action. To refer to the entities, the alias given in the behavior definition should be used (if one exists).

Note that the syntax of an action import parameter definition differs a little from the type definition of those of the standard operation-related parameters.

```
TYPES type_for_import_parameter TYPE TABLE FOR {OPERATION} entity_name.  
TYPES type_for_action_import TYPE TABLE FOR ACTION IMPORT  
entity_name~action_name.
```

{OPERATION} is one of the following:

- CREATE
- UPDATE
- DELETE
- LOCK
- READ IMPORT

The name of the import parameter type `type_for_import_parameter` can be freely selected, for example `it_create_travel` or `it_read_travel_id`.

The placeholder `entity_name` refers to the name of the entity (such as a CDS view) as it is defined in the behavior definition.

Type Declaration for Export Parameters

Similarly, local and global types can be defined for export parameters:

```
TYPES type_for_action_result_parameter TYPE TABLE FOR ACTION RESULT  
entity_name~action_name.
```

{RESULT_PARAMETER} is one of the following:

- MAPPED [LATE] - The mapped result parameters provide the consumer with ID mapping information. By default, the mapping information is already available in the interaction phase (early mapped). The `CID` is then mapped to the real key or to the `PID`. Using the addition `LATE`, you specify that the mapping information is only available in the save sequence. This plays a role when providing the late numbering (see also: [Method ADJUST_NUMBERS \[page 733\]](#)) where the `PID` is mapped to the real key.
- FAILED [LATE] - The failed parameters include information for identifying the data set where an error occurred. (Early) `FAILED` is provided during the interaction phase and contains the `CID` or the `KEY` to

indicate instances for which an operation failed. FAILED with the additional specification LATE is only provided during the save sequence and contains the PID or the KEY, but not the CID.

- REPORTED [LATE] - The reported parameters are used to return messages in case of failure. (Early) REPORTED is provided during the interaction phase and contains the CID or the KEY to indicate instances for which an operation failed. REPORTED with the additional specification LATE is only provided during the save sequence and contains the PID or the KEY, but not the CID.
- READ RESULT

Type Declaration for Changing Parameters

Similarly, local and global types can be defined for export parameters:

```
TYPES type_for_export_parameter TYPE TABLE FOR {RESULT_PARAMETER} entity_name.  
TYPES type_for_action_result_parameter TYPE TABLE FOR ACTION RESULT  
entity_name~action_name.
```

{RESULT_PARAMETER} is one of the following:

- MAPPED [LATE] - The mapped result parameters provide the consumer with ID mapping information. By default, the mapping information is already available in the interaction phase (early mapped). The CID is then mapped to the real key or to the PID. Using the addition LATE, you specify that the mapping information is only available in the save sequence. This plays a role when providing the late numbering (see also: [Method ADJUST_NUMBERS \[page 733\]](#)) where the PID is mapped to the real key.
- FAILED [LATE] - The failed parameters include information for identifying the data set where an error occurred. (Early) FAILED is provided during the interaction phase and contains the CID or the KEY to indicate instances for which an operation failed. FAILED with the additional specification LATE is only provided during the save sequence and contains the PID or the KEY, but not the CID.
- REPORTED [LATE] - The reported parameters are used to return messages in case of failure. (Early) REPORTED is provided during the interaction phase and contains the CID or the KEY to indicate instances for which an operation failed. REPORTED with the additional specification LATE is only provided during the save sequence and contains the PID or the KEY, but not the CID.
- READ RESULT

The name of the export parameter type type_for_export_parameter or type_for_action_result_parameter can be freely selected, for example et_create_travel or et_read_travel_id_out.

entity_name refers to the name of the entity (such as a CDS view) or to the alias as it is defined in the behavior definition.

Examples

```
TYPES:  
  et_travel_mapped          TYPE TABLE FOR MAPPED travel_root,  
  et_travel_mapped_late     TYPE TABLE FOR MAPPED LATE travel_root,  
  et_booking_failed         TYPE TABLE FOR FAILED booking_item,  
  et_booking_reported_late  TYPE TABLE FOR REPORTED LATE booking_item,  
  et_booking_read_out       TYPE TABLE FOR READ RESULT booking_item,  
  et_travel_set_booked_out  TYPE TABLE FOR ACTION RESULT travel_root~set_booked.
```

Derived Types for Key and the Data Structures

In addition to derived parameter types, you can also define specific derived data types for the IDs and data fields:

```
TYPES type_for_update TYPE TABLE FOR UPDATE entity_name.  
TYPES type_for_id      TYPE LINE OF type_for_update-%key.  
TYPES type_for_data    TYPE LINE OF type_for_update-%data.
```

Examples

```
TYPES:  
  it_booking_update      TYPE TABLE FOR UPDATE booking_item,  
  ltype_for_update       TYPE LINE OF it_booking_update,  
  ltype_for_key          TYPE ltyp_for_update-%key,  
  ltype_for_data         TYPE ltyp_for_update-%data.
```

Explicit Usage of Derived Types

For modularization, it may be necessary to declare variables with derived types, even outside the reserved handler methods. For this purpose, there is an explicit syntax that is supported in the statements TYPES, DATA, and CREATE DATA. The syntax always has the form ... TYPE TABLE FOR....

```
TYPES dtype TYPE TABLE FOR {OPERATION | RESULT_PARAMETER | ACTION ...}  
entity_name.  
DATA dtype TYPE TABLE FOR {OPERATION | RESULT_PARAMETER | ACTION ...}  
entity_name.  
CREATE DATA dref TYPE TABLE FOR {OPERATION | RESULT_PARAMETER | ACTION ...}  
entity_name.
```

In this case, only the entity_name, but not an alias, can be used to refer to the entity. This is because, unlike in the handler methods, no reference to a particular behavior definition is given.

Examples

```
TYPES      it_item_c      TYPE TABLE FOR CREATE SalesOrderItem.  
DATA       it_root_u      TYPE TABLE FOR UPDATE SalesOrder.  
CREATE DATA rt_item_ri   TYPE TABLE FOR READ IMPORT  
SalesOrderItem.  
DATA       et_item_rr     TYPE TABLE FOR READ RESULT SalesOrderItem  
TYPES      et_root_f     TYPE TABLE FOR FAILED SalesOrder.  
TYPES      it_root_l     TYPE TABLE FOR LOCK SalesOrder.  
TYPES      it_item_a_in   TYPE TABLE FOR ACTION IMPORT  
SalesOrderItem~add_supplement.  
DATA       et_root_a_out  TYPE TABLE FOR ACTION RESULT  
SalesOrder~set_final_status.
```

8.2.1.4 Implicit Returning Parameters

When implementing a BO contract, you make use of implicit returning parameters. These parameters do not have fixed data types and instead are assigned by the compiler with the types derived from behavior definition.

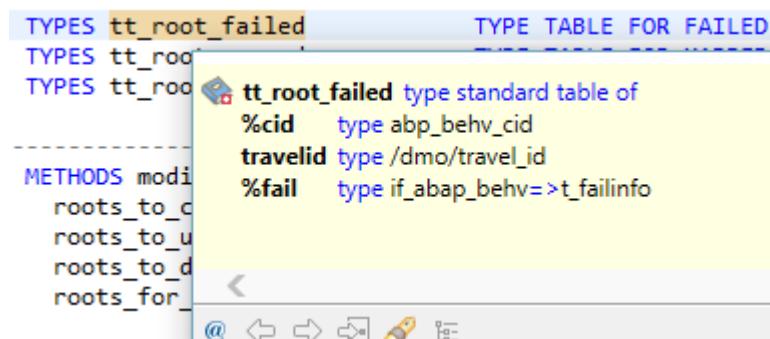
The implicit parameters can be declared explicitly as CHANGING parameters in the method signature of the handler classes by using the generic type DATA:

```
METHODS method_name FOR MODIFY | READ | LOCK
  [IMPORTING]
    <operation>_import_parameter      FOR <OPERATION> entity
    ...
  CHANGING failed      TYPE DATA
    [mapped      TYPE DATA] "Relevant for CREATE only
    reported     TYPE DATA.
```

The ABAP compiler replaces the type DATA with the respective **derived types** resulting from the concrete behavior definition.

Implicit Parameters

Parameter	Description
FAILED	This exporting parameter is defined as a nested table which contains one table for each entity defined in the behavior definition. The failed tables include information for identifying the data set where an error occurred: <ul style="list-style-type: none">• %CID and• ID of the relevant BO instance. The reason for the failure is specified by the predefined component: <ul style="list-style-type: none">• %FAIL, which stores the symptom of the failure.



Accessing Element Information for Failed Parameter Type (F2)

Parameter	Description
REPORTED	<p>This exporting parameter is used to return messages. It is defined as a nested table which contains one table for each entity defined in the behavior definition.</p> <p>The reported tables include data for instance-specific messages.</p> <p>The data set for which the message is relevant is identified by the following components:</p> <ul style="list-style-type: none"> • %CID • ID of the relevant instance • %MSG with an instance of the message interface <code>IF_ABAP_BEHV_MESSAGE</code> • %ELEMENT which refers to all elements of an entity. <p>Accessing Element Information for a Reported Parameter Type (F2)</p>
MAPPED	<p>This mapped parameter is defined as a nested table which contains one table for each entity defined in the behavior definition.</p> <p>The mapped parameters provide the consumer with ID mapping information. They include the information about which key values were created by the application for given content IDs. The BO runtime passes the created key values in any subsequent calls in the same request and in the response.</p> <p>The relevant data set is identified by the following components:</p> <ul style="list-style-type: none"> • %CID • %KEY

Components of Derived Data Types

All derived data types in the context of the ABAP RESTful programming model also contain components that do not originate from the row type of the entity and begin with the character % to avoid naming conflicts with original components. For example, the row type of a failed table contains a component %fail to store the

symptom for a failed instance and also an include structure %key that contains all primary key fields of the entity.

EXAMPLE: Usage of %... components in a failed parameter

```
APPEND #VALUE(%KEY = ... %FAIL = ...) TO failed-entity.
```

The following list provides you with a description of the most common %... components:

Component	Description
%CID	<p>The content ID %CID is a temporary primary key for an instance, as long as no primary key was created by the BO runtime.</p> <p>The content ID is always provided by the SADL [page 818] framework. It is only needed in case of internal numbering and/or late numbering. The content ID provides the reference between the related entity instances. A good example is a <i>DEEP INSERT</i> for multiple parent/child instances with internal numbering and/or late numbering. In this case, the references between the child and parent instances are established using the content ID %CID.</p>
%CID_REF	A reference to the content ID %CID.
%KEY	<p>Contains all key elements of an entity (CDS view).</p> <p>%key is part of almost all derived types, including trigger parameters in the <code>for modify()</code> method.</p>
%PID	<p>Defines the preliminary ID.</p> <p>It is only used if the application does not provide a temporary primary key. It is designed for use in draft data use cases (which are not yet supported in the context of the ABAP RESTful programming model).</p>

i Note

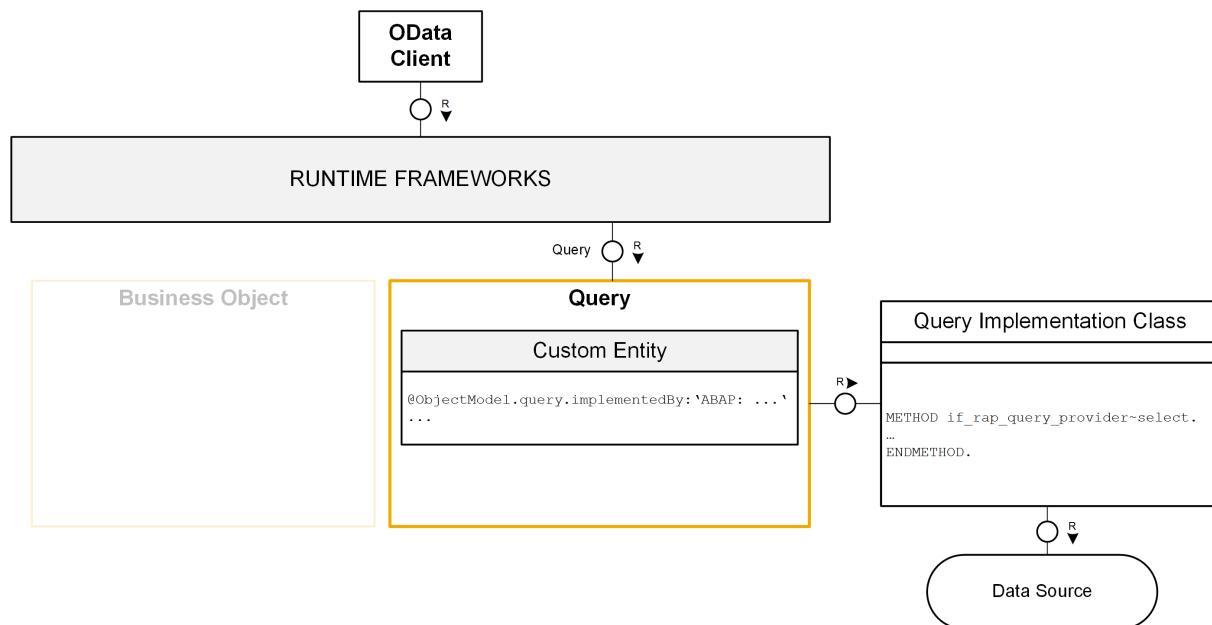
The preliminary ID is only available when `LATE NUMBERING` is defined in the behavior definition without the addition `IN PLACE`.

Component	Description
%CONTROL	<p>Reflects which elements are requested by the consumer.</p> <p>The fields of the %CONTROL structure provide information, depending on the operation, about which elements of the entity are supplied in the request (for CREATE and UPDATE operations) or which elements are requested in the read request (for READ operations).</p> <p>For each entity element, this control structure contains a flag which indicates whether the corresponding field was provided/requested by the consumer or not.</p> <p>The element names of the entity have the uniform type ABP_BEHV_FLAG.</p> <p>i Note</p> <p>The possible constants are defined in the basis handler class <code>if_abap_behv=>mk-<...></code>. For example, the elements that have the value <code>if_abap_behv=>mk-on</code> in the %CONTROL structure are used to handle delta updates within the UPDATE operation.</p>
%DATA	Contains all data elements of an entity (CDS view).
%FAIL	Stores the symptom for a failed data set (BO instance).
%MSG	<p>Provides an instance of the message interface IF_ABAP_BEHV_MESSAGE.</p> <p>→ Tip</p> <p>The component %MSG of type REF TO IF_ABAP_BEHV_MESSAGE includes IF_T100_DYN_MSG. If you do not need your own implementation of this interface, then you can benefit from the provided standard implementation by using the inherited methods <code>new_message()</code> or <code>new_message_with_text()</code>.</p>
%ELEMENT	Refers to all elements of an entity.
%PARAM	Holds the import/result type of actions.

8.2.2 Unmanaged Query API

In contrast to managed queries, in which a framework assumes the implementation tasks to select the requested data from the data source, the implementation of the unmanaged query must be done by the application developer. For this, all desired query capabilities (paging, filtering, sorting, ...) must be implemented in a query implementation class, which is referenced in a CDS custom entity.

The following diagram illustrates the runtime of an unmanaged query:



The query request is delegated to the query implementation class which must implement the `select` method of the interface `IF_RAP_QUERY_PROVIDER`. This API is described in following.

Interfaces

These interfaces define methods for the unmanaged query API.

- [Interface IF_RAP_QUERY_PROVIDER \[page 743\]](#)
 - [Interface IF_RAP_QUERY_REQUEST \[page 744\]](#)
 - [Interface IF_RAP_QUERY_FILTER \[page 749\]](#)
 - [Interface IF_RAP_QUERY_PAGING \[page 752\]](#)
 - [Interface IF_RAP_QUERY_AGGREGATION \[page 753\]](#)
 - [Interface IF_RAP_QUERY_RESPONSE \[page 755\]](#)

For more conceptual information about the unmanaged query, see [Query Runtime Implementation \[page 50\]](#).

For an example on how to implement an unmanaged query, see [Implementing an Unmanaged Query \[page 494\]](#).

For an example on how to implement the unmanaged query contract in a development scenario, see [Implementing the Query for Service Consumption \[page 386\]](#).



i Note

Before SAP Cloud Platform ABAP Environment Release 1908, `IF_A4C_RAP_QUERY_PROVIDER` and the related interfaces was the API to implement an unmanaged query. This API is deprecated as of 1908, but still available in ABAP Environment. However, it is recommended to use only the new interface `IF_RAP_QUERY_PROVIDER`.

The interfaces differ in some aspects, for example the handling of filter requests. The new interface offers some more methods to reflect the query requests in more detail, for example `get_aggregation` or `get_parameters`, which facilitates the implementation.

8.2.2.1 Interface `IF_RAP_QUERY_PROVIDER`

This interface defines a method that is used for requesting and responding to OData query requests in an unmanaged query.

Method `select`

The method `select` must be implemented in custom entity scenarios. It replaces the SQL-SELECT of a CDS view to retrieve and return data. The `select` method must be called by the query implementation class, which is referenced in the custom entity annotation `@ObjectModel.query.implementedBy`.

i Note

Before SAP Cloud Platform ABAP Environment 1908 the annotation `@QueryImplementedBy` was in use, which is deprecated as of 1908.

The `select` imports an interface instance for the request data and one for the response data:

Signature

```
METHODS select IMPORTING io_request TYPE REF TO if\_rap\_query\_request \[page 744\]
          io_response TYPE REF TO if\_rap\_query\_response \[page 755\]
          RAISING cx_rap_query_provider.
```

[Interface IF_RAP_QUERY_REQUEST \[page 744\]](#)

[Interface IF_RAP_QUERY_RESPONSE \[page 755\]](#)

Parameter

<code>IO_REQUEST</code>	Interface instance for gathering request information that are used as input for the <code>select</code> implementation. The request interface provides methods for implementing query options, like filtering or sorting.
<code>IO_RESPONSE</code>	Interface instance for the result output of the <code>select</code> implementation.

Exception

<code>CX_RAP_QUERY_PROVIDER</code>	Exception that can be raised if there is an error during the query execution.
------------------------------------	---

• Example

See [Implementing an Unmanaged Query \[page 494\]](#).

8.2.2.1.1 Interface `IF_RAP_QUERY_REQUEST`

The interface defines methods to parametrize a query request in an unmanaged query. It is used to handle OData query options for data retrieval.

Method `get_entity_id`

This method returns the CDS entity name of the requested entity set of an OData request in an unmanaged query.

With this method, you can ensure that the query implementation is only executed if the correct entity for this query implementation set is called.

Signature

```
METHODS get_entity_id RETURNING VALUE(rv_entity_id) TYPE string.
```

`rv_entity_id` CDS entity name of the requested entity set.

• Example

See [Returning Requested Entity in an Unmanaged Query \[page 499\]](#).

Method `is_data_requested`

This method returns a boolean value to indicate if data is requested.

i Note

If this method is used to indicate the request for data, the method [set_data \[page 755\]](#) must be called.

Signature

```
METHODS is_data_requested RETURNING VALUE(rv_is_requested) TYPE abap_bool.
```

Parameter

`rv_is_requested`

If data needs to be returned, the value is `abap_true`. If no data needs to be returned, the value is `abap_false`.

• Example

See [Requesting and Setting Data or Count in an Unmanaged Query \[page 500\]](#).

Method `is_total_numb_of_rec_requested`

This method returns a boolean value to indicate if the total number of records is requested. The total number of records is requested by the query option `$inlinecount` or a `$count` request.

i Note

If this method indicates the request for the total number of records, the total count needs to be returned by the method [set_total_number_of_records \[page 755\]](#).

Signature

```
METHODS is_total_numb_of_rec_requested RETURNING VALUE(rv_is_requested) TYPE abap_bool.
```

Parameter

`rv_is_requested`

If the total number of records needs to be returned the value is `abap_true`. If the total number of records is not requested the value is `abap_false`.

• Example

See [Requesting and Setting Data or Count in an Unmanaged Query \[page 500\]](#).

Method `get_filter`

This method returns a filter object. This filter object is an interface instance of `IF_RAP_QUERY_FILTER`. If a filter is requested, its methods return the filter information. Only records that match this filter condition must be returned or counted.

Signature

```
METHODS get_filter RETURNING VALUE(ro_filter) TYPE REF TO if\_rap\_query\_filter \[page 749\].
```

[Interface IF_RAP_QUERY_FILTER \[page 749\]](#)

Parameter

RO_FILTER	Contains the filter condition.
-----------	--------------------------------

• Example

See [Implementing Filtering in an Unmanaged Query \[page 501\]](#).

Method get_paging

This method returns an object with paging information. The paging object is an interface instance of `IF_RAP_QUERY_PAGING`. It limits the number of records to be returned as response data with offset and page size.

Signature

```
METHODS get_paging RETURNING VALUE(ro.paging) TYPE REF TO if\_rap\_query\_paging [page 752].
```

Interface [IF_RAP_QUERY_PAGING \[page 752\]](#)

Parameter

RO_PAGING	Contains the paging information.
-----------	----------------------------------

• Example

See [Implementing Paging in an Unmanaged Query \[page 507\]](#).

Method get_sort_elements

This method returns the sort order for the sort elements.

Signature

```
METHODS get_sort_elements RETURNING VALUE(rt_sort_elements) TYPE tt\_sort\_elements.
```

Parameter

rt_sort_elements

Contains the elements to be sorted with their sort direction. It is an ordered list to define the ranking order, the first element being the primary sort criteria. The table indicates the names of the sort element and the sort order with a boolean value in the column `descending`. The following table illustrates how the returning value looks like.

tt_sort_elements

ELEMENT_NAME	DESCENDING
<i>string</i>	<i>abap_bool</i>

• Example

For a filter request like

```
<service_root_url>/<entity_set>?$orderby=Customer_ID desc
```

the method `get_sort_elements` returns the following entries in the returning table:

rt_sort_elements

ELEMENT_NAME	DESCENDING
CUSTOMER_ID	X

• Example

See [Implementing Sorting in an Unmanaged Query \[page 509\]](#).

Method `get_parameters`

This method returns a list of the entity parameters and their values.

Signature

```
METHODS get_parameters RETURNING VALUE(rt_parameters) TYPE tt_parameters.
```

Parameter

rt_parameters

Contains a list of parameters and their given values.

tt_parameters

PARAMETER_NAME	VALUE
<i>string</i>	<i>string</i>

• Example

For a filter request like

```
<service_root_url>/  
<entity_set>(p_start_date=datetime'2016-07-08T12:34',p_end_date=datetime'2019-  
07-08T12:34')/Set
```

the method `get_parameters` returns the following table:

`rt_parameters`

PARAMETER_NAME	VALUE
P_START_DATE	20160708
P_END_DATE	20190708

• Example

See [Using Parameters in an Unmanaged Query \[page 503\]](#).

Method `get_aggregation`

This method returns an aggregation object. This object is an interface instance of `IF_RAP_QUERY_AGGREGATION` which contains methods to indicate which elements need to be aggregated or grouped.

Signature

```
METHODS get_aggregation RETURNING VALUE(ro_aggregation) TYPE REF TO  
if_rap_query_aggregation [page 753].
```

Interface [IF_RAP_QUERY_AGGREGATION \[page 753\]](#)

Parameter

`ro_aggregation` Interface instance for information about aggregation and grouping.

• Example

See [Implementing Aggregations in an Unmanaged Query \[page 511\]](#).

Method `get_search_expression`

This method returns the requested search string.

Signature

```
METHODS get_search_expression RETURNING VALUE(rv_search_expression) TYPE  
string.
```

Parameter

rv_search_expression	Contains a free search expression with unspecified format.
----------------------	--

• Example

See [Implementing Search in an Unmanaged Query \[page 505\]](#).

Method `get_requested_elements`

This method returns the requested elements, which need to be given to the response.

Signature

```
METHODS get_requested_elements RETURNING VALUE(rt_requested_elements) TYPE  
tt_requested_elements.
```

rt_requested_elements	Contains a list of the requested elements.
-----------------------	--

• Example

See [Considering Requested Elements in an Unmanaged Query \[page 510\]](#).

8.2.2.1.1.1 Interface `IF_RAP_QUERY_FILTER`

This interface is a filter criteria provider for the unmanaged query. The methods provide different representations for the filter criteria.

Method `get_as_ranges`

This method returns the filter as a list of simultaneously applicable range tables. The table is initial if no filter is supplied.

Signature

```
METHODS get_as_ranges RETURNING VALUE(rt_ranges) TYPE tt_name_range_pairs  
RAISING cx_rap_query_filter_no_range.
```

Parameter

`rt_ranges`

Contains a list of filter conditions in name-range-table pairs. That means, every requested filter element is related to a ranges table that indicates the filter conditions. The returning value is in a ranges-table-compatible format. The following table illustrates the list of name and ranges table.

The columns of the ranges tables have the semantics of selection table criteria. They are defined as follows:

- **SIGN**: Contains the values `I` for inclusive or `E` for exclusive consideration of the defined range
- **OPTION**: Contains the operator values. Valid operators are `EQ`, `NE`, `GE`, `GT`, `LE`, `LT`, `CP`, and `NP`, if the column high is initial, and `BT`, `NB`, if column high is not initial.
- **LOW**: Contains the comparison value or the lower interval limitation.
- **HIGH**: Contains the upper interval limitation.

`tt_name_range_pairs`

NAME	RANGE		
<code>string</code>	<code>tt_range_option</code>		
SIGN	OPTION	LOW	HIGH
<code>c(1)</code>	<code>c(2)</code>	<code>string</code>	<code>string</code>

Example

For a filter request like

```
<service_root_url>/<entity_set>?$filter= Agency_ID eq '070031'  
                                and (Begin_Date ge datetime 2019-01-01T00  
                                and Begin_Date le datetime 2019-12-31T00)
```

the method `get_as_ranges` returns the following entries in the range table:

`tt_name_range_pairs`

NAME	RANGE		
<code>AGENCY_ID</code>	<code>tt_range_option</code>		
SIGN	OPTION	LOW	HIGH
<code>I</code>	<code>EQ</code>	<code>070031</code>	

NAME	RANGE			
	SIGN	OPTION	LOW	HIGH
Begin_Date	tt_range_option	I	BT	20190101 20191231

Exception

`cx_rap_query_filter_no_ran ge` This exception is thrown if the filter cannot be converted into a ranges table.
In this case the developer can try to use the method `get_as_sql_string` as a fall back or throw an error.

• Example

See [Implementing Filtering in an Unmanaged Query \[page 501\]](#).

Method `get_as_sql_string`

This method returns the filter as an SQL string. The string is initial if no filter is supplied.

Signature

```
METHODS get_as_sql_string RETURNING VALUE(rv_string) TYPE string.
```

Parameter

`rv_string` Contains the filter conditions as an SQL string. The variable can be used directly in the WHERE clause of an SQL statement to select data.

• Example

For a filter request like

```
<service_root_url>/<entity_set>?$filter= Agency_ID eq '070031'  
and (Begin_Date ge datetime 2019-01-01T00  
and Begin_Date le datetime 2019-12-31T00)
```

the method `get_as_sql_string` returns `BEGIN_DATE BETWEEN '20190101' AND '20191231' AND AGENCY_ID = '070031'`. This string has the correct syntax to be used in an SQL statement.

See [Implementing Filtering in an Unmanaged Query \[page 501\]](#).

8.2.2.1.1.2 Interface `IF_RAP_QUERY_PAGING`

This interface provides the information for paging requests. The methods provide the offset and the page size for one OData request.

Method `get_offset`

This method indicates the number of records to drop from the list of data records in the data source. In an OData query request, the offset is requested by the query option `$skip`.

Signature

```
METHODS get_offset RETURNING VALUE(rv_offset) TYPE int8.
```

Parameter

`rv_offset` Contains the number of records that are dropped from the result list.

❖ Example

If `rv_offset` is 2, the first record in the result list is the data record on position 3.

❖ Example

See [Implementing Paging in an Unmanaged Query \[page 507\]](#).

Method `get_page_size`

This method indicates the maximum number of records that are to be returned. In an OData query request, the page size is requested by the query option `$top`.

Signature

```
METHODS get_page_size RETURNING VALUE(rv_page_size) TYPE int8.
```

Parameter

`rv_page_size` Contains the number of records that are returned.

i Note

`rv_page_size if_rap_query_pagin=>page_size_unlimited`
if no limit is requested.

• Example

See [Implementing Paging in an Unmanaged Query \[page 507\]](#).

8.2.2.1.1.3 Interface `IF_RAP_QUERY_AGGREGATION`

This interface provides methods to receive information about the requested aggregation and grouping requests.

Method `get_aggregated_elements`

This method returns the requested aggregated elements with their aggregation method and the output elements in a string table. These values can then be extracted and used in the query implementation.

Signature

```
METHODS get_aggregated_elements RETURNING VALUE(rt_aggregated_elements) TYPE  
tt_aggregation_elements.
```

Parameter

`rt_aggregated_elements`

Contains the aggregation method, the input element, and the output element.

The constants for the available predefined aggregation methods are:

- **COUNT**: for returning the number of values of the input element in the output element.
The constant `co_count_all_identifier` as value for `input_element` denotes the counting of all rows.
- **COUNT_DISTINCT**: for returning the number of unique values of the input element in the output element.
- **SUM**: for returning the sum of the input element in the output element.
- **MIN**: for returning the minimum of the input element in the output element.
- **MAX**: for returning the maximum of the input element in the output element.
- **AVG**: for returning the average of the input element in the output element.

The input element is the element whose values are aggregated and the output element is the element, which contains the aggregated value. The output element can be the same as the input element.

❖ Example

`rt_aggregated_elements`

<code>aggregation_method</code>	<code>input_element</code>	<code>result_element</code>
SUM	BOOKING_FEE	TOTAL_BOOKING_FEE

Signature

❖ Example

See [Implementing Aggregations in an Unmanaged Query \[page 511\]](#).

Method `get_grouped_elements`

This method returns the requested elements by which the result is to be grouped.

Signature

```
METHODS get_grouped_elements RETURNING VALUE(rt_grouped_elements) TYPE  
tt_grouped_elements.
```

Parameter

`rt_grouped_elements`

Returns an ordered list of the elements by which the result is to be grouped. The elements are listed in the order of grouping priority.

• Example

See [Implementing Aggregations in an Unmanaged Query \[page 511\]](#).

8.2.2.1.2 Interface IF_RAP_QUERY_RESPONSE

This interface provides methods to return data and the count for the query response. The results of the methods of interface `IF_RAP_QUERY_REQUEST` are integrated in the response.

Method `set_data`

This method provides the response for the method `if_rap_query_request~is_data_requested`. If this method is called, the table of result data must be provided (empty if there is no result data).

Signature

```
METHODS set_data IMPORTING it_data TYPE STANDARD TABLE  
                    RAISING   cx_rap_query_response_set_twic.
```

Parameter

`it_data` Contains a table of the data records for the query response.

Use the type of your custom entity for the response to be compatible with the request.

Exception

`cx_rap_query_response_set_twic` Exception is raised when the result table is set more than once.

• Example

See [Requesting and Setting Data or Count in an Unmanaged Query \[page 500\]](#).

Method `set_total_number_of_records`

This method provides the response for the method `if_rap_query_request~is_total numb_of_rec_requested`. If this method is called, the count needs to be set for the response.

Signature

```
METHODS set_total_number_of_records IMPORTING iv_total_number_of_records TYPE  
int8  
RAISING cx_rap_query_response_set_twic.
```

Parameter

iv_total_number_of_records Contains the total number of records. If no records match the given request criteria, the value zero must be passed.

Exception

cx_rap_query_response_set_twic Exception is raised when the number of records is set more than once.

• Example

See [Requesting and Setting Data or Count in an Unmanaged Query \[page 500\]](#).

8.2.3 API for Virtual Elements

Virtual elements are defined in CDS projection views and implemented in related ABAP classes that calculate their values and transform filtering and sorting requests. For this calculation and transformation, the generic orchestration framework calls the ABAP classes that implement the methods of the virtual elements API.

- Calculation: [Interface IF_SADL_EXIT_CALC_ELEMENT_READ \[page 756\]](#)

For more information about virtual elements, see [Using Virtual Elements in CDS Projection Views \[page 440\]](#).

8.2.3.1 Interface IF_SADL_EXIT_CALC_ELEMENT_READ

Interface that must be implemented by calculation classes for virtual elements that are referenced at the level of the CDS projection view element with the annotation@ObjectModel.virtualElementCalculatedBy.

This interface defines the following methods for implementing the field calculation:

Method get_calculation_info

This method provides a list of all elements that are required for calculating the values of the virtual elements in the requested entity.

This method is called during runtime before the retrieval of data from the database to ensure that all necessary elements for calculation are filled with data.

Signature

```
TYPES tt_elements TYPE SORTED TABLE OF string WITH UNIQUE DEFAULT KEY .  
METHODS get_calculation_info IMPORTING !it_requested_calc_elements TYPE  
tt_elements  
          !iv_entity TYPE string  
          EXPORTING !et_requested_orig_elements TYPE  
          RAISING   cx_sadl_exit.
```

Parameter

importing	it_requested_calc_elements	List of the virtual elements that are requested by the client and intended for calculation. The names of the virtual elements are listed as upper case strings.
	iv_entity	Name of the CDS entity that contains the requested virtual elements in it_requested_calc_elements. The name of the CDS entity is a string in upper case.
exporting	et_requested_orig_elements	List of the CDS elements that are needed for the calculation. The CDS elements are listed as upper case strings.
raising	cx_sadl_exit	Abstract exception class that can be used to raise exceptions and return messages related to the processing of the virtual element calculation.

i Note

Only elements of the requested entity that are needed for the calculation of the virtual elements. iv_entity can be appended for data retrieval.

Example

See [Implementing the Calculation of Virtual Elements \[page 443\]](#).

Method calculate

Executes the value calculation for the virtual element.

This method is called during runtime after data is retrieved from the database. The elements needed for the calculation of the virtual elements are already inside the data table passed to this method. The method returns a table that contains the values of the requested virtual elements.

Signature

```
TYPES tt_elements TYPE SORTED TABLE OF string WITH UNIQUE DEFAULT KEY .  
METHODS calculate IMPORTING !it_original_data      TYPE STANDARD TABLE  
          !it_requested_calc_elements TYPE tt_elements  
          CHANGING  !ct_calculated_data    TYPE STANDARD TABLE
```

```
RAISING    cx_sadl_exit.
```

importing	it_original_data	Result table of the retrieved values for the elements requested by the client, in addition to the elements that are needed for calculation, which are requested for data retrieval by the method <code>get_calculation_info</code> .
	it_requested_calc_elements	List of virtual elements that are requested by the client and are intended for calculation. The names of the virtual elements are listed as upper case strings.
exporting	ct_calculated_data	Table of the virtual element for each entity instance corresponding to the table <code>it_original_data</code> by index.
raising	cx_sadl_exit	Abstract exception class that can be used to raise exceptions and return messages related to the processing of virtual element calculation.

With this method you ensure that the virtual element is calculated and that its values are passed to the runtime engine to process the data in the requested entity.

• Example

See [Implementing the Calculation of Virtual Elements \[page 443\]](#).

8.3 Tool Reference

The following chapter provides information about specific tools and features that are enabled for the ABAP RESTful programming model in ABAP Development Tools (ADT):

- [Exploring Business Objects \[page 759\]](#)
- [Working with Behavior Definitions \[page 760\]](#)
- [Working with Business Services \[page 767\]](#)
- [Creating Projection Views \[page 773\]](#)

In accordance with the relevant ABAP development scenario, you can find further information about ABAP development using ADT features in the following development user guides:

-
-
-

8.3.1 Exploring Business Objects

Context

You have a business object containing several nodes that are represented as CDS entities and stored in DDL sources. The corresponding behavior is distributed among the behavior definition and several behavior implementation classes.

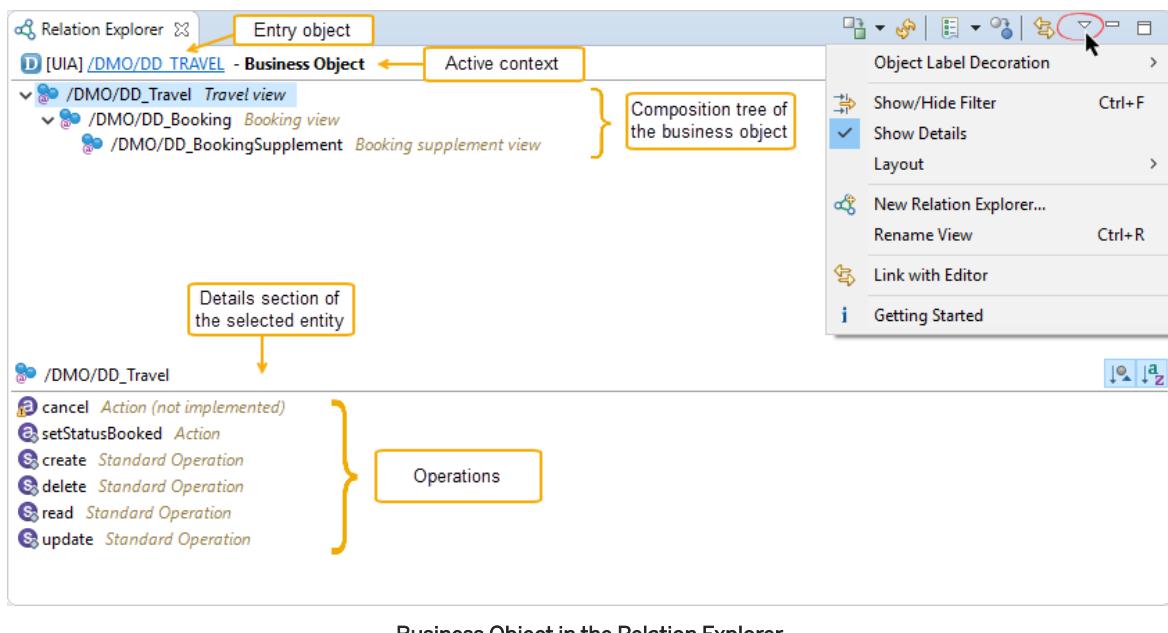
Now, you want to understand the entire structure of this business object at a glance and explore the operations of each business object node.

To get an overview of the business object, proceed as follows.

Procedure

1. In the Editor, open the *root entity* or any other object that is related to the business object such as *behavior definition* or *behavior implementation class*.
2. From the context menu, select **Show in > Relation Explorer**. Alternatively, use **Alt + Shift + W**.

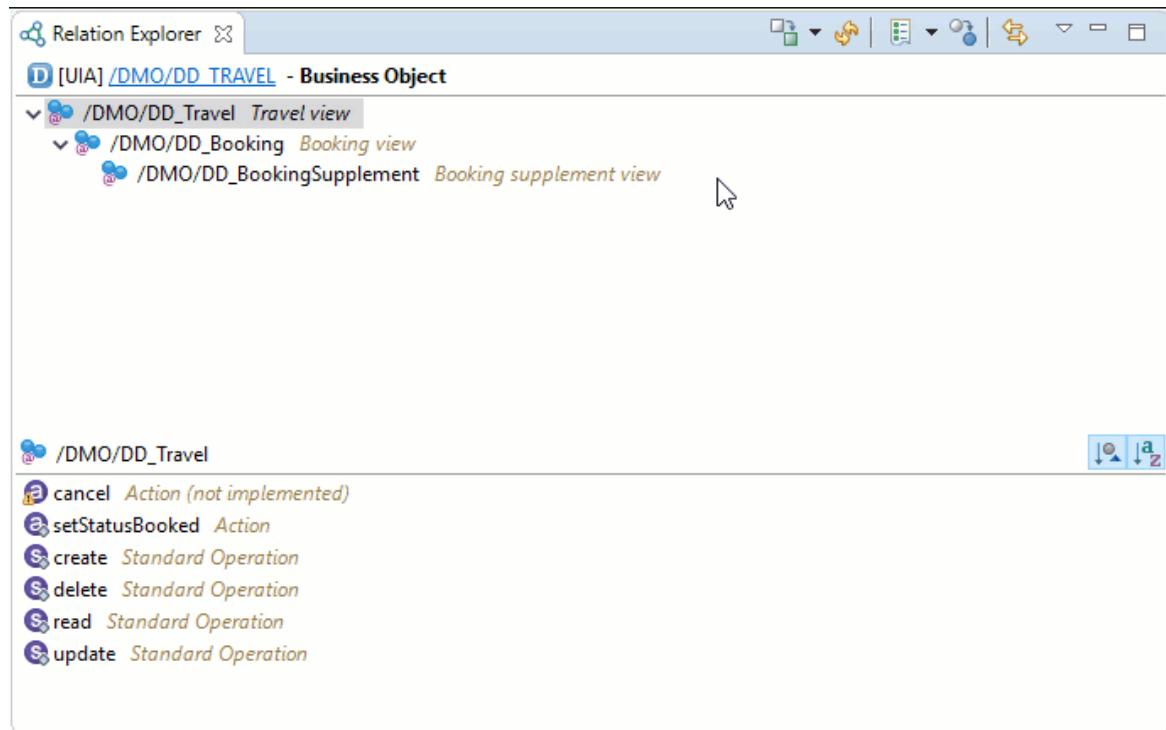
The Relation Explorer displays a composition tree of the business object in the *Business Object* context. In the details section, you can view the operations of the selected entity. On each entity in the object's composition tree and on each operation in the details section, you can trigger the context menu.



Changing the Context

You can select other contexts using from the toolbar.

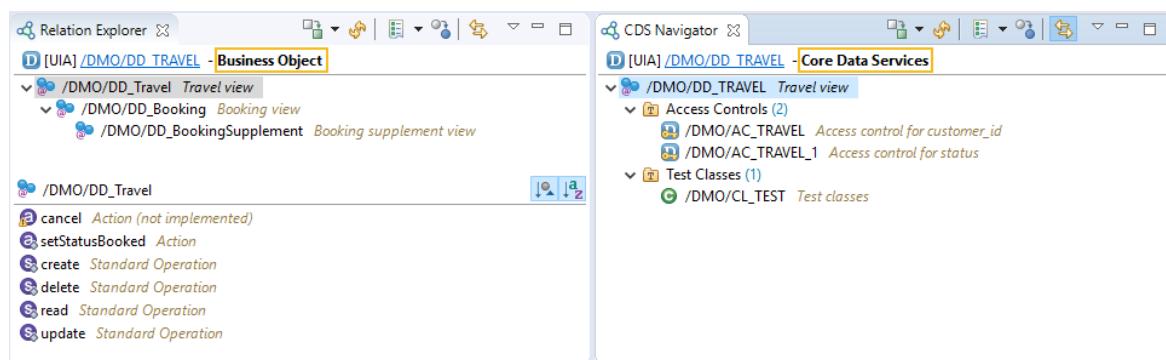
Select the *Core Data Services* context to display CDS-specific related objects such as access controls.



Changing the Context in the Relation Explorer

New Relation Explorer

If you want to see both contexts at once, instead of switching between them, . For example, if you have the object opened in the *Business Object* context, you can create a new instance of the *Relation Explorer* to display the CDS-specific related objects for the same business object in the *Core Data Services* context.



Contexts of the Business Object

8.3.2 Working with Behavior Definitions

Based on the existing CDS data model, you can create and edit behavior definitions to define the behavior of business objects in the ABAP RESTful programming model. To implement the behavior, create a behavior implementation class.

Related Information

- [Creating Behavior Definitions \[page 761\]](#)
- [Editor Features \[page 763\]](#)
- [Creating Behavior Implementations \[page 764\]](#)
- [Business Object \[page 55\]](#)
- [Documenting Behavior Definitions \[page 766\]](#)

8.3.2.1 Creating Behavior Definitions

To define the behavior of business objects in the ABAP RESTful programming model, create a behavior definition as the corresponding ABAP repository object. You create a behavior definition as follows:

1. In the *Project Explorer*, select the relevant node for the data definition that contains the CDS root entity for which you want to create a behavior definition.
2. Open the context menu and select *New Behavior Definition* to launch the creation wizard.
3. The *Project* and *Package* are inserted automatically. You can change them if needed.

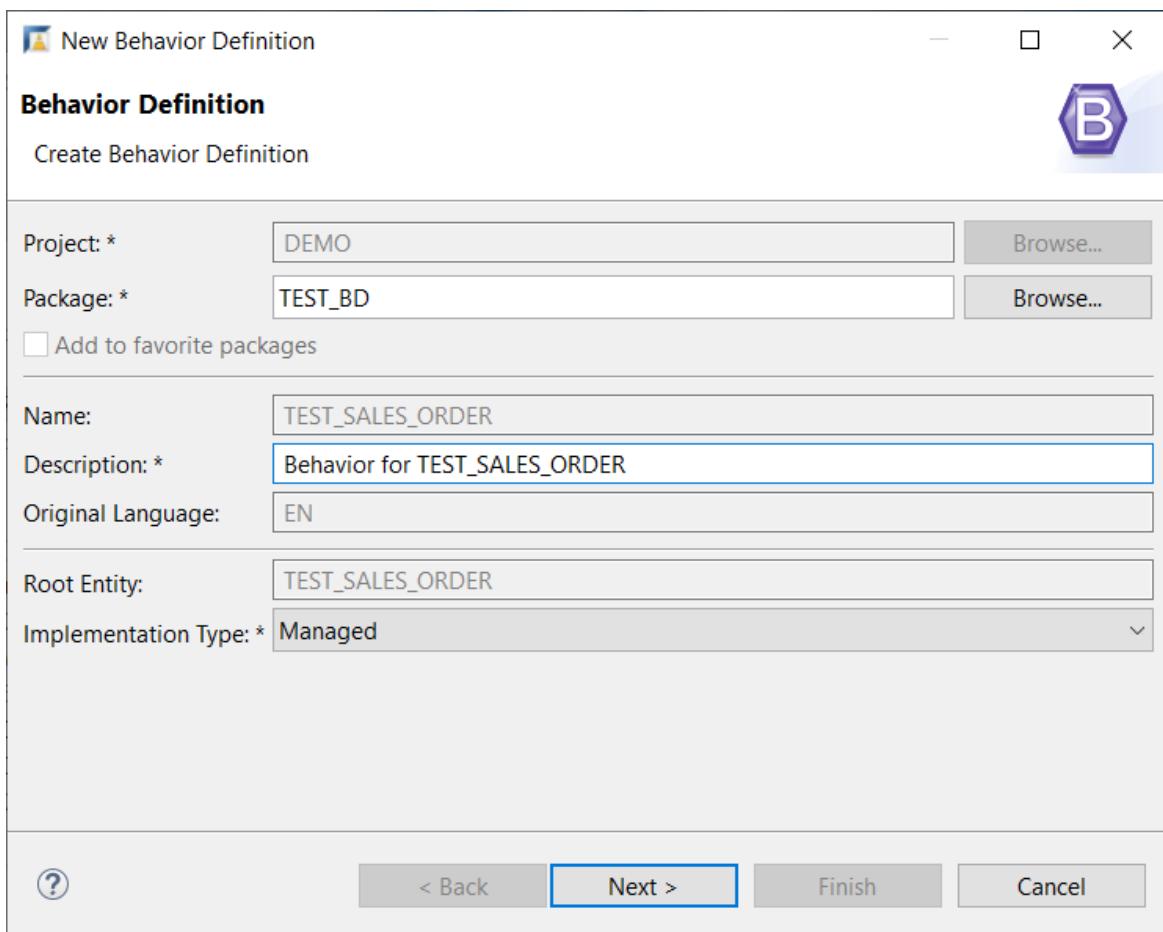
i Note

The *Name* of the behavior definition is the same as the name of its root entity. It is automatically inserted and cannot be modified.

4. Enter a *Description* for the behavior definition.
5. The *Root Entity* that you selected in Project Explorer is automatically inserted.

i Note

If you triggered the *New Behavior Definition* wizard not from the referenced CDS view, the *Root Entity* is not filled in automatically. In this case, select the *Root Entity* manually by using the *Browse* button.



Creation wizard

- From the drop-down list of the *Implementation Type* field, select *Managed* or *Unmanaged* implementation type.

i Note

If you create a Behavior Definition from a projection root entity, the implementation type *Projection* is inserted automatically.

- Select *Next*.
- Assign a transport request.
- Select *Finish*.

Result

The created behavior definition object represents the root node of a new business object in ABAP RESTful programming model.

In the Project Explorer, the new behavior definition is added to the Core Data Services folder.

In the editor, a new behavior definition is created with a basic structure. You can now start editing or refining the behavior definition using predefined [language elements](#).

Related Information

[Editor Features \[page 763\]](#)

[Creating Behavior Implementations \[page 764\]](#)

[Business Object \[page 55\]](#)

8.3.2.2 Editor Features

You can define and edit the behavior of the business object in the created behavior definition.

The following table gives you an overview of the supported features.

Availability of features

Feature Types	Features	Key Shortcuts	Availability
Standard	Activation	[Ctrl] + [F3]	✓
	Deleting		✓
	Duplicating		✗
Search	Editing		✓
	Where-Used Search	[Ctrl] + [Shift] + [G]	✓
Convenience	Source Search	[Ctrl] + [H]	✓
	Formatting	[Shift] + [F1]	✓
	Navigation	[F3]	✓
	Outline		✓
	Quick Outline	[Ctrl] + [O]	✓

Feature Types	Features	Key Shortcuts	Availability
	Quick Assists	[<code>Ctrl</code>] + [<code>1</code>]	✓
	Code Completion	[<code>Ctrl</code>] + [<code>Space</code>]	✓
	Syntax Highlighting		✓
	Automatic Syntax Check		✓
	Element Information	[<code>F2</code>]	✓
Others	Comparing Source Code		✓
	Version History		✓
	Share Link		✓

Related Information

[Working with Behavior Definitions \[page 760\]](#)
[Creating Behavior Definitions \[page 761\]](#)
[Creating Behavior Implementations \[page 764\]](#)

8.3.2.3 Creating Behavior Implementations

Prerequisites

You created a behavior definition object. The object is activated.

Context

To implement the behavior of business objects defined in the behavior definition, create a behavior implementation class.

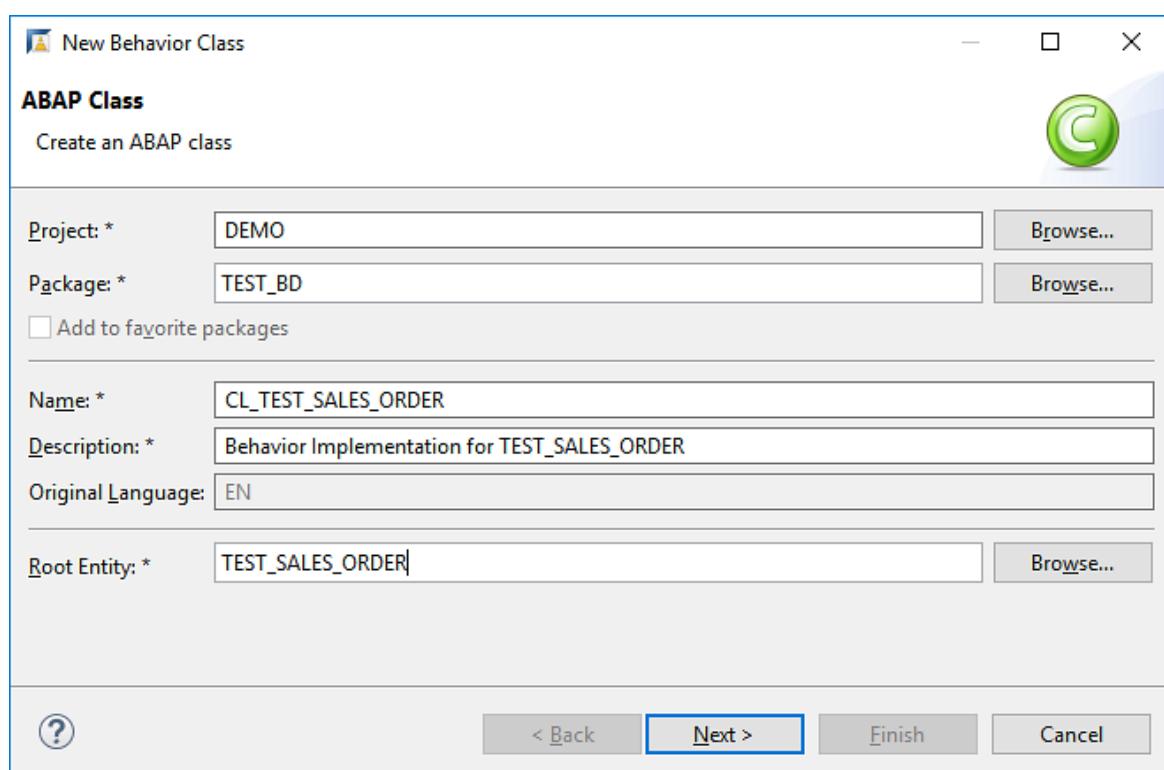
Procedure

1. In the *Project Explorer*, select the relevant behavior definition for which you want to create a behavior implementation class.
2. Open the context menu and select *New Behavior Implementation* to launch the creation wizard.

The *Project* and *Package* are automatically inserted. If needed you can change them.

3. Enter a *Name* and *Description* for the behavior implementation class.

The name of the *Behavior Definition* is automatically inserted. If needed you can change it.



Creation wizard for a behavior implementation class

4. Select *Next*.
5. Assign a transport request.
6. Select *Finish*.

Results

The behavior implementation class is created. In *Project Explorer*, the new behavior implementation is added to the corresponding folder.

The *Local Types* tab is automatically opened. In the tab, local *Icl_handler* and *Icl_saver* classes are created. These local classes are used to implement the interaction phase and the save sequence in an unmanaged transactional scenario.

You can now start implementing the behavior using predefined *language elements*.

Related Information

[Working with Behavior Definitions \[page 760\]](#)

[Creating Behavior Definitions \[page 761\]](#)

[Editor Features \[page 763\]](#)

[Business Object \[page 55\]](#)

8.3.2.4 Documenting Behavior Definitions

You can document behavior definitions as follows:

1. From the Project Explorer, select the behavior definition you want to document. Use the context menu to .
The created document is opened in the *Knowledge Transfer Document* Editor.
2. The behavior definition is displayed with its elements, such as actions, functions, associations, and standard operations, structured as a tree in the *Object Structure* section. In the *Documentation* section, you can .

8.3.3 Working with Business Services

In the context of the ABAP RESTful programming model, a business service is a RESTful service which can be called by a consumer. It is defined by exposing its data model together with the associated behavior. A business service consists of a service definition and a service binding.

8.3.3.1 Creating Service Definitions

A service definition provides the CDS entities that are part of the data model to be exposed as a business service.

Prerequisites

You need the standard developer authorization profile to create ABAP development objects.

Context

You want to define the data to be exposed as a business service by one or more service bindings.

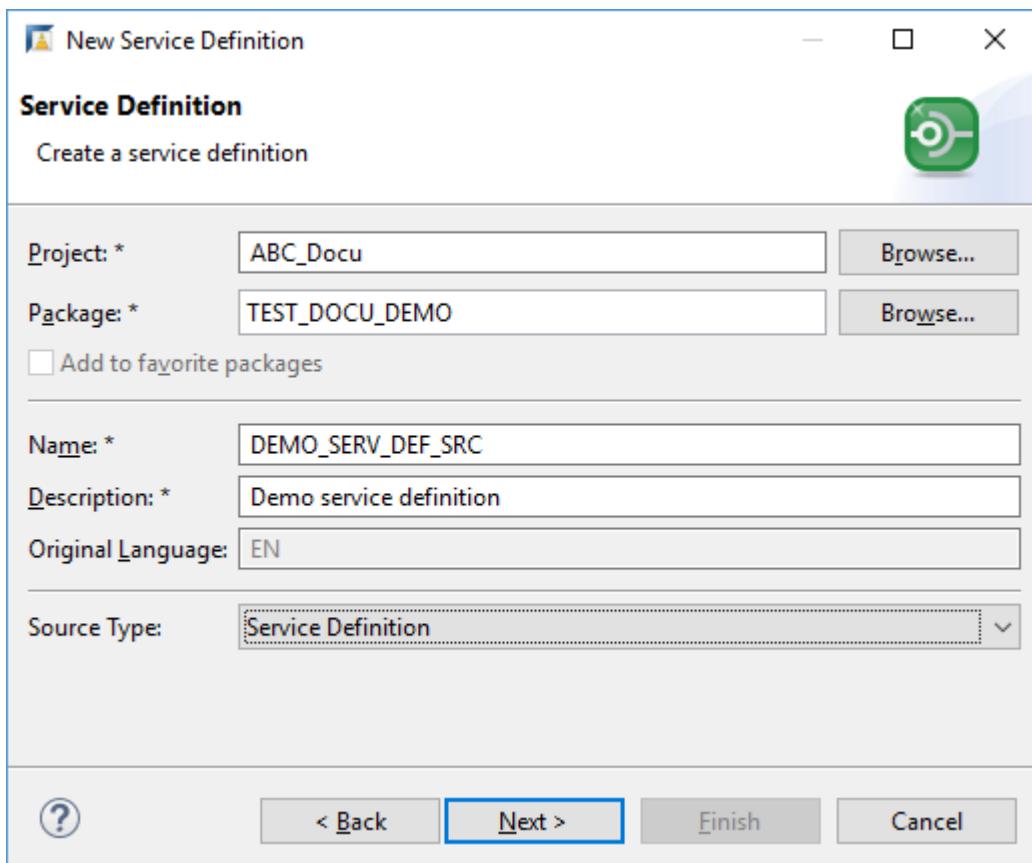
Procedure

1. In your ABAP project, select the relevant package node in the *Project Explorer*.
2. Open the context menu and choose **New > ABAP > Business Services > Service Definition** to launch the creation wizard.
3. In addition to the *Project* and *Package*, enter the *Name* and the *Description* for the service definition to be created.

i Note

The maximum length for names of data definition is 30 characters.

4. Choose *Next*.



Wizard page when creating a service definition

5. Assign a transport request.
6. Choose *Next*.
7. Choose a template which you want to base your service definition.
8. Choose *Finish*.

Results

In the selected package, the ABAP back-end system creates an inactive version of a service definition and stores it in the ABAP Repository.

In the *Project Explorer*, the new service definition is added to the *Business Services* folder of the corresponding package node. As a result of this creation procedure, the source editor will be opened. Here, you can start enter the CDS entities to be exposed as a business service.

8.3.3.2 Creating Service Binding

Using a service binding you can enable a service definition to create a business service with a protocol of your choice.

Prerequisites

- You need the standard developer authorization profile to create ABAP development objects.
- You have created the relevant [Service Definition \[page 767\]](#).

Context

You can use an existing service definition to create a business service with an OData V2 protocol for example.

Procedure

1. In your ABAP project, select the relevant package node in the *Project Explorer*.
2. Open the context menu and choose to launch the creation wizard.
3. In addition to the *Project* and *Package*, enter the *Name* and the *Description* for the service binding to be created.

Note

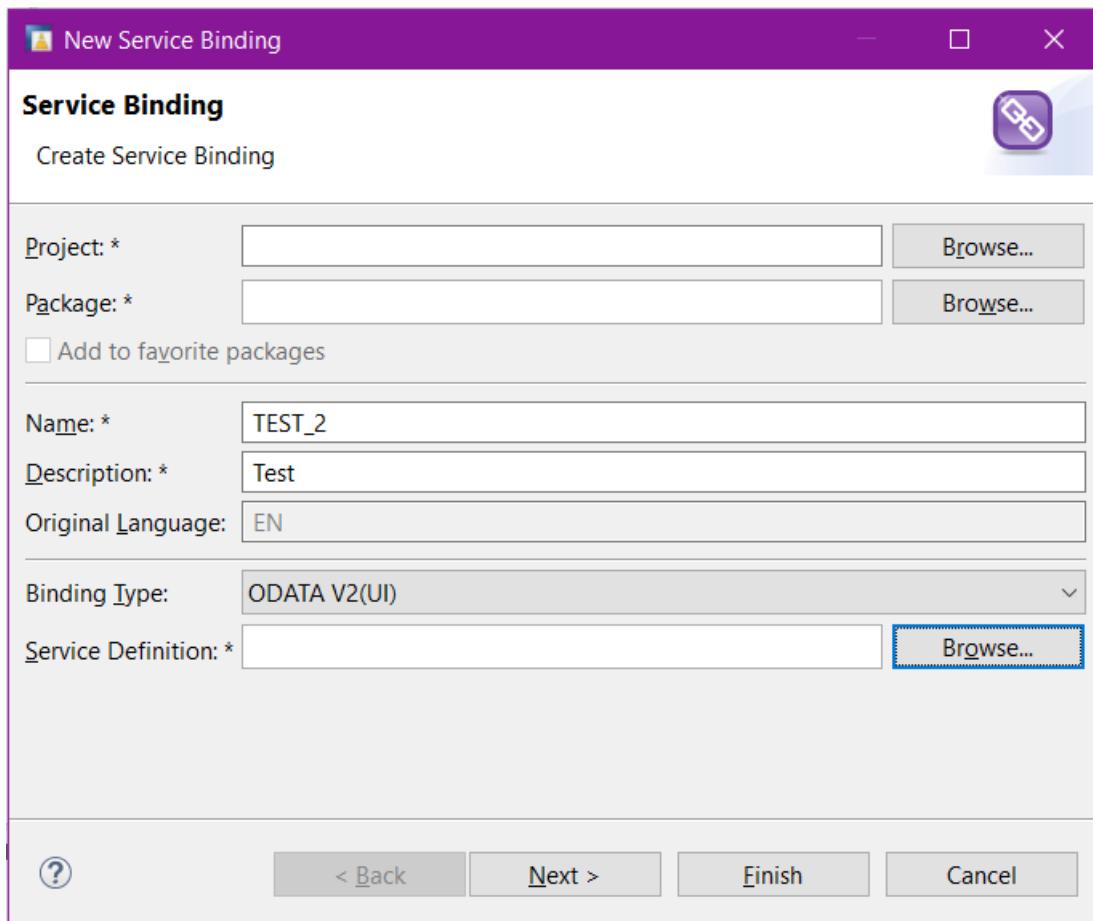
The maximum length for names of a service binding is 26 characters.

4. Select the *Binding Type*.

i Note

The protocol is OData V2 and the available categories are UI and Web API. An UI based OData V2 service can be consumed by any SAP UI5 application. An Web API based OData V2 service can be used for providing APIs and not UI based applications.

5. If not yet specified, search for the *Service Definition* that you want to use as a base for your service binding.
6. Choose *Next*.



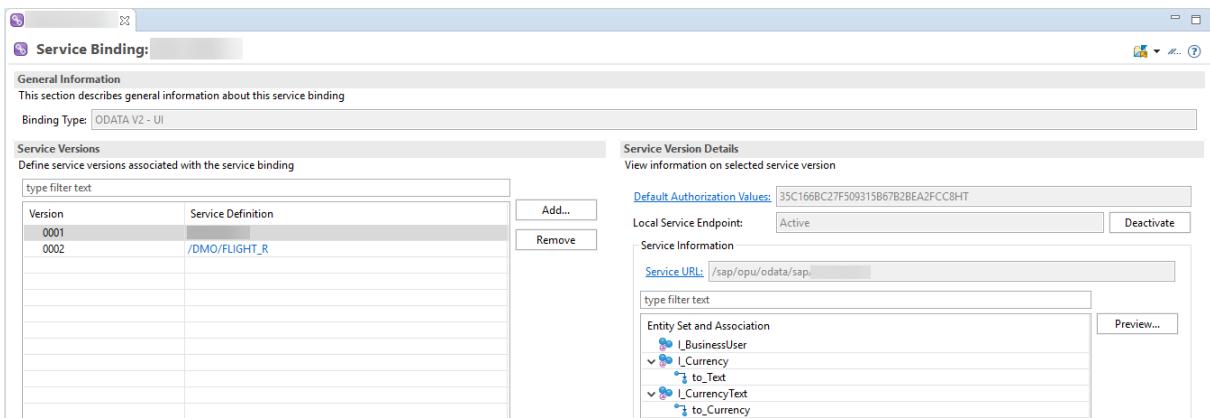
Wizard page when creating a service binding

7. Choose *Next*.
8. Assign a transport request.
9. Choose *Finish*.

Results

In the selected package, the ABAP back end system creates a service binding and an OData V2 service.

In the *Project Explorer*, the new service binding is added to the *Business Services* folder of the corresponding package node. As a result of this creation procedure, the form editor will be opened. Here, you can activate your OData V2 service.



Service Binding Editor after activating the OData V2 - UI business service

Related Information

[Business Service \[page 98\]](#)

[Service Binding \[page 111\]](#)

8.3.3.2.1 Using Service Binding Editor

Use the Service Binding editor to preview the business service.

After you create a service binding, the editor is displayed. The following actions can be done here:

Type	Action
OData V2 (UI)	<ul style="list-style-type: none"> The Service Versions section shows the service version and the associated service definition. You can add a new service version for an active service definition by clicking Add.... Similarly, choose Remove to remove a service version. The Service Version Details section shows the local service endpoint and lists the entity sets. You can click Service URL to view the service document. Select an entity set and click Preview to open a preview of the Fiori elements app in an external browser. Alternately, you can do this by right-clicking an entity set and selecting Open Fiori Elements App Preview. For each entity set, the navigation shown represents the association with another entity set. Use the Activate button to see the service details for each service version. After you've activated the service, you can choose Deactivate to revert to the inactive state. The service information doesn't appear for a service that is deactivated. For each service version, an authorization value is generated. Choose Default Authorization Values to open the Authorization Default Values editor. For maintaining default authorization values, see SAP Cloud Platform - ABAP Development User Guide.

Type	Action
OData V2 (Web API)	<ul style="list-style-type: none"> The Service Versions section shows the service version, API state, and the associated service definition. You can add a new service version for an active service definition by clicking Add..... Similarly, choose Remove to remove a service version. <p>i Note</p> <p>You can only remove a service version that was created last, provided it is in the state Not Released.</p> <ul style="list-style-type: none"> You can change the API state for a service version using the context menu under API State and selecting API State. For more information, see Released APIs. The Service Version Details section shows the service information and lists the entity sets. For each entity set, the navigation shown represents the association with another entity set. Use the Activate button to see the service details for each service version. After you've activated the service, you can choose Deactivate to revert to the inactive state. The service details do not appear for a service that is deactivated. For each service version, an authorization value is generated. Choose Default Authorization Values to open the Authorization Default Values editor. For maintaining default authorization values, see SAP Cloud Platform - ABAP Development User Guide.
Test Class Generation	<p>You can generate automated tests for the OData service you've created using service binding. The test provides guidance on how to access the OData service using ABAP Units and provides the test code for performing CRUD operations on an entity set. Perform the following steps to generate a test class for a selected entity set:</p> <ol style="list-style-type: none"> Choose Activate next to the local service endpoint. Select an entity set, right click and then, choose New ABAP Test Class. Provide a name and description for the test class. Choose Next and then, Finish. <p>A test class is generated in which local test classes for each CRUD operation in the selected entity set can be viewed. You can either create a separate test class for each entity set or copy and paste the generated code, then change the name of the entity set accordingly for writing ABAP Units for other entity sets.</p> <p>i Note</p> <p>The OData service tests and the OData service run in the same session. Thus, these tests can use various test double mechanisms available in ABAP Unit for isolating database dependencies.</p> <p>i Note</p> <p>If you create the service binding object using abapGit flow, you get an error that the service binding is inconsistent and must be repaired. You must activate the service definitions and click on the Repair Service Binding link. This repairs the service binding and makes the service binding consistent.</p>

8.3.4 Creating Projection Views

A projection view enables you to expose a subset of data from an underlying data model, for example in an OData service.

Prerequisites

You need the standard developer authorization profile to create development objects.

Context

In a transactional scenario, you want to create and define, for example, a consumption-specific OData service that only exposes relevant data of an underlying data model using a service definition and service binding.

i Note

A projection view can be used in one or more service definitions.

Procedure

1. In your ABAP cloud project, select a package node in the *Project Explorer*.
2. To launch the creation wizard, open the context menu and choose ► *New* ► *Other...* ► *Core Data Services* ► *Data Definition* ▾.
3. In addition to the *Project* and *Package*, enter a *Name* and *Description* for the data definition you want to create.

i Note

The maximum length for names of data definitions is 30 characters.

4. Choose *Next*.
5. Assign a transport request.
6. Choose *Next*.
7. Select the *Define Projection View* template.

i Note

By default, ABAP Development Tools uses the last selected template for creation.

8. Choose *Finish*.

Results

In the selected package, the ABAP cloud system creates an inactive version of a data definition, and stores it in the ABAP repository.

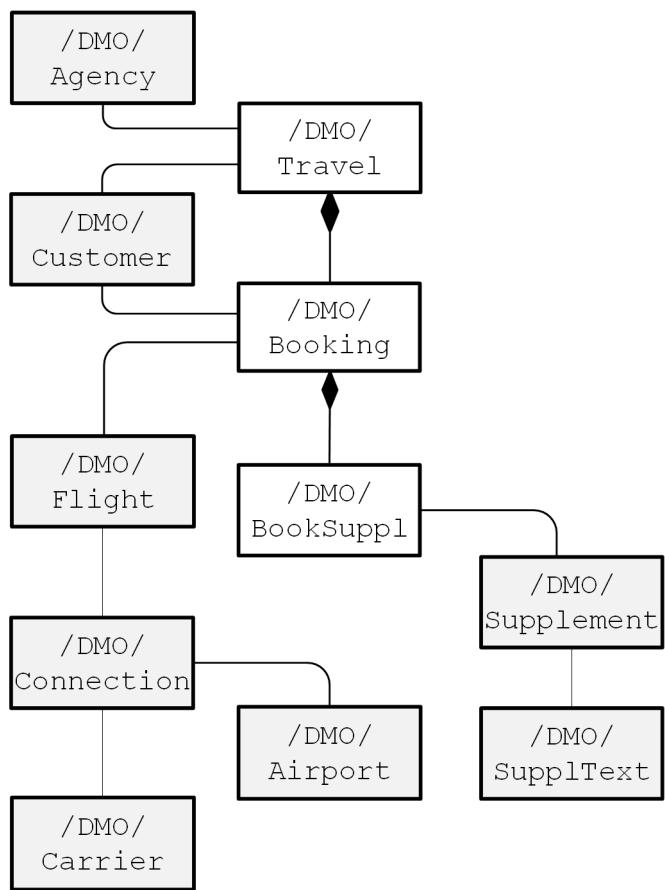
In the *Project Explorer*, the new data definition is added to the *Core Data Services* folder of the corresponding package node. As a result of this creation procedure, the source editor is opened, where you can start completing the template.

You can now define the elements from a datasource from which you want to expose data, for example, in an OData service.

8.4 ABAP Flight Reference Scenario

The ABAP Flight Reference Scenario helps you to get started with development in the context of the ABAP RESTful Programming Model. It contains demo content that you can play around with and use to build your own sample applications..

The reference scenario is based on multiple database tables which you can use to build your application. These tables are also used in the development guides in the [Develop \[page 121\]](#) section.



- [#unique_18/unique_18_Connect_42_subsection-im1 \[page 775\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im2 \[page 776\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im3 \[page 776\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im4 \[page 777\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im5 \[page 777\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im6 \[page 778\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im7 \[page 777\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im8 \[page 779\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im9 \[page 778\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im10 \[page 778\]](#)
- [#unique_18/unique_18_Connect_42_subsection-im11 \[page 779\]](#)

Hover over each element for a description of the related database table. Click the element to view the database table fields.

/DMO/AGENCY

The database table **/DMO/AGENCY** stores general data about the travel agency that operates travels for customers.

The database table has the following fields:

- agency_id (key)
- name
- street
- postal_code
- city
- country_code
- phone_number
- email_address
- web_address

The key field is the unique ID for the travel.

/DMO/TRAVEL

The database table */DMO/TRAVEL* stores general travel data. In addition, it includes administrative data about the creation and changing of instances.

The database table has the following fields:

- travel_id (key)
- agency_id
- customer_id
- begin_date
- end_date
- booking_fee
- total_price
- currency_code
- description
- status
- createdby
- createdat
- lastchangedby
- lastchangedat

The key field is the unique ID for the travel.

/DMO/CUSTOMER

The database table */DMO/CUSTOMER* stores general data about customers. In addition, it stores administrative data about the creation and changing of instances.

The database table has the following fields:

- customer_id
- first_name
- last_name
- title
- street
- postal_code

- city
- country_code
- phone_number
- email_address
- createdat
- createdby
- lastchangedby
- lastchangedat

The key field is the unique ID for the customer.

[*/DMO/BOOKING*](#)

The database table [*/DMO/BOOKING*](#) stores data about a booked flight for a certain travel instance. Apart from general flight and booking data, it includes the customer ID for whom the flight is booked as well as the travel ID to which the booking belongs.

The database table has the following fields:

- travel_id (key)
- booking_id (key)
- booking_date
- customer_id
- carrier_id
- connection_id
- flight_date
- flight_price
- currency_code

The key fields are the travel ID for the travel it belongs to and the booking ID, which are unique in combination.

[*/DMO/FLIGHT*](#)

The database table [*/DMO/FLIGHT*](#) stores general data about flights.

The database table has the following fields:

- carrier_id (key)
- connection_id (key)
- flight_date (key)
- price
- currency_code
- plane_type_id
- seats_max
- seats_occupied

The key fields are the IDs for carrier and connection as well as the flight date, which makes the flight unique.

[*/DMO/BOOK_SUPPL*](#)

The database table [*/DMO/BOOK_SUPPL*](#) stores data of booking supplements that can be booked for flights, for example meals or insurances.

The database table has the following fields:

- travel_id (key)
- booking_id (key)
- booking_supplement_id (key)
- supplement_id
- price
- currency_code

The key fields are the travel ID, the booking ID and the booking supplement ID, which are unique in combination.

[*/DMO/CONNECTION*](#)

The database table [*/DMO/CONNECTION*](#) stores general data about flight connections.

The database table has the following fields:

- carrier_id (key)
- connection_id (key)
- airport_from_id
- airport_to_id
- departure_time
- arrival_time
- distance
- distance_unit

The key fields are the IDs of carrier and connection, which are unique in combination.

[*/DMO/CARRIER*](#)

The database table [*/DMO/CARRIER*](#) stores data about flight carriers.

The database table has the following fields:

- carrier_id (key)
- name
- currency_code

The key field is the unique ID of a carrier.

[*/DMO/AIRPORT*](#)

The database table [*/DMO/AIRPORT*](#) stores data about airports.

The database table has the following fields:

- airport_id (key)
- name
- city
- country

The key field is the unique airport ID.

/DMO/SUPPLEMENT

The database table `/DMO/SUPPLEMENT` stores general data about the supplement, which can be booked for flights.

The database table has the following fields:

- `supplement_id` (key)
- `price`
- `currency_code`

The key field is the unique ID for the supplement.

/DMO/SUPPL_TEXT

The database table `/DMO/SUPPL_TEXT` stores the readable texts for the supplements in different languages.

The database table has the following fields:

- `supplement_id` (key)
- `language_code` (key)
- `description`

The key fields are the IDs of the supplement and the language, which are unique in combination.

Downloading the ABAP Flight Reference Scenario from GitHub

You can download the complete ABAP Flight Reference Scenario for the ABAP RESTful Application Programming Model from GitHub.

↗ <https://github.com/SAP-samples/abap-platform-refscen-flight/tree/Cloud-Platform> ↗ .

The steps to include the development objects in your ADT are described in the `README.md` file.

→ Remember

The namespace `/DMO/` is reserved for the demo content. Apart from the downloaded ABAP Flight Scenario, do not use the namespace `/DMO/` and do not create any development objects in the downloaded packages. You can access the development objects in `/DMO/` from your own namespace.

8.5 Naming Conventions for Development Objects

Naming conventions facilitate the development. An addition to the name of development objects conveys standardized meaning and generates consistency in your development.

General Rules

→ Remember

The general guideline for development objects is the following: [/<namespace>/]<prefix>_<object_name>_<suffix>.

- Use your own namespace that is reserved for your organization.

i Note

Consider that the namespace /`DMO`/ is reserved for demo purposes. Do not use this namespace in your productive development.

- A prefix is used for cases when there are generically different types of one development object. Then, this prefix states the semantic difference that cannot be conveyed through the object type.
For example, a service binding can expose an OData service for UI purposes and as a [Web API \[page 822\]](#). That is why, for service bindings we introduce the prefixes `UI_` and `API_` to differentiate the semantics of service bindings.
- A suffix is used for additional differentiation between different types of development objects. It helps to recognize more subtle or secondary differences in development objects.
For example, a UI service can be bound against the [OData protocol \[page 816\]](#) `OData, version 2` and `OData, version 4`. This difference can also be manifested by suffixing the name with `_02` or `_04`.

i Note

In the ABAP Flight Reference Scenario we use another suffixed character (`_R`, `_U`, `_M`, `_C`). This character identifies the development object to belong to one specific development guide (`read-only`, `unmanaged`, `managed`, service `consumption`).

The following list provides an overview of the prefixing and suffixing guidelines on naming specific development objects.

ABAP Dictionary Objects

SQL View (Database View of CDS view)

The SQL view of a CDS view must be defined in the [data definition \[page 811\]](#) of a CDS view. It cannot have the same name as the CDS view itself. Use the prefix

- `I` (without underscore) for SQL views of CDS interface views.

Example: `/DMO/I_TRAVEL_U`

CDS Objects

CDS Entity

Use the prefix

- `I_` for an interface view.
- `C_` for a projection view. The character `C` represents the consumption layer. If there are multiple projections of one CDS entity, the object name should semantically represent the projection role.

Example: `/DMO/I_Travel_U, /DMO/C_Travel_Processor_M`

Behavior Definition

A behavior definition has always the same name as the root entity of the business object.

Example: `/DMO/I_Travel_U, /DMO/C_Travel_M`

Metadata Extension

A metadata extension has the same name as the CDS entity it relates to. If you use more than one metadata extension for one CDS entity, you can add a numbered suffix.

Example: `/DMO/C_TRAVEL_U, /DMO/C_BOOKING_U_M2`

Business Services

Service Definition

Since a [service definition \[page 807\]](#) - as a part of a business service - does not have different types or different specifications, there is (in general) no need for a prefix or suffix to differentiate meaning.

Example: `/DMO/TRAVEL_U`

However, in use cases where no reuse of the same service definition is planned for UI and API services, the prefix may follow the rules of the service binding.

Example: `/DMO/UI_TRAVEL_U`

Service Binding

Use the prefix

- `UI_` if the service is exposed as a UI service.
- `API_` if the service is exposed as [Web API \[page 822\]](#).

Use the suffix

- `_O2` if the service is bound to OData protocol version 2.

- `_O4` if the service is bound to OData protocol version 4.

Example: [`/DMO/UI_TRAVEL_U_O2`](#)

Source Code Objects

Behavior Pool

Use the prefix

- `BP_` for an ABAP class that implements the behavior of a business object.

Example: [`/DMO/BP_TRAVEL_U`](#)

Handler and Saver Classes

Use the prefix

- `LHC_` for a local handler class.
- `LSC_` for a local saver class.

Depending on the modularization of your behavior implementation, you can provide the semantics of the coding in the name of the classes.

Example: [`LHC_TRAVEL_CREATE`](#)

Example: [`LHC_BOOKING_CUD`](#)

9 What's New

Here are descriptions of some of the changes of interest (delta information) to developers made to ABAP RESTful programming model:

- Version 2005 [page 783]
- Version 2002 [page 786]
- Version 1911 [page 788]
- Version 1908 [page 791]
- Version 1905 [page 796]
- Version 1902 [page 799]
- Version 1811 [page 802]

9.1 Version 2005

Implementing Own Locking Logic for Managed Business Objects

You can now define an **unmanaged lock** in a managed scenario. The unmanaged lock mechanism must be defined in the behavior definition with the syntax `lock master unmanaged` and implemented in the behavior pool in the method `FOR LOCK`, just like in an unmanaged scenario. The method is then invoked during runtime.

 For more information, see [Pessimistic Concurrency Control \(Locking\) \[page 91\]](#).

Using `dependent by _Association` in Behavior Definition

You can now use the syntax `dependent by _Association` for lock, ETag, or authorization dependent entities. The association to the respective master entity must be explicitly specified in the behavior definition and implemented when using the unmanaged scenario.

 For more information, see [Defining Elementary Behavior for Ready-to-Run Business Object \[page 165\]](#) or [Adding Behavior to the Business Object \[page 289\]](#).

Using Client-Independent Tables in Managed Scenarios

The RAP managed BO runtime now supports client-independent database tables.

i For more information, see [Using Client-Independent Database Tables in Managed Transactional Apps \[page 159\]](#).

Dynamic Operation Control for Create By Association

You can now dynamically control the create by association operation. The syntax for defining dynamic feature control is the following:

```
_association {create (features: instance); }
```

Like other feature controls, the control for the create by association must be implemented in the behavior pool with the method `FOR FEATURES`.

i For more information, see [Adding Feature Control \[page 459\]](#).

New Read-Only Associations in CDS Projection Views

It is now possible to define new read-only associations in the projection view. These associations can be reused to display additional information in a UI service, for example analytical data which is not part of the transactional basic business object. With associations in projection views, you can model new service-specific relationships.

The syntax for the association definition is the same as in CDS views:

```
association [min..max] to TargetEntity [as _Alias] on OnCondition
```

i For more information, see [CDS Projection View \[page 103\]](#).

Filtering for Null Values in OData V2 Services

OData does not foresee initial values for the following EDM data types:

- Boolean
- GUID
- Time
- DateTime
- DateTimeOffset.

Fiori Elements UIs offer the `empty` operator in filter bars for these fields. Although they appear as visually empty in the UIs, the actual value sent to the client is `null`.

To be able to select records with empty values for these data types, the client filters null values and the filter is transformed to filter for initial values for the backend. Real null values cannot be retrieved from the database by filtering for them in OData V2.

i For more information, see <https://www.odata.org/documentation/odata-version-2-0/>.

Understanding Runtime Processes

The concepts section in the ABAP RESTful Application Programming Model has been enhanced with interactive diagrams so you can get a better understanding of the runtime processes for OData requests.

i For more information, see

- [Query Runtime \[page 53\]](#)
- [Create Operation \[page 65\]](#)
- [Update Operation \[page 67\]](#)
- [Delete Operation \[page 69\]](#)
- [Create by Association Operation \[page 71\]](#)
- [Action Runtime \[page 81\]](#)
- [Save Sequence \[page 83\].](#)

Cloud Control Structure for Import Parameter in CREATE Operations

The control structure %control for import parameters of CREATE and CREATE by Association operations is now filled according to the elements that were sent by the OData client or were marked in the EML call. In particular this means that the application developer knows which elements are relevant for the create.

• Example

In the following example the elements agencyid, customerid, begindate, and enddate were filled by the OData client and thus are marked in the control structure, while the other elements are not.

```

METHOD create_travel.

DATA lt_messages    TYPE /dmo/t_message.
DATA ls_travel_in   TYPE /dmo/travel.
DATA ls_travel_out  TYPE /dmo/travel.

LOOP AT it_travel create ASSIGNING FIELD-SYMBOL(<fs_travel_create>).
  ls_trav ...
  CALL F ...
  EXP ...
  is ...
  IMPO ...
  es ...
  et ...
  IF lt ...
  INSE ...
  ELSE ...
  /dmo ...
  EX ...
  CH ...
  ENDIF.
ENDLOOP.
ENDMETHOD.

```

The screenshot shows an SAP ABAP code editor with a context menu open over a line of code. The menu path 'LOOP AT it_travel create' is highlighted. The context menu includes options like 'IT_TRAVEL_CREATE = [1x13(144)]Standard Table', '[1] = Structure: deep', and 'ENTITY USING CONTROL'. A callout box points to the 'Structure: deep' option with the text 'Elements not filled by Client'. The code itself defines a method 'create_travel' that loops through travel records, creating them and assigning fields from a field symbol to a standard table structure.

i For more information, see [<method> FOR MODIFY \[page 721\]](#).

9.2 Version 2002

Automatically Drawing Primary Key Values for Managed Business Objects

The managed runtime framework can now automatically generate key values in scenarios with UUID keys if managed numbering is defined in the behavior definition.

Syntax for defining managed numbering in the behavior definition:

```
[implementation] managed [implementation in class ABAP_CLASS [unique]];
define behavior for Entity [alias AliasedName]
  lock (master|dependent() )
  ...
{
  ...
  field ( [read only,] numbering:managed ) KeyField1, [KeyField2];
  ...
}
```

i For more information, see [Automatically Drawing Primary Key Values in Managed BOs \[page 455\]](#).

New Options for Action and Function Results

The BO runtime framework now supports results for actions and functions other than `$self`. The results can be entities of the same BO the action is defined for, entities of other BOs, or result structures. To differentiate between result entities and result structures the syntax element `entity` has been introduced.

In addition, you can now create actions, for which the action consumer can decide whether the result shall be returned completely or only parts of it, for example the keys only. Such an action must be marked with the keyword `selective` in the behavior definition.

```
define behavior for CDSEntity
...
** Action with result entity
  action ActionName result [cardinality] entity OutputEntity
** Action with result structure
  action ActionName result [cardinality] OutputStructure
** Action with selective result
  action ActionName result selective [cardinality] entity OutputEntity
...
```

i For more information, see [Actions \[page 73\]](#).

Reporting Messages in `ADJUST_NUMBERS` and `SAVE`

The implicit returning parameter `REPORTED` is now available for the methods `adjust_numbers` and `save`. By filling this parameter you can report information or success messages after the point of no return in the `save` sequence.

i For more information, see [Method ADJUST_NUMBERS \[page 733\]](#) and [Method SAVE \[page 734\]](#).

Documenting Behavior Definitions

Now you can document behavior definitions in the *Knowledge Transfer Document* editor.

From the [Project Explorer](#), select the behavior definition you want to document. Use the context menu to create a knowledge transfer document.

Knowledge Transfer Document Editor

 For more information, see [Documenting Behavior Definitions \[page 766\]](#).

9.3 Version 1911

Defining the Service Namespace for OData Services

You can now define the OData service namespace in service definitions with the annotation `@OData.schema.name`.

 For more information, see [OData Annotations \[page 568\]](#).

Filling the Result Parameter for Actions

SAP Cloud Platform, ABAP Environment 1911 enforces the proper assignment of result values in action implementations when result parameters are declared in the corresponding action definition. Whereas actions returned the input entity if the result parameter was not filled in the action implementation before 1911, they

now return the value of the result parameter as specified in the action definition and exactly according to its implementation.

To avoid that a Fiori UI shows initial values, if nothing is returned, you must fill the result parameter for all actions.

i For more information, see [Developing Actions \[page 179\]](#) and [Implementing the SET_STATUS_BOOKED Action \[page 333\]](#).

Navigating More Than One Step in OData Requests

It is now possible to navigate to associated entities using more than one navigation for both, V2 and V4 services.

Example

To request the booking supplements for the booking entity with BookingID 5 of the travel with TravelID 1, use the following syntax.

```
<service_URL>/Travel('1')/to_Booking('5')/to_BookingSupplement
```

You can link an unlimited number of entities in one OData request.

Outsourcing UI Metadata in Metadata Extensions

You can now use metadata extensions to define CDS annotations outside of the corresponding data definition of the CDS projection view. The use of metadata extensions allows the separation of concerns by separating the data model from domain-specific semantics, such as UI-related information for UI consumption.

i For more information, see [Adding UI Metadata to the Data Model \[page 284\]](#).

Using Type and Control Mapping

You can now define a mapping contract for applications that include unmanaged or managed business objects based on CDS entities on the one hand, and legacy data types that are generally older and implement the behavior or parts of it.

The general syntax for field and control (information) mapping in a behavior definition is the following:

```
mapping for LegacyType control ControlType corresponding
{
    EntityField1 = LegacyField1;
    EntityField2 = LegacyField2;
    ...
}
```

 For more information, see [Using Type and Control Mapping \[page 477\]](#).

Integrating Additional Save in Managed Business Objects

You can now integrate Additional Save for each entity of a given business object with managed implementation type. This is done in the behavior definition of the business object by adding the keyword `with additional save`. The actual implementation of Additional Save takes place in a local saver class as part of the behavior pool.

 For more information, see [Integrating Additional Save in Managed Business Objects \[page 217\]](#).

Integrating Unmanaged Save in Managed Business Objects

You can now integrate Unmanaged Save within the transactional life cycle of managed business objects. In order to integrate Unmanaged Save into the save sequence as a part of the managed runtime, you must first add the corresponding syntax (`with unmanaged save`) to the behavior definition and then implement the saver handler method as a part of the behavior pool.

 For more information, see [Integrating Unmanaged Save in Managed Business Objects \[page 227\]](#).

Using Groups for Large Implementations

Until now, the implementation of business object entity's operations and actions had to be done in the *Local Types* include of the behavior pool associated with that entity. Since the *Local Types* include can only be changed by one developer at a time, the efficiency of development would be significantly reduced in case of larger implementations. As a solution, the groups can be now used to divide operations, actions and other implementation-relevant parts of the business logic into several groups for behavior implementation.

 For more information, see [Using Groups in Large Development Projects \[page 481\]](#).

Using Virtual Elements

You can now define elements in CDS projection views that are not persisted on the database. These virtual elements are defined with the new syntax element `virtual`. Their calculation is done in a separate ABAP class, which is called during runtime via an ABAP code exit for virtual elements.

 For more information, see [Using Virtual Elements in CDS Projection Views \[page 440\]](#).

Cloud Using Service Binding Editor

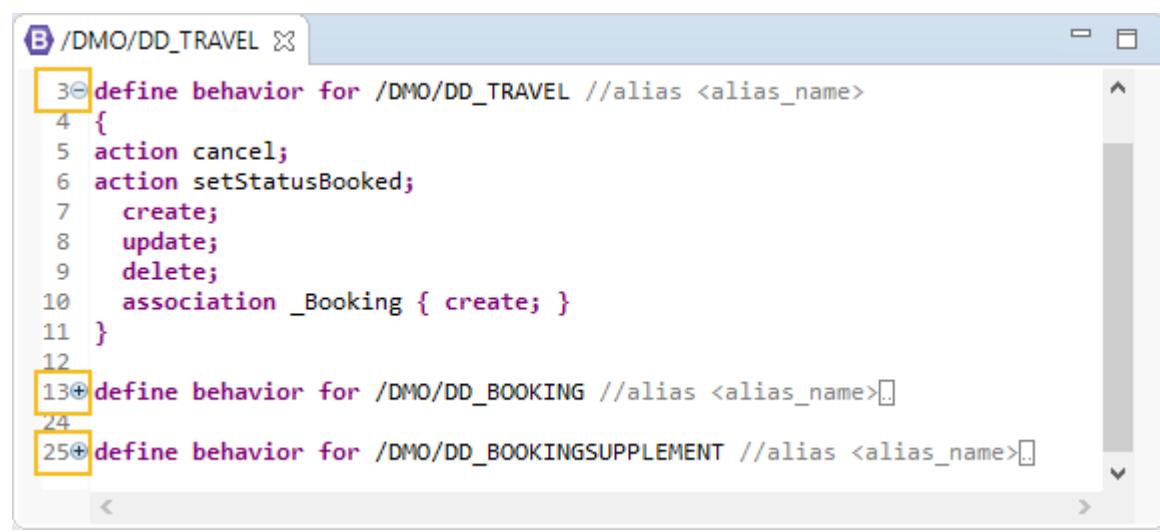
Previewing Fiori Elements App

In the Service Binding editor, the preview of the Fiori elements app now automatically opens in the logon language.

i For more information, see [Using Service Binding Editor \[page 771\]](#).

Cloud Using Code Folding for Behavior Definitions

Code folding is now supported for behavior definitions.



Code Folding in the Behavior Definition

9.4 Cloud Version 1908

Cloud New URL for the Documentation of the ABAP RESTful Programming Model in SAP Cloud Platform ABAP Environment

The documentation of the [ABAP RESTful Programming Model](#) for SAP Cloud Platform ABAP Environment on the SAP Help Portal has been moved.

i The new URL is <https://help.sap.com/viewer/923180ddb98240829d935862025004d6/Cloud/en-US/289477a81eec4d4e84c0302fb6835035.html>.

Developing New Managed Transactional Apps

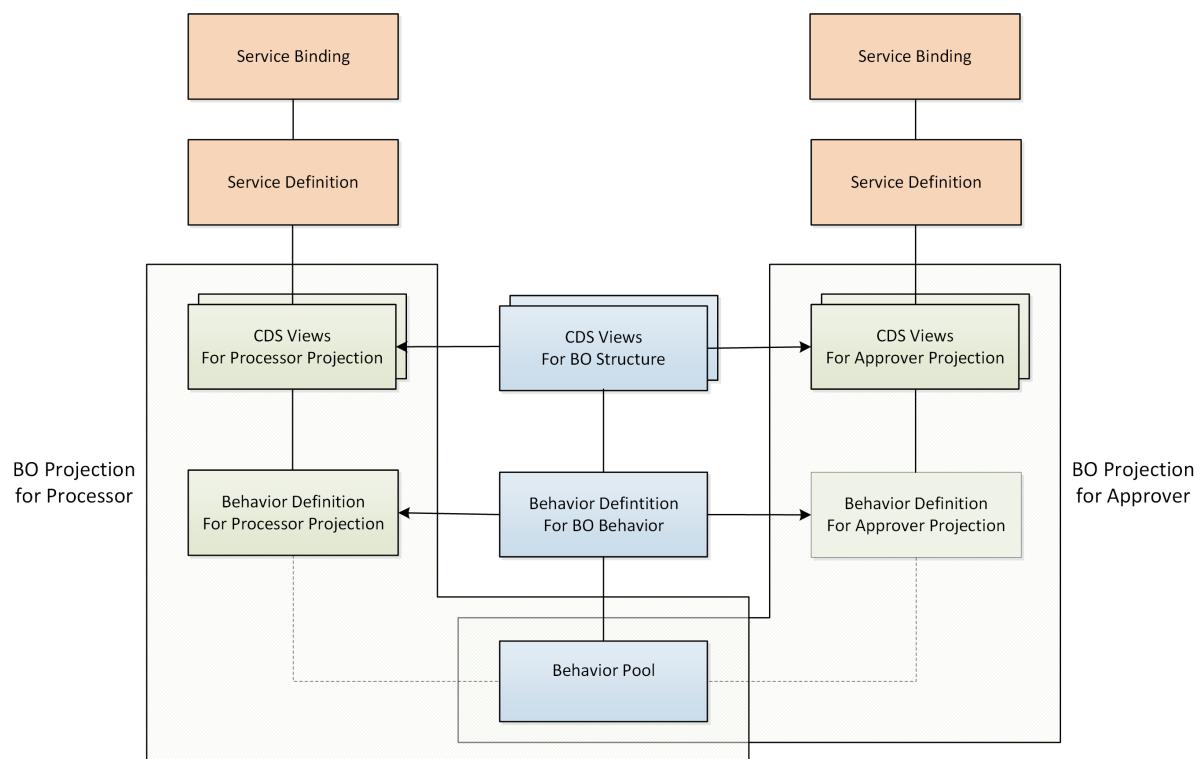
The ABAP RESTful Programming Model now supports the managed implementation type for developing new transactional apps. This scenario aims at use cases to develop new transactional apps from scratch. All required standard operations must only be specified in the behavior definition to obtain a ready-to-run business object. The business logic is implemented using actions, validations and determinations.

 For more information, see [Developing Managed Transactional Apps \[page 144\]](#).

Business Object Projection

You can now use the ABAP-native approach to project and to alias a subset of the business object for a specific business service. The projection enables flexible service consumption as well as role-based service designs.

With projections, it is possible to project one business object for different role-based UIs. An example is given in [Developing a Projection Layer for Flexible Service Consumption \[page 237\]](#). The travel business object is exposed for a processor Fiori UI and for an approver Fiori UI.



 For conceptual information about projections, see [Business Object Projection \[page 100\]](#).

Implementing an Unmanaged Query

A new API is available to implement an unmanaged query in a query implementation class. In an unmanaged query, the request is delegated to the query implementation class, which must implement the `select` method

the interface `IF_RAP_QUERY_PROVIDER`. The new interface provides methods to implement query capabilities, such as paging, filtering, sorting, or counting. It replaces the interface `IF_A4C_RAP_QUERY_PROVIDER`, which is deprecated as of 1908.

i For more information about the new API, see [Unmanaged Query API \[page 741\]](#).

The implementation of an unmanaged query with the interface `IF_RAP_QUERY_PROVIDER` is described in a new topic in the common task section.

i For more information, see [Implementing an Unmanaged Query \[page 494\]](#).

Server-Side Paging

Server-side paging has now been implemented to improve performance for OData calls. If the OData client does not provide paging query options, the default paging restricts the response to 100 data records. If `$top` is set to values greater than 5000, the response is reduced to 5000 data records.

Following from this, unmanaged queries require at least the implementation for paging. If the corresponding methods are not implemented and the unmanaged query does not return the respective information, there will be a dump during runtime.

i For more information, see [Implementing an Unmanaged Query \[page 494\]](#).

Additions to the Unmanaged Reference Scenario

The sources for the unmanaged reference scenario has been updated with the latest features. You can explore the new features in the sources that you can download from GitHub and in the related description in the guide on how to develop unmanaged transactional apps based on existing application logic.

- **Mapping CDS names to database field names**

By defining a mapping specification in the behavior definition, you can use the mapping operator in the behavior implementation to write records to the database that have a discrepancy between the names of the database table fields and the CDS view field names.

i For more information, see the information for mapping in [Adding Behavior to the Business Object \[page 289\]](#) and the implementation in [Implementing the CREATE Operation for Travel Instances \[page 305\]](#).

- **Message handling**

The behavior processing framework provides a message object that can be used for message handling in the behavior pool. New message handling has been added to the unmanaged scenario. The corresponding methods are implemented in an auxiliary class that inherits from `cl_abap_behv` and called from the behavior handler.

i For more information, see the information about message handling in [Implementing the CREATE Operation for Travel Instances \[page 305\]](#).

Authorizations Checks for Modifying Operations in Managed Business Objects

To protect data from unauthorized read access, the ABAP CDS provides its own authorization concept based on a data control language (DCL). The authorization checks for read operations allow you to limit the results returned by an entity to those results you authorize a user to see.

For business objects that are implemented for managed contract, also modifying operations in such as standard operations create, update, delete, create by associations, and actions must be checked against unauthorized access. With the current release, the instance-based authorization control is supported.

i For more information, see [Adding Authorization Control to Managed Business Objects \[page 487\]](#).

Adding Dynamic Feature Control

Apart from static feature control, you can now also use dynamic feature control. In this case, it depends on a state of the node instance if certain elements or actions are available.

Dynamic feature control is defined in the behavior definition:

```
...
define behavior for /DMO/I_TRAVEL_M
  field   (features : instance ) travel_id;
  action  ( features: instance ) acceptedTravel result [1] $self;
...
...
```

The implementation for the control must then be implemented in the respective methods in the behavior pool. For example, for this behavior definition, you can implement that the field `travel_id` is mandatory on create, but read-only on the update operation. The action can be implemented as disabled if the travel entity is already set to accepted, but enabled if it is not yet accepted.

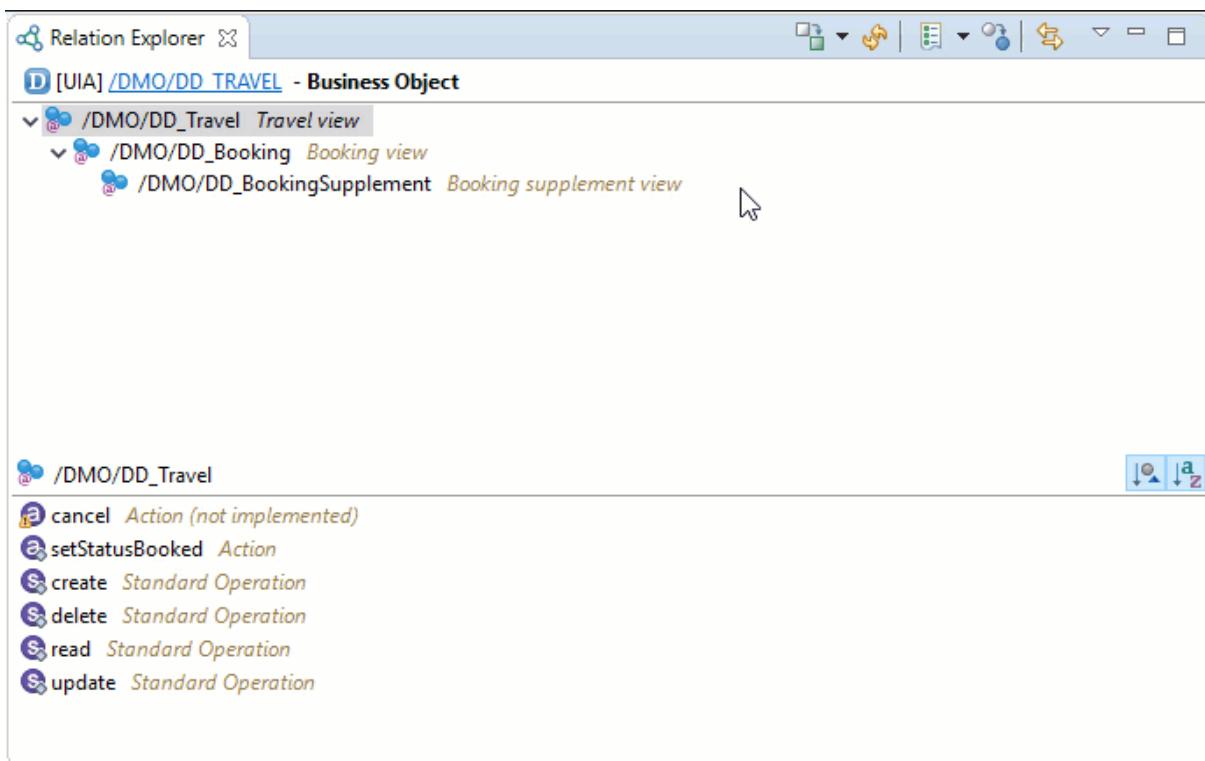
i For more information, see [Dynamic Feature Control \[page 463\]](#).

Exploring Business Objects

A business object consists of hierarchical connected entities. The behavior for each entity is defined in the behavior definition object and implemented in the behavior classes. In the [Relation Explorer](#), you can see structure and behavior of a certain business object independent of the technical location. You can navigate to all the entities and the corresponding behavior (definition and implementation).

Sometimes you might be interested in more CDS-specific aspects and want to see access control lists or test classes. You can achieve this by switching the context from **Business Object** to **Core Data Services** context, as you can see in the following animation.

Switching the context in the Relation Explorer



Relation Explorer provides much more features such as further contexts, for example, to display used or using objects for a certain class.

i For further information, see [Exploring Business Objects \[page 759\]](#).

Cloud Service Consumption Model Wizard

In the service consumption model creation wizard:

1. ETag support can now be selected for any entity set. If ETag support is marked in your edmx file, the Etag support checkbox is selected by default.
2. An entity set can now be selected for the generation. You can only edit the ABAP artifact name for the entity set that you've selected for generation.
3. Issues in an entity set are displayed and these entity sets cannot be selected for generation.

New Service Consumption Model

Define Entity Set

Select entity sets for generation, edit the ABAP artifact name and select ETag support. Entity sets with issues are not generated.

	Service Entity Set	ABAP Artifact Name	ETag S...	Issue
2	<input checked="" type="checkbox"/> A_BusinessPartner	ZA_BUSINESSPARTNER739111184C	<input checked="" type="checkbox"/>	1
	<input checked="" type="checkbox"/> A_BusinessPartnerAddress	ZA_BUSINESSPARTNERADDRESS	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_BusinessPartnerBank	ZA_BUSINESSPARTNERBA10EF53F5...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_BusinessPartnerRole	ZA_BUSINESSPARTNERRO84E9222...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_BusinessPartnerTaxNu...	ZA_BUSINESSPARTNERTAB0E4B215...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_CustSalesPartnerFunc	ZA_CUSTSALESPARTNERF159B283...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_Customer	ZA_CUSTOMERD6C7ECF4B8	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_CustomerCompany	ZA_CUSTOMERCOMPANY3D79525...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_CustomerDunning	ZA_CUSTOMERDUNNING55ECBD...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_CustomerSalesArea	ZA_CUSTOMERSALESAREA96C421...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_CustomerSalesAreaTax	ZA_CUSTOMERSALESAREA7A7938...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_CustomerWithHolding...	ZA_CUSTOMERWITHHOLDI5302D...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_Supplier	ZA_SUPPLIER617002ED75	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_SupplierCompany	ZA_SUPPLIERCOMPANY1BBDA39...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_SupplierDunning	ZA_SUPPLIERDUNNING8194E0C2D6	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_SupplierPartnerFunc	ZA_SUPPLIERPARTNERFUB236BA0...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_SupplierPurchasingOrg	ZA_SUPPLIERPURCHASIN8CD8D1...	<input type="checkbox"/>	
	<input checked="" type="checkbox"/> A_SupplierWithHoldingTax	ZA_SUPPLIERWITHHOLDIACA225...	<input type="checkbox"/>	
	<input type="checkbox"/> A_BPContactToAddress	ZA_BPCONTACTTOADDRESS	<input type="checkbox"/>	Type Edm.DateTime of element ValidityEnd...
	<input type="checkbox"/> A_BPContactToFuncAnd...	ZA_BPCONTACTTOFUNCANDDEPT	<input type="checkbox"/>	Type Edm.DateTime of element ValidityEnd...
	<input type="checkbox"/> A_BusinessPartnerContact	ZA_BUSINESSPARTNERCONTACT	<input type="checkbox"/>	Type Edm.DateTime of element ValidityEnd...
	<input type="checkbox"/> A_AddressHomePageURL	ZA_ADDRESSHOMEPAGEURL	<input type="checkbox"/>	Type Edm.DateTime of element ValidityStart...

Select All Deselect All

?

< Back Next > Finish Cancel

For further information, see .

9.5 Version 1905

Understanding Concepts

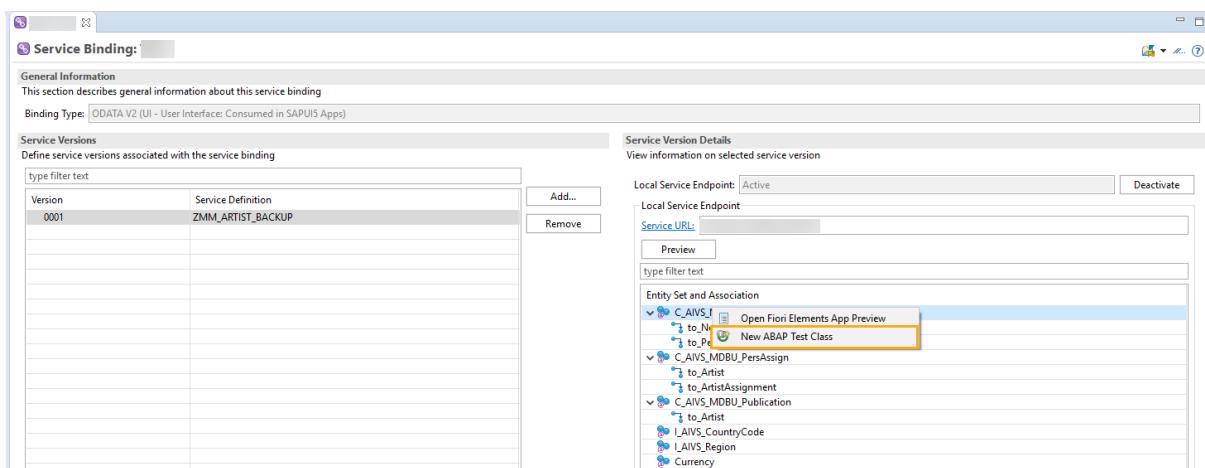
As an application developer, you may not only be interested on how you can implement different scenarios and use cases for your business applications by following the stateless programming paradigm of the ABAP RESTful programming model. It may also be important for you to understand the main concepts behind it.

The concepts section provides you with background information of ABAP RESTful programming model and helps you to understand main concepts from both, the design time, and runtime perspective.

i For further information, see [Concepts \[page 43\]](#).

Test Class Generation for Service Binding

You can now generate automated tests for an OData service that you've created using service binding. The test class provides guidance on how to access the OData service using ABAP Units and provides the test code for performing CRUD operations on an entity set.



i For more information, see [Using Service Binding Editor \[page 771\]](#).

Remote OData Access for Service Consumption Model

You can now view the code sample for performing the CRUD operations on an entity set belonging to a remote OData service.

Service Consumption Model: General Information

Object List

Service Entity Set	ABAP Artifact Name	Type
A_AddressFaxNumber	ZA_ADDRESSFAXNUMBER293CA760FA	Data Definition
A_SupplierPartnerFunc	ZA_SUPPLIERPARTNERFU09E3ABC63E	Data Definition
A_SupplierWithHoldingTax	ZA_SUPPLIERWITHHOLD4E59E6D7AE	Data Definition
	ZA_ADDRESSFAXNUMBER293CA760FA	Behavior Definition
	ZA_SUPPLIERPARTNERFU09E3ABC63E	Behavior Definition
	ZA_SUPPLIERWITHHOLD4E59E6D7AE	Behavior Definition

Code Sample for Entity Set Consumption

```

DATA: ls_entity_key      TYPE ZA_ADDRESSFAXNUMBER293CA760FA,
      ls_business_data  TYPE ZA_ADDRESSFAXNUMBER293CA760FA,
      lo_http_client     TYPE REF TO if_web_http_client,
      lo_client_proxy    TYPE REF TO /iwbep/if_cp_client_proxy,
      lo_resource        TYPE REF TO /iwbep/if_cp_resource_entity,
      lo_request         TYPE REF TO /iwbep/if_cp_request_read,
      lo_response        TYPE REF TO /iwbep/if_cp_response_read.

TRY.
  " Create http client
  " Details depend on your connection settings
  " lo_http_client = cl_web_http_client_manager->create_by_http_destination(
  "   " cl_http_destination_provider->create_by_cloud_destination(
  "     " i_name           = '<Name of Cloud
  Destination>'
  "   " i_service_instance_name = '<Service Instance Name>
  ) ).

  lo_client_proxy = cl_web_odata_client_factory->create_v2_remote_proxy(
    EXPORTING
      iv_service_definition_name = ' '
      iv_http_client            = lo_http_client
      iv_relative_service_root  = '<service_root>' ).

  " Set entity key
  " ls_entity_key = value #((
  "   addressid = 'Addressid'
  "   person      = 'Person'
  "   ordinalnumber = '1'
  "   ).

  " Navigate to the resource
  lo_resource = lo_client_proxy->create_resource_for_entity_set( 'A_ADDRESSFAXNUMBER'
) ->navigate_with_key( ls_entity_key ).
```

For more information, see

Defining Names for OData Entity Sets and Entity Types

You can now define external names for OData entity sets and entity types that are then used in the OData service metadata. The annotations `@OData.entitySet.name` and `@OData.entityType.name` can now be used in any CDS entity.

CDS Entity

```

@OData.entitySet.name: 'New_EntitySet_Name'
@OData.entityType.name: 'New_EntityType_Name'
define view /DMO/_Customer_OData
as select from /dmo/customer
{
  key customer_id,
  first_name,
  last_name
}

```

OData Metadata

```

<EntityType Name="New_EntityType_Name" sap:label="Test for Renaming" sap:content-version="1">
  <Key>
    <PropertyRef Name="customer_id"/>
  </Key>
  <Property Name="customer_id" Type="Edm.String" Nullable="false" MaxLength="6" sap:display-format="NonNegative" sap:label="Customer ID" sap:quickinfo="Flight Reference Scenario: Customer ID"/>
  <Property Name="first_name" Type="Edm.String" MaxLength="40" sap:label="First Name" sap:quickinfo="Flight Reference Scenario: First Name"/>
  <Property Name="last_name" Type="Edm.String" MaxLength="40" sap:label="Last Name" sap:quickinfo="Flight Reference Scenario: Last Name"/>
</EntityType>
<EntityContainer Name="cds_xdmcustomer_odata_Entities" m:isDefaultEntityContainer="true" sap:message-scope-supported="true" sap:supported-formats="atom json xslx">
  <EntitySet Name="New_EntitySet_Name" EntityType="cds_xdmcustomer_odata_New_EntityType_Name" sap:createable="raise" sap:updatable="false" sap:deletable="false" sap:content-version="1" />
</EntityContainer>

```

For more information, see [ObjectModel Annotations \[page 565\]](#).

9.6 Version 1902

Downloading the ABAP Flight Reference Scenario

You can now download the relevant development objects that are used in the course of the development guides via GitHub. This enables you to import a complete OData service into your system, which you can use and reuse for learning purposes.

Limitation: The import of a service binding artifact is not possible. To complete the OData service, you need to create service binding in your own package.

For further information, see [Downloading the ABAP Flight Reference Scenario \[page 13\]](#).

Entity Manipulation Language (EML)

Entity Manipulation Language (in short: EML) is a part of the ABAP language that is used to implement the business object's behavior in the context of ABAP RESTful programming model. It provides a type-safe read and modifying access to data in transactional development scenarios.

For further information, see [Entity Manipulation Language \(EML\) \[page 116\]](#).

Extension of Transactional Development Guide

The development guide [Developing Unmanaged Transactional Apps \[page 263\]](#) was extended with a 3-tier entity hierarchy. Booking Supplements are now part of the business object.

For further information, see [Adding Another Layer to the Transactional Data Model \[page 354\]](#).

In addition, the travel business object can now be consumed using EML syntax.

For further information, see [Consuming Business Objects with EML \[page 469\]](#).

Service Binding UI

The service binding tools come with a new UI design:

1. The binding type does not have a value populated by default.
2. The *Publish* and *Unpublish* buttons have been renamed to *Activate* and *Deactivate*.
3. Entity Set and Association was displayed with the service URL as the root node. Now, the fields are separate and it is displayed only if the local service endpoint has been activated.

4. The Preview button is now available on the interface for previewing the SAP Fiori Elements App. This is applicable for OData V2 UI service only.
5. The local service endpoint information now displays the service URL when it is in the activated state.

The top screenshot shows the 'New Service Binding' wizard. It has fields for Project, Package, Name, Description, Original Language, Binding Type (marked with a red circle 1), and Service Definition. The bottom screenshot shows the 'Service Binding' configuration screen. It has sections for General Information, Service Versions, and Service Version Details. The Service Version Details section highlights the 'Local service endpoint: Active' status (red circle 2), the 'Preview' button (red circle 4), and the 'Entity Set and Association' list which includes 'SalesOrder_TP_SOL' (red circle 3).

For further information, see [Service Binding \[page 111\]](#).

Consuming Services

The service consumption model replaces the OData client proxy to generate service artifacts for an OData service. It comes with a new wizard and an editor to work on the generated artifacts. The service consumption model artifact provides an overview of all abstract entities and generated behavior and service definitions that belong to the imported service.

For further information, see .

Transactional Behavior for the Service Consumption Scenario

The guide on how to consume a remote service has been extended with transactional capabilities for additional data. The use case of maintaining discount data in a local database is exemplified in this scenario.

For further information, see [Adding Transactional Behavior to the Business Object \[page 401\]](#).

Using Aggregate Data in SAP Fiori Apps

Aggregate functions, such as sum, maximum, minimum and average, as well as a counting option are now available in the ABAP environment to be implemented in CDS and displayed in your SAP Fiori App. Annotations are used to mark the elements as measures, whose values can be aggregated.

For further information, see [Using Aggregate Data in SAP Fiori Apps \[page 446\]](#).

Adding Static Feature Control in SAP Fiori Apps

In a typical transactional scenario, you have to specify which operations should be provided by the whole entity or you must specify which fields of an entity have specific access restrictions (read-only or mandatory fields).

For further information, see [Static Feature Control \[page 460\]](#).

Freely Selectable Name for Handler Methods in Behavior Pools

The method name in handler classes is now freely selectable. What kind of method it is, is expressed by the FOR clause.

The old syntax METHODS modify FOR BEHAVIOR ... becomes now: METHODS FreeMethodName FOR MODIFY

The old syntax METHODS read FOR BEHAVIOR ... becomes now: METHODS FreeMethodName FOR READ

The old syntax METHODS lock FOR BEHAVIOR ... becomes now: METHODS FreeMethodName FOR LOCK

Note that the old syntax remains valid but is no longer recommended!

For further information, see [Handler Classes \[page 719\]](#).

9.7 Version 1811

Defining Service Versions

The service binding tools come with a new UI design and some additional functions for versioning of (business) services.

For further information, look at [Service Binding \[page 111\]](#).

Support for Compositions

The current version of the ABAP RESTful programming model supports compositions: A business object consists of a tree of nodes where the nodes are linked by means of a special kind of associations, the compositions. A composition is specialized association that defines a whole-part relationship. A composite part only exists together with its parent entity (whole).

For further information, look at [Providing CDS Data Model with Business Object Structure \[page 269\]](#).

Developing an A2X Service

It is possible to publish an OData service as an application-to-cross application (A2X) service. That means the service is published without information relevant for a UI service (for example Value Helps or annotation to define a UI). An A2X service facilitates the exchange of business information between an application and any client, including from a different system or server.

For more information, look at [Developing a Web API \[page 365\]](#).

Developing a UI Service with Access to a Remote Service

With the help of the service consumption model it is now possible to consume a Web API service and build a new SAP Fiori application by consuming the remote service. This development guide includes the definition of a custom entity and implementing an custom query.

For more information, look at [Developing a UI Service with Access to a Remote Service \[page 369\]](#)

10 Glossary

ABAP Compiler

ABAP compiler creates a byte code as interim code when generating a program from the ABAP source code. This interim code is stored in the database as a load program and is loaded to Program Execution Area (memory for managing the fixed data of an ABAP program while it is being executed) when required.

ABAP Development Tools (ADT)

An ABAP-integrated development environment built on top of the Eclipse platform. Its main objective is to support developers by offering state-of-the-art ABAP development tools. These tools include strong and proven ABAP life-cycle management on the open Eclipse platform with powerful UI capabilities.

ABAP Dictionary

Persistent storage for data types that are visible in all repository objects. In addition, the database tables of the central database, views, and lock objects are managed in the ABAP Dictionary - among other things.

ABAP Flight Reference Scenario (in short: Flight Scenario)

SAP's reference scenario based on an updated flight data model. It is intended to be used for demonstration and learning purposes in the context of the ABAP RESTful programming model.

ABAP RESTful Programming Model

An ABAP programming model for browser-based applications that are optimized for SAP HANA.

ABAP Runtime Environment (Virtual Machine)

Processes of the ABAP runtime environment control the execution of an ABAP program by calling the processing blocks of the program. The ABAP runtime environment is provided by the Application Server ABAP.

ABAP SQL

A subset of SQL realized using ABAP statements. ABAP SQL is used to read (`SELECT`) and modify (`INSERT`, `UPDATE`, `MODIFY`, or `DELETE`) data in database tables defined in ABAP Dictionary. Database tables, views, and all non-abstract CDS entities can be accessed directly.

Action

A [modify operation \[page 816\]](#) that is a part of the behavior of a business object. Actions can be related to the [instances of a business object \[page 806\]](#) (default) or are static. Actions can have input parameters and a result with a cardinality.

Additional Save

A processing step within the transactional life cycle of a [managed business object \[page 815\]](#). It allows an external functionality to be invoked during the save sequence (after the managed runtime has written the changed data of the business object's instances to the database, but before the final commit work is executed).

Additional save is defined in the [behavior definition \[page 806\]](#) of a managed business object and implemented in the related behavior pool.

Association

A relationship between two entities of a business object's [data model \[page 811\]](#).

An association is a directed connection between two nodes (source and target) of BO structures.

Association Path

An association path is a sequence of [associations \[page 804\]](#) connecting entities with each other.

Before Image

The before image denotes a data set (data image) without transactional changes in the current LUW.

Behavior Definition Language (in short: BDL)

Declarative language for behavior modeling of business objects in the context of ABAP RESTful programming model. The language is syntactically oriented to CDS. Technically however, BDL artifacts are not managed by the [ABAP Dictionary \[page 803\]](#), but by the [ABAP compiler \[page 803\]](#).

The corresponding source code artifact in ABAP repository is the [\(business object\) behavior definition \[page 806\]](#).

Behavior Characteristic

A part of the business object's behavior that specifies general properties of an entity such as late numbering, ETag, draft handling, or feature control.

Behavior Pool

A special global ABAP class that implements the [business object's behavior \[page 806\]](#) specified in the [behavior definition \[page 806\]](#).

The real substance of a behavior pool is located in Local Types. Here, the ABAP developer can define two types of special local classes, namely handler classes for the operations within the interaction phase and saver classes for the operations within the save sequence. These classes can be instantiated or invoked only by the kernel.

Business Object (in short: BO)

In the ABAP RESTful programming model, a business object provides the following:

- A [data model \[page 811\]](#) which explicitly defines the structure of the data (the relationships within the data, the semantics of the data and the data constraints)
- The [behavior \[page 806\]](#) which defines the
 - capabilities of the data (create, update, or delete)
 - An association is a directed connection between two nodes (source and target) of BO operations which can be performed on the data (actions, determinations, or validations)
 - transactional properties of the data model (such as draft enabled, or the implementation type).
- The runtime Implementation

The ABAP RESTful programming model uses ABAP CDS to define the data model for business objects. Each BO contains one distinguished root node which is the leading entity within the BO. Furthermore, nodes within a BO are connected by [compositions \[page 808\]](#). All entities which can be reached by the transitive tree of compositions starting at the root entities belong to the BO structure.

The data model's [behavior \[page 806\]](#) is defined and implemented in a [behavior definition \[page 806\]](#) and [behavior implementation \[page 806\]](#) respectively.

(Business Object) Behavior

A behavior characterizes a business object in the ABAP RESTful programming model.

It includes a [behavior characteristic \[page 805\]](#) and a set of [operations \[page 816\]](#) for each [entity \[page 806\]](#) of the BO. To specify the business object's behavior, the [behavior definition \[page 806\]](#) as the corresponding development object is used.

(Business Object) Behavior Definition

The behavior definition is an ABAP repository object that is used to specify the [business object's behavior \[page 806\]](#).

(Business Object) Behavior Implementation

The behavior implementation is an ABAP class that implements the [business object's behavior \[page 806\]](#).

(Business Object) Entity

A node in a [business object's \[page 805\]](#) structure. In the ABAP RESTful programming model, an entity is used as the [composition \[page 808\]](#) unit of a business object structure.

An entity can be a root, parent, child, or a leaf entity.

(Business Object) Instance

A concrete occurrence of an [entity \[page 806\]](#).

(Business Object) Metadata

Meta information about an [entity of a business object \[page 806\]](#).

For example: what actions belong to a certain entity (BO)?

Business Object Projection

A subset of a business object data model and/or business object behavior. A business object that is designed for general purpose can be restricted for a specific business service in a BO projection. One of the most prominent examples is the `Business Partner`, which can be projected as `Customer`, `Supplier`, or `Vendor`.

Business Service

A business service is a RESTful service which can be called by a client. It consists of a [service definition \[page 807\]](#) and a [service binding \[page 807\]](#).

(Business) Service Binding

A service binding is an ABAP repository object used to bind a [service definition \[page 807\]](#) to a client server communication protocol such as OData (HTTP).

(Business) Service Definition

A service definition is an ABAP repository object defining the [CDS entities \[page 809\]](#) that are exposed for an OData service, including their [behavior \[page 806\]](#).

Child Entity

In ABAP CDS, [entities \[page 806\]](#) are connected using compositions. A child entity is a CDS entity which is the target of a [composition \[page 808\]](#).

Composition

A specialized association [page 804] that has a whole-part relationship. A composite part only exists together with its [parent entity](#) [page 817] (whole). Compositions are defined in [CDS entities](#) [page 809] using the keyword COMPOSITION OF.

Composition Path

The composition path is a sequence of [compositions](#) [page 808] connecting nodes with each other.

Composition Tree

A [composition](#) [page 808] tree represents the hierarchy of nodes in a [business object's](#) [page 805] structure where the nodes are linked by the composition relationship.

Each node of a composition hierarchy has entities that are modeled in the ABAP RESTful programming model using [CDS entities](#) [page 809] where the [root](#) [page 818] is the top node in the business object's structure.

Core Data Services (CDS)

CDS provides an infrastructure for defining and consuming semantically rich [data models](#) [page 811] in SAP HANA.

In particular, ABAP CDS provides a framework for defining and consuming semantic data models on the central database of the application server AS ABAP. The specified data models are based on the data definition language (DDL) and the data control language (DCL).

CDS Abstract Entity

A [CDS entity](#) [page 809] defined using the keyword `DEFINE ABSTRACT ENTITY` in a [CDS data definition](#) [page 811].

An abstract entity defines the type attributes of a CDS entity without creating an instance of a database object.

CDS Access Control

Concept for implicit restrictions on access to CDS entities.

CDS access control can be applied to CDS entities and is enabled by default for every CDS entity. It can be disabled for individual entities using an entity annotation.

CDS Annotations

CDS annotations describe semantics related to business data.

An annotation enriches a definition of a [CDS \[page 808\]](#) object with metadata going beyond the syntactical features offered by SQL. It can be specified for specific scopes of a CDS object, namely specific places in a piece of CDS source code.

CDS Custom Entity

A [CDS entity \[page 809\]](#) defined using the keyword `DEFINE CUSTOM ENTITY` in a [CDS data definition \[page 811\]](#). A CDS custom entity is a non-SQL CDS entity with an [unmanaged query \[page 821\]](#) runtime implemented in ABAP.

CDS Entities

ABAP CDS entities (also referred to as CDS entities) are [data models \[page 811\]](#) based on the DDL (Data Definition Language) specification and are managed by ABAP Dictionary.

Currently, the following types of ABAP CDS entities are supported:

- [CDS view \[page 810\]](#)
- CDS table function
- [CDS abstract entity \[page 808\]](#)
- [CDS custom entity \[page 809\]](#).

CDS Metadata Extension

A CDS object (of the ABAP CDS) defined in a piece of DDL source code using language elements in CDS DDL. In a metadata extension (MDE), [CDS annotations \[page 809\]](#) are specified for a CDS entity outside of the corresponding data definition. A CDS metadata extension is always assigned to a layer such as industry, partner or customer and can be joined using a CDS variant, in order to control its evaluation.

CDS Projection View

Result of CDS view projection. A CDS projection is defined in a data definition in which you can define the service-specific projected data model, a subset of the data model of the general business object.

CDS Views

An ABAP CDS view (also referred to as a CDS view) is defined for existing database tables, database views, or for other CDS views by using the ABAP CDS statement `DEFINE VIEW` within a DDL source.

A CDS view defines the structure of an SQL view and represents a projection onto one or more ABAP Dictionary tables or ABAP Dictionary views.

For each CDS view, two objects are created in the ABAP Dictionary:

- An SQL view
- The actual CDS entity.

Create Operation

A create operation is an [operation \[page 816\]](#) that implements the creation of persistent instances of entities (BOs).

Create-by-Association Operation

A [modify operation \[page 816\]](#) that is used to create instances of the associated (child) entity by the source of the association (parent entity).

CSDL XML File

The Common Schema Definition Language (CSDL) defines specific representations of the entity data model (EDM) exposed by an OData service, for example in an XML format.

Data Control Language (DCL)

A subset of SQL statements for executing authorization and consistency checks in relational databases.

The Application Server ABAP maps the functions of the data control language onto constructs such as authorizations objects and locks.

Data Definition

ABAP development object used to define an ABAP [CDS entity \[page 809\]](#) (for example, a CDS view).

After creating a *data definition*, the developer is able to use the standard functions of the ABAP Workbench - such as syntax check, activation, or connecting to the *Transport Organizer*. The developer creates a *data definition* using a wizard in *ABAP Development Tools*.

Data Model

Set of entities that represents a specific self-contained business object and is used to define a people-centric view of respective business information.

Delete Operation

A delete operation is an [operation \[page 816\]](#) that implements the deletion of persisted instances of entities (BOs).

Derived (Data) Type

The ABAP compiler allows the creation of derived types for the type-safe parametrization of the BO provider code. Such data types are referred to as derived types because they are implicitly derived by the compiler from CDS entity types and their [behavior definition \[page 806\]](#).

Determination

A determination is an implicitly executed action that is used to handle side effects of modifications by changing instances and returning messages.

Early Numbering

A numbering concept by which newly created entity instances are given a definitive key value during the [interaction phase \[page 814\]](#) on the [create operation \[page 810\]](#).

Element

An integral part of an [entity \[page 806\]](#). An element can be a [field \[page 813\]](#) or an [association \[page 804\]](#).

EML

Entity Manipulation Language (in short: EML) is a part of the ABAP language that is used to implement the [business object's behavior \[page 806\]](#) in the context of ABAP RESTful programming model. It provides a type-safe read and modifying access to data in transactional development scenarios.

ETag (Entity Tag)

An ETag is a [field \[page 813\]](#) that is used to determine changes to the requested resource. Usually, fields like last changed timestamp, hash values, or version counters are used as ETags.

An ETag can be used for optimistic concurrency control in the OData protocol to help prevent simultaneous updates of a resource from overwriting each other. An ETag check is used to determine whether two representations of a business [entity \[page 806\]](#), are the same. Whenever the representation of the entity changes, a new and different ETag value is assigned.

External Numbering

A numbering concept by which newly created entity instances are given their values by external BO consumers, e.g. a [Fiori \[page 819\]](#) UI.

Factory action

A special action that is used for creating new instances.

Feature Control

A functionality that provides property settings for [fields \[page 813\]](#), [entities \[page 806\]](#), [actions \[page 804\]](#), or [associations \[page 804\]](#) of a given [business object \[page 805\]](#)

These settings control the [behavior of a business object \[page 806\]](#) when it is in a certain state.

On the user interface, these settings control, for example, the following:

- Make fields mandatory, read only, editable, and/or invisible
- Enable/disable buttons

The feature control is either static (valid for all instances of an entity) or dynamic (depends on the state of the node instances).

Field

An [element \[page 812\]](#) of an [entity \[page 806\]](#) (business object), which represents a data object.

Fields are either [persistent \[page 817\]](#) or [virtual \[page 822\]](#).

Full Text Searching

Full text searching (or just text search) provides the capability to identify natural-language terms that satisfy a query and, optionally, to sort them by relevance (ranking) to the query.

Function

A [read operation \[page 818\]](#) that is a part of a business object's behavior. Functions are defined similarly to [actions \[page 804\]](#), but they do not cause any side effects.

Fuzzy Search

Fuzzy search is a fast and fault-tolerant search feature of SAP HANA. The concept behind the fault-tolerant search means that a database query returns records even if the search term (user input) contains additional or missing characters, or other types of spelling errors.

Instance Action

An [action \[page 804\]](#) that operates on a specific instance of a BO entity.

Interaction Phase

A part of the BO runtime where a consumer calls the business object's operations to modify or read business data in a transactional context.

A user triggers the interaction phase by clicking the [EDIT](#) button on UI. The interaction phase ends when the user clicks the [SAVE](#) button on UI.

Internal Action

An action that can only be executed from the business logic inside the same business object the action is assigned to, such as from a [determination \[page 811\]](#) or another action.

Internal Numbering

A numbering concept by which newly created entity instances are given their values by BO internal logic, e.g. by the managed runtime framework.

Late Numbering

Late numbering is a concept by which new entity [instances \[page 806\]](#) are given a definitive key just before they are saved on the database.

Leaf Entity

The leaf entity is an [entity \[page 806\]](#) in a business object's structure without any [child entities \[page 807\]](#).

A leaf entity is a CDS entity which is the target of a [composition \[page 808\]](#) (a child entity) but does not connect further entities (does not contain a composition definition).

Lock

The ability to protect data of entities from concurrent accesses by multiple users.

An [entity \[page 806\]](#) is locked using the enqueue mechanism.

Lock Master

A lock master defines the property of entities to be locked on themselves. This is currently only supported for [root entities \[page 818\]](#) of [business objects \[page 805\]](#).

Lock Dependent

A lock dependent is an [entity \[page 806\]](#) that depends on the locking status of a [parent \[page 817\]](#) or [root \[page 818\]](#) entity.

LUW (Logical Unit of Work)

When data in database tables is modified by application programs, it must be ensured that the data is consistent after the changes have been made. This is particularly important when data is edited in the database. The time span in which a consistent data state is transferred to another consistent state is known as an LUW (Logical Unit of Work).

Managed

The `managed` property defines an implementation type of a [business object \[page 805\]](#) or a [query \[page 817\]](#) provider in the context of the ABAP RESTful programming model. See also: [unmanaged \[page 821\]](#)

For the implementation type `managed`, the generic runtime framework assumes standard implementation tasks. The business logic is implemented by the application developer via `validation`, `determinations` and `actions`.

Managed Business Object

Business object with implementation type [managed \[page 815\]](#).

Managed Runtime

Runtime for processing business objects with implementation type [managed \[page 815\]](#).

Managed Save

A processing step within the save sequence of a [managed business object \[page 815\]](#).

Modify Operations

Umbrella term for [operations \[page 816\]](#) causing business data changes in the context of [behavior implementation \[page 805\]](#). It includes standard operations (create, update and delete) and [action \[page 804\]](#) execution. implementation.

OData

The Open Data (in short: OData) protocol is a Web protocol for querying and updating data. It applies several Web technologies, such as HTTP, Atom Publishing Protocol, and JSON to provide access to information from a variety of applications.

OData is based on industry standards and offers database-like access to business data using a REST-based (**Representational State Transfer**) architecture.

OData Client Proxy

A proxy for an OData client that acts as an intermediary to forward and transform requests from one service to another one.

OData Service

A service that is implemented in accordance with [OData \[page 816\]](#) protocol. OData services are used to expose data to consumers.

Operation

A procedure performed on an [entity \[page 806\]](#) (or a set of entities).

Example are changing operations create, update, delete that are performed within a transactional life cycle of a [business object \[page 805\]](#).

Orchestration Framework

Runtime framework for request dispatching and runtime checks within the ABAP RESTful programming model.

The technical term for this framework is [SADL \[page 818\]](#).

Parent Entity

The parent entity is an [entity \[page 806\]](#) in a [business object's \[page 805\]](#) structure that is directly connected to another entity when moving towards the root node.

In ABAP CDS, entities are connected using [compositions \[page 808\]](#). A parent entity is a CDS entity which contains a composition definition (keyword COMPOSITION OF).

Persistent Field

A [field \[page 813\]](#) of a CDS entity that is persisted in a database table.

Projected Entity

CDS entity whose elements are projected in a projection view. The projected entity is specified in the PROJECTION ON clause of the CDS projection view.

Projection

Building a subset of a BO data model or BO behavior.

Query

In the ABAP RESTful programming model, a query provides the following:

- A CDS [data model \[page 811\]](#)
- Capabilities that are either explicitly modeled via [CDS annotations \[page 809\]](#) (search, aggregation, ...) or generally applicable (paging, sorting, filtering)
- A runtime that is either managed by the query framework or unmanaged, which means implemented by the developer

In contrast to the [BO \[page 805\]](#) transactional capabilities, query capabilities are always read-only and do not modify data on the database

Query Implementation Class

The query implementation class is the class that is referenced by a [custom entity \[page 809\]](#) to implement its [query \[page 817\]](#).

Read Operation

Umbrella term for [operations \[page 816\]](#) that do no change any business data in the context of [business object behavior implementation \[page 806\]](#). It included operations such as read, read by association, and functions.

Create-by-Association Operation

A [read operation \[page 818\]](#) that is used used to read instance data of the associated (child) entity by the source of the association (parent entity).

Root Entity

The root entity is the top [entity \[page 806\]](#) in a [business object's \[page 805\]](#) structure. In ABAP CDS, a root entity is defined using the keyword `ROOT` in the [data definition \[page 811\]](#).

SADL

Service Adaptation Description Language (in short: SADL) is an ABAP technology that enables the consumption of entity relationship-like [data models \[page 811\]](#) in ABAP based on a model-driven approach.

In the context of SAP HANA, SADL enables fast read access to database data for scenarios on mobile and desktop applications using query push-down.

SAP Cloud Platform

SAP Cloud Platform is an open platform as a service (PaaS) that provides customers and partners with in-memory capabilities, core platform services, and unique business services for building and extending personalized, collaborative, mobile-enabled cloud applications.

SAP Cloud Platform ABAP Environment

SAP Cloud Platform ABAP Environment is part of the [SAP Cloud Platform \[page 818\]](#) and offered as Platform as a Service (PaaS). ABAP Environment provides a special variant of the ABAP platform and supports a subset of the ABAP language.

SAP Fiori (UX)

SAP Fiori is a new user experience (UX) for SAP software that applies modern design principles. SAP solutions, such as the SAP Business Suite powered by SAP HANA, use the SAP Fiori UX to provide a personalized, responsive, and simple user experience.

SAP Gateway

An ABAP infrastructure that exposes back-end services to consumer applications.

SAP Web IDE

A browser-based development tool set for modeling and developing Fiori UIs.

Save Sequence

Part of the BO runtime when data is persisted after all changes were performed.

Service Consumption Model

A set of artifacts that are generated in ABAP Development Tools on the basis of an entity data model XML (CSDL) file and are used to provide a generic client for remote [OData service \[page 816\]](#) consumption.

Service Definition Language (SDL)

Declarative language for defining [service definition \[page 807\]](#) objects. The language is syntactically oriented to CDS.

Service Proxy API

An ABAP API using the service consumption model [page 819] to consume remote OData services [page 816].

Static Action

An action [page 804] that operates independent of a specific instance of an entity.

To-Parent Association

A to-parent association in ABAP CDS is a specialized association [page 804] which can be defined to model the parent-child relationship between two CDS entities [page 809].

Compositions [page 808] and to-parent associations are used to define the structure of a business object [page 805] which can be used in the ABAP RESTful Programming model.

Transactional Buffer

A part of the BO runtime used to store the state of the BO data that is used in the interaction phase [page 814] for modifying and read operations (in a transactional context) and which can be persisted during the save sequence [page 819].

Trigger Condition

The condition that needs to be fulfilled to execute a validation [page 822] or a determination [page 811].

A trigger condition consists of a trigger operation [page 821] (create, update, create by association) and a list of entity fields (trigger elements [page 820]) belonging to the same entity the validation/determination is assigned to.

Trigger Element

Elements of the assigned entity that trigger the validation [page 822]/determination [page 811] when affected by the trigger operations [page 821].

Trigger Operation

Operations on which a [validation \[page 822\]](#)/[determination \[page 811\]](#) is executed, for example create or update.

Trigger Time

The point in time when the [validation \[page 822\]](#)/[determination \[page 811\]](#) is executed during the [BO \[page 805\]](#) lifecycle. The trigger time is declared in the definition of the [validation \[page 822\]](#)/[determination \[page 811\]](#).

Unmanaged

The `unmanaged` property defines an implementation type of a [business object \[page 805\]](#) or a [query \[page 817\]](#) provider in the context of the ABAP RESTful programming model.

For the implementation type `unmanaged`, the application developer must implement essential components of the REST contract itself.

In this case, all required BO [operations \[page 816\]](#) (create, update, delete, or application-specific actions) must be specified in the corresponding [behavior definition \[page 806\]](#) for a BO before they are implemented in ABAP. In `managed` implementation types, on the other hand, a behavior definition is already sufficient to obtain a ready-to-run [business object \[page 805\]](#).

Unmanaged Business Object

Business object with implementation type [unmanaged \[page 821\]](#).

Unmanaged Query

An implementation type for the runtime of a [query \[page 817\]](#). In an unmanaged query, the query contract must be implemented by the application developer and is not managed by the query framework.

Unmanaged Lock

A feature of a [managed business object \[page 821\]](#) with which you can implement the lock mechanism manually. Like, in the [unmanaged \[page 822\]](#) scenario, the method `FOR LOCK` is called during runtime.

Unmanaged Runtime

Runtime for processing business objects with implementation type [unmanaged \[page 821\]](#).

Unmanaged Save

A processing step within the transactional life cycle of a [managed business object \[page 821\]](#) that prevents the business object's managed runtime from saving business data (changes) during the [save sequence \[page 819\]](#). In this case, the function modules (for update task) are called to save data changes of the relevant business object.

Unmanaged save is defined in the [behavior definition \[page 806\]](#) of a managed business object and implemented in the related behavior pool.

Update Operation

An update operation is an [operation \[page 816\]](#) that implements the update of instance data of entities (BOs).

Validation

A validation is an implicitly executed function that checks the consistency of entity instances that belong to a business object.

It is defined in the behavior definition and implemented in the related behavior pool.

Virtual Element

Element that is not persisted on the database but calculated during runtime.

A virtual element is declared with the statement `VIRTUAL` in [CDS projection views \[page 810\]](#).

Web API

An [OData service \[page 816\]](#) that is published without any UI specific metadata. It is not exposed for a UI context. Instead it provides an API to access the service by another client, including from a different system or server.

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.

About the icons:

- Links with the icon  : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
 - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
 - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon  : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

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.

Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

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.

